Burroughs

B 1000 Systems

FORTRAN 77

REFERENCE MANUAL

(RELATIVE TO MARK 10.0 RELEASE)

PRICED ITEM

Printed in U.S. America March 1983 1108867

Burroughs cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded using the Remarks form at the back of the manual, or may be addressed directly to Corporate Documentation, Burroughs Corporation, 1300 John Reed Court, City of Industry, California 91745, U.S.A.

LIST OF EFFECTIVE PAGES

Page	Issue	Page	Issue
Title	Original	10-1 thru 10-4	Original
ii	Original	11-1 thru 11-26	Original
iii	Original	12-1 thru 12-28	Original
iv	Blank	13-1 thru 13-32	Original
v thru xv	Original	14-1 thru 14-14	Original
xvi	Blank	A-1 thru A-11	Original
1-1 thru 1-4	Original	A-12	Blank
2-1 thru 2-2	Original	B-1	Original
3-1 thru 3-5	Original	B-2	Blank
3-6	Blank	C-1	Original
4-1 thru 4-6	Original	C-2	Blank
5-1 thru 5-7	Original	D-1 thru D-7	Original
5-8	Blank	D-8	Blank
6-1 thru 6-20	Original	E-1 thru E-3	Original
7-1 thru 7-5	Original	E-4	Blank
7-6	Blank	F-1	Original
8-1 thru 8-3	Original	F-2	Blank
8-4	Blank	G-1 thru G-59	Original
9-1 thru 9-13	Original	G-60	Blank
9-14	Blank	Index 1 thru 20	Original

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

LIST OF ILLUSTRATIONS

LIST OF TABLES

INTRODUCTION

The purpose of this manual is to provide an explanation of the implementation and use of the Burroughs B 1000 FORTRAN 77 programming language. The language is designed along the guidelines of the American National Standards Institute committee for FORTRAN 77 (ANSI X3.9-1978), along with extensions provided by Burroughs as programming aids, and to conform with the B 1000 system architecture.

This manual is designed to provide the FORTRAN 77 programmer with a source of reference information and is not a primer in the language. The manual is organized in a manner that provides ease of use as a reference document, beginning with basic concepts and proceeding to more complex concepts.

BASIC FORTRAN 77 CONCEPTS

Certain basic concepts concerning the FORTRAN 77 language are presented here prior to the description of the B 1000 implementation of this language.

A problem-solving system written in the FORTRAN 77 language is called a source program; a program which constitutes a self-contained processing structure is called an executable source program. Every executable FORTRAN 77 program consists of one or more program units which combine to form the complete processing structure. Among the program units are the required main program and as many subprogram units as necessary to complete the source program.

Each program unit is constructed of a series of items called statements. These statements specify the arithmetic operations which are to be executed, control the order in which program statements are to be performed, accomplish various program input and output functions (such as reading data records and printing the results of computations), or describe program data items and provide other program information without directly producing any actions during program execution.

Each program statement is constructed of a string of appropriate characters which are contained on one or more physical records (for example, punched cards). A set of these physical records can be input as a file to a special computer program called a compiler. The compiler first verifies that each source statement is syntactically correct, and then converts each program unit into FORTRAN 77 S-code and places this Intermediate Code Module (ICM) into an intermediate code file along with other ICMs of the same source program. When the S-code has been generated for the program units, the main program is reexamined to determine which subprograms are needed to execute the FORTRAN 77 program. The intermediate code file is then searched for the intermediate code modules of these subprograms. These subprograms, and any intrinsics from the intrinsics library file that are needed, are bound together with the S-code for the main program to create an executable program. The executable program can then be executed on the B 1000 system using the FORTRAN 77 interpreter. The interpreter causes the system hardware to perform the operations specified by the S-code and thus, the source program. For more detailed information regarding the function of S-code and its relation to the interpreter and the hardware, refer to the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982.

The B 1000 FORTRAN 77 compiler operates under the control of a Master Control Program (MCP). Similarly, the S-code generated by the compiler is executed under control of the MCP.

A FORTRAN program that was compiled with the FORTRAN 66 compiler must be recompiled with the FORTRAN 77 compiler to be able to run with the FORTRAN 77 interpreter.

SECTION 1
SYNTAX CONVENTIONS

RAILROAD DIAGRAMS

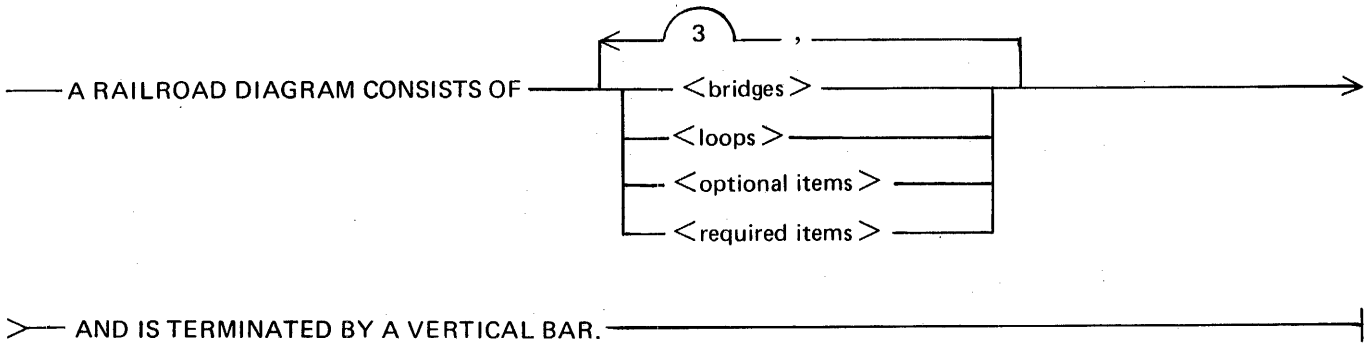
Railroad diagrams show how syntactically valid statements can be constructed.

Traversing a railroad diagram from left to right, or in the direction of the arrowheads, and adhering to the limits indicated by bridges produces a syntactically valid statement. Continuation from one line of a diagram to another is represented by a right arrow (→) or by a letter (for example, A, B, C) appearing at the end of the current line and beginning of the next line. The complete syntax diagram is terminated by a vertical bar (|).

Items contained in broken brackets (< >) are syntactic variables which are further defined, or require the user to supply the requested information.

Uppercase items must appear literally. Minimum abbreviations of uppercase items are underlined.

Example:



G50051

The following syntactically valid statements can be constructed from the above diagram:

A RAILROAD DIAGRAM CONSISTS OF <bridges> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <optional-items> AND IS TERMINATED BY A VERTICAL BAR.

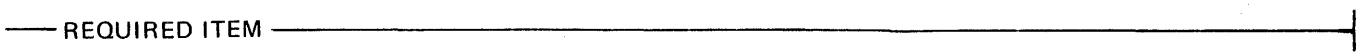
A RAILROAD DIAGRAM CONSISTS OF <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <optional-items>, <required-items>, <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR.

Required Items

No alternate path through the railroad diagram exists for required items or required punctuation.

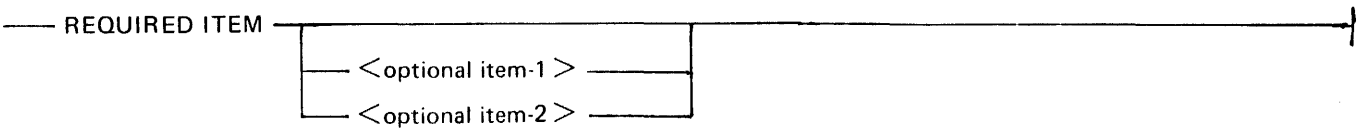
Example:



G50052

Optional Items

Items shown as a vertical list indicate that the user must make a choice of the items specified. An empty path through the list allows the optional item to be absent.



G50053

The following valid statements can be constructed from the above diagram:

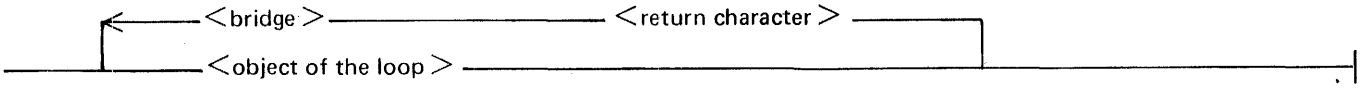
REQUIRED ITEM

REQUIRED ITEM <optional-item-1>

REQUIRED ITEM <optional-item-2>

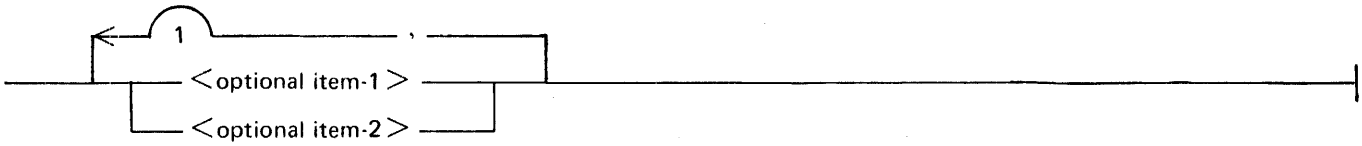
Loops

A loop is a recurrent path through a railroad diagram and has the following general format:



G50054

Example:



G50055

The following statements can be constructed from the previous railroad diagram:

<optional-item-1>

<optional-item-1>,<optional-item-1>

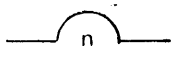
<optional-item-2>,<optional-item-1>

A <loop> must be traversed in the direction of the arrowheads, and the limits specified by bridges cannot be exceeded.

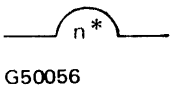
Bridges

A bridge indicates the minimum or maximum number of times a path can be traversed in a railroad diagram.

The following are two forms of <bridges>:



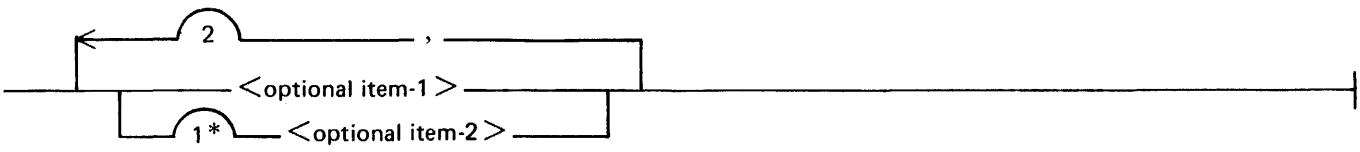
n is an integer which specifies the maximum number of times the path can be traversed.



n is an integer which specifies the minimum number of times the path must be traversed.

G50056

Example:



G50057

The loop can be traversed a maximum of two times; however, the path for <optional-item-2> must be traversed at least one time.

The following statements can be constructed from the railroad diagram in the example:

<optional-item-2>

<optional-item-1>,<optional-item-2>

<optional-item-2>,<optional-item-2>,<optional-item-1>

<optional-item-2>,<optional-item-2>,<optional-item-2>

SECTION 2
CHARACTER SET

Characters are the elements from which a language is constructed. The B 1000 FORTRAN 77 language is based upon a prescribed character set which is described in this section. Each type of character within this FORTRAN 77 character set is described in this section.

B 1000 FORTRAN 77 CHARACTER SET

For source program input, the B 1000 FORTRAN 77 character set consists of the following types of characters:

1. Digits

   1) Decimal digits
   2) Hexadecimal digits

2. Letters
3. Special Characters

Digits

Two types of digits are employed in the B 1000 FORTRAN 77 language: decimal digits and hexadecimal digits. Decimal digits are defined as consisting of the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. These digits are generally used to define program values in terms of the decimal (radix 10) number system. When the term "digit" is used in this manual, it refers to a member of the set of decimal digits.

Hexadecimal digits are defined as consisting of the characters in the decimal digit set plus the characters A, B, C, D, E, and F. These digits are generally used to define program values in terms of the hexadecimal (radix 16) number system; where A is equivalent to 10 in the decimal system, B is equivalent to 11 in the decimal system, and so forth.

These two digit types are used to represent numerical values in the B 1000 FORTRAN 77 language.

Letters

For the B 1000 FORTRAN 77 language, letters consist of the following 26 characters:

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

Special Characters

Special characters for the B 1000 FORTRAN 77 language consist of the following 16 characters:

	blank or space
.	period or decimal point
(	left parenthesis
+	plus sign
&	ampersand
$	dollar sign
*	asterisk
)	right parenthesis
–	minus sign
/	slash
,	comma
%	percent symbol
:	colon
'	apostrophe
=	equal sign
"	quotation mark

The blank character has a specific meaning only in string literals and in the FILE declaration statement (two blanks must follow the FILE statement). Blanks can be used throughout the program to improve readability.

Collating Sequence

The collating sequence of the character set is such that special characters are less than letters and letters are less than digits. Within each of these three groups, the collating sequence is the following:

1. Digits. Have the sequence as normally assigned to numbers; 1 is less than 2, 9 is greater than 7, and so forth.
2. Letters. Listed in ascending order under Letters, in this section.
3. Special Characters. Proper sequence, in ascending order, is listed under Special Characters in this section.

SECTION 3
PROGRAM STRUCTURE

The FORTRAN 77 programming language consists of procedures containing statements conforming to a general order. The classes of statements and their relative sequence in the FORTRAN 77 program are described in this section.

STATEMENTS

Every executable FORTRAN program (refer to Basic FORTRAN 77 Concepts in the Introduction) consists of a sequence of statements, with each statement physically contained on one or more lines, or on card images. These statements are classified as executable and nonexecutable statements.

Executable Statements

An executable statement is an instruction that causes action to be taken at the point in the program where the statement is executed. The FORTRAN 77 executable statements described in this document are as follows:

Assignment statement
BACKSPACE statement
CALL statement
CLOSE statement
CONTINUE statement
DO statement
ELSE statement
ELSE IF statement
END statement
ENDFILE statement
END IF statement
GO TO statement
IF statement
PAUSE statement
PRINT statement
PUNCH statement
READ statement
RETURN statement
REWIND statement
STOP statement
WRITE statement

Nonexecutable Statements

A nonexecutable statement is an instruction which gives information to the compiler regarding storage allocation, data initialization, I/O editing specifications, and program units. The FORTRAN 77 nonexecutable statements described in this document are as follows:

BLOCK DATA statement
COMMON statement
DATA statement
DIMENSION statement
ENTRY statement
EQUIVALENCE statement
Explicit type statement
EXTERNAL statement
FORMAT statement
FUNCTION statement
IMPLICIT statement
INTRINSIC statement
PARAMETER statement
PROGRAM statement
SAVE statement
Statement function statement
SUBROUTINE statement

Statement Ordering

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

1. Comment statements and Compiler Control Images can appear, according to their respective rules, at any point within a program. (For the sake of brevity, the rules following do not describe the relationship of comment statements and Compiler Control Images to the other valid FORTRAN 77 statements, but as stated previously, comment statements and Compiler Control Images can appear at any point.)
2. FILE declaration statements must precede all other statements of the main program.
3. PROGRAM is the first statement in the program following any FILE declaration statements. In a subprogram unit, the SUBROUTINE, FUNCTION, or BLOCK DATA statement must be first.
4. All specification statements must precede all DATA statements, statement function declaration statements, and executable statements. Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements except PARAMETER statements. Any specification statement that specifies the type of a symbolic name of a constant must precede the PARAMETER statement that defines that particular symbolic name of a constant. The PARAMETER statement must precede all other statements containing the symbolic names of constants that are defined in that PARAMETER statement.
5. All statement function declaration statements must precede all executable statements.
6. FORMAT statements can appear anywhere within a program unit.
7. ENTRY statements can appear anywhere within a program unit except between a block IF statement and the corresponding END IF statement, or between a DO statement and the terminal statement of the DO loop.
8. The last line of a program unit must be an END statement.

Figure 3-1 shows the required order of statements and comment lines.

Vertical lines delineate varieties of statements that can be interspersed. For example, FORMAT statements can be interspersed with statement function statements and executable statements. Horizontal lines delineate varieties of statements that must not be interspersed. For example, statement function statements must not be interspersed with executable statements. An END statement is also an executable statement and must appear only as the last statement of a program unit.

The recommended order of appearance of FORTRAN 77 statements in a program unit is as follows:

1. FILE declaration statement.
2. PROGRAM statement (This statement appears only in the main program; otherwise, the FUNCTION or SUBROUTINE statement appears here.)
3. IMPLICIT statements and PARAMETER statements.
4. DIMENSION, COMMON, INTRINSIC, EXTERNAL, or explicit type statements in any order.
5. EQUIVALENCE statements.
6. DATA statements.
7. Statement function declaration statements.
8. Remainder of program unit.
9. END statement.

Statement Labels

Statement labels provide a means of referring to individual statements. Any statement can be labeled, but only executable statements and FORMAT statements can be referred to by the use of statement

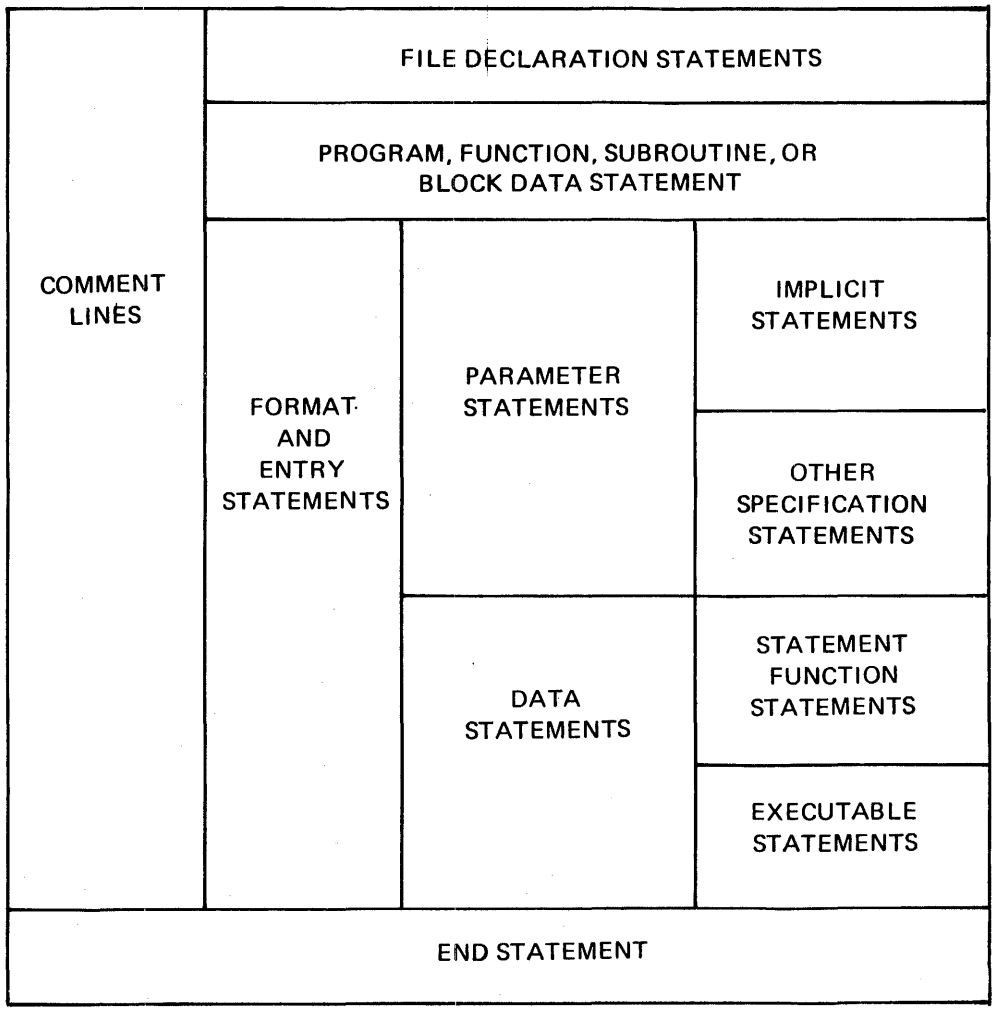


Figure 3-1. Required Order of Statements and Comments

labels. The form of a statement label is a sequence of one to five digits, at least one of which must be nonzero. The statement label can be placed anywhere in columns one through five of the initial line of the statement. The same statement label must not be given to more than one statement in a program unit. Blanks and leading zeros are not significant in distinguishing between statement labels.

Examples:

```
100 A=A+1
200 FORMAT (5I4,F7.2)
300 STOP
```

PROGRAM UNITS

Every executable FORTRAN 77 program consists of a main program unit which can be preceded and/or followed by as many subprograms as necessary.

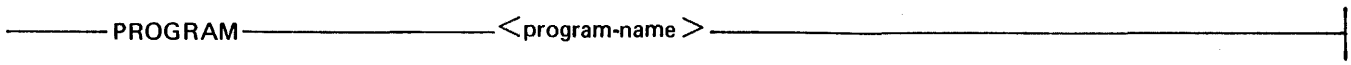
Main Program

A main program unit is a program unit that does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as the first statement. It can have a PROGRAM statement as the first statement.

There must be exactly one main program unit in an executable program. Execution of an executable program begins with the execution of the first executable statement of the main program.

PROGRAM Statement

The PROGRAM statement has the following form:



G50291

<program-name> is the symbolic name of the main program unit in which the PROGRAM statement occurs. The program name can contain up to six characters.

A PROGRAM statement is not required to appear in an executable program. If it does appear, it must precede any statement in the main program unit except any FILE statements.

The symbolic name <program-name> is global to the executable program and must not be the same as the name of an external procedure, block data subprogram, or common block in the same executable program. <program-name> must not be the same as any local name in the main program.

Examples of PROGRAM statements follow:

```
PROGRAM INVENT
PROGRAM HYPER
```

Main Program Restrictions

The PROGRAM statement can appear only as the first statement of a main program. A main program can contain any other statement except a BLOCK DATA, FUNCTION, SUBROUTINE, ENTRY, or RETURN statement. A main program cannot be referenced from a subprogram or from itself.

Subprograms

Subprograms (other than block data subprograms) are independent program units. A subprogram is referenced by a CALL statement or indirectly as a function in an expression. A subprogram can contain any FORTRAN 77 statement except a PROGRAM statement and a BLOCK DATA statement (unless the subprogram is a block data subprogram). For additional information on subprograms refer to section 13.

A block data subprogram is a special type of subprogram. Block data subprograms are described in section 13.

An END statement is required to complete every program unit. Refer to section 9 for a full description of the END statement.

SOURCE INPUT FORMAT

The compiler must receive FORTRAN 77 statements from cards, tape, or disk. Source input records are, in general, free-form format, with the following exceptions:

1. Columns 1 through 5 of a card can contain a statement label (refer to Statement Labels in this section). This field is recognized as a label on the first card only of an executable or FORMAT statement. Statement labels must not occur on continuation lines. A label without an associated statement causes a syntax error. Blanks and preceding zeros are ignored.
2. Column 6 of the first card of a statement must be blank or contain a zero. A statement can be continued on up to 19 records by placing any nonblank and nonzero character in column 6 of the continuation cards.
3. Columns 7 through 72 of a card contain the FORTRAN 77 statement.
4. Columns 73 through 80 can contain sequence numbers. This field is checked for ascending sequence numbering when $ MERGE or $ SEQCHECK is set; otherwise, the field is ignored.

A card containing a $ in column 1 is a Compiler Control Image as described in section 14.

Blank characters are significant only in column 6 of a statement, columns 5 and 6 of a FILE declaration, and in string literals. With these exceptions, blanks can be used freely without affecting the meaning of the FORTRAN 77 program.

Comments

If a line contains the letter C or an asterisk (*) character in column 1, or is entirely blank, the line is considered a comment line and is not interpreted. Any characters can follow the letter C or the asterisk on the same line without affecting program execution.

Comment lines can appear anywhere in the program unit and can precede the initial line of the first statement of any program unit. Comment lines can appear between an initial line and the first continuation line or between two continuation lines.

SECTION 4
CONSTANTS

This section explains the constants available in FORTRAN 77. Constants are formed from the FORTRAN 77 character set according to prescribed rules.

Constants function as FORTRAN 77 value data items used in problem solving and related operations such as input/output (I/O). Rules governing usage are described in this section.

Constants are classified into three types: 1) numeric constants, 2) logical constants, and 3) character constants.

NUMERIC CONSTANTS

A constant numeric data item can be expressed by a variety of constant representations which are grouped into the following categories:

1. Integer constants.
2. Real constants.
3. Double-precision constants.
4. Complex constants.
5. Hexadecimal constants.

These five constant data constructs are described in the following paragraphs and the internal storage requirements are described in appendix D.

Integer Constants

An integer constant consists of a string of decimal digit characters which can be preceded by a sign character (+ or −). If the constant is nonzero and unsigned, it is interpreted as representing a positive value. A zero has the same value whether signed or unsigned. From one to ten decimal digit characters are permitted and accuracy is ensured providing the value does not exceed −2,147,483,648 (for negative values) or 2,147,483,647 (for positive values). If this limit is exceeded, a syntax error results.

Several examples of valid integer constants follow:

```
0
+0
−43
17711
999999999
03770
2089934591
−5708
```

Several examples of invalid integer constants follow:

1.0	Decimal point not permitted, interpreted as a real constant.
3,000	No commas or other punctuation permitted.
2222222222	Exceeds the largest integer value allowed.
−0	Minus zero is invalid.

Real Constants

A real constant is stored in the B 1000 processor as an approximation of the actual constant. It can assume a positive, negative, or zero value.

The three forms of a real constant follow:

1. Basic real constant.
2. Basic real constant followed by a real exponent.
3. Integer constant followed by a real exponent.

The form of a basic real constant is an optional sign, an integer part, a decimal point, and a fractional part, in that order. Both the integer part and the fractional part are strings of decimal digits; either of these parts can be omitted, but not both. A basic real constant can be written with more digits than the B 1000 processor will use to approximate the value of the constant. The number of significant digits that the processor uses is approximately seven. A basic real constant is interpreted as a decimal number.

Examples of valid basic real constants follow:

```
3.141592
0.
0.0
.075
00000000000007.
-253.
-.075
```

The second and third types of real constants are combinations of a basic real constant or an integer constant and a real exponent.

The form of a real exponent is the letter E followed by an optionally signed integer constant. A real exponent denotes a power of ten.

The value of a real constant that contains a real exponent is the product of the constant that precedes the E and the power of ten indicated by the integer following the E. The integer constant part of the third type of real constant can be written with more digits than the processor uses to approximate the value of the constant.

Examples of valid basic real constants followed by a real exponent:

```
205.E-3
.01E3
6.02E23
345.280E-28
4291.0234E+8
2.9979E08
32.5E007
```

Examples of valid integer constants followed by a real exponent:

```
2E3
602E-19
-8E-43
1E-9
1245748E+27
```

Examples of invalid real constants follow:

-1597	No decimal point or E portion, interpreted as an integer constant.
8.2E+77	Exceeds maximum size limit.
4.2E-79	Smaller than minimum size limit.
E22	No integer or real part, interpreted as a variable name.
2.7E1.2	Exponent part must be an integer.
1E2E3	Only one E portion allowed per constant.
2,765,987.	No commas or other punctuation, except decimal point, permitted.

The range for the magnitude of a real constant is approximately .5397605E-78 .LT. X .LT. .7237005E+76, where X is the real constant. If this limit is exceeded, a syntax error results. For more information on the internal format of a real constant refer to appendix D.

Double-Precision Constants

A double-precision constant must be written using scientific notation and is stored in the B 1000 processor as an approximation of the actual constant. It can assume a positive, negative, or zero value. A double-precision constant uses two consecutive words of storage.

The two forms of a double-precision constant follow:

1. Basic real constant followed by a double-precision exponent.
2. Integer constant followed by a double-precision exponent.

The form of a double-precision exponent is the letter D followed by an optionally signed integer constant. A double-precision exponent denotes a power of ten. The form and interpretation of a double-precision exponent are identical to those of a real exponent, except that the letter D is used instead of the letter E.

The value of a double-precision constant is the product of the constant that precedes the D and the power of ten indicated by the integer following the D. The integer constant part of the second form can be written with more digits than the processor uses to approximate the value of the constant. The number of significant digits that the processor uses is approximately 14.

Examples of valid basic real constants followed by a double-precision constant:

```
3.141592653589793D0   }
3.141592653589793D-0  } equivalent
+1.D+3
1234567890.123456D+29
6.63D-03
9.80665D+0
```

Examples of valid integer constants followed by a double-precision constant:

```
1D3
+1D+03    } equivalent
-363354D-10
1D50
```

Examples of invalid double-precision constants follow:

3.14159	No D portion, interpreted as a real constant.
2.7 D 99	Exceeds maximum size limit.
2.7 D-99	Smaller than minimum size limit.
1,234,567,890,123.	Commas not permitted, no D portion.
1.3E45	No D in exponent part.
123456789.12345678901	No D portion, interpreted as a real constant.

The range of values for double-precision constants is approximately .5397605346E-78 .LT. X .LT. .7237005577E+76. If this limit is exceeded, a syntax error results. For more information on the internal format of a real constant, refer to appendix D.

Complex Constants

The form of a complex constant is a left parenthesis followed by an ordered pair of real or integer constants, separated by a comma, and followed by a right parenthesis. The first constant of the pair is the real part of the complex constant and the second is the imaginary part.

Examples of valid complex constants follow:

```
(6,0.7)
(12.93,14)
(65,27)
(.004,3.141)
(.1234567890,1)
```

Examples of invalid complex constants follow:

12	No parentheses, and no imaginary part.
(,5.4)	No real part.

Hexadecimal Constants

An alternate representation of program values consists of the hexadecimal constant which corresponds to digits of base 16. Hexadecimal constants can only be used as data initialization values in a DATA statement. For more information on the machine representation of the various data types refer to appendix D.

A hexadecimal constant consists of the letter Z followed by one or more hexadecimal digits. The hexadecimal constant assigns a value to the entire storage location used by the variable. Variables that use one storage unit (INTEGER, REAL, and LOGICAL) can contain eight hexadecimal digits. Variables that use two storage units (DOUBLE PRECISION and COMPLEX) can contain 16 hexadecimal digits. Any excess digits are truncated from the right (low-order digits). When a hexadecimal value does not fill the variable to which it is assigned, the variable is padded on the left

with hexadecimal zeros. CHARACTER variables must have two hexadecimal digits assigned for each character in the string or substring. Specifying too few or too many hexadecimal digits in a CHARACTER variable results in a syntax error.

The hexadecimal notation employed by the B 1000 system conforms to the standard form whereby each hexadecimal digit corresponds to a unique pattern of four bits within a data word. A list of these 4-bit patterns follows with the corresponding hexadecimal (hex) digits denoted:

Hex Digit	Bit Pattern	Hex Digit	Bit Pattern
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Examples of valid hexadecimal constants follow:

Z50180000	Valid for any variable of a numeric type, type LOGICAL, and CHARACTER * 4.
Z0123456789ABCDEF	Valid for any variable of a numeric type, type LOGICAL, and CHARACTER * 8.
Z0ACDEFA11	Valid for any variable of a numeric type and for LOGICAL. Invalid for CHARACTER because of odd number of digits.
ZFABZFAC4D5671234B90F	Valid for any variable of a numeric type, type LOGICAL, and CHARACTER * 10.
Z00000001	Valid for any variable of a numeric type, type LOGICAL, and CHARACTER * 4.

Examples of invalid hexadecimal constants follow:

FFF60	The Z is missing.
Z−1	A minus sign is not permitted in a hexadecimal constant.
Z0ABCDEFGF	The character G is not a hexadecimal digit character.
Z333.330033	A decimal point is not allowed in a hexadecimal constant.

LOGICAL CONSTANTS

FORTRAN 77 allows the use of logical operations through the medium of the logical expression. Two logical constants are provided to represent the logical values TRUE and FALSE.

These two logical constants are represented in the source code of a FORTRAN 77 program in the following manner:

.TRUE.
.FALSE.

The use of these logical constants is restricted to certain types of expressions. Refer to Logical Expressions in section 7 for details. The internal machine representation of these two constants is such that the data words corresponding to the constant .TRUE. and the integer constant −1 (all bits set) are identical; the data word corresponding to the constant .FALSE. and the integer constant 0 are identical. Refer to appendix D for more information on the internal representation of LOGICAL constants.

CHARACTER CONSTANTS

The form of a character constant is an apostrophe followed by a nonempty string of characters, followed by an apostrophe. The string can consist of any character capable of being represented in the B 1000 processor. The delimiting apostrophes are not part of the datum represented by the constant. An apostrophe within the datum string is represented by two consecutive apostrophes with no intervening blanks. In a character constant, blanks embedded between the delimiting apostrophes are significant.

One additional form of a character constant is allowed. A character constant can be of the form as described in the preceding paragraph except that quotation marks replace the apostrophes in the description. When an apostrophe is used as the string delimiter, a quotation mark within the datum string is represented by a quotation mark. When a quotation mark is used as the string delimiter, an apostrophe within the datum string is represented by an apostrophe.

The length of a character constant is the number of characters between the delimiting apostrophes or quotation marks, except that each pair of consecutive apostrophes or quotation marks counts as a single character. The delimiting apostrophes or quotation marks are not counted. The length of a character constant must be greater than zero and no greater than 255.

Examples of valid character constants (b represents a blank character) follow:

```
"DON'T"              'DON''T'    (equivalent)
'ABC123bbbDEF'
"""""
'?*−$3'
"b"
```

Examples of invalid character constants follow:

"ABC"DEFG"	Two adjacent quotation marks are needed if quotation marks are used as delimiters.
'POIU'b'YT'	This is interpreted as two strings since the inner apostrophes are not immediately adjacent.

SECTION 5
VARIABLES AND ARRAYS

FORTRAN 77 variable names and array names are symbolic names which are constructed from the FORTRAN 77 character set according to appropriate rules. Variables and arrays represent values which can be altered during program execution.

These constructs are used to identify one or more storage locations for purposes of data storage and retrieval. The constants of these storage locations are accessed by referencing the associated variable or array element name.

This section contains a description of variable name construction which extends to array names and function names. A description of the construction and use of arrays and substrings is also presented. The internal handling of variables and arrays is described in appendix D.

VARIABLE NAMES

A FORTRAN 77 variable name is an identifier which consists of a string of one to six alphanumeric characters (letters or digits), with the leading character being a letter. Special characters cannot be used in variable names.

If the variable name is more than six characters long, a syntax error results. Embedded blanks are acceptable but are removed by the system. Variables are classified into six fundamental types.

Type	Memory Required
INTEGER	4 bytes
REAL	4 bytes
DOUBLE PRECISION	8 bytes
COMPLEX	8 bytes
LOGICAL	4 bytes
CHARACTER	1 byte per character

There is no variable of type hexadecimal. Hexadecimal constants can only be used as data initialization values in DATA statements and explicit type statements.

The value represented by a variable of each of these types can be expressed by a constant of the same type. Thus, the value represented by an integer variable can be expressed by an integer constant, the value represented by a real variable can be expressed by a real constant, and so forth. Therefore, the values represented by each variable type must conform to the magnitude and significant digit restrictions governing the corresponding type of constant.

Unless declared otherwise in an explicit type statement or an IMPLICIT statement, the identifier is assigned a type according to the initial character. If this initial character is the letter I, J, K, L, M, or N, then the variable, by default, is of INTEGER type. If this initial character is any other letter, the variable, by default, is of REAL type. No such defaults exist for DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER variables. Variables of these types must be declared as such by explicit type statements.

Examples of valid variable names (type is assigned according to the first letter, as described in the preceding paragraph) follow:

Variable Name	Description
LNO599	This variable is of type INTEGER.
IF	This variable is of type INTEGER. There are no reserved words in B 1000 FORTRAN 77.
OF TEN	This variable is of type REAL. It is interpreted as OFTEN (blank ignored).
LOOP3	This variable is of type INTEGER.

Examples of invalid variable names follow:

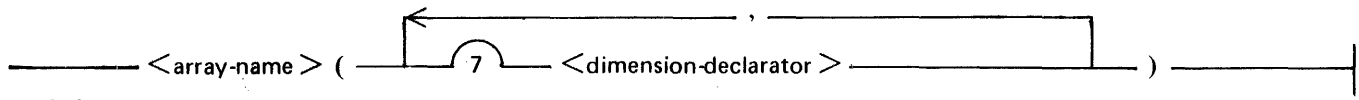
Invalid Variable Name	Description
3LOOP	Variable names cannot begin with a digit.
BE-GIN	Characters other than letters, digits, or blanks are not allowed in a variable name.
REALNUMBER	There are too many characters, only six are permitted.
END$Q	The dollar sign ($) character is not a legal character.

ARRAYS

An array is an ordered data set corresponding to an n-dimensional organization such that each member can be referenced by an array element, with each of the n subscripts in the element denoting a location in the appropriate dimension. In FORTRAN 77, an array can have a maximum of seven dimensions.

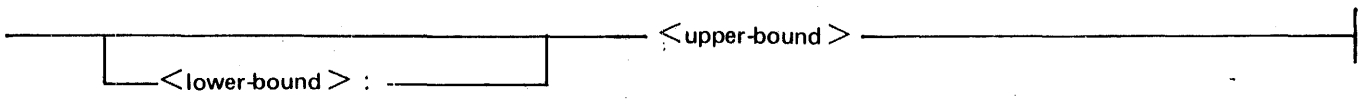
Array Declarator

An array declarator appears in a DIMENSION, explicit type, or COMMON statement and specifies the symbolic name of an array within a program unit and specific attributes for that array. Only one array declarator can be specified for a given array in a program unit. The array declarator specifies the number of dimensions for the array and the bounds on each of those dimensions. An array declarator has the following form:



G50294

<array-name> has the same restrictions as a variable name and uniquely identifies the array. <dimension-declarator> specifies bounds for each dimension. The number of dimension declarators in the array gives the number of dimensions in the array. <dimension-declarator> contains an <upper-bound> declarator and, optionally, a <lower-bound> declarator. A dimension declarator has the following form:



G50295

Both the <lower-bound> declarator and the <upper-bound> declarator can be integer expressions and are called dimension-bound expressions. If <lower-bound> is omitted, the lower bound for that dimension is 1. The values of these expressions can be positive, negative, or zero, with one restriction: the value of the upper bound must not be less than the value of the lower bound. The upper-dimension bound of the last dimension can be an asterisk in assumed-size array declarators.

Examples of statements that use array declarators follow:

```
DIMENSION LO(-3:-1,-7:0), ALPHA(14,15:20)
LOGICAL EL(0:99,3,27:28)
COMMON NI(1,2,3,4,5), BETA(2)
REAL N(-2:2)
```

The number of elements in an array can be determined by using the following formula:

$$E = ((u1 - l1) + 1) * ((u2 - l2) + 1) * \ldots * ((un - ln) + 1)$$

E is the number of elements in the array, the u's are the upper-bound declarators for each dimension, the l's are the lower-bound declarators for each dimension, and n is the number of dimensions of the array.

Types of Arrays

The upper-bound declaration of the final dimension declarator can be an asterisk (*), in which case the array is an assumed-size array. If the array declaration contains integer variables in the dimension-bound expressions, the array is an adjustable array. If the array contains only integer constant expressions in the dimension-bound expressions, the array is a constant array. Only a dummy array can be an adjustable array or an assumed-size array. Dummy arrays are explained under Arguments in section 13.

Examples:

DIMENSION AL(-1:I,2:J)	AL is an adjustable array.
REAL BE(14,*), CE(I,2:J,*)	BE is an assumed-size array. CE is an adjustable array and an assumed-size array.
COMMON DE(18)	DE is a constant array. A constant array is the only type of array that can be in COMMON storage.

Array Elements

Each member of an array is called an array element. The following is the proper form of an array element:

G50296

<subscript-list> consists of as many arithmetic expressions (subscripts), separated by commas, as there are array dimensions.

Each member of an array is referenced by means of an array element with appropriate subscripts. Each arithmetic expression in the subscript list of this construct must be of type INTEGER only. The expression can contain any of the arithmetic operators, integer functions, or subscripted integer variables. A subscript within an array reference must be greater than or equal to the lower bound declared for that dimension in the array declarator for that array. The subscript must also be less than or equal to the upper bound declared for that dimension in the array declarator for that array. The number of subscripts in an array reference must be equal to the number of dimensions in the array declarator for the referenced array.

Whenever an array name appears in a program, this array name must be immediately followed by a subscript list, except when the array name appears in the following:

1. The dummy argument list of a subprogram reference.
2. The actual argument list of a subprogram reference.
3. The variable list of an input/output statement, unless the array is an assumed-size array.
4. As a unit identifier or format identifier in an input/output statement, unless the array is an assumed-size array.
5. A COMMON, DATA, EQUIVALENCE, or explicit type statement.

An array can never contain fewer subscripts than are declared for that array in an array declaration. However, a dummy array can have fewer declared dimensions than the actual array with which it is associated.

Examples of valid array elements follow:

B(I)	
LNO 599(-6)	This array element is interpreted as LNO599(-6).
I5(IT(3))	The subscript is an array element.
ARRAY2(1,0,1,0)	This array element is valid only if the given subscripts are within the ranges of the dimensions declared for ARRAY2.
A(M*N)	

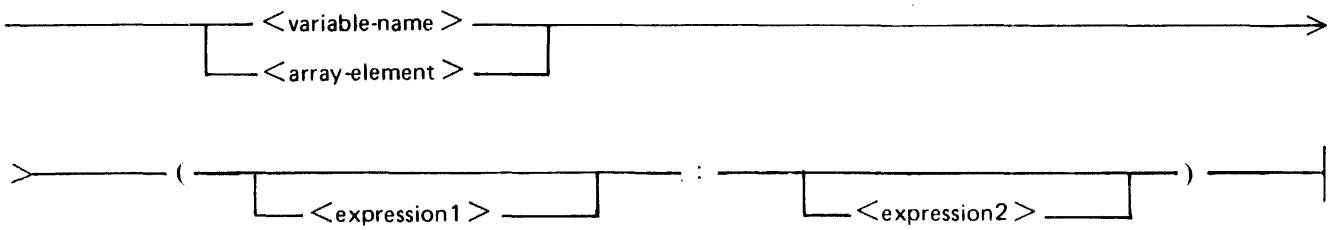
Examples of invalid array elements follow:

I(I)	A subscript must be a valid arithmetic expression; an array name does not constitute such an expression.
ARRAY3(0)	This array element is invalid only if ARRAY3 does not contain 0 in the dimension range (example: ARRAY3(-4:-2)).
3ARRAYS(6)	An array name cannot violate the rules governing variable names.
ARRAY(3.6)	The subscript must be INTEGER type only.

A detailed description of the internal representation of FORTRAN 77 arrays is contained in appendix D.

CHARACTER SUBSTRINGS

A character variable can either be referenced as a complete entity or any part of the variable can be referenced using a substring name. A character substring name has the following format:



G50297

The variable referenced can be either a simple character <variable-name> as in the first option, or a character <array-element> (a character array name followed by a subscript expression). <expression1> is the character position within the variable where the substring begins and <expression2> is the character position within the character variable where the substring ends. <expression1> and <expression2> are integer expressions which have the following restriction:

1 .LE. expression1 .LE. expression2 .LE. len

The expression len is the length of the character variable from which the substring is being taken. If <expression2> is omitted, the substring is assumed to be all of the characters from character position expression1 to the end of the character variable. The form A(:) is equivalent to A, which is assumed to be the entire character variable, and the form B(s1,s2,...) (:) is assumed to be the entire character array element.

Examples of valid character substrings follow:

B(2:4)	Character positions 2, 3, and 4 in character variable B.
B(2:)	All the characters in B from character position 2 to the end of B.
B(:I+5)	From the beginning of B to character position I+5 in B. Same as B(1:I+5).
B(:)	All the characters in B. Same as B.
B(3:3)	Character position 3 in B.
C(2,3)(5:9)	Character positions 5 through 9 in element (2,3) of character array C.

Examples of invalid character substrings follow:

D()	No colon.
D(-1:)	Negative not permitted.
D(5:4)	Expression2 is less than expression1.
D(6.3 + X:)	Real expression not allowed.
D(2:8)	Invalid only if D contains less than eight characters.
E(3:5)(1,I)	Subscript expression must precede substring expression.

SECTION 6
SPECIFICATION STATEMENTS

Specification statements are employed to supply compile-time information about program variables pertaining to variable types and storage allocation. All specification statements must precede the first executable statement in a program unit. The specification statements are comprised of the following:

Explicit type statements
COMMON statement
DATA statement
DIMENSION statement
EQUIVALENCE statement
EXTERNAL statement
IMPLICIT statement
INTRINSIC statement
PARAMETER statement
PROGRAM statement
SAVE statement

These statements are described in the following paragraphs in the order just listed.

EXPLICIT TYPE STATEMENTS

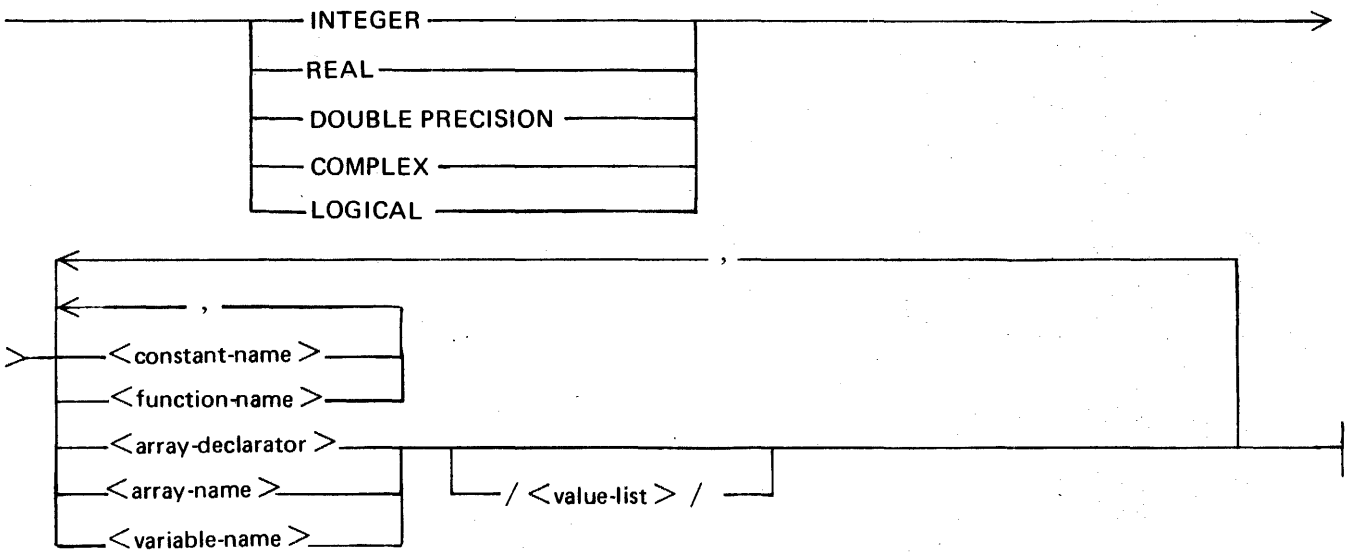
The explicit type statements allow the type of a program variable to be explicitly specified for a program unit and can also specify dimension information. The type assigned is only recognized in the program unit in which it occurs (main program, subroutine, function, or block data subprogram). A value can be assigned to the variable within the explicit type statement. Explicit type specifications override any default specifications due to the initial character in the symbolic name of the variable. Refer to Variable Names in section 5 for additional information.

Program variables can be assigned the following types:

INTEGER
REAL
DOUBLE PRECISION
COMPLEX
LOGICAL
CHARACTER

Numeric and Logical Type Statements

An explicit type statement specifying a type of DOUBLE PRECISION, INTEGER, REAL, or COMPLEX is a numeric type statement. There is only one logical type statement: LOGICAL. These types of variables have an implied length. Numeric and logical type statements have the following form:



G50298

<constant-name> is the symbolic name of a constant that is to be given a value in a subsequent PARAMETER statement. <value-list> is a list of initial values for the entity. Initial values for numeric entities must be numeric constants or the symbolic names of numeric constants. A complex constant must only initialize a variable or array of type COMPLEX. <constant-name> or <function-name> must not have an associated <value-list>. If the entity is a simple variable, <value-list>, if specified, must contain only one value. If the entity being typed is an array, <value-list> contains the number of elements in the array. A dummy variable or dummy array declaration must not contain a <value-list>.

An array declarator can appear only once in a program unit for a specific array. Therefore, if the dimensions for a given array are given in a DIMENSION statement, the array name, without an array declarator, must be used in the explicit type statement.

Examples of explicit type statements follow:

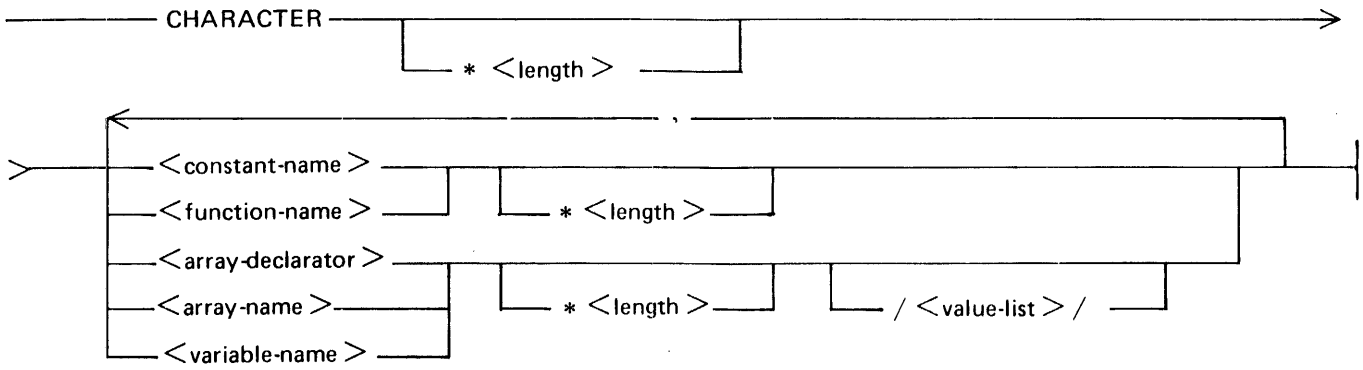
```
REAL IZE/134.99/, LEMON(12:15,-13:1)
LOGICAL CONC/.FALSE./, LUSION(12)
INTEGER NO, DECI, MAL, POINT(-3:0)/1,2,3,4/
DOUBLE PRECISION MORE(3), EXACT
```

An explicit type statement, if used, must appear before any other statements referencing the variable.

Character Type Statement

The character type statement is used to specify that a specific variable can contain string values only. A variable that is not of type CHARACTER must not be assigned string values at any time. A character variable can have a specified length, or the variable can assume the default length of one byte. This length attribute, when specified, is given in the explicit type statement for the character variable.

The CHARACTER explicit type statement has the following form:



G50299

<length> is the length in characters of the variable or array being declared and has a minimum value of 1 and a maximum value of 255. <length> has one of the following three forms:

1. An integer constant.
2. An integer constant expression (expression containing only integer constants and the symbolic names of integer constants, no variables) in parentheses.
3. An asterisk (*) character in parentheses.

The first two forms of <length> specify a constant length for the character entity. The third form specifies an assumed length for a dummy variable or dummy array. The dummy variable or dummy array assumes the length of the actual argument with which it is associated.

If the CHARACTER type statement does not have a length attribute, the default length is one character. A length following an entity in the list overrides the length given for the CHARACTER type statement. Any entry in the list that is not followed by a length attribute takes on the length given following the word CHARACTER, if one is given.

<constant-name> is the symbolic name of a constant to be given a value in a subsequent PARAMETER statement. <variable-name>, <array-name>, and <array-declarator> are described in section 5. <function-name> is described in section 13. <value-list> is a list of string constants separated by commas which initialize the entity being typed. There must be only one value in the <value-list> if the entity is not an array. If the entity is an array, the list contains the number of elements in the array. A dummy variable or dummy array declaration must not contain a value list.

Example:

```
CHARACTER * 9 A, B * 4 /'XYZA'/, C /'ABCDEFGHI'/,X * 2
CHARACTER G, I(4:13) * 7, K(2) /'O','D'/
```

In this example, A has a length of nine characters, B has a length of four characters and is assigned an initial value, C has a length of nine and has an initial value, and X has a length of two. In the second statement, G has a length of one. I is a character array; each element contains seven characters. K is a 2-element character array, each element containing one character and an initial value.

An example of a partial function subprogram of type CHARACTER with dummy variable declarations follows:

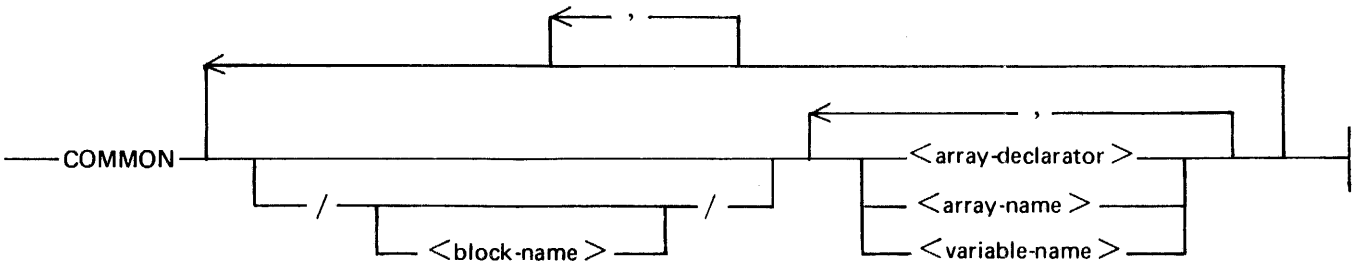
```
CHARACTER * (*) FUNCTION T(E, F, G)
CHARACTER * (4) E(3)
CHARACTER * (*) F, G
```

In this example, T is a variable length CHARACTER FUNCTION containing four dummy parameters (the length of the value returned is dependent on the length declaration for the function in the calling program unit; refer to section 13). The element length of dummy array E is 4.

Dummy variables F and G have no explicit length and are dependent on the length of the corresponding actual arguments. Refer to section 13 for more information on dummy parameters.

COMMON STATEMENT

The COMMON statement allows values to be shared among program units without employing entries in SUBROUTINE and FUNCTION statement argument lists, while permitting these data items to be referenced in each program unit. The proper form of the COMMON statement follows:



G50300

Common Names

A symbolic name is associated with each block of COMMON storage; this name is called a COMMON name or block name. Any program unit can access the block of storage associated with this name by means of a COMMON statement employing this name. COMMON storage associated with a COMMON name is referred to as COMMON.

A COMMON name is constructed in the same manner as a variable name, except that no type is associated with a COMMON name. A COMMON block need not be named; COMMON storage associated with no name is called blank COMMON and is assigned the internal identifier &BLANK. If the specification for blank COMMON is the first specified in a COMMON statement, the two slashes enclosing the COMMON name can be omitted. Thus, these two statements are equivalent:

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

COMMON block names are unique only within COMMON statements. Outside the COMMON statement, a COMMON block name can be reused as another element within the program unit (for example, a simple variable name, an array name, and so forth).

Use of Array Declarators

Array declarators can be used in COMMON statements to declare the dimensions of arrays in the same manner as type statements or DIMENSION statements. Refer to Array Declarator in section 5 for an explanation of array declarators.

Storage Assignments

Each element of a COMMON block is allocated storage in COMMON storage once for an entire executable program. Each program unit can reference a COMMON block (and hence each location in the block) by means of an appropriate COMMON statement. The contents of the locations referenced can be changed in the same manner as the contents of any location local to the program unit.

Variables and arrays are assigned contiguous locations in COMMON storage in the order of appearance in a COMMON statement. The size of each block of COMMON storage is either as large as the maximum specification indicated by a COMMON statement referencing the block name in any program unit, or as large as the maximum length to which the block is extended by an EQUIVALENCE statement. Refer to EQUIVALENCE Statement in this section for additional information.

Assume that the following statements are the initial statements of a program unit:

```
SUBROUTINE MSG
DOUBLE PRECISION D
LOGICAL FLAG(6)
COMMON WORD1,WORD2, D,FLAG,TEXT(20)
COUNT = 1
```

Assume that the preceding COMMON statement is the largest description of the size of the unlabeled COMMON block in a given program. The total size of this COMMON block is 30 words. These words are recognized in the MSG subprogram as the words assigned to the REAL variables WORD1 and WORD2, the word pair assigned to the DOUBLE PRECISION variable D, the six words assigned to the LOGICAL array FLAG, and the 20 words assigned to the REAL array TEXT. These data words are contained at relative locations within the COMMON block in the order listed.

The unlabeled COMMON block just described can be referenced, for example, by a COMMON statement within another program unit as follows:

```
      SUBROUTINE DUMP
      COMMON T(10)
      WRITE (6,1)T
    1 FORMAT(1X.10Z8)
      RETURN
      END
```

In this example, T is a REAL array. The elements of this array are assigned the data words contained in the COMMON block, beginning with the initial word of the block and proceeding for 10 words. Thus, WORD1 and WORD2 are equivalent to the array elements T(1) and T(2), respectively; D is equivalent to elements T(3) through T(4); FLAG(1) through FLAG(6) are equivalent to element T(5) through T(10). The data words allocated to the TEXT array in the MSG subprogram are not accessed in the DUMP subprogram.

Entire arrays, but not individual array elements, can be assigned storage locations in COMMON storage.

If the same COMMON name appears more than once in a program unit, the COMMON elements associated with one appearance are considered extensions to the list of the previous appearance.

Data initialization can be performed by means of a BLOCK DATA subprogram. The BLOCK DATA subprogram is described in section 13. A DOUBLE PRECISION variable in a COMMON block must not cross a data segment boundary. Each data segment contains up to 256 words.

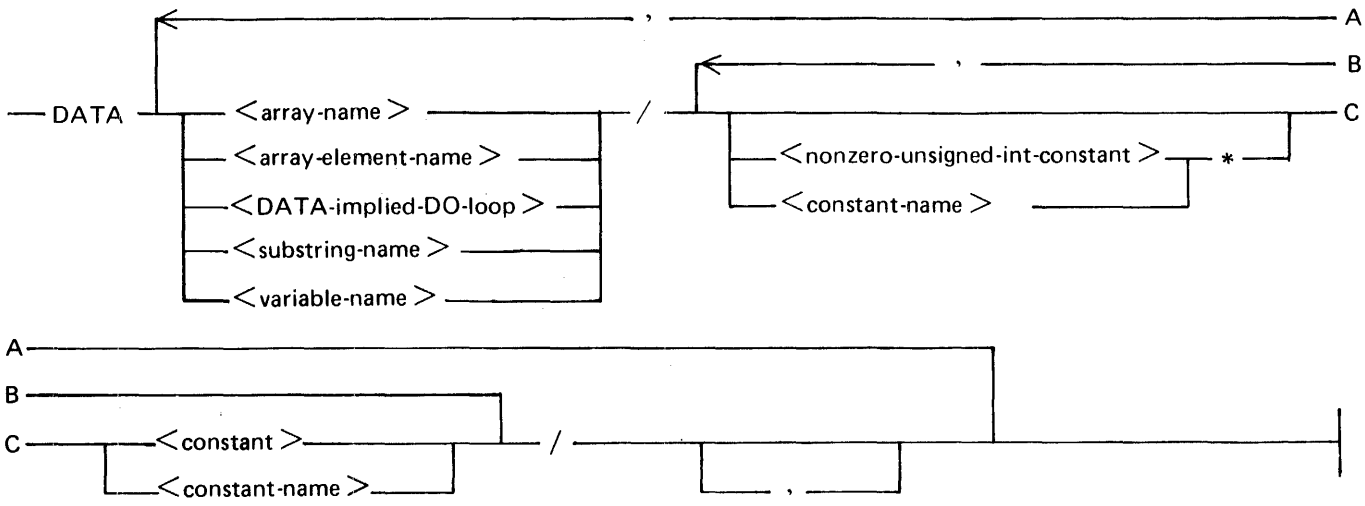
Variables and array names cannot be duplicated in COMMON statements. One variable cannot be assigned to more than one block of COMMON storage within a program unit. No dummy arguments can appear in a COMMON statement. A COMMON block can contain CHARACTER type data, but if so, it must contain only CHARACTER type data, and any variable name associated with the block must be of type CHARACTER.

Examples of COMMON statements follow:

```
COMMON/BLOCK1/A,B(10),C//G,HOLD/BLOCK2/Q(3)
COMMON D
COMMON T1/CMN/T2,T3,X(-4:-1,8)
```

DATA STATEMENT

The DATA statement is provided to allow compile-time initialization of program variables. All variables are initialized to 0 if not specified in a DATA statement or a <value-list> in an explicit type statement. The proper form of the DATA statement follows:



<constant-name> is the symbolic name of a constant described in the PARAMETER statement. The items concerning constants are described in section 4. Items concerning arrays, variables, and substrings are described in section 5. <DATA-implied-DO-loop>s are described in the following subsection entitled Variable Lists.

If a DATA statement is used in a program unit, it must appear after all specification statements and before the END statement in the program unit. The DATA statement has effect only at compilation time. Elements of a COMMON block can appear in DATA statements only in a BLOCK DATA subprogram or in a main program.

Variable Lists

A variable list in a DATA statement consists of the following: <array-name>, <array-element-name>, <DATA-implied-DO-loop>, <substring-name>, and <variable-name>. Each element of the variable list can occur only once. When an array name is written without a subscript, each element of the array is initialized with an element of the initial value list in the order in which the array elements are stored. Refer to appendix D for more details.

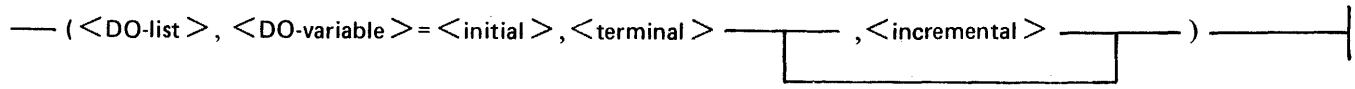
Each substring expression in the variable list must be an integer constant expression. Each subscript expression in the variable list must be an integer constant expression except for implied-DO variables that can appear within the expression.

Example of a variable list:

```
K, M, A(3), B(2,4,11), ((X(J,I),I=1,J),J=1,5)
```

DATA Implied-DO Loop

A DATA implied-DO loop is used to specify the elements of an array which are to be initialized. Giving an array name without an implied-DO loop specifies that every element of the array is to be initialized. A DATA implied-DO loop has the following form:



G50302

The range of the implied-DO loop is the list <DO-list>. In the diagram, <DO-list> is a list of array elements, and/or DATA implied-DO loops separated by commas. The <DO-variable> to the left of the equal sign is described under the DO statement in section 7. The <initial>, <terminal>, and <incremental> parameters are any integer constant expressions or integer expressions with this implied-DO loop within their range. The parameters and <DO-variable> in the implied-DO are handled in the same manner as a DO loop. Refer to section 7 for additional information.

The iteration count (section 7) must be positive. With each iteration of the implied-DO, each item in the <DO-list> is assigned a value from the initial value list (refer to following subsection), and any list items accessing the <DO-variable> (a parameter in another implied-DO within the range of the outer implied-DO, or an array containing the <DO-variable> as a subscript) are assigned the new value of the <DO-variable>.

An example of DATA implied-DO loops within DATA statements follows:

```
DIMENSION A(20), B(6), C(12,-4:10), D(100), E(2,2)
CHARACTER * 5 F(6)
DATA (A(I),I=4,15)/12 * 1.5/
DATA (B(J),(C(J,I),I = -4,J,2), D(J),J = 1,6),E/43 * 1.0/
DATA (F(I)(2:4),I=1,3)/3 * 'ABC'/
```

In this example, the first DATA statement initializes elements 4 through 15 of array A to 1.5. The second DATA statement initializes all six elements of array B to 1.0; elements (1,−4), (1,−2), (1,0), (2,−4), (2,−2), (2,0), (2,2), (3,−4), (3,−2), (3,0), (3,2), (4,−4), (4,−2), (4,0), (4,2), (4,49), (5,−4), (5,−2), (5,0), (5,2), (5,4), (6,−4), (6,−2), (6,0), (6,2), (6,4), and (6,6) of array C to 1.0, the first six elements of array D to 1.0, and all of array E to 1.0. The third DATA statement initializes character positions 2 through 4 of the first three elements of array F to 'ABC'.

Each element of an array must only be initialized once in a DATA statement in an executable program. A <DO-variable> in a DATA implied-DO loop does not affect the value of a program variable with the same name.

Initial Value Lists

The constant values contained within the slashes comprise the initial value list of the DATA statement. The values in the list consist of numeric constants and strings.

Repeat Counts

The constants can optionally be preceded by a repeat count of the form n*, where n is an unsigned nonzero integer constant, or constant name defined in a PARAMETER statement. This repeat count indicates the number of times the immediately following constant is to be used for assignment.

Data Assignment

Constant values in the initial value list are assigned to elements of the variable list in the order of occurrence. For example, the following DATA statement initializes the variables A and B to the values 2 and 3, respectively, and initializes C and D to 4.

```
DATA A,B/2,3/,C,D/2*4/
```

All elements of the variable list must be matched to the constants in the initial value list, and all constants must be used; the number of items in the variable list to be initialized must equal the number of items in the initial value list. A repeat count (n*) counts for n occurrences of the immediately succeeding entry in the initial value list. An implied-DO has a similar effect on an item in the variable list. If an entire array is specified in the variable list, but there are not enough constants to completely initialize it, an error is given.

Character Strings

The initial value list can contain strings of up to 255 characters. Character constants (refer to Character Constants in section 4) in DATA statements initialize character variables, character substrings, character arrays, or character array elements. Numeric-typed variables must not be assigned character values, and character entities must never be assigned numeric values unless the values are hexadecimal. If the length of the string is less than the length of the character entity to which the string is being assigned, the string is assigned left-justified with blanks filled in to character storage locations that are unassigned. If the value being assigned is greater than the length of the character entity to which it is being assigned, excess characters in the value are truncated from the right.

Examples of valid DATA statements involving CHARACTER entities follow:

```
CHARACTER *4 X, Y, Z(-17:4) *2
DATA X, B, C, Y, Z/'ABCD', 2*4.3, 'EGH', 20*'IT',2*ZC1C3/
```

In this example, X is initialized to the value 'ABCD', Y to the value 'EGH ', and the first 20 elements of array Z to the value 'IT'. The final two elements of Z are initialized to the value 'AC'.

If a long string is being assigned to an array, excess characters in the string after assignment to the first element of the array are not assigned to the next element of the array. One string can only be assigned to one variable unless there is a repeat count preceding the string.

Hexadecimal Initialization

Hexadecimal (hex) constants (refer to Hexadecimal Constants in section 4) can be used to initialize either numeric or character variables, arrays, array elements, or substrings. The exact value represented in the hex string is assigned, without conversion or regard to type, to the entity in the variable list. When a hex value is assigned to a numeric entity (COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, REAL), if the hex value is too small, it is filled on the left with zeroes until it fills the entity. If the hex value is too large to completely fit in the variable item (more than eight hex digits, or more than 16 for DOUBLE PRECISION), truncation is performed from the left (most significant digits) until the value can fit into the variable.

Hex constants which are assigned to character entities must exactly fit the character entity; for each character in the character entity (variable, array element, substring) there must be two hex digits in the hex string being assigned.

Conversion During Assignment

Table 6-1 indicates the type conversion to be performed on a constant appearing in an initial value list when assigned as the initial value of a variable.

The CMPLX, DBLE, INT, and REAL functions have the same effect as the CMPLX, DBLE, INT, and REAL intrinsic functions. Refer to section 13 for more information about these intrinsic functions.

Table 6-1. DATA Statement Type Conversions

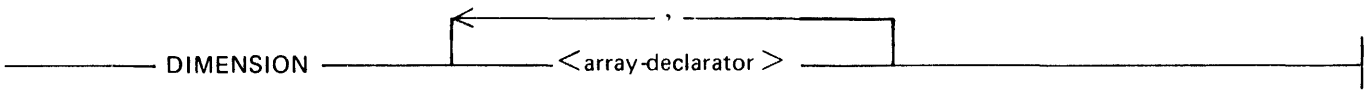
Constant Type	Variable Type					
	INTEGER	**REAL**	**DOUBLE**	**LOGICAL**	**CHARACTER**	**COMPLEX**
INTEGER	None	REAL	DBLE	Invalid	Invalid	CMPLX
REAL	INT	None	DBLE	Invalid	Invalid	CMPLX
DOUBLE	INT	REAL	None	Invalid	Invalid	CMPLX
LOGICAL	Invalid	Invalid	Invalid	None	Invalid	Invalid
CHARACTER	Invalid	Invalid	Invalid	Invalid	None	Invalid
COMPLEX	INT	REAL	DBLE	Invalid	Invalid	None
Hex	None	None	None	None	None	None

The following notation is used in table 6-1.

Table	Meaning
None	No conversion.
Invalid	Invalid combination resulting in a syntax error.
CMPLX	Perform REAL and assign to real portion; assign 0. to imaginary portion.
DBLE	Convert to DOUBLE PRECISION.
INT	Truncate.
REAL	Convert to REAL.

DIMENSION STATEMENT

The DIMENSION statement specifies the size and number of dimensions of a program array. The following is the proper form of the DIMENSION statement:



G50303

<array-declarator> is described in section 5. Each array referenced in a program unit must have the array bounds specified exactly once in that program unit. This specification can be accomplished by means of a DIMENSION, explicit-type, or COMMON statement.

For an array which is not a dummy argument, an array declaration specifies exactly the amount of internal storage to be allocated to the array and the number of subscripts an element of that array must have. Refer to ARRAYS in section 5 for additional information.

Only an array declaration appearing in a subprogram can have dimensions which are variables. The array name and the variable names appearing in the array declaration must appear in a dummy argument list within the subprogram. Refer to section 13 for more information on dummy arguments.

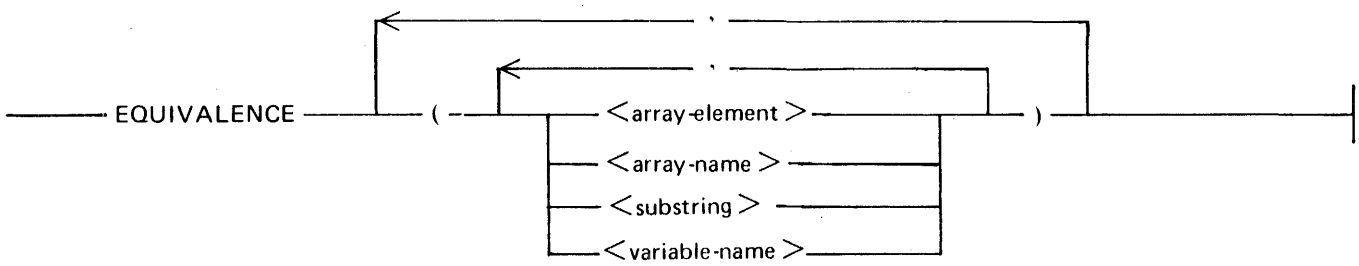
Examples of DIMENSION statements follow:

```
DIMENSION A(-12:10),B(3,3)
DIMENSION C(N,4:J)
DIMENSION D(13,*)
```

Arrays C and D in the example are dummy arrays.

EQUIVALENCE STATEMENT

The EQUIVALENCE statement allows the user to assign a number of program data items to a single unit of internal storage. Thus, more than one symbolic name can refer to one storage location. The following is the proper form of the EQUIVALENCE statement:



G50304

Each data item grouping in the EQUIVALENCE statement is enclosed in parentheses. Each such grouping is assigned storage locations to share. The subscripts of array elements in the list must be integer constants, symbolic names of integer constants, or integer constant expressions and must correspond in number to the number of dimensions declared for the array. Two elements of the same array cannot be equivalenced. Thus, EQUIVALENCE (A(3),B), (A(6),B) is invalid.

No dummy argument or subprogram name can appear in an EQUIVALENCE statement. A list item of type CHARACTER must only be associated with other items of type CHARACTER.

Single Storage Locations – Numeric

The least complicated use of the EQUIVALENCE statement involves the assignment of data items requiring a single word (REAL, INTEGER, LOGICAL) of storage to mutual storage location.

As an example, assume that the following statements are the first statements of an executable program:

```
INTEGER A, AR(2)
LOGICAL L,AL
EQUIVALENCE (A, AR(2), B2), (AL,L)
```

The EQUIVALENCE statement causes the INTEGER variable A, the INTEGER array element AR(2), and the REAL variable B2 to be assigned to one data word. The first element of AR is not affected by this specification statement. A change in the value of any one of the three equivalenced items produces a simultaneous change in the value of the other two items; however, only variables of the same type contain equivalent changes, and variables of different types become undefined. In this example, if INTEGER variable A is assigned a value, INTEGER array element AR(2) is assigned the same value, and REAL variable B2 becomes undefined.

The EQUIVALENCE statement also causes the LOGICAL variables AL and L to be assigned to the same data word. As the variable L changes value, AL also changes value. For example, the following assignment statement places the logical value TRUE into variable AL.

```
L = .TRUE.
```

Multiple Storage Locations – Numeric

EQUIVALENCE statements can also involve data items requiring more than one word of storage. As an example, assume that the following statements are the first statements of a program:

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

This EQUIVALENCE statement causes the REAL array A, the DOUBLE PRECISION variable D, and the REAL variable B to be assigned to identical data words. As A and D both require two data words, the first and second elements of A become equivalent to the first and second words, respectively, of the storage unit assigned to D. The variable B requires only one data word and is assigned to the same location as A(1) and the first word of D. If D was declared as COMPLEX, the preceding EQUIVALENCE statement would have the same effect.

Array Handling – Numeric

The EQUIVALENCE statement can be used to assign a single group of contiguous storage locations to a number of arrays. The following discussion illustrates the effect of the appearance within an EQUIVALENCE list of each of these two types of possible array references:

1. An array name.
2. An array element with the same number of subscripts as contained in the declaration declaring the array.

Assume that the following statements are the first statements of a program unit.

```
REAL A(4), B(10, 10), C(100), D(50), E(3, 3), F(50)
DOUBLE PRECISION DP(2)
EQUIVALENCE (A,DP(1)), (B(1,1),C(1)),(D,F(26), E(2,1))
```

The first list in the above EQUIVALENCE statement ((A,DP(1))) causes the REAL array A and the DOUBLE PRECISION array DP to share four storage words. The first two elements of A (A(1) and A(2)) become equivalent to the two words of the first element of DP (DP(1)), and the last two elements of A (A(3) and A(4)) become equivalent to the two words of the last element of DP (DP(2)). The appearance of array names only in an EQUIVALENCE list causes equivalencing to begin with the first element of each array. The second list in the sample EQUIVALENCE statement ((B(1,1),C(1))) causes the REAL arrays B and C to share 100 storage words. Each element in the 100-element array C (beginning with C(1)) is assigned to the same storage location as a unique element of B. The elements of the 2-dimensional array are stored internally in a column-wise fashion (refer to appendix D). The internal storage locations assigned to C occur in the same order as the elements of array B. Hence, each C(I) is equivalenced to the I-th internal element of B. If the following two WRITE statements occur in the same program unit, identical output is produced:

```
WRITE (6, 10)(C(I),I = 1,100)
WRITE (6,10) B
```

The final list in the sample EQUIVALENCE statement, (D,F(26),E(2,1)), indicates that the elements of the arrays D, F, and E are to be equivalenced in such a manner that D(1), F(26), and the second internal element of the 2-dimensional array E are to be assigned identical internal locations. The last 25 elements of F, F(26) through F(50), become equivalent to the first 25 elements of D, D(1) through D(25). Since E is stored internally in the manner described in the explanation of arrays in this section, equivalencing is handled in the manner illustrated in the following diagram. Each of the lines denotes a single storage location, and the array element(s) on a line is assigned to the corresponding location.

	F(1) through F(25)	E(1,1)
D(1)	F(26)	E(2,1)
D(2)	F(27)	E(3,1)
D(3)	F(28)	E(1.2)
D(4)	F(29)	E(2,2)
D(5)	F(30)	E(3,2)
D(6)	F(31)	E(1,3)
D(7)	F(32)	E(2,3)
D(8)	F(33)	E(3,3)
D(9) through D(25)	F(34) through F(50)	
D(26) through D(50)		

The following DATA statement initializes the elements E(1,2), E(2,2), F(28), F(29), D(3), and D(4) with the value 6.

```
DATA E(1,2), E(2,2) / 2*6/
```

Character Association

Character storage locations can be associated with more than one character variable, character array, or character substring due to an equivalence relation.

A character array can be viewed as a contiguous sequence of n character storage locations, where n is the number of elements in the character array multiplied by the length of an element.

An example of an EQUIVALENCE statement with CHARACTER arguments follows:

```
CHARACTER * 5 A, B(2) * 2
EQUIVALENCE (A(3:4), B(1))
```

In this example, character locations 3 and 4 in A share the same storage locations as B(1). The fifth character storage location in A (A(5:5)) is the same as the storage location for B(2)(1:1).

Interaction with Common Storage

The EQUIVALENCE statement can be used to associate additional elements with a COMMON block. This can extend the block beyond its former terminal point, increasing the size of the block. It is possible to EQUIVALENCE the beginning of an item representing more than one storage location (such as an array) to an element of the COMMON block, resulting in the addition of storage locations at the end of the block. The following example illustrates the manner in which a COMMON block can be extended by the EQUIVALENCE statement.

Assume that the following statements form two units of an executable program:

```
      FUNCTION SUM(N)
      COMMON GR1/IT(3,3)
      DO 1 I=1,3
      DO 1 J=1,3
    1 SUM=SUM+IT(I,J)
      SUM=SUM*N
      RETURN
      END
C
      LOGICAL FUNCTION TEST(L)
      LOGICAL X(6)
      COMMON GR1/K(9)
      EQUIVALENCE (K(6),X)
      DO 1 I=1,9
    1 S=S+K(I)
      TEST= S.EQ.L.AND.X(1).AND.X(6)
C     ELSE TEST IS .FALSE.
      RETURN
      END
```

The COMMON block referenced by these two sample program units is labeled GR1. The function SUM accesses the first nine locations of this block through the 2-dimensional INTEGER array IT. The function TEST accesses the first nine locations of the block using the INTEGER array K. In addition, the following two locations of the GR1 block are referenced as the LOGICAL array elements X(5) and X(6), since the X array is equivalenced to the K array starting at the element K(6).

The elements of the array K occur in the same order as the contiguous storage locations assigned to the array IT but allow these locations to be referenced using only one subscript. Equivalenced portions of the X and K arrays allow various elements of K to be handled as both INTEGER and LOGICAL type items.

COMMON blocks cannot be extended backwards by the EQUIVALENCE statement. The following combination of statements is invalid:

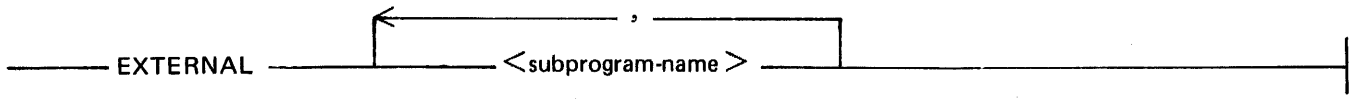
```
LOGICAL X(6)
COMMON/GR1/K (9)
EQUIVALENCE (X(3),K)
```

Two elements of COMMON storage cannot be made equivalent to each other, either directly or indirectly, by an EQUIVALENCE statement.

A local variable equivalenced to a COMMON variable becomes a part of the COMMON block for that program unit. This variable cannot be initialized in a DATA statement or an explicit type statement in the program unit because it is in COMMON storage. It cannot be initialized in a BLOCK DATA subprogram because it is not explicitly named in a COMMON statement in that program unit.

EXTERNAL STATEMENT

The EXTERNAL statement is used to identify a subprogram name as representing an external procedure and to specify to the compiler binding information relating to the subprogram. The proper form of the EXTERNAL statement follows:



G50306

The EXTERNAL statement has two basic purposes: 1) to identify subprogram names to be passed as actual parameters in a subprogram invocation, and 2) to override intrinsic function selection in a program unit.

Subprograms as Actual Parameters

When a subprogram name is used as an actual parameter, it must appear in an EXTERNAL statement. The invocation of the subprogram associates the dummy subprogram name in the dummy parameter list with the actual subprogram name. A call to the dummy subprogram is a call to the subprogram named in the actual parameter list.

Example:

```
      EXTERNAL A
      CALL B(A)
        .
        .
      SUBROUTINE C(D)
100   CALL D
```

In this example, the CALL in subprogram C of subprogram D is actually a CALL to subprogram A. When line 100 is executed, control passes to subprogram A. A subprogram in an actual parameter list can also be the name of a dummy subprogram in the calling program unit. A block data subprogram must never appear in the actual parameter list of a subprogram reference.

User-defined Intrinsic Functions

Intrinsic function selection can be overridden through use of the EXTERNAL statement. User functions can replace intrinsic functions for a program unit by specifying the user defined function in an EXTERNAL statement. The user-defined function with the same name as the default intrinsic function is substituted during the subprogram unit in which the EXTERNAL statement containing the user-defined subprogram name occurred. A user could, for example, write a SIN function that is different from the SIN intrinsic function normally used. If the user only wanted to use the new SIN function in certain cases, an EXTERNAL statement could be used in each subprogram where the new SIN function is desired.

An example of the EXTERNAL statement used to override intrinsic functions follows:

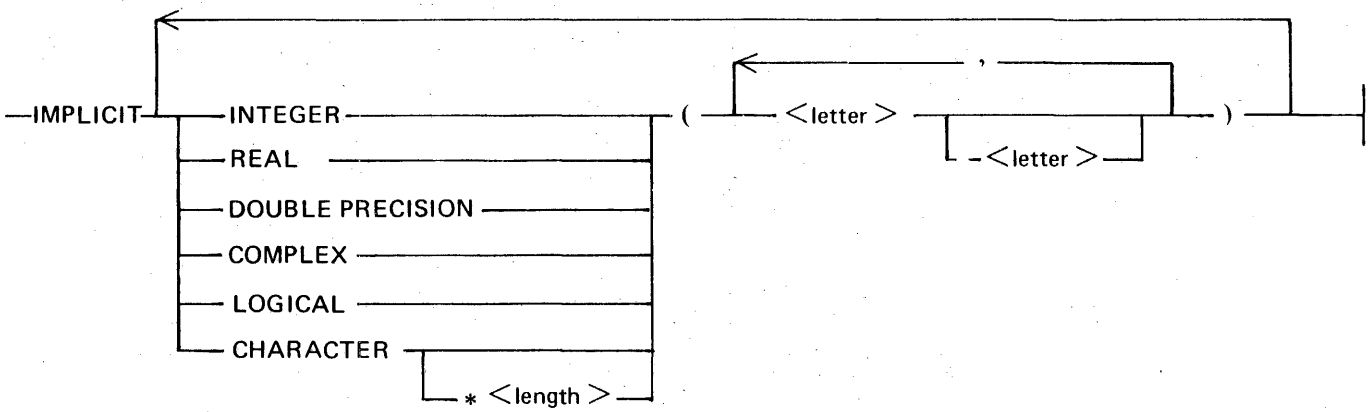
```
                SUBROUTINE A(X,Y)
                EXTERNAL SIN, COS
          100   X=SIN(Y)+COS(Y)
                  .
                  .
                  .
                SUBROUTINE B(X,Y)
                X=SIN(Y)+COS(Y)
                  .
                  .
                  .
                REAL FUNCTION SIN(A)
                  .
                  .
                  .
                REAL FUNCTION COS(B)
                  .
                  .
                  .
```

The subprograms specified in the EXTERNAL statement are searched for in the intermediate code files. Refer to Intermediate Code Modules in section 14. In the previous example, subprogram A uses the SIN and COS functions contained within the user program while subprogram B binds in the SIN and COS functions in the intrinsic function file and references them.

IMPLICIT STATEMENT

The IMPLICIT statement allows the default type assigned to a variable, due to the initial character, to be altered.

The following is the proper form of the IMPLICIT statement:



G50307

<length> is an integer constant or integer constant expression in parentheses and is the length of the character entity that assumes a default type of CHARACTER when it begins with one of the letters specified in the IMPLICIT statement. <letter> is a letter of the alphabet.

A program unit can contain one or more IMPLICIT statements. No letter can appear in more than one IMPLICIT statement in a program unit. If used, IMPLICIT statements must appear before any other statements except FILE statements (in the main program), Compiler Control Images, comments, FUNCTION statements, SUBROUTINE statements, and optionally, PARAMETER statements. The letters used as the first letter of a symbolic constant in a PARAMETER statement preceding the IMPLICIT statement(s) must not appear in an IMPLICIT statement in that program unit. The IMPLICIT statement applies only to symbolic names in the program unit in which the statement appears, including function and dummy arguments.

Symbolic names, whose initial character lies between or is the same as one of the indicated letters, are to be of the specified type. Each element of the list can be one or two letters separated by a hyphen. If the element is a letter, a name must begin with that letter to be assigned the specified default type. If the element is a hyphenated letter pair, the letter pair indicates a range of initial characters with which the default type is associated. The second letter in a hyphenated pair must be greater in the collating sequence than the first letter.

Examples of valid IMPLICIT statements follow:

```
IMPLICIT REAL (I-N)
IMPLICIT CHARACTER * 10 (A-Z)
IMPLICIT DOUBLE PRECISION (D)
IMPLICIT LOGICAL (A-C,L), REAL(D-F), COMPLEX(X)
```

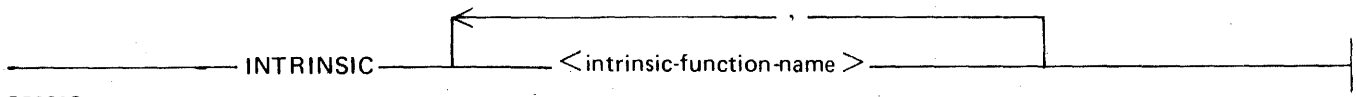
An IMPLICIT statement, occurring in a function subprogram, applies to the name of the function, any entry names, and all other symbolic names in the function subprogram, unless an explicit type is specified in an EXPLICIT type statement or in the FUNCTION statement.

IMPLICIT ranges that overlap (for example, REAL (A-K) INTEGER (I-M)) generate an error message. The first specification is used to determine the variable type.

INTRINSIC STATEMENT

The INTRINSIC statement permits the specific name of an intrinsic function to be used as an actual argument. If the specific name of an intrinsic function is used as an actual parameter, it must appear in an INTRINSIC statement.

The proper form of the INTRINSIC statement follows:



G50308

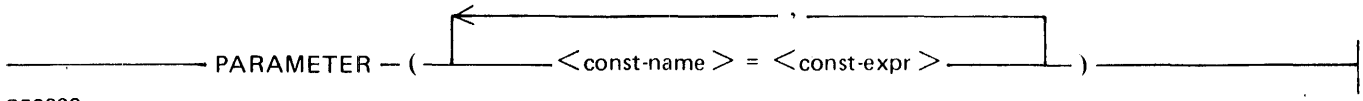
<intrinsic-function-name> is the name of an intrinsic function in the F77INTRIN file. The intrinsic functions MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, and MIN1 cannot be used as actual arguments (however, this does not preclude usage in an expression that is an actual argument).

An example of the INTRINSIC statement follows:

```
INTRINSIC DCOS, NINT
```

PARAMETER STATEMENT

A PARAMETER statement is used to assign a symbolic name to a constant. The following is the proper form of the PARAMETER statement:



G50309

<const-name> is the symbolic name of a constant that becomes defined with the value determined from <const-exp>, an expression involving only constants and other symbolic names of constants as operands, in accordance with the rules for assignment statements as shown in table 6-1. If the constant name is of type INTEGER, REAL, DOUBLE PRECISION, or COMPLEX, the corresponding constant expression must be an arithmetic constant expression. If the constant name is of type CHARACTER or LOGICAL, the corresponding constant expression must be a character constant expression or a logical constant expression, respectively.

Any symbolic name of a constant that appears in a constant expression must have been defined previously in the same or a different PARAMETER statement in the same program unit.

If a symbolic name of a constant is not of default implied type, the type must be specified by an explicit type statement or IMPLICIT statement prior to the first appearance in a PARAMETER statement.

Once such a symbolic name is defined, the name can appear in that program unit in any subsequent statement as a constant in an expression or in a DATA statement. A symbolic name of a constant must not be part of a format specification. A symbolic name of a constant must not be used to form part of another constant.

An example of a PARAMETER statement follows:

```
PARAMETER (I=10)
    .
    .
    .
EXAMPL= I.0
```

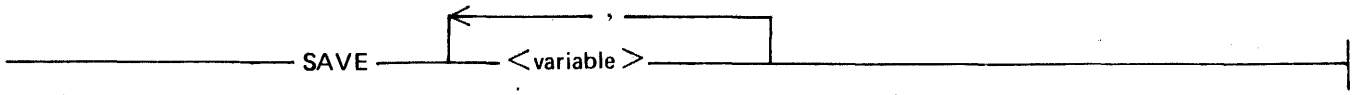
A symbolic name of a constant must not be given a value more than once in a program unit. A symbolic name in a PARAMETER statement can identify only the corresponding constant in that program unit.

Examples of the PARAMETER statement follow:

```
CHARACTER *3 A
LOGICAL L
PARAMETER (A="ABC", X=1.414, I=1)
PARAMETER (L=I.GT.2, Y=14.051)
    .
    .
    .
Z=X+Y-12.3
```

SAVE STATEMENT

The SAVE statement is used to retain a variable and its value after the execution of a RETURN or END statement in a subprogram. The form of the SAVE statement follows:



G50310

<variable> is a named COMMON block name preceded and followed by a slash, a variable name, or an array name. Redundant appearances of an item are not permitted. Dummy argument names, procedure names, and names of entities in a COMMON block must not appear in a SAVE statement.

Within a function or subroutine subprogram, an entity specified by a SAVE statement does not become undefined as a result of the execution of a RETURN or END statement in the subprogram. However, such an entity in a COMMON block can become undefined or redefined in another program unit.

A SAVE statement in a main program is optional and has no effect on program execution.

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

The appearance of a COMMON block name preceded and followed by a slash (/) character in a SAVE statement has the effect of specifying all of the entities in that COMMON block.

If a particular COMMON block name is specified by a SAVE statement in a subprogram of an executable program, it must be specified by a SAVE statement in every subprogram in which that COMMON block appears.

If a named COMMON block is specified in a SAVE statement within a subprogram, the current values of the entities in the COMMON block at the time a RETURN or END statement is executed are made available to the next program unit that specifies that COMMON block name at execution time.

If a named COMMON block is specified in the main program unit, the current values in the COMMON block are made available to each subprogram that specifies that named common block. A SAVE statement in the subprogram that specifies this named COMMON block has no effect on program execution.

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

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

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

SECTION 7
EXPRESSIONS

The manner in which expressions are constructed and the general features of the statements that form the basis of the FORTRAN 77 language are described in this section.

GENERAL

The purpose of expressions is to specify equation-oriented rules whereby a unique data value can be obtained as a result of operations performed on other data values.

An expression is any valid constant, variable, function reference, or any combination of these items separated by appropriate operators and parentheses. The expression represents the value obtained when the indicated operations are performed on the indicated values.

Expressions are divided into three basic types: arithmetic, character, and logical.

OPERATORS

The operators which can be employed by a FORTRAN 77 expression are listed in table 7-1. The relative precedence assigned to each operator by the compiler is shown. The highest relative precedence is eight.

The presence of these operators in an expression indicates that an arithmetic, character, or logical operation or a relational comparison is to be performed. Operations of equal precedence are performed from left to right, except exponentiation which is carried out from right to left. The unary + operator is ignored. Parentheses can be used to override operator precedence. A character expression (an expression which returns a character value) must not contain any arithmetic operators. Likewise, an arithmetic expression (an expression which returns a numeric value) must not contain any character operators.

Table 7-1. Operators Used in FORTRAN 77 Expressions

Operator	Type	Relative Precedence	Function Represented
**	Arithmetic	8	Exponentiation
– (Unary)	Arithmetic	7	Change of sign
/	Arithmetic	6	Division
*	Arithmetic	6	Multiplication
–	Arithmetic	5	Subtraction
+	Arithmetic	5	Addition
//	Character	5	Concatenation
.NE.	Relational	4	Not equal to
.GE.	Relational	4	Greater than or equal to
.GT.	Relational	4	Greater than
.EQ.	Relational	4	Equal to
.LE.	Relational	4	Less than or equal to
.LT.	Relational	4	Less than

Table 7-1. Operators Used in FORTRAN 77 Expressions
(continued)

Operator	Type	Relative Precedence	Function Represented
.NOT.	Logical	3	Logical negation
.AND.	Logical	2	Logical conjunction
.OR.	Logical	1	Logical disjunction
.EQV.	Logical	0	Logical equivalent
.NEQV.	Logical	0	Logical nonequivalent

ARITHMETIC EXPRESSIONS

An arithmetic expression is a rule for computing a numeric value.

An arithmetic expression can contain only arithmetic operators and numeric constants, symbolic names of constants, variables, array elements, function references, and arithmetic expressions in parentheses. Logical or character operands of any type are not permissible in arithmetic expressions. In general, mixed arithmetic operand types are permissible.

Immediately adjacent operators are not permissible and parentheses must be used to avoid adjacent operators (for example, A**(-2)).

Expression Types

The types of operands in an arithmetic expression determine the type of the value obtained from the evaluation of the expression. When a COMPLEX value is combined with any other type of value in an operation, the result is of COMPLEX type. If none of the operands in an arithmetic operation are COMPLEX and at least one is DOUBLE PRECISION, the result is of DOUBLE PRECISION type. If none of the operands in an arithmetic operation are COMPLEX or DOUBLE PRECISION and at least one of them is REAL, the result is of REAL type. Only if all of the operands in an arithmetic operation are INTEGER, is the result of type INTEGER. Tables 7-2 and 7-3 illustrate the resultant types of arithmetic operations depending upon the types of operands and the operator involved. DOUBLE indicates DOUBLE PRECISION type.

For the operators +, −, *, and /, the result of the operation is of the following type:

Table 7-2. Resultant Types of Arithmetic Operations

Type of First Operand	Type of Second Operand			
	INTEGER	REAL	DOUBLE	COMPLEX
INTEGER	INTEGER	REAL	DOUBLE	COMPLEX
REAL	REAL	REAL	DOUBLE	COMPLEX
DOUBLE	DOUBLE	DOUBLE	DOUBLE	COMPLEX
COMPLEX	COMPLEX	COMPLEX	COMPLEX	COMPLEX

For exponentiation (**), the result of the operation is of the following type:

Table 7-3. Resultant Types for Exponentiation

Type of Base	Type of Exponent			
	INTEGER	**REAL**	**DOUBLE**	**COMPLEX**
INTEGER	INTEGER	REAL	DOUBLE	COMPLEX
REAL	REAL	REAL	DOUBLE	COMPLEX
DOUBLE	DOUBLE	DOUBLE	DOUBLE	Prohibited
COMPLEX	COMPLEX	COMPLEX	Prohibited	COMPLEX

In the case of a divide operation involving two integer operands, the result is an integer value. Thus, the expression 3/2 represents the value 1 and the expression 3.0/2 represents the value 1.5.

Examples of valid arithmetic expressions follow (all variables are nonlogical, noncharacter):

```
6
1+6
SIN(3.14159*(-A)+2)
BID(M(1),N(2))
-B*A
A+(-P)
6**X
```

CHARACTER EXPRESSIONS

A character expression returns a character value of variable length. If a character expression is used in an assignment statement, the resultant value must be assigned to a character variable, character substring, character array element, character array element substring, or character function currently being defined. The operands in the expression can be character constants, symbolic names of character constants, variables, substrings, array elements, array element substrings, function references, or other character expressions. The operator used for combining operands in a character expression is concatenation (//). A character expression must not involve concatenation of an operand whose length specification is an asterisk in parentheses, unless the operand is the symbolic name of a constant. The value returned by the character expression has the total length of all the operands involved in the expression.

Examples of valid character expressions follow:

'DEFGHIJKLM'	Simplest form. The expression length equals 10.
D	Expression length is the length of variable D.
X // 'ABCD'	Expression length equals the length of X plus 4.
Y // Z // A(2:3) // 'X'	Expression length equals the length of Y plus the length of Z plus 2 plus 1.
F // FUN(A,B,C)	FUN is a function call (or character array element). The expression length equals the length of F plus the length of the value returned by the function FUN.

Each variable and function in the above expressions must be of type CHARACTER. For more information on character assignment, refer to section 8.

LOGICAL EXPRESSIONS

The value TRUE or FALSE is returned by logical expressions. The operands in a logical expression consist of the following:

1. Logical constant.
2. Symbolic name of a logical constant.
3. Logical variable name.
4. Logical array element reference.
5. Logical function reference.
6. Relational expression.
7. Logical expression in parentheses.

If a logical expression is used in an assignment statement, the result must only be assigned to a logical variable, logical array element, or logical function currently being defined.

Logical Operators

The .NOT. operator expresses logical negation. It changes the value of a logical operand to its complement. For example, if A is TRUE, the value of .NOT.A is FALSE.

The .AND. operator produces the logical product of two logical expressions. The operation A.AND.B is TRUE if both A and B are TRUE; the operation is FALSE if either A or B or both are FALSE.

The .OR. operator produces the logical sum of two logical expressions. The operation A.OR.B is TRUE if either A or B or both are TRUE; the operation is FALSE if both A and B are FALSE.

The .EQV. operator returns the value TRUE when both operands have the same value (are equivalent) and returns the value FALSE when the operands have different values. For example, if A and B are both FALSE, or if A and B are both TRUE, the operation A.EQV.B is TRUE; if A and B have different values, the operation is FALSE.

The .NEQV. operator is opposite the .EQV. operator and returns the value TRUE only when the two operands have different values (are not equivalent).

Table 7-4 summarizes the the preceding explanation of the logical operators.

Table 7-4. Logical Expression Constructs

A	B	.NOT.A	.NOT.B	A.AND.B	A.OR.B	A.EQV.B	A.NEQV.B
T	T	F	F	T	T	T	F
T	F	F	T	F	T	F	T
F	T	T	F	F	T	F	T
F	F	T	T	F	F	T	F

Examples of logical expressions follow (variables A, B, C, and array L are of type LOGICAL):

```
A
A.OR.L(3)
A.OR.B.AND.C      (A.OR.B).AND.C (equivalent expressions)
A.EQV.L(1).NEQV.L(2).OR..NOT.C
```

Relational Expressions

Relational expressions provide the capability to compare numeric or character values and return the value TRUE or FALSE depending on the result of the comparison. A relational expression must only appear in a logical expression. Except when used in a relational expression, numeric or character expressions cannot appear in a logical expression.

When numeric or character operands are used in a relational expression, numeric operands must be compared with numeric operands and character operands must be compared with character operands. Character operands in relational expressions are compared lexically. Each character storage location is compared with the corresponding character storage location in the other half of the relational expression according to the relative location in the EBCDIC collating sequence. In this sequence, A is less than Z, Z is less than 0, and 0 is less than 9.

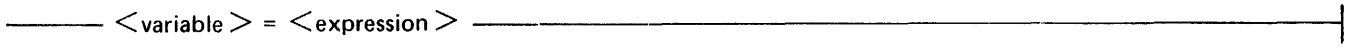
Parentheses can be used to override operator precedence.

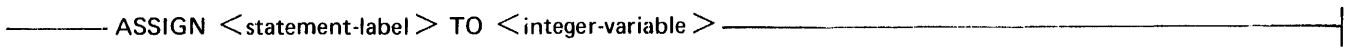
Examples of logical expressions involving relational expressions follow (B is LOGICAL):

```
A.GT.(F+G).OR.B
C.LE.I-J
(6*K).LT.(4-T)
'HENRY'.EQV.'FRED'
```

SECTION 8
ASSIGNMENT STATEMENTS

Assignment statements allow arithmetic, logical, character, or label values to be assigned to program variables. The two proper forms of the assignment statement follow:





In the first form, <variable> is a variable name as described in section 5. <variable> can be a simple variable name, array element name, or character substring (in a character assignment statement only). <expression> determines the type of assignment to be made. <expression> can be numeric, in which case the <variable> must be of numeric type (REAL, INTEGER, DOUBLE PRECISION, or COMPLEX); <expression> can be logical, in which case <variable> must be of type LOGICAL; <expression> can be a character expression, in which case the <variable> must be of type CHARACTER.

In the second form, <statement-label> must be the label of a statement that appears in the same program unit as the ASSIGN statement. <statement-label> must be the label of an executable statement or a FORMAT statement.

ARITHMETIC ASSIGNMENT STATEMENT

An arithmetic assignment statement involves an arithmetic expression which returns a numeric value that is assigned to a numeric variable or array element. When such a statement is executed, the arithmetic expression is evaluated and the value obtained is placed into the storage word or word pair allocated to the variable or array element.

The variable and the arithmetic expression need not be of the same type. If the types are different, the expression is first evaluated and automatic conversion is subsequently performed on the value obtained to agree with the type of variable to be assigned the value. This automatic conversion proceeds according to the rules indicated in table 8-1.

Table 8-1. Type Conversions in Assignment Statements

Type of Variable	Type of Expression			
	Integer	**Real**	**Double Precision**	**Complex**
INTEGER	None	INT	INT	INT
REAL	REAL	None	REAL	REAL
DOUBLE PRECISION	DBLE	DBLE	None	DBLE
COMPLEX	CMPLX	CMPLX	CMPLX	CMPLX

The following notation is used in this table:

Word	Meaning
None	No conversion.
CMPLX	Perform REAL and assign to real portion; assign 0. to imaginary.
DBLE	Convert to DOUBLE PRECISION.
INT	Round to nearest integer.
REAL	Convert to REAL.

The CMPLX, DBLE, INT, and REAL functions have the same effect as the CMPLX, DBLE, INT, and REAL intrinsic functions. Refer to section 13 for more information about these intrinsic functions.

The means of determining the type of an expression is given in tables 7-2 and 7-3. The internal storage formats of the various data types are described in appendix D.

Examples of valid arithmetic expressions follow:

```
O(IROW+2,-4) = IROW-K(-137,0)
N=I+2+B/3.7
L=2.6+(1/2.3)
```

LOGICAL ASSIGNMENT STATEMENT

A logical assignment statement involves a logical expression (refer to Logical Expressions in section 7) which is assigned to a variable or array element of type LOGICAL. When such a statement is executed, the logical expression is evaluated, and the logical value is placed into the storage word allocated to the logical variable.

Examples of valid logical assignment statements follow:

```
L=.TRUE.
LOGIC(2,4) = LOGIC(1,1).AND.L(1,2)
L=G.GT.H.EQV..NOT.B.EQ.C
L='FORT'.LE.'RAN'
```

CHARACTER ASSIGNMENT STATEMENT

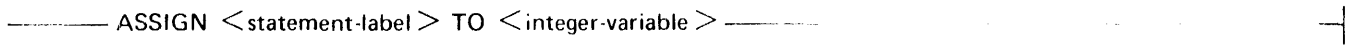
A character assignment statement involves a character expression which returns a character value. This value is assigned to a variable, substring, or array element of type CHARACTER. When the character value returned is not of the same size as the variable to which it is to be assigned, padding or truncation occurs. If the value returned by the character expression is larger than the character variable to the left of the equal sign, characters in the returned value are truncated from the right until the value is the same size as the variable that is to receive it. If the value is smaller than the variable, the value is assigned left-justified to the variable and any unassigned character storage locations in the variable are padded with blanks.

Examples of valid character assignment statements follow:

```
CHARACTER *3 A,B(-2:14),C*6
A='TG'                          A contains 'TG '
B(0)='LE' // A                  B(0) contains 'LET'
B(-1)(1:1)='A'                  B(-1) contains 'A '
C=A(2:2) // 'O' // B(0) // B(-1)  C contains 'GOLETA'
```

ASSIGN STATEMENT

The ASSIGN statement stores the label of an executable statement in an integer variable. The syntax of the ASSIGN statement follows.



Execution of an ASSIGN statement causes <statement-label> to be stored in <integer-variable>. <statement-label> must be the label of an executable statement or a FORMAT statement that appears in the same program unit as the ASSIGN statement.

Execution of an ASSIGN statement is the only way to store a statement label value in a variable. A variable must be defined with a statement label value when referenced in an assigned GO TO statement (refer to section 9) or when referenced as a format identifier (refer to section 12) in an input/output statement.

An integer variable defined with a statement label value can be redefined with the same or different statement label value, or with an integer value. When defined with a statement label value, the variable must not be referenced in any other way.

Example:

```
ASSIGN 250 TO LABEL
GO TO LABEL (150,250,350)
```

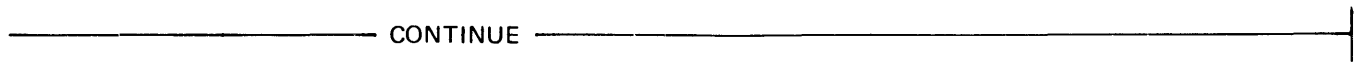
SECTION 9
CONTROL STATEMENTS

The executable control statements are used to alter the normal flow of the program, terminate or suspend execution, or control iterative processes. Control can be transferred to labeled executable statements only. The control statements are described in the following paragraphs in the order listed:

CONTINUE statement
DO statement
END statement
GO TO statement
IF statement
ELSE IF statement
ELSE statement
END IF statement
PAUSE statement
STOP statement

CONTINUE STATEMENT

The executable CONTINUE statement has no effect on program execution. The following is the proper form of the CONTINUE statement:



G50312

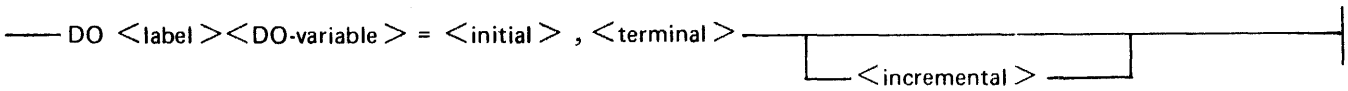
The CONTINUE statement is a dummy executable statement allowing the programmer to position a label at any desired point within a program. This facilitates transfers to that point and allows the range of a DO loop to be clearly delimited.

An example of two CONTINUE statements within a partial FORTRAN 77 program follows:

```
      .
      .
      .
      DO 10 I=2,10,2
        A(I)=I/M
        WRITE (6,100)A(I)
        IF (A(I)) 30,10,10
10    CONTINUE
      M=-M
30    CONTINUE
      .
      .
      .
```

DO STATEMENT

The DO statement is a control statement provided to alter the order of the execution of program statements. The DO statement allows a series of statements to be repeatedly executed while the value of a specified program variable is varied between specified limits. The number of times a DO loop is executed is dependent upon an iteration count. The following is the proper form of the DO statement:



G50313

<label> in the diagram is the statement label of the terminal statement of the DO loop. The <initial>, <terminal>, and <incremental> parameters are any arithmetic expressions of type INTEGER, REAL, or DOUBLE PRECISION. The <DO-variable> is an integer, real, or double-precision variable which is assigned the value of the <initial> parameter upon execution of the DO statement. The <incremental> parameter is added to the <DO-variable> after execution of the terminal statement of the DO loop. Assignments are made according to the rules established in section 8, ASSIGNMENT STATEMENTS. If the <incremental> parameter is left out the value 1 is assumed. The <terminal> parameter is used in loop execution control to determine the number of times the DO loop is executed. Loop execution control is described later in this section.

Range of a DO Loop

The range of a DO loop consists of all the executable statements following the DO statement up to and including the terminal statement specified in that DO statement.

The range of a DO loop occurring within the range of another DO loop must be entirely contained within the range of the outer DO loop. This is referred to as nesting of DO loops. More than one DO loop can have the same terminal statement.

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

Example:

```
            IF (I.EQ.5) THEN
              DO 10 J=14,1,-1
                IF (R.LE.12.9) THEN
                  X=7.1+X
                ELSE
                  X=3.22
                END IF
10            CONTINUE
            ELSE
              X=1.01
            END IF
```

DO Statement Execution

The execution of a DO statement causes the following to occur:

1. The DO loop becomes active.
2. The initial, terminal, and incremental parameters are evaluated.
3. The <DO-variable> is assigned the initial parameter value.
4. The iteration count is determined.

DO Loop Activation

A DO loop becomes active when the corresponding DO statement is executed. The DO loop becomes inactive when the iteration count is determined to be 0, when a branch is made to a statement outside the range of the DO loop, or when a RETURN or STOP statement is executed within the range of the DO loop. Branching to a statement outside the range of a DO loop from within the range of the DO loop is permitted; however, it is prohibited to branch into the range of a DO loop from outside the range of the DO loop.

Parameter Evaluation

When the DO statement is executed, the values of the <initial>, <terminal>, and <incremental> parameters (DO-parameters) are determined. If necessary, the values are converted to the type of the <DO-variable>. Any variables used in the parameter expression can be altered within the range of the DO loop without affecting loop execution control or iteration processing.

DO-variable Initialization

After determining the values of the DO-parameters, the <DO-variable> is assigned the value of the <initial> parameter. The value of the <DO-variable> can be accessed within the range of the DO loop by the program; however, the <DO-variable> must never be assigned another value within the range of the DO loop.

Iteration Count Initialization

The iteration count determines the number of times the DO loop is executed (barring a branch to a statement outside the range of the DO loop). The initial value of the iteration count is established by evaluating the following expression:

MAX (INT ((<terminal> − <initial> + <incremental>) / <incremental>), 0)

The iteration count is zero whenever:

<initial> > <terminal> and <incremental> > 0, or
<initial> < <terminal> and <incremental> < 0.

At completion of execution of the DO statement, loop execution control begins.

Loop Execution Control

Loop execution control determines whether or not all of the statements in the range of the DO loop are to be executed. The iteration count is tested, and if nonzero, execution continues with the first executable statement within the range of the DO loop. If the iteration count is zero, the DO loop becomes inactive. If, as a result of this inactivation, all DO loops sharing the same terminal statement become inactive, execution continues with the first executable statement after the terminal statement.

If some of the DO loops sharing the terminal statement are active, execution continues as described in Iteration Processing in this section.

Execution of Statements in the Range

Statements in the range of a DO loop are executed until the terminal statement is reached. A subprogram reference is not a transfer of control outside the range of the DO loop. If the <DO-variable> is passed as a parameter, it must not be assigned another value within the subprogram.

Terminal Statement Execution

Execution of the terminal statement occurs as a result of the normal execution sequence or as a result of transfer of control from within the DO loop to the terminal statement of the same DO loop. If the execution of the terminal statement does not cause transfer of control, execution continues with iteration processing.

Iteration Processing

Iteration processing causes the following four steps to be performed:

1. The <DO-variable>, iteration count, and <incremental> parameter of the last active DO statement executed are chosen for processing.
2. The iteration count is decremented by one.
3. The <DO-variable> is incremented by the value of the <incremental> parameter.
4. Control is passed to the loop execution control of the DO loop that was chosen for iteration processing.

Consider the following two examples:

Example:

```
      DO 10 F=3.7, 9.81, 2.03
        X=F
        DO 10 I=14, -3, -2
          J=J+1
10        K=K-I
      CONTINUE
```

Upon execution of the CONTINUE statement F=11.82, X=9.79, I=-4, J=9, and K=-54.

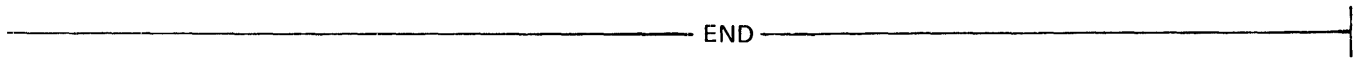
Example:

```
        DO 20 L=10, 1
        M=L
20      CONTINUE
```

Upon execution of the CONTINUE statement L=10, and M=0. The statement in the range of the DO loop is never executed.

END STATEMENT

Each program unit consists of a sequence of statements terminated by an END statement. The END statement is provided for use as the terminal statement of a program unit. The proper form of the END statement is as follows:



G50314

Every program unit must contain exactly one END statement. If an END statement is encountered during execution of a subprogram, a RETURN is implied; if an END statement is encountered in a main program, a STOP is implied.

GO TO STATEMENT

The GO TO statement can be used to transfer control from one point of an executing program to another point in the same program unit. The GO TO statement has three forms: 1) the unconditional GO TO, 2) the computed GO TO, and 3) the assigned GO TO. These three forms of the GO TO statement are described in the following paragraphs.

Unconditional GO TO

The simplest form of the GO TO statement is the unconditional GO TO which has the following form:

G50315

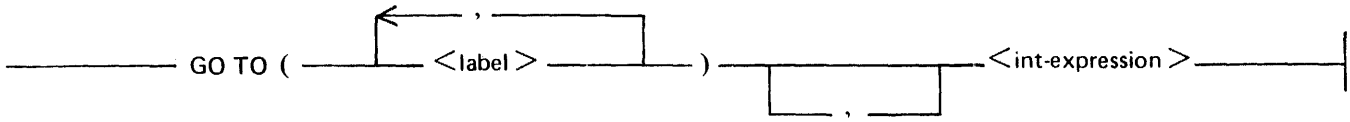
<label> is the statement label of an executable statement in the same program unit as described in section 3.

Execution of this control statement causes the executable statement bearing the indicated label in the program unit to be the next statement executed. For example, the statement GO TO 23 causes program flow to continue at the statement labeled 23.

The statement following a GO TO statement must have a label unless it is an END statement or END IF statement. This is a syntactical requirement since a GO TO statement breaks the sequential flow of execution. It is never possible to return to execute the statement following the GO TO unless that subsequent statement has a label.

Computed GO TO

The second form of the GO TO statement is the computed GO TO statement. The execution of this statement causes control to be transferred to a statement whose label appears in the list portion of the statement or to the next executable statement following the GO TO. How control is transferred depends on the value of the integer arithmetic expression following the label list. The expression is evaluated and the result is used to select one of the labels in the list. The computed GO TO has the following form:



G50316

If the <int-expression> has the value n when computed, control passes to the n-th label in the label list. If there are fewer than n labels in the list or if n is less than or equal to zero, control passes to the next executable statement.

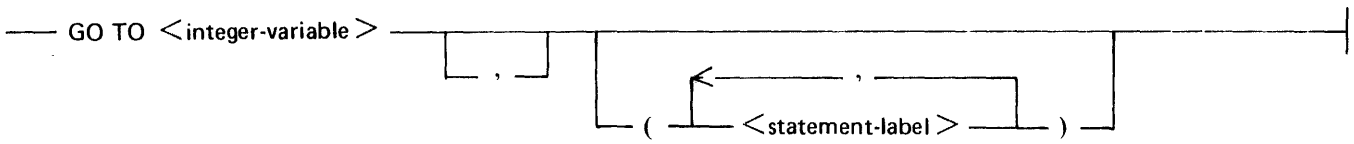
An example of a computed GO TO statement follows:

```
GO TO (1,25,3,6,1,17),I+1
```

At execution time, the value of I+1 is computed. If I+1 has the value n, then control passes to the n-th statement in the list. For example, if I + 1 = 4 (I = 3), then control passes to the statement labeled 6, the fourth label in the list. If I + 1 = 1 or 5 (I = 0 or 4) in this particular example, control passes to the statement labeled 1, since both the first and fifth elements of the label list are 1. If I + 1 is less than 1 or greater than 6 (I < 0 or > 5), control passes to the next executable statement after the computed GO TO.

Assigned GO TO Statement

The syntax of an assigned GO TO statement follows:



<integer-variable> is an integer variable name. <statement-label> is the statement label of an executable statement that appears in the same program unit as the assigned GO TO statement. The same <statement-label> can appear more than once in the same assigned GO TO statement.

At the time of execution of an assigned GO TO statement, <integer-variable> must be defined with the value of a statement label of an executable statement that appears in the same program unit. <integer-variable> can be defined with a statement label value only by an ASSIGN statement (refer to section 8) in the same program unit as the assigned GO TO statement. The execution of the assigned GO TO statement causes a transfer of control to the statement identified by the statement label.

If the parenthesized list of <statement-label>s is present, the statement label assigned to <integer-variable> must be one of the <statement-label>s in the list.

An example of an assigned GO TO statement follows:

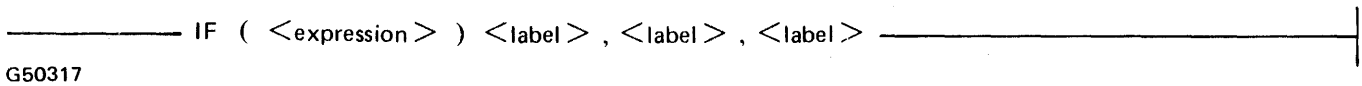
```
ASSIGN 250 TO LABEL
GO TO LABEL (150,250,350)

      .
      .
      .

GO TO LABEL
```

IF STATEMENT

The IF statement is a control statement provided to cause conditional execution of program statement(s), depending upon the value of an arithmetic or logical expression.

Arithmetic IF Statement

The arithmetic IF statement causes conditional branching depending on the value of an arithmetic expression within the statement. The arithmetic IF statement has the following form:



G50317

<expression> is an arithmetic expression of INTEGER, REAL, or DOUBLE PRECISION type. <label> is a statement label as described in section 3.

The arithmetic IF statement is a 3-way branch. The arithmetic expression inside the parentheses following the IF is evaluated and control is transferred to the statement identified by the first, second, or third label, depending on whether the expression is negative, zero, or positive, respectively.

An example of an arithmetic IF follows:

```
IF (I-J) 10, 20, 30
```

If I – J is negative, control is transferred to the statement labeled 10. If I – J is zero, control is transferred to the statement labeled 20. If I – J is positive, control is transferred to the statement labeled 30. Notice that this is actually a test of whether or not J is greater than, equal to, or less than I.

Not all three statement labels of an arithmetic IF need to be different.

Example:

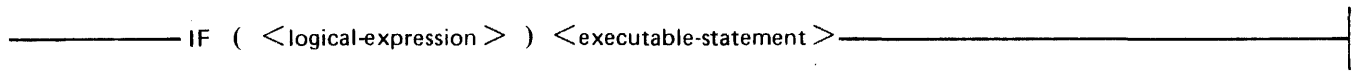
```
IF((A-2)*(B-3)) 10, 10, 3
```

In this example, control passes to the statement labeled 3 only if (A-2)*(B-3) is greater than zero; otherwise, control passes to the statement labeled 10.

The statement following an arithmetic IF must have a label unless it is an END statement.

Logical IF Statement

The logical IF statement conditionally executes a statement depending on the result of a logical expression. The logical IF statement has the following form:



G50318

<logical-expression> is a logical expression as described in section 7. <executable-statement> is any executable statement described in section 3 except a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement.

<logical-expression> is evaluated. If the value is TRUE, the statement following <logical-expression> is executed. If the value is FALSE, the statement is ignored. In either case, control passes to the next statement, unless the statement following <logical-expression> was executed and caused a branch to another point in the program.

The following are examples of logical IF statements:

Example:

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

If A equals B, or C equals D (or both), then G is incremented by 1; otherwise, G remains unchanged. In any event, control passes to the next statement.

Example:

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

If L1, which must be declared to be a LOGICAL variable, is TRUE, control passes to the statement labeled 97. If L1 is FALSE, control passes to the next statement.

Example:

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

If A is less than or equal to 97, the arithmetic IF is executed and control passes to statement number 12 or 13, depending on the value of B. If A is greater than 97, control passes to the next statement.

Block IF Statement

The block IF statement is used along with an END IF statement to control the program execution sequence. The ELSE IF and ELSE statements are optionally used to further segment the block IF statement. The block IF statement has the following form:

G50319

<logical-expression> is a logical expression as described in section 7.

The block IF statement permits the execution of multiple statements depending on the result of a single condition or multiple conditions within the block IF. Any executable statements can be used following a block IF statement including another block IF statement.

Example (L is a logical variable):

```
      IF (J.EQ.7) THEN
        IF (L) GO TO 50
        IF (X.GT.0.25) I=I+1
        X=5.5
      END IF
      X=6.5
50    CONTINUE
```

Nesting Level

When a block IF statement occurs within another block IF statement, the second block IF statement is nested within the first. The nesting level of a statement is l = n1 − n2, where n1 is the number of block IF statements occurring up to and including the statement, and n2 is the number of END IF statements occurring before the statement. Every statement must have a nesting level that is positive or zero. An ELSE IF statement, ELSE statement, or END IF statement only controls execution of the block IF statement at the same nesting level. A block IF statement is terminated by the first succeeding END IF statement with the same nesting level. Transfer of control to a statement within an IF-block from outside the IF-block is prohibited.

An example of invalid transfer of control follows:

```
      IF (Y.EQ.8) THEN
10      IF (X.EQ.5) GO TO 20
        IF (X.EQ.7) THEN
20        U=V**W
30        GO TO 40
        END IF
40      V=V+5.4
      END IF
```

Statement 10 is invalid since it is branching into the range of a block IF from outside the range of the block IF. Statement 30, however, is valid since it is not entering another block IF, only leaving a block IF.

Block IF Statement Execution

Execution of a block IF statement causes evaluation of <logical-expression>. If the value of <logical-expression> is TRUE, execution continues with the first executable statement following the block IF statement and proceeds until the next ELSE IF statement, ELSE statement, or END IF statement on the same nesting level. The statements between the block IF statement and the next ELSE IF statement, ELSE statement, or END IF statement on the same nesting level are referred to as the IF-block. If the execution of the last statement in the IF-block does not result in a transfer of control, control is transferred to the next executable statement following the END IF statement that terminates the block IF statement.

```
10    IF (X.GT.Y) THEN
20      K=5
30      IF (T.LE.0) THEN
40        X=9
50      END IF
60      Y=7
70    END IF
```

The END IF statement at line 50 terminates the IF-block beginning at line 30. The END IF statement at line 70 terminates the IF-block beginning at line 10. If the value of the logical expression in line 10 is TRUE, lines 20 through 60 are executed. If the value of the logical expression in line 30 is TRUE, line 40 is executed.

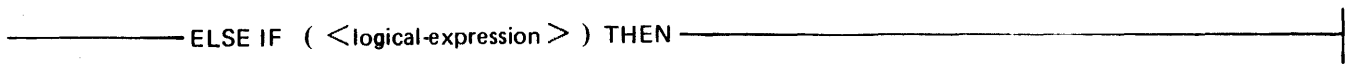
An IF-block can be empty, in which case control passes to the next executable statement following the END IF statement that terminates the block IF statement. If the value of <logical-expression> is FALSE, control passes to the next ELSE IF statement, ELSE statement, or END IF statement with the same nesting level.

Example (L is a logical variable):

```
IF (L) THEN
ELSE
  X=5
END IF
```

ELSE IF Statement

The ELSE IF statement is used to specify an alternate execution sequence within a block IF statement. The ELSE IF statement has the following form:



G50320

<logical-expression> is a logical expression as described in section 7.

The ELSE IF statement must only occur within a block IF statement. There can be any number of ELSE IF statements between the block IF statement and the next ELSE statement, or the END IF statement of the same nesting level. An ELSE IF statement must not occur within an ELSE block.

ELSE IF Statement Execution

When an ELSE IF statement is executed, <logical-expression> is evaluated and, if TRUE, program execution continues with the first executable statement of the ELSE IF-block. The statements between the ELSE IF statement and the next ELSE IF statement, ELSE statement, or END IF statement are referred to as the ELSE IF-block. If the last statement in the ELSE IF-block does not cause transfer of control, or if the ELSE IF-block is empty, control is transferred to the first executable statement following the END IF statement that terminates the block IF statement at the same nesting level.

If the value of <logical-expression> is FALSE, control is transferred to the next ELSE IF statement, ELSE statement, or END IF statement at the same nesting level.

Example:

```
IF (W.EQ.1) THEN
  X=1
ELSE IF (W.EQ.2) THEN
  Y=2
  W=3
ELSE IF (W.EQ.3) THEN
  Z=3
END IF
```

X is assigned a value only if W is equal to 1. Y is assigned a value only if W is equal to 2. Z is assigned a value only if W is equal to 3 and none of the preceding conditions are TRUE. Note that if W is equal to 2, only the first ELSE IF-block is executed even though W is equal to 3 upon leaving the second ELSE IF-block. No more than one ELSE IF-block can be executed. An ELSE IF statement is executed only when the block IF statement and all other preceding ELSE IF statements at the same nesting level return the value FALSE.

Transfer of control into an ELSE IF block from outside the ELSE IF-block is prohibited. The statement label, if any, of the ELSE IF statement must not be referenced by any statement.

ELSE Statement

The ELSE statement delimits a segment of the block IF statement. The ELSE statement has the following form:

G50321

The ELSE statement must only occur within a block IF statement. There can be only one ELSE statement at the same nesting level within a block IF statement. The ELSE statement is only executed if a value of FALSE is returned for the logical expression in the block IF statement and for the logical expressions in every ELSE IF statement at the same nesting level as the ELSE statement.

ELSE Statement Execution

The execution of an ELSE statement has no effect. The statements between the ELSE statement and the following END IF statement at the same nesting level are called the ELSE-block. The ELSE-block can contain any executable statements except an ELSE IF statement or an ELSE statement at the same nesting level. If an executable statement in the ELSE-block does not cause transfer of control, control is transferred to the first executable statement following the END IF statement at the same nesting level.

Transfer of control into an ELSE-block from outside of the ELSE-block is prohibited. The statement label, if any, of an ELSE statement must not be referenced by any other statement. An ELSE IF-block cannot occur within an ELSE-block at the same nesting level.

An example of the ELSE statement follows:

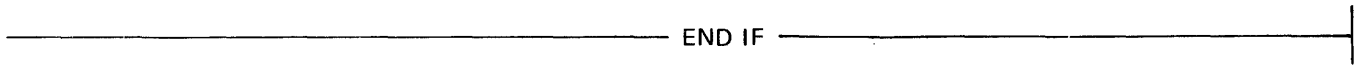
```
 10      IF (A.EQ.7) THEN
 20         X=1
 30      ELSE IF (A.EQ.6) THEN
 40         X=2
 50      ELSE IF (A.EQ.5) THEN
 60         IF (A.EQ.4) THEN
 70            X=3
 80         ELSE
 90            X=4
100         END IF
110      ELSE
120         IF (A.EQ.3) THEN
130         ELSE IF (A.EQ.2) THEN
140            X=5
150         ELSE
160            X=6
170         END IF
180      END IF
190      CONTINUE
```

Line 20 is only executed when the value of the logical expression in line 10 is TRUE. After line 20 is executed, a transfer of control is made to line 190. Line 40 is only executed when the value of the logical expression in line 10 is FALSE and the value of the logical expression in line 30 is TRUE. In this case, line 20 is not executed. Line 60 begins a nested IF-block. Line 60 is only executed if the value of the logical expression in line 50 is TRUE and the previous conditions at lines 10 and 30 are FALSE. The END IF statement at line 100 ends the IF-block beginning at line 60. If line 60 is executed and the value of the logical expression is TRUE, line 70 is executed and control is transferred to line 190; however, if the value of the logical expression is FALSE, lines 80 and 90 are executed and control passes to line 190.

If the values of the logical expressions in lines 10, 30, and 50 are all FALSE, control passes to line 110 and the IF-block beginning at line 120 is evaluated.

END IF Statement

The END IF statement terminates the block IF statement. The END IF statement has the following form:

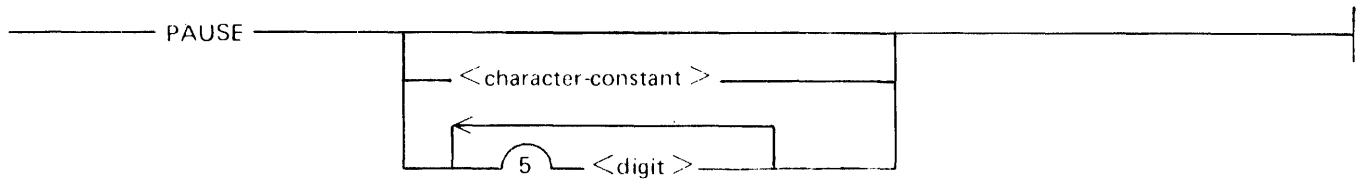


G50322

For each block IF statement there must be a corresponding END IF statement. The corresponding END IF statement is the next END IF following the block IF statement at the same nesting level. Execution of the END IF statement has no effect.

PAUSE STATEMENT

The PAUSE statement is provided to allow an executing program to be suspended indefinitely. The proper form of the PAUSE statement is as follows:



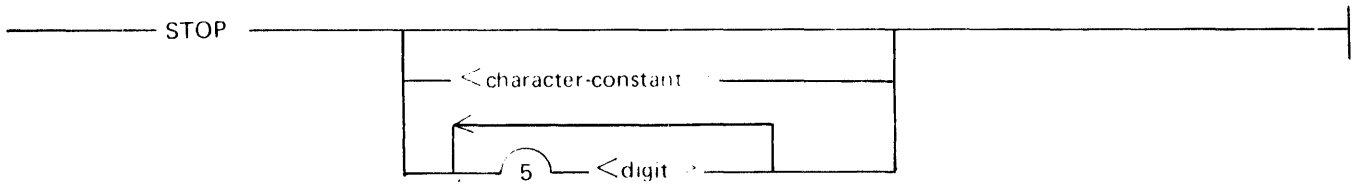
G50323

The PAUSE statement can be followed by a decimal string of up to five <digits>, or a <character-constant> (as described in section 4).

The execution of the PAUSE statement causes the unconditional suspension of the program being executed, pending operator action. The required operator action is <job #>OK. In addition to suspending the program, the execution of this statement causes the optional integer or string following the PAUSE statement to be displayed at the operator display terminal (ODT).

STOP STATEMENT

The STOP statement is provided to allow the termination of an executing program. The following is the proper form of the STOP statement:



G50324

The STOP statement can be followed by a decimal string of up to five <digits>, a <character-constant> (as described in section 4).

Execution of the STOP statement causes the unconditional termination of the program being executed. The execution of the STOP statement is the generally accepted manner in which a program can reach error-free termination. The optional integer or string following the STOP statement is displayed on the ODT.

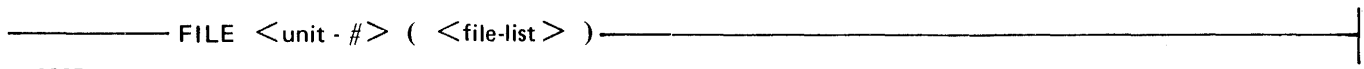
SECTION 10
FILE DECLARATIONS

The FILE declaration statement associates a unit number with an external file and assigns values to certain file specifiers and file attributes for the unit. An external file is referenced in a FORTRAN 77 I/O statement with a unit number. By default, the B 1000 FORTRAN 77 compiler associates the unit numbers 5, 6, and 7 with the card reader, line printer, and card punch files, respectively. When associations other than these are required, or additional files are desired, FILE declaration statements must be used to inform the compiler of the attributes of the files.

The values assigned to file attributes and file specifiers in a FILE declaration statement can be overridden in several ways. A list of ways in which file attributes and file specifiers can be assigned follows. The list is arranged in order of precedence, with the first item having the highest precedence.

1. A value can be assigned in an OPEN statement (refer to section 11).
2. A value can be assigned by file-equating a file in an EXECUTE statement (refer to B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982).
3. A value can be assigned by modifying the object file using the MODIFY command (refer to B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982).
4. A value can be assigned in a FILE declaration statement.

The following is the proper form of the FILE declaration statement:



G50325

The word FILE must be in columns 1 through 4 of the card image and must be followed by at least two blanks. The rest of the line is free-form format. <unit-#> is an integer expression which specifies the number of the unit to which a file is connected. <file-list> is a list of values for file attributes and certain file specifiers separated by commas.

The following paragraphs describe the file attributes and specifiers that can appear in <file-list>. Only one of each of the following specifiers is allowed for each FILE statement.

ACCESS = <access-type>

The ACCESS specifier determines whether the access mode for the file is sequential or direct. <access-type> is either a character constant expression having the value SEQUENTIAL or DIRECT, or is a string of characters spelling the word SEQUENTIAL or DIRECT. Once a file is declared to have a specific access mode, it can only be accessed in that manner. The default value of <access-type> is SEQUENTIAL.

BLANK = <blnk>

The BLANK specifier determines how blank characters are interpreted in an input file. <blnk> is either a character constant expression having the value NULL or ZERO, or is a string of characters spelling the word NULL or ZERO. If NULL is specified, blank characters not enclosed in quotes are ignored. If ZERO is specified, blank characters not enclosed in quotes are interpreted as zeros. The default value of <blnk> is NULL.

BLOCKSIZE = <block-size>

The BLOCKSIZE attribute specifies the length of a block. <block-size> is an integer constant expression evaluating to a blocksize in bytes and having a maximum value of 65,535. The default value of <blocksize> is the value of the RECL specifier.

FILE = <file-name>

The FILE specifier gives the external name of the file to be accessed. <file-name> must be enclosed in quotation marks. B 1000 naming conventions are described in the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982.

If the FILE specifier is not given, the external name is assumed to be the same as the internal name, and a search for the internal name is made on the device specified. The internal name is constructed by concatenating the word FILE with the file number specified by the user (for example, FILE3).

FORM = <form>

The FORM specifier determines whether the file is being connected for formatted or unformatted I/O. <form> is either a character constant expression having the value FORMATTED or UNFORMATTED, or is a string of characters spelling the word FORMATTED or UNFORMATTED. If the FORM specifier is omitted, a value of UNFORMATTED is assumed if the file is being connected for direct access, and a value of FORMATTED is assumed if the the file is being connected for sequential access. For a new file, the processor creates the file with a set of allowable forms that includes the specified form. For an existing file, <form> must be included in the set of allowable forms for the file.

KIND = <hardware-type>

The KIND specifier determines the device to which the file is connected. <hardware-type> is either a character constant expression having the value DISK (disk pack), PRINTER (line printer), READER (card reader), TAPE (magnetic tape), PUNCH (card punch), ODT (operator display terminal), or REMOTE (remote terminal), or is a string of characters spelling one of the values. The default value of <hardware-type> is DISK.

MYUSE = <use-type>

The MYUSE attribute specifies how the file will be used. <use-type> is either a character constant expression having the value IN (for input only), OUT (for output only), or IO (for both input and output), or is a string of characters spelling one of the values. Refer to table 10-1 for additional information.

RECL = <record-length>

The RECL specifier gives the record length for all records of a file. <record-length> is an integer constant expression evaluating to a record length in bytes. The default value of <record-length> depends on the value of the KIND specifier. Refer to table 10-1 for additional information.

STATUS = <file-status>

The STATUS specifier gives the file status. <file-status> is either character constant expression having the value NEW, OLD, SCRATCH, or UNKNOWN, or is a string of characters spelling one of the values. When a file is opened which is declared to have a STATUS value of OLD, the declared device

is searched to locate the file. If the file is not found, a NO FILE condition results. When the file is closed by the program, the file, including any updates that were made, is saved. When a file is opened and is declared to have a STATUS value of NEW, the declared device is not searched to locate the file. A new file is created. When closed at the end of the program, the new file is saved on the device specified. If SCRATCH is specified, a new file is created for use by the executable program, but is deleted when the file is closed. If UNKNOWN is specified, the file is assumed to exist. If the MYUSE specifier is not used in the FILE declaration statement with the STATUS specifier, using SCRATCH causes the MYUSE attribute of IO to be set. Using UNKNOWN without the MYUSE specifier causes the MYUSE attribute of IN to be set. The default value of <file-status> is OLD, unless the file KIND is PRINTER or PUNCH, in which case the default value is NEW.

Association of a unit number with a hardware device (by means of a FILE declaration statement) associates the unit with the default attributes indicated in table 10-1. Individual attributes can be redefined either in the FILE declaration for the file, or with a MODIFY (MO) MCP command. Refer to the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982 for an explanation of the MODIFY command. Those file attributes not specifically stated in a FILE statement are given the default values. File attributes which cannot be specified in the FILE statement must be changed with the MODIFY command.

Table 10-1. Default Attributes

Specifier	DISK	TAPE	READER	PRINTER	PUNCH	REMOTE	ODT
ACCESS	SEQUENTIAL for all hardware types						
BLANK	NULL for all hardware types						
BLOCKSIZE	The value of the RECL specifier						
FILE	See Note 1						
FORM	See Note 2		FORMATTED				
RECL	180	180	80	132	80	80	80
MYUSE	See Note 3		IN	OUT	OUT	IO	IO
STATUS	OLD	OLD	OLD	NEW	NEW	OLD	OLD

Notes for table 10-1:

1. The default for all hardware types is "FILE" // <unit-#>.
2. The default is SEQUENTIAL for FORM = FORMATTED, and DIRECT for FORM = UNFORMATTED.
3. The defaults for MYUSE depend on the STATUS specifier.

STATUS	MYUSE	
NEW	IO	
OLD	IN	
SCRATCH	IO	(STATUS=NEW)
UNKNOWN	IN	(STATUS=OLD)

Each FILE declaration statement can contain the description of only one data file. Each data file must have a unique unit number associated with that file; no unit number can refer to more than one data file. The unit number must always be included in a FILE declaration statement.

FILE declaration statements must precede all other statements except comments and Compiler Control Images.

The default file types associated with unit numbers 5, 6, and 7 are shown in table 10-2.

Table 10-2. Unit Number/Hardware Default Associations

Unit Number	Hardware Type	Internal/External File-Name
5	Card Reader	FILE5
6	Line Printer	FILE6
7	Card Punch	FILE7

An example of two FILE declaration statements follows:

```
FILE 4 (FILE= 'FORTRAN77/SAMPLE/FILE', RECL= 6, KIND= 'TAPE')
FILE 5 (FILE='DATA', STATUS=NEW, ACCESS=DIRECT, RECL=112, KIND=DISK)
      .
      .
      .
    N=4
    READ (N,100) A, B, C
    WRITE (5,200,REC=I) A, B, C
    WRITE (6,300) D, E, F, G
```

In the example, file 5 no longer defaults to the card reader but is attached to a disk file that is being created (NEW attribute). The name of the disk file associated with file 4 can also be specified as 'SAMPLE/FILE ON FORTRAN77'. File 6 in the second WRITE statement gets the default device for that file (the line printer). Thus, no FILE declaration statement is necessary.

SECTION 11
INPUT/OUTPUT

Input/output statements permit the transfer of data between a program and various peripheral devices or, as in the case with internal files, between character entities and other program variables. There are four input/output statements in FORTRAN 77: READ, WRITE, PRINT, and PUNCH.

ACCESS METHODS

Two access methods are available in FORTRAN 77. An input/output statement can specify sequential access or direct access to a file. A file that has been declared with one access type cannot be referred to using a different access type.

Sequential

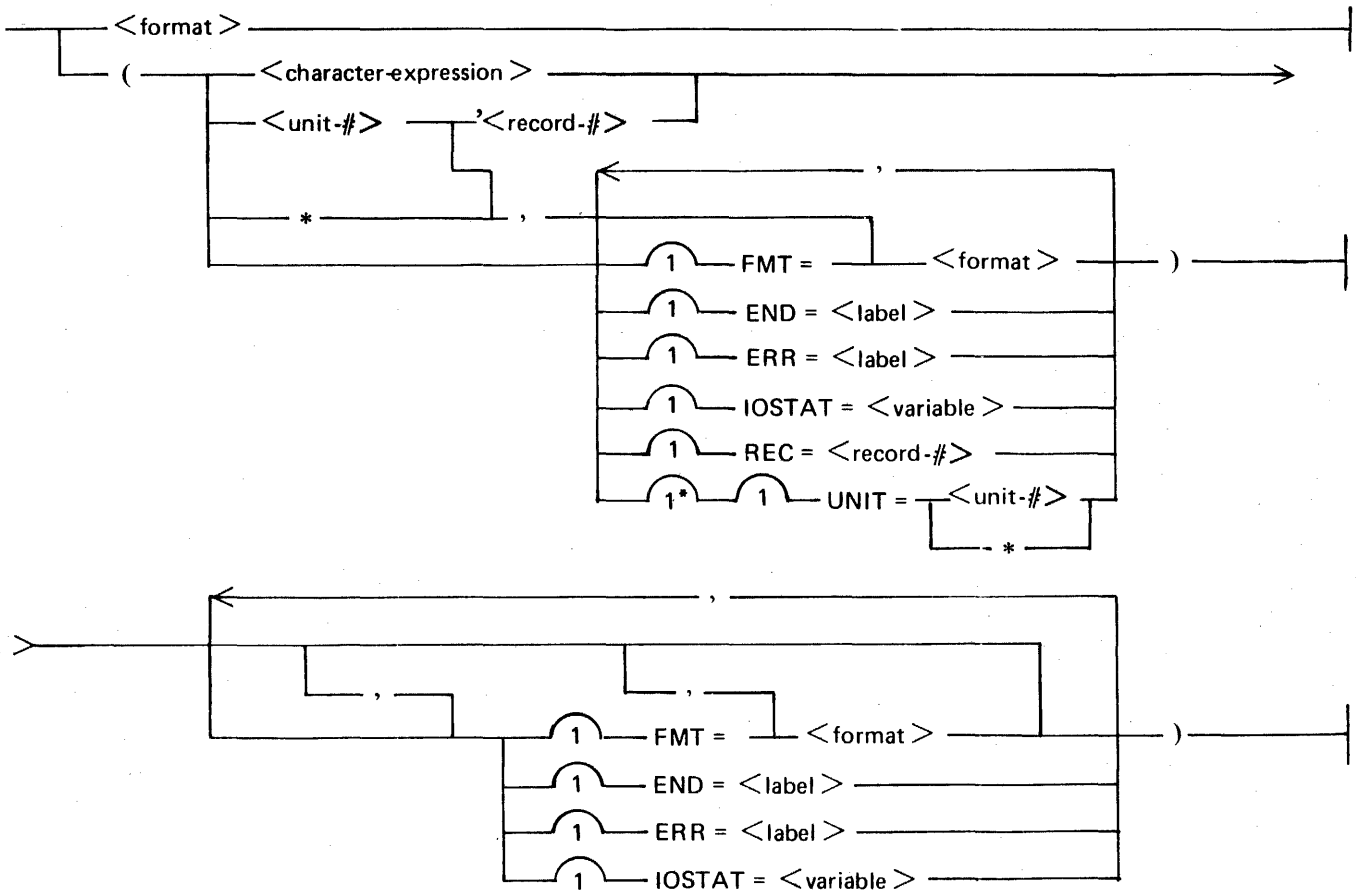
A file is opened by the first I/O statement accessing that file. When the file is opened for sequential access, the file pointer is pointing at the beginning of the first record. When an I/O statement has finished execution, the file pointer points at the beginning of the next record. A file that does not have a DIRECT declaration in the FILE statement can only be accessed sequentially. A record number must never appear in an I/O statement specifying a file opened for sequential access.

Direct

For a file to be accessed directly, it must be declared as DIRECT in a FILE declaration statement. A file declared to be DIRECT must always have a record number specified in every control list referencing the file. The record number is a relative pointer to a record in the file. Any record of the file can be accessed directly without first accessing the preceding records by giving the relative record number in a control list referencing the file containing the record.

CONTROL LIST

Every input/output statement has a control list that gives information concerning which unit the I/O operation is being performed on, the format of the data to be transferred, which record of the file is to be accessed, and what to do if an error condition occurs. A control list has the following form:



G50326

<format> is the format specifier for the input/output statement; it determines the manner in which the data is transmitted.

The unit specifier, <unit-#>, is an integer expression giving the unit number of the file on which the I/O operation is being performed. An asterisk (*) character appearing in the place of <unit-#> specifies unit 5 if the control list appears in a READ statement or unit 6 if the control list appears in a WRITE statement. A <character-expression> can replace the unit number, in which case internal I/O is specified. Refer to Internal Files in this section. There can be only one unit specifier in a control list.

<record-#> is an integer expression giving the number of the desired record in a file. This is only meaningful with direct access I/O.

The END= and ERR= specifiers are used when an abnormal condition results from the I/O operation. The IOSTAT= specifier provides a variable to store information regarding the result of the I/O operation.

Unit

The unit is the unit number of the file as declared in a FILE statement. There must be only one unit number specified in a control list for an I/O statement. If the UNIT= specifier is omitted, the unit number must be the first item in the list.

A character variable, character array, character array element, or character substring can replace the unit number. This is referred to as internal I/O and is explained under Internal Files in this section. If a character entity is used, do not specify a record number.

Format

The format in which data is to be transmitted is specified in the format portion of the control list. If the FMT= specifier is omitted, the <format> must follow the first comma in the list; the unit number, without UNIT=, must be the first item in the list.

The format specifier must be one of the following: 1) the label of a FORMAT statement, 2) a character constant, 3) a character expression giving a valid format specifier (except an expression that involves concatenation of a character entity with an asterisk (*) character as the length specification), 4) an asterisk (*) character for list-directed formatting, or 5) a namelist name for namelist formatting.

Record Number

A record number is specified when direct access of a file is required. The record number is a positive integer expression specifying the record number from/to which data is to be transmitted. REC= must always appear before the record number unless the form <unit-#>'<record-#> is used. The apostrophe in this shortened form separates the unit number from the record number. The unit number must be the first item in the control list and must not be preceded by the word UNIT=.

Action Specifiers

The entries END=, ERR=, and IOSTAT= are action specifiers used to control program flow dependent on the result of the I/O operation.

END= <label>

The END= option causes a branch to the statement with the specified label when one of the following conditions occurs:

1. Attempted to read beyond the last record previously written on disk.
2. Read the EOF mark.
3. Attempted to write beyond the end of the designated number of areas for a disk file.
4. Read a record beyond the end of an internal file.

If an END= specifier is not given in the control list for an I/O statement and one of the four stated conditions occurs, program execution is halted at the point the exception condition occurs. The END= specifier is not permitted with direct access I/O.

ERR = <label>

The ERR = option causes a branch to the statement with the specified label when one of the following conditions occurs:

1. For a READ (formatted or unformatted random) when the I/O variable list requests more data than the logical record contains.
2. For direct access files or sequential formatted files, the logical record size is greater than the declared record size. For sequential unformatted files, the I/O can request more data than the declared record size.
3. When the input data for a formatted read does not meet the requirements of the format specifier.
4. On a formatted write when the type of the variable does not match the format specifier.
5. When the random record key is less than 1.
6. When the format specification exceeds the record size.
7. When a parity error occurs during the data transfer.

If an ERR = specifier is not given for an I/O statement and one of the seven exception conditions occurs within the statement, program execution is terminated at the point where the exception condition occurs.

IOSTAT = <variable>

The IOSTAT = action specifier assigns a value to an integer <variable> or integer array element name depending on the result of the I/O operation in which the option occurs. If no error condition occurs, the <variable> or array element name is assigned the value 0. If one of the conditions specified under the END = option occurs, the variable is assigned a value according to the following table:

Condition	Value
No error	0
End of file	−1
Error	2

Examples of control lists:

```
(6,100)
(4'12,200,ERR=300)
(FMT=400, IOSTAT=J, UNIT=8, REC=103, ERR=800)
(*,'(2I4)')
(A(1),100)
```

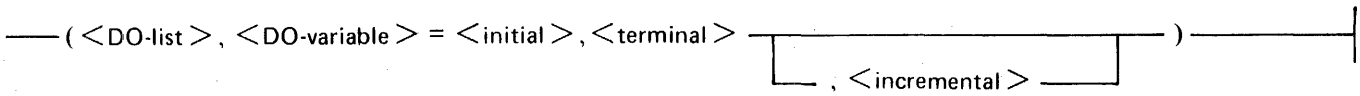
A(1) is a character array element being used as an internal file.

I/O LIST

In addition to having a control list, every input/output statement can have an I/O list associated with it. An I/O list specifies either the variables or array elements that are to be assigned the values of the data received from the file on input, or the data that is to be transmitted to the file on output. The items of the list are separated by commas, and the list can optionally contain implied-DO loops.

I/O Implied-DO Loop

During input, an I/O implied-DO loop permits assignment to a specified group of elements in an array (or arrays) in the input list. During output, an I/O implied-DO list permits the value of specified elements of an array (or arrays) to be transmitted, and allows the value of other list items in the output list to be transmitted repetitively. An I/O implied-DO loop has the following form:



G50327

The range of the implied-DO loop is the list <DO-list>. The parameters <initial>, <terminal>, and <incremental> can be any integer expressions. The I/O implied-DO loop is executed in the same manner as the DATA implied-DO loop. Refer to DATA Statement in section 6 for additional information.

With each iteration of the DO loop, each item in the <DO-list> is used once with every new value assigned to the <DO-variable>.

For input, members of the <DO-list> must be either array element names that optionally use the DO-variable as subscripts, or other I/O implied-DO loops that optionally use the outer <DO-variable> in their parameter lists. Each element of the <DO-list> must be separated by commas. The <DO-variable> of an implied-DO loop must not appear in the <DO-list> on input.

For output, the <DO-list> can contain array element names or I/O implied-DO loops, as stated above, as well as any constants or expressions separated by commas.

The most deeply nested I/O implied-DO loop is completed before the next outer <DO-variable> is incremented, at which time incrementation processing returns to the deeper DO loop level.

Example:

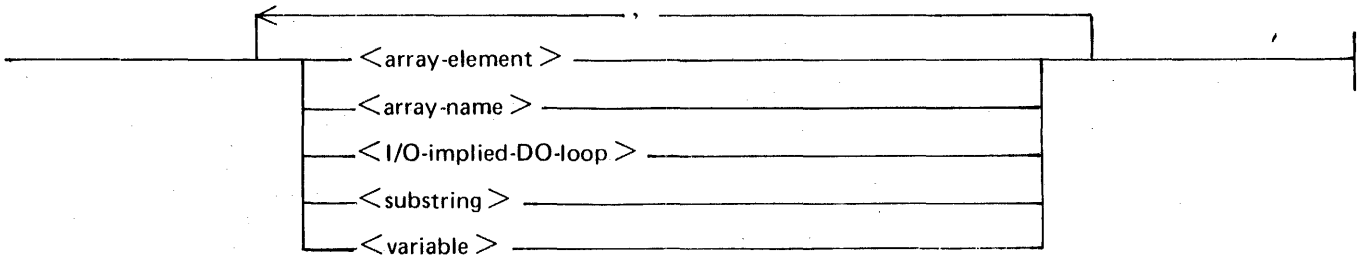
```
((A(I,J),I = 1,3),J = 1,10)
```

The array in the above example is accessed in the following sequence:

```
A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), ...
```

Input List

An input list contains the variables, arrays, array elements, and/or substrings that are to be assigned the values of the data received from the file. An input list has the following form:



G50328

The actual value assigned to an item in the input list is dependent upon the data type of the list item, the format specifier associated with the list item (refer to Format Specifications in section 12), and the value of the item in the data file.

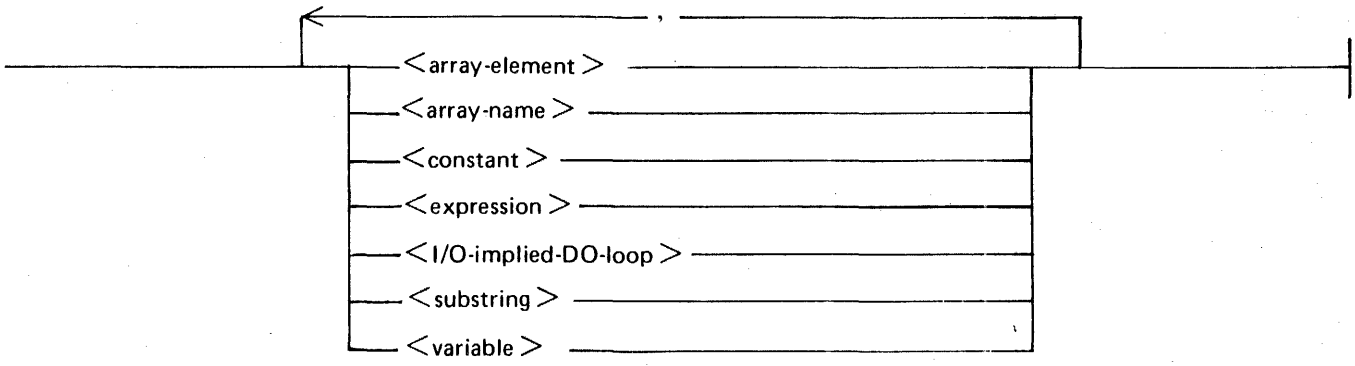
If an array name is specified in the input list (not an array element name or an array element name in an I/O implied-DO loop), values are assigned to each element of the array in a column-wise order (refer to appendix D) until the array is filled, or until the input record is exhausted. If the input record is exhausted, an error occurs (refer to ERR= Specifier in this section) unless there is a slash in the associated format.

Example of an input list:

```
(A(I),I=3,15), D(7), S(1:5), X(12,12)(4:5), (Y(J)(3:9),J=3,5), Z
```

Output List

An output list contains variables, arrays, array elements, substrings, constants, and expressions (except character expressions containing a character entity with a length attribute of asterisk (*)) to be transmitted to a file. An output list has the following form:



G50329

The number of significant digits in the value that is output from the items in the output list depends on the corresponding format specifier for the statement.

If an array name is given, every element of the array is output in the order in which the array is stored (refer to appendix D).

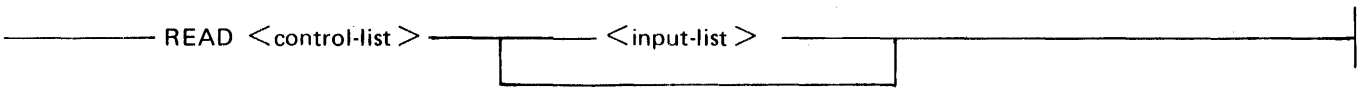
An example of an output list follows:

```
X+5, Y, (D(I),C,I=1,13), FUN(X, Y+4) + 3
```

READ STATEMENT

The READ statement reads data from peripheral storage or internal files, converts the data, and assigns the data to internal storage locations indicated by the input list and format portion of the statement. Provision can also be made to handle errors incurred during the read using action specifiers.

The proper form of the READ statement follows:



G50330

If only a format appears as the <control-list>, unit 5 (the card reader) is assumed.

There are two types of data access used with the READ statement: sequential and direct. Sequential access is assumed when no record number appears in the <control-list>.

Sequential READ

In a sequential READ operation, an entire block of records is brought into the file buffer when the first READ occurs. The file pointer is positioned at the beginning of the first record, and the first record is scanned and edited. The data is then assigned to the items in the input list. If there is more data in the record than is required by the READ input list, the remaining data items are skipped. When the READ operation is completed, the pointer is positioned at the beginning of the next record.

If a slash character appears in the format for a READ statement, the rest of the record (or all of the record if the slash is encountered when the record pointer is at the beginning of the record) is skipped and the record pointer is positioned at the beginning of the next record. The remaining items in the input list are filled from this point unless another slash appears. Refer to Slash Editing in section 12 for additional information.

With each subsequent READ or when a slash appears in the format specifier, a new record is accessed and assigned to the items in the input list. When all the records in a block have been read, a new block is brought into the buffer by the first READ that accesses a record of the new block.

Examples of sequential READ statements follow:

```
      READ (6,100) (A(I),I=1,M), C, D(2:4)
10    READ (*,'(I2,A4)')B, C
      READ 4, 200, I, J, K
```

Direct-Access READ

A direct-access READ can be performed on a file declared as DIRECT. Direct-access READ permits the access of any record within the file without first reading the records that precede the desired record. The direct-access READ has the same general form as a sequential READ. They differ in that a direct-access READ contains a record number in the control list.

When the initial READ operation occurs in DIRECT mode, the specified file is opened and the block containing the specified record is brought into the file buffer. Each time a subsequent READ occurs, the file buffer is checked to see if the specified record is already in the buffer. If the record is found in the buffer, the record is read in from that buffer. If the record is not in the buffer, the file is read again and a new block containing the desired record is brought into the buffer.

The record number specified in the control list must be between 1 and the number of records in the file.

Examples of direct-access READ statements follow:

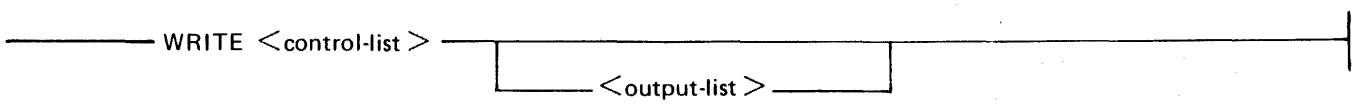
```
READ (6'5,100) A
READ (UNIT=7, FMT=200, REC=12-J, ERR=300) A, B, C(3)
```

If a slash (/) character does not appear in the format for the READ, the record pointer for the file is positioned at the beginning of the next record. If a slash does appear in the format for the READ, the pointer is positioned at the beginning of the next record, and the READ continues from that point.

WRITE STATEMENT

The WRITE statement writes data to peripheral storage, or to an internal file from the internal storage locations indicated by the output list of the statement. The data can be converted during the transfer process and positioned within records of a file depending on the specifications in the format identifier. Provision can also be made to handle error conditions using action specifiers.

The WRITE statement has the following form:



G50331

A WRITE statement writes one or more records to a file in either SEQUENTIAL or DIRECT mode. A file connected for sequential access must be written sequentially. A file connected for direct access must be written using a record number specifier in the control list.

Sequential WRITE

A sequential WRITE operation to an external file transmits the data in the output list to one or more records in a file buffer. The initial write opens the file and positions the file pointer at the beginning of the first record in the file. The file pointer is positioned at the beginning of the next record whenever a slash (/) character occurs in the format for the WRITE statement and when the WRITE operation is completed. When a block of records has been written to the buffer, the block is transferred to the actual hardware device.

If a slash (/) character appears in the format for the WRITE statement, the remaining character positions in the record are filled with blanks, the record is transmitted, the file pointer is positioned at the beginning of the next record, and the WRITE continues.

Examples of WRITE statements follow:

```
WRITE (6,100) A, B, 3+D, FUN(A,V), (A(I),I=1,15)
WRITE 5, 200, 7, 6, 5
WRITE (UNIT=5, FMT=100, ERR=300) ALPHA
WRITE (*, 400) BETA
```

An asterisk (*) character in the control list refers to file 6 (the line printer by default). An END= action specifier cannot appear in the control list of a WRITE statement.

Direct-Access WRITE

A direct-access WRITE operation transmits data to a specified record of the file being accessed. A record number must always appear in the control list for the direct-access WRITE.

When the initial WRITE operation occurs, the file is opened and the record specified is the initial record to receive the data in the output list, unless a slash (/) character appears in the format for the WRITE. In this case, the current record is filled with blanks and is transmitted. The file pointer is positioned at the beginning of the next record. The WRITE continues from this point.

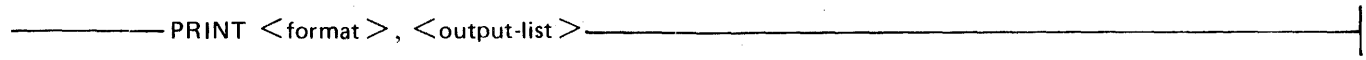
Examples of direct-access WRITE statements follow:

```
WRITE (UNIT=4, FMT=700, REC=X) A, D, 45, 'GARBLED'
WRITE (9'DAT-1, 300) X
```

PRINT STATEMENT

The PRINT statement obtains data from the internal storage locations indicated by the output list, converts the data, and writes it to unit 6 (the line printer in the default condition). This statement is a variation of the formatted WRITE statement; no action specifier is allowed and the unit number is not explicitly specified.

The proper form of the PRINT statement follows:



G50332

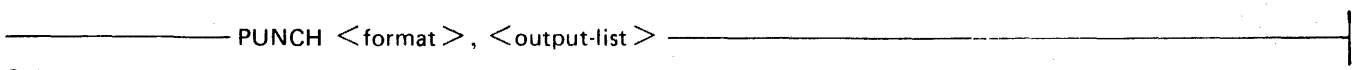
Execution of a PRINT statement writes data from internal storage to one or more records (if slash (/) characters appear in the format) of unit number 6. The position of the data file is unchanged prior to the execution of the PRINT statement. After such a statement is executed, the file pointer is positioned at the record immediately after the record(s) written.

<format> specifies the manner in which the transferred data is edited. <format> is the label of a FORMAT statement or a character expression (refer to section 12).

PUNCH STATEMENT

The PUNCH statement obtains data from the internal storage locations indicated by the output list of the statement and punches to the card punch by default. This statement is a variation of the formatted WRITE statement. No action specifier is allowed with this statement.

The proper form of the PUNCH statement follows:



G50333

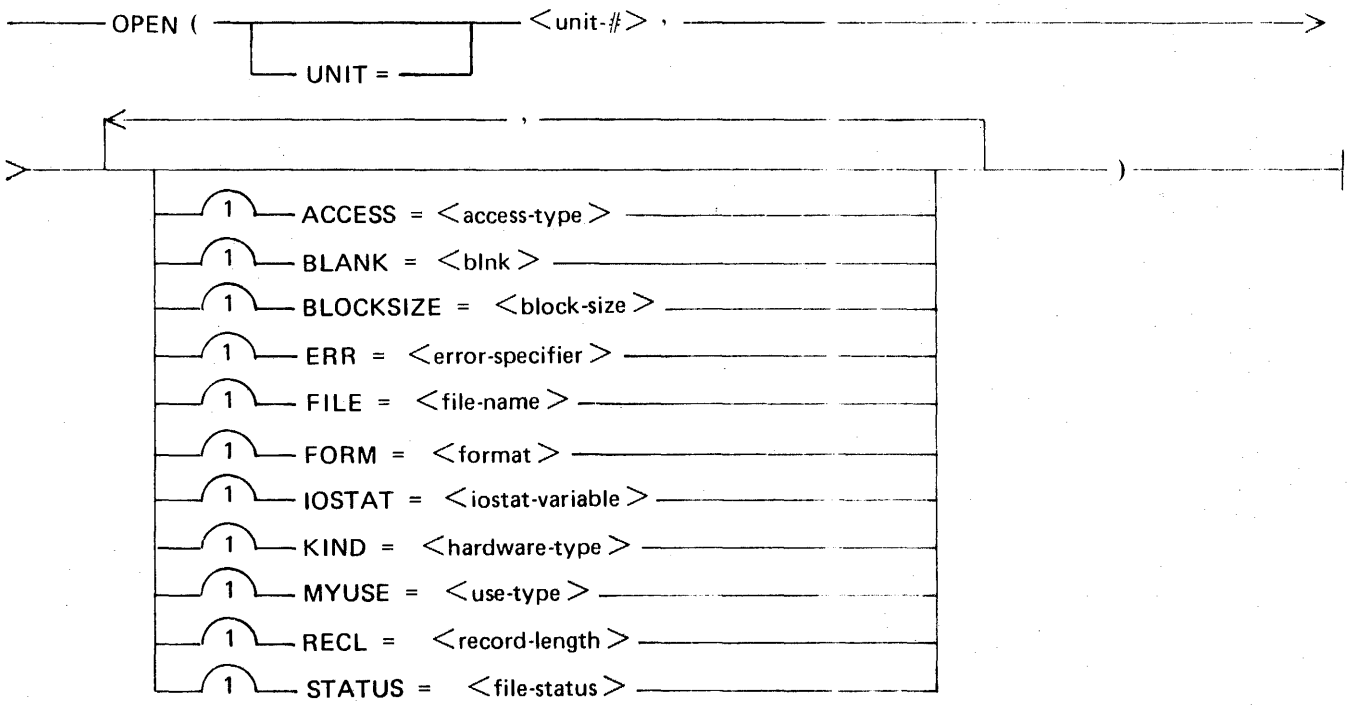
The operation of this statement is identical to the operation of the PRINT statement, except for the unit number. Each PUNCH statement has the implied unit number 7, a card punch file by default. This default condition can be overridden by redefining unit 7 in a FILE statement.

OPEN STATEMENT

The OPEN statement is used to: 1) connect an existing file to a unit, 2) create a file that was connected at the start of program execution, 3) create a file and connect it to a unit, or 4) change certain specifiers of a connection between a file and a unit.

The execution of the OPEN statement can occur in any program unit of an executable program and once the file is connected, it can be referenced in any program unit of the executable program.

The OPEN statement has the following form:



The following paragraphs describe the file attributes and specifiers that can appear in an OPEN statement.

UNIT = <unit-#>

<unit-#> is an integer expression that specifies the unit number to which a file is connected or to which a file is to be connected.

ACCESS = <access-type>

<access-type> is a character expression whose value is either SEQUENTIAL or DIRECT. The value of <access-type> specifies the access method for the connection of the file as being sequential or direct. For an existing file, the specified access method must be included in the set of allowed access methods for the file. For a new file, the processor creates the file with a set of allowed access methods that includes the specified method. The default value of <access-type> is SEQUENTIAL.

BLANK = <blnk>

<blnk> is a character expression whose value is either NULL or ZERO. If NULL is specified, all blank characters in numerically formatted input fields on the specified unit are ignored. The only exception is a field having all blank characters which is given a value of 0. If ZERO is specified, all blank characters other than leading blank characters are treated as zeros (0). The default value of <blnk> is NULL. The BLANK specifier is permitted only for a file being connected for formatted input/output.

BLOCKSIZE = <block-size>

<block-size> is an integer expression that specifies the length of a block in bytes. The maximum value is 65535. The default value is dependent upon the physical unit (KIND) assigned to the file.

ERR = <error-specifier>

<error-specifier> is a statement label of an executable statement that appears in the same program unit as <error-specifier>. If the OPEN statement contains <error-specifier> and an error occurs during the execution of the OPEN statement, the following occurs:

1. Execution of the OPEN statement terminates.
2. The position of the file specified in the OPEN statement becomes indeterminate.
3. If the OPEN statement contains the IOSTAT specifier, <iostat-variable> is defined with an integer value as described in the paragraph on <iostat-variable> that follows.
4. Execution continues with the statement whose label is <error-specifier>.

FILE = <file-name>

<file-name> is a character expression whose value is the name of the file to be connected to the specified unit. If the FILE specifier is omitted and the unit is not connected to a file, the unit becomes connected to a file whose name is formed by the concatenation of FILE and <unit-#>, for example, FILE8.

FORM = <format>

<format> is a character expression whose value is either FORMATTED or UNFORMATTED. The FORM specifier indicates that the file is being connected for either formatted or unformatted input/output. If this specifier is omitted, a value of UNFORMATTED is assumed if the file is being connected for direct access, and a value of FORMATTED is assumed if the file is being connected for sequential access. For an existing file, the specified form must be included in the set of allowed forms for the file. For a new file, the processor creates the file with a set of allowed forms that includes the specified form.

IOSTAT = <iostat-variable>

<iostat-variable> is an integer variable or integer array element. <iostat-variable> is assigned the value zero (0) if no input/output error condition exists and it is assigned a processor-dependent positive integer if an input/output error condition does exist.

KIND = <hardware-type>

<hardware-type> is a character expression which specifies the device to which the file is connected. The allowable values for <hardware-type> follow:

DISK
ODT
PRINTER
PUNCH
READER
REMOTE
TAPE

The default value of <hardware-type> is DISK.

MYUSE = <use-type>

<use-type> is a character expression which specifies how the file is to be used for input/output. The allowable values for <use-type> follow:

Value	Definition
IN	Input only
OUT	Output only
IO	Input or output

The default for <use-type> depends on <file-status> as follows:

<file-status>	<use-type>
NEW	IO
OLD	IN
SCRATCH	IO
UNKNOWN	IN

RECL = <record-length>

<record-length> is an integer expression whose value must be positive. It specifies the length of each record in a file being connected for direct access. If the file is being connected for formatted input/output, the length is the number of characters. If the file is being connected for unformatted input/output, the length is measured in processor-dependent units. For an existing file, the <record-length> value must be included in the set of allowed record lengths for the file. For a new file, the processor creates the file with a set of allowed record lengths that includes the specified value. The RECL specifier must be given when a file is being connected for direct access; otherwise, it must be omitted.

STATUS = <file-status>

<file-status> is a character expression whose value is OLD, NEW, SCRATCH, or UNKNOWN. If OLD or NEW is specified, a FILE specifier must be given. If OLD is specified, the file must exist. If NEW is specified, the file must not exist. Successful execution of an OPEN statement with NEW specified creates the file and changes the status to OLD. If SCRATCH is specified with an unnamed file, the file is connected to the specified unit for use by the executable program but is deleted either at the execution of a CLOSE statement referring to the same unit or at the termination of the executable program. SCRATCH must not be specified with a named file. If UNKNOWN is specified, the file is assumed to exist. If this specifier is omitted, a value of UNKNOWN is assumed.

OPEN of a Connected Unit

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

If the file to be connected to the unit does not exist but is the same as the file to which the unit was connected at the start of program execution, the properties specified by the OPEN statement become a part of the connection.

If the file to be connected to the unit is not the same as the file to which the unit is connected, the effect is as if a CLOSE statement without a STATUS specifier had been executed for the unit immediately prior to the execution of the OPEN statement.

If the file to be connected to the unit is the same as the file to which the unit is connected, only the BLANK specifier can have a value different from the one currently in effect. Execution of the OPEN statement causes the new value of the BLANK specifier to be in effect. The position of the file is unaffected.

If a file is connected to a unit, execution of an OPEN statement on that file and a different unit is not permitted.

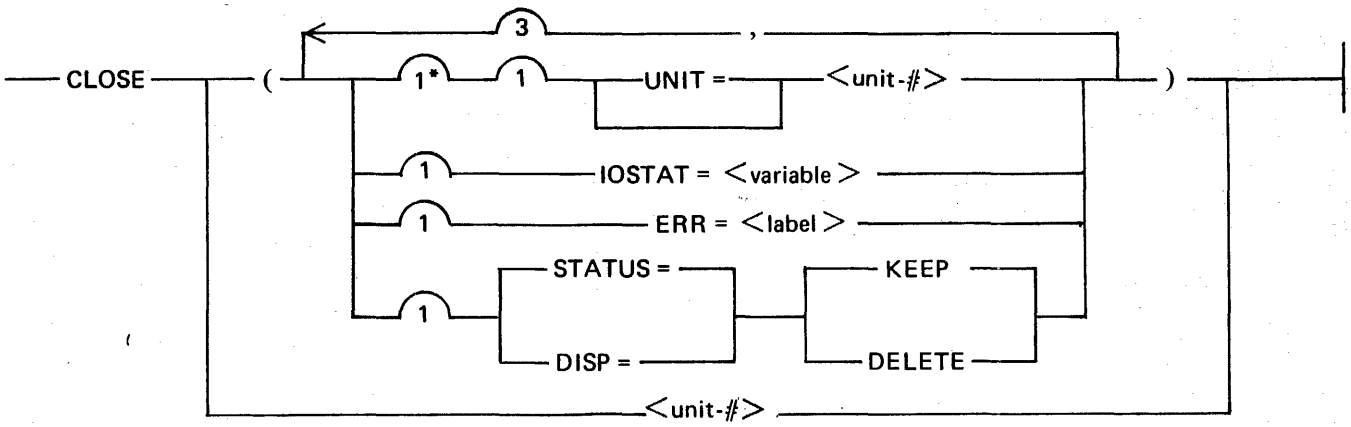
Examples of valid OPEN statements follow:

```
OPEN (8)
OPEN (UNIT=8)
OPEN (3,IOSTAT=IERR)
OPEN (UNIT=7,ERR=850)
OPEN (IUNIT,FILE=FNAME,KIND='DISK',MYUSE=IN)
OPEN (10,FILE='TEST.OUT',STATUS='NEW',BLOCKSIZE=90,BLNK='ZERO')
```

CLOSE STATEMENT

The CLOSE statement terminates the connection of a particular file to a unit.

The proper form of the CLOSE statement follows:



G50334

<unit-#> is an integer expression giving the unit number of the file as declared in a FILE statement. There must be one and only one unit number specified in the CLOSE statement. If UNIT= is omitted, <unit-#> must be the first item in the list; otherwise, the specifiers can occur in any order.

Execution of a CLOSE statement containing IOSTAT= <variable> causes <variable> to receive an integer value depending on the outcome of the CLOSE operation. The possible values follow:

Condition	Value
No error	0
End of file	−1
Error	2

If a CLOSE statement contains ERR= <label> and an error occurs during the CLOSE operation, the operation is terminated and execution continues with the statement labeled <label> which must be an executable statement that appears in the same program unit as ERR= <label>.

STATUS and DISP are equivalent. Use of these statements in a CLOSE operation determines the disposition of the file that is connected to the specified unit. KEEP must not be specified for a file whose status, prior to execution of the CLOSE statement, is SCRATCH. If KEEP is specified for a file that exists, the file continues to exist after the execution of the CLOSE statement. If KEEP is specified for a file that does not exist, the file will not exist after the execution of the CLOSE statement. If DELETE is specified, the file will not exist after execution of the CLOSE statement. If this specifier is omitted, the assumed value is KEEP unless the file status, prior to execution of the CLOSE statement, is SCRATCH. In this case, the assumed value is DELETE.

Examples of valid CLOSE statements follow:

```
CLOSE 6
CLOSE (UNIT = 2)
CLOSE (3,IOSTAT=I)
CLOSE (UNIT=5,ERR=500,IOSTAT=J)
CLOSE (9,STATUS=KEEP)
CLOSE (UNIT=8,IOSTAT=I,ERR=7500,DISP=DELETE)
```

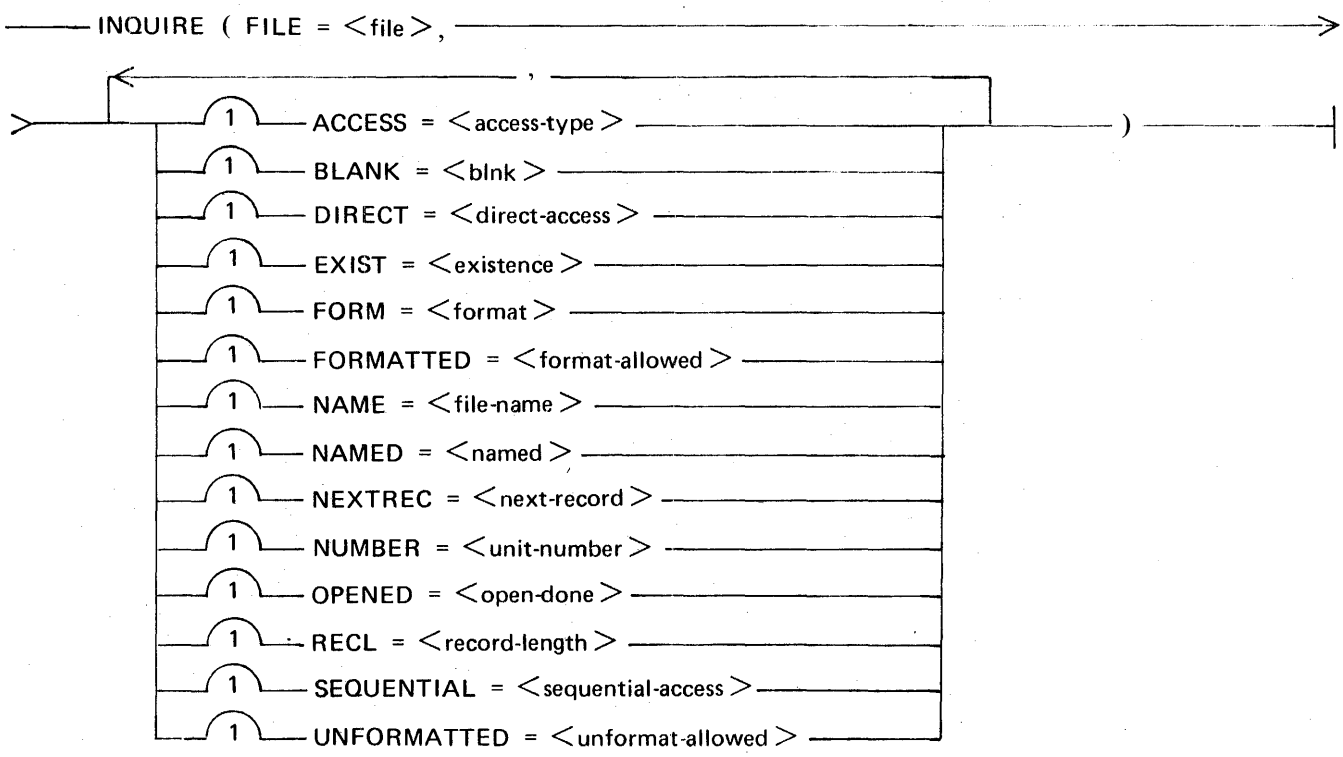
INQUIRE STATEMENT

The INQUIRE statement inquires about properties of a named file or of a file connected to a particular unit. The INQUIRE statement has two forms: inquire by file and inquire by unit. All value assignments are made according to the rules for assignment statements.

The INQUIRE statement can be executed before, while, or after a file is connected to a unit. All values assigned by the INQUIRE statement are current at the time the statement is executed.

INQUIRE by File Statement

The form of the INQUIRE by file statement follows:



The following paragraphs describe the file attributes and specifiers that can appear in an INQUIRE by file statement.

FILE = <file>

<file> is a character expression that specifies the name of the file on which an inquiry is being made. The named file need not exist nor be connected to a unit. The value of <file> must have one of the following forms:

<multi-file-id> / <file-id> ON <pack-name>

<pack-name> / <multi-file-id> / <file-id>

ACCESS = <access-type>

<access-type> is a character variable or character array element that is assigned the value SEQUENTIAL if the file is connected for sequential access and DIRECT if the file is connected for direct access. If there is no connection, <access-type> becomes undefined.

BLANK = <blnk>

<blnk> is a character variable or character array element that is assigned one of the following values: 1) NULL, if null blank control is in effect and the file is connected for formatted input/output, and 2) ZERO, if zero blank control is in effect and the file is connected for formatted input/output. If there is no connection or if the connection is not for formatted input/output, <blnk> becomes undefined.

DIRECT = <direct-access>

<direct-access> is a character variable or character array element that is assigned one of the following three values: 1) YES, if DIRECT is included in the set of allowed access methods for the file, 2) NO, if DIRECT is not included in the set of allowed access methods for the file, and 3) UNKNOWN, if the operating system is unable to determine whether or not DIRECT is included in the set of allowed access methods for the file.

EXIST = <existence>

<existence> is a logical variable or logical array element that is assigned the value TRUE if there exists a file with the specified name; otherwise, <existence> is assigned the value FALSE.

FORM = <format>

<format> is a character variable or character array element that is assigned the value FORMATTED if the file is connected for formatted input/output, and is assigned the value UNFORMATTED if the file is connected for unformatted input/output. If there is no connection, <format> becomes undefined.

FORMATTED = <format-allowed>

<format-allowed> is a character variable or character array that is assigned one of the following three values: 1) YES, if FORMATTED is included in the set of allowed forms for the file, 2) NO, if FORMATTED is not included in the set of allowed forms for the file, and 3) UNKNOWN, if the operating system is unable to determine whether or not FORMATTED is included in the set of allowed forms for the file.

NAME = <file-name>

<file-name> is a character variable or character array element that is assigned the value of the name of the file if the file has a name; otherwise, <file-name> becomes undefined. The value of <file-name> is not necessarily the same as the name given in the FILE specifier. For example, the processor can return a file name that is qualified by user identification. However, the value returned is suitable for use as the value of the FILE specifier in an OPEN statement.

NAMED = <named>

<named> is a logical variable or logical array element that is assigned the value TRUE if the file has a name; otherwise, it is assigned the value FALSE.

NEXTREC = <next-record>

<next-record> is an integer variable or integer array element that is assigned the value n+1, where n is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records have been read or written since the connection, <next-record> is assigned the value 1. If the file is not connected for direct access or if the position of the file is indeterminate because of a previous error condition, <next-record> becomes undefined.

NUMBER = <unit-number>

<unit-number> is an integer variable or integer array element that is assigned the value of the external unit identifier of the unit that is currently connected to the file. If there is no unit connected to the file, <unit-number> becomes undefined.

OPENED = <open-done>

<open-done> is a logical variable or logical array element that is assigned the value TRUE if the file specified is connected to a unit; otherwise, <open-done> is assigned the value FALSE.

RECL = <record-length>

<record-length> is an integer variable or integer array element that is assigned the value of the record length of the file connected for direct access. If the file is connected for formatted input/output, the length is the number of characters. If the file is connected for unformatted input/output, the length is measured in processor-dependent units. If there is no connection or if the connection is not for direct access, <record-length> becomes undefined.

SEQUENTIAL = <sequential-access>

<sequential-access> is a character variable or character array element that is assigned one of the following three values: 1) YES, if SEQUENTIAL is included in the set of allowed access methods for the file, 2) NO, if SEQUENTIAL is not included in the set of allowed access methods for the file, and 3) UNKNOWN, if the operating system is unable to determine whether or not SEQUENTIAL is included in the set of allowed access methods for the file.

UNFORMATTED = <unformat-allowed>

<unformat-allowed> is a character variable or character array element that is assigned one of the following three values: 1) YES, if UNFORMATTED is included in the set of allowed forms for the file, 2) NO, if UNFORMATTED is not included in the set of allowed forms for the file, and 3) UNKNOWN, if the operating system is unable to determine whether or not UNFORMATTED is included in the set of allowed forms for the file.

A variable or array element that is defined or undefined as a result of its use as a specifier in an INQUIRE statement, or any associated entity, cannot be referenced by any other specifier in the same INQUIRE statement.

Execution of an INQUIRE by file statement causes the specifier variables or array elements <named>, <file-name>, <sequential-access>, <direct-access>, <format-allowed>, and <unformat-allowed> to be assigned values only if the value of <file> is a valid file name and if there exists a file by that name; otherwise, they become undefined. <unit-number> is defined only if <open-done> is defined with the value TRUE. The specifier variables or array elements <access-type>, <format>, <record-length>, <next-record>, and <blnk> are defined only if <open-done> is defined with the value TRUE.

If an error condition occurs during execution of the INQUIRE statement, all of the inquiry specifier variables and array elements become undefined.

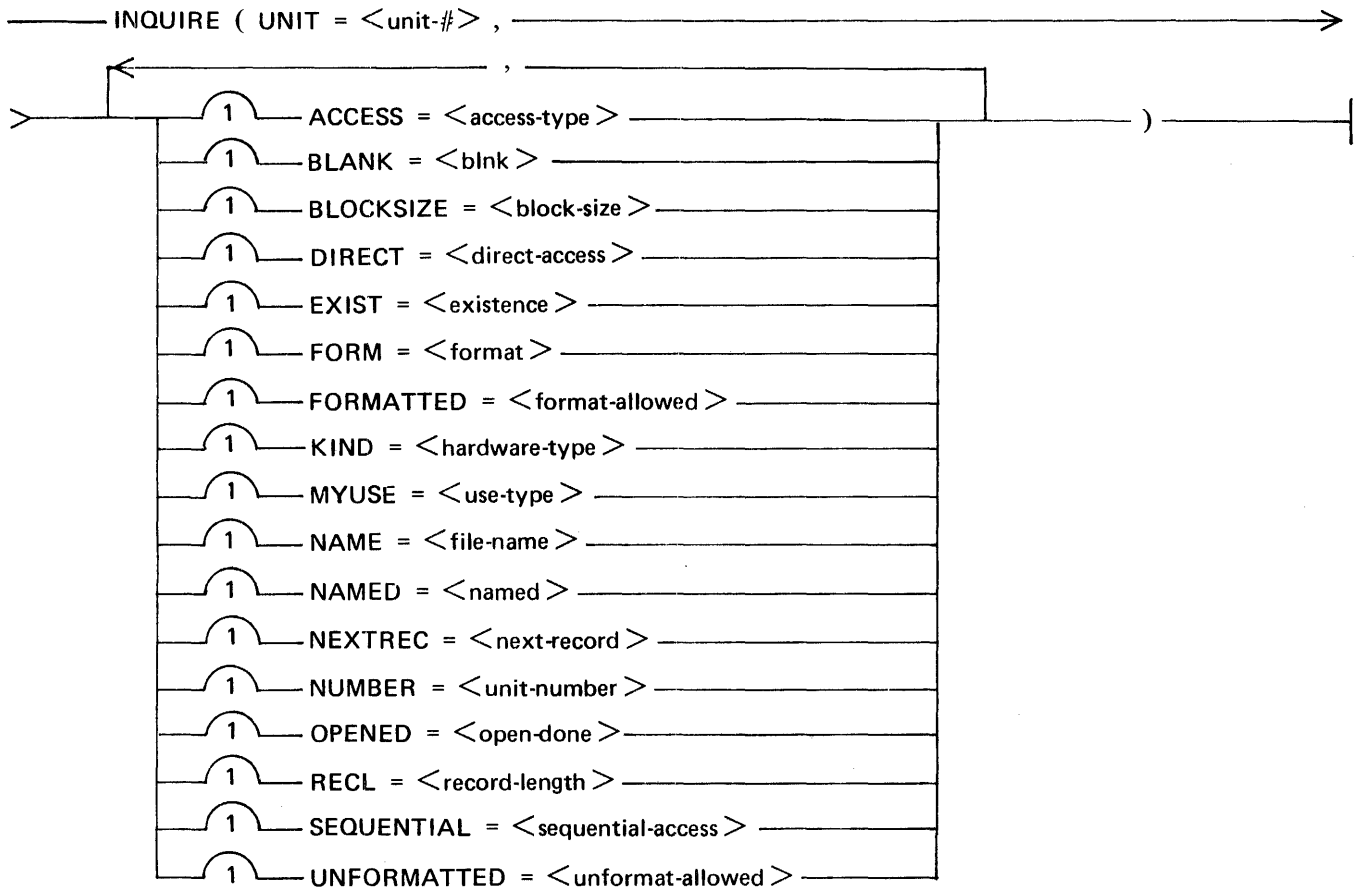
The specifier variables or array elements <existence> and <open-done> are always defined unless an error condition occurs.

Examples of INQUIRE by file statements follow:

```
INQUIRE (FILE = 'F77FILE', ACCESS = ACCTYP)
INQUIRE (FILE = 'TEST',RECL = IREC,  NEXTREC = NEXREC)
INQUIRE (FILE = FNAME, BLANK = INQ(1),DIRECT = INQ(2), FORM = INQ(3))
INQUIRE (FILE = 'A/B ON C',NAME=FNAME, NUMBER=1UNIT,UNFORMATTED=UFORM)
```

INQUIRE by Unit Statement

The form of the INQUIRE by unit statement follows:



The following paragraphs describe the file attributes and specifiers that can appear in an INQUIRE by unit statement.

UNIT = <unit-#>

<unit-#> is either an external unit identifier or an internal file identifier. An external unit identifier refers to an external file and is an integer expression whose value is positive or zero. An internal file identifier refers to an internal file and is the name of a character variable, character array, character array element, or character substring.

ACCESS = <access-type>

<access-type> is a character variable or character array element that is assigned the value SEQUENTIAL if the file is connected for sequential access and DIRECT if the file is connected for direct access. If there is no connection, <access-type> becomes undefined.

BLANK = <blnk>

<blnk> is a character variable or character array element that is assigned the value NULL if null blank control is in effect and the file is connected for formatted input/output, and is assigned the value ZERO if zero blank control is in effect and the file is connected for formatted input/output. If there is no connection or if the connection is not for formatted input/output, <blnk> becomes undefined.

BLOCKSIZE = <block-size>

<block-size> is an integer variable or integer array element that is assigned the value of the block size of the file.

DIRECT = <direct-access>

<direct-access> is a character variable or character array element that is assigned one of the following three values: 1) YES, if DIRECT is included in the set of allowed access methods for the file, 2) NO, if DIRECT is not included in the set of allowed access methods for the file, and 3) UNKNOWN, if the operating system is unable to determine whether or not DIRECT is included in the set of allowed access methods for the file.

EXIST = <existence>

<existence> is a logical variable or logical array element that is assigned the value TRUE if the specified unit exists; otherwise, <existence> is assigned the value FALSE.

FORM = <format>

<format> is a character variable or character array element that is assigned the value FORMATTED if the file is connected for formatted input/output and is assigned the value UNFORMATTED if the file is connected for unformatted input/output. If there is no connection, <format> becomes undefined.

FORMATTED = <format-allowed>

<format-allowed> is a character variable or character array that is assigned one of the following three values: 1) YES, if FORMATTED is included in the set of allowed forms for the file, 2) NO, if FORMATTED is not included in the set of allowed forms for the file, and 3) UNKNOWN, if the operating system is unable to determine whether or not FORMATTED is included in the set of allowed forms for the file.

KIND = <hardware-type>

<hardware-type> is a character variable or character array element that is assigned the name of the hardware device to which the file is connected. The following are the possible values:

DISK
PRINTER
PUNCH
ODT
READER
REMOTE
TAPE

MYUSE = <use-type>

<use-type> is a character variable or character array element that is assigned the value IN if only input is allowed for the file, OUT if only output is allowed, and IO if both input and output are allowed.

NAME = <file-name>

<file-name> is a character variable or character array element that is assigned the value of the name of the file if the file has a name; otherwise, it becomes undefined. The value of <file-name> is not necessarily the same as the name given in the FILE specifier. For example, the operating system can return a file name that is qualified by user identification. However, the value returned is suitable for use as the value of the FILE specifier in an OPEN statement.

NAMED = <named>

<named> is a logical variable or logical array element that is assigned the value TRUE if the file has a name; otherwise, it is assigned the value FALSE.

NEXTREC = <next-record>

<next-record> is an integer variable or integer array element that is assigned the value n + 1, where n is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records have been read or written since the connection, <next-record> is assigned the value 1. If the file is not connected for direct access or if the position of the file is indeterminate because of a previous error condition, <next-record> becomes undefined.

NUMBER = <unit-number>

<unit-number> is an integer variable or integer array element that is assigned the value of the external unit identifier of the unit that is currently connected to the file. If there is no unit connected to the file, <unit-number> becomes undefined.

OPENED = <open-done>

<open-done> is a logical variable or logical array element that is assigned the value TRUE if the specified unit is connected to a file; otherwise, <open-done> is assigned the value FALSE.

RECL = <record-length>

<record-length> is an integer variable or integer array element that is assigned the value of the record length of the file connected for direct access. If the file is connected for formatted input/output, the length is the number of characters. If the file is connected for unformatted input/output, the length is measured in processor-dependent units. If there is no connection or if the connection is not for direct access, <record-length> becomes undefined.

SEQUENTIAL = <sequential-access>

<sequential-access> is a character variable or character array element that is assigned one of the following three values: 1) YES, if SEQUENTIAL is included in the set of allowed access methods for the file, 2) NO, if SEQUENTIAL is not included in the set of allowed access methods for the file, and 3) UNKNOWN, if the operating system is unable to determine whether or not SEQUENTIAL is included in the set of allowed access methods for the file.

UNFORMATTED = <unformat-allowed>

<unformat-allowed> is a character variable or character array element that is assigned one of the following three values: 1) YES, if UNFORMATTED is included in the set of allowed forms for the file, 2) NO, if UNFORMATTED is not included in the set of allowed forms for the file, and 3) UNKNOWN, if the operating system is unable to determine whether or not UNFORMATTED is included in the set of allowed forms for the file.

A variable or array element that is defined or undefined as a result of its use as a specifier in an INQUIRE statement, or any associated entity, cannot be referenced by any other specifier in the same INQUIRE statement.

Execution of an INQUIRE by unit statement causes the specifier variables or array elements <unit-number>, <named>, <file-name>, <access-type>, <sequential>, <direct-access>, <format>, <format-allowed>, <unformat-allowed>, <record-length>, <next-record>, and <blnk> to be assigned values only if the specified unit exists and if a file is connected to the unit; otherwise, they become undefined.

If an error condition occurs during execution of the INQUIRE statement, all of the inquiry specifier variables and array elements become undefined.

The specifier variables or array elements <existence> and <open-done> are always defined unless an error condition occurs.

The unit specified need not exist or be connected to a file. If it is connected to a file, the inquiry operation refers to the connection and to the file connected.

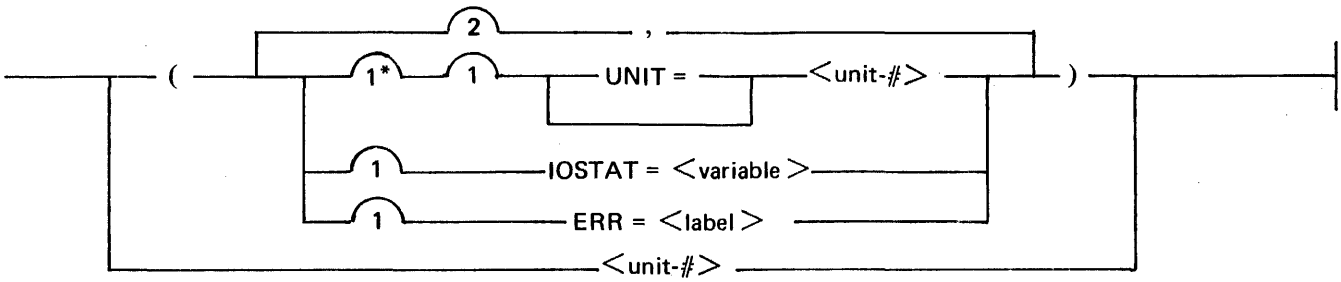
Examples of INQUIRE by unit statements follow:

```
INQUIRE (UNIT=3,NAME=FNAME)
INQUIRE (UNIT=7,NEXTREC=NREC,BLOCKSIZE=IBLK,RECL=ILEN)
INQUIRE (UNIT=IUNIT,FORM=INQ(1),FORMATTED=INQ(2),SEQUENTIAL=INQ(3))
INQUIRE (UNIT=10,KIND=HTYPE,MYUSE=USE,BLANK=BLNK)
```

CONTROL LIST FOR FILE POSITIONING STATEMENTS

There are four statements that control the positioning of a file: 1) BACKSPACE, 2) ENDFILE, 3) REWIND, and 4) FIND. Associated with the file positioning statements BACKSPACE, ENDFILE, and REWIND is a list of specifiers that specify which unit is to be accessed, the status of the operation, and where execution continues in case of an error. This section describes these specifiers.

The format of the control list for the BACKSPACE, ENDFILE, and REWIND file positioning statements follows:



G50335

<unit-#> is an integer expression giving the unit number of the file as declared in a FILE statement. Only one unit number can be specified in a file positioning statement. If UNIT= is omitted, <unit-#> must be the first item in the list; otherwise, the specifiers can occur in any order.

Execution of a file positioning statement containing IOSTAT= <variable> causes <variable> to receive an integer value depending on the outcome of the file positioning operation. The possible values of <variable> follow:

Condition	Value
No error	0
Error	2

If a file positioning statement contains an ERR= specifier and an error occurs, the operation is terminated and execution continues with the statement labeled <label>, which must be an executable statement that appears in the same program unit as the ERR= specifier.

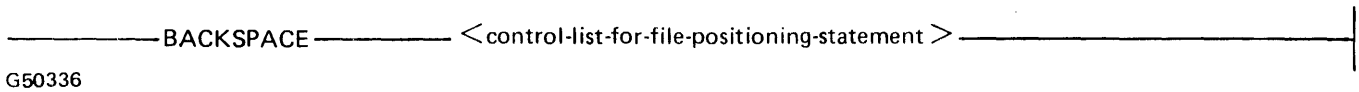
The control list must contain exactly one external unit specifier and can contain, at most, one of each of the other specifiers.

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

The specifiers, as described in this subsection, are essentially the same as those described under Control List in this section and are re-explained to relate them specifically to file positioning statements.

BACKSPACE STATEMENT

The BACKSPACE statement backspaces the specified file one record. The proper form of the BACKSPACE statement follows:



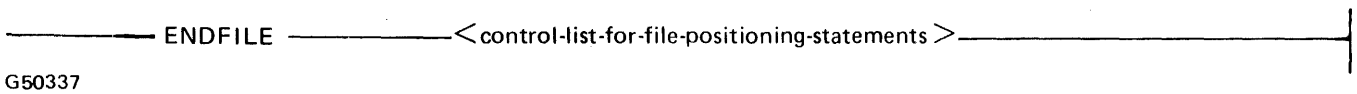
G50336

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

Backspacing a file that is connected, but does not exist, is prohibited.

ENDFILE STATEMENT

The ENDFILE statement writes an endfile record to the specified file. The proper form of the ENDFILE statement follows:



G50337

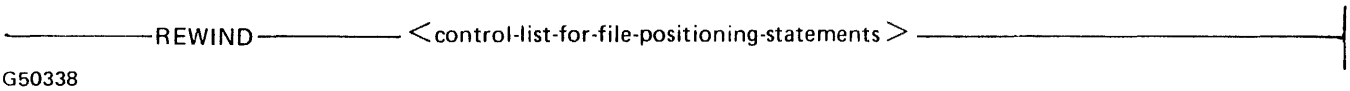
Execution of an ENDFILE statement writes an endfile record as the next record of the file. The file pointer is then positioned after the endfile record. If the file is a direct-access file, only those records before the endfile record can be read through subsequent direct access to the file.

Before subsequent I/O operations can be performed on a file with an endfile, a BACKSPACE or REWIND statement must be used to reposition the file to a point where access can occur.

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

REWIND STATEMENT

The REWIND statement causes the specified file to be repositioned to the initial point. The proper form of the REWIND statement follows:



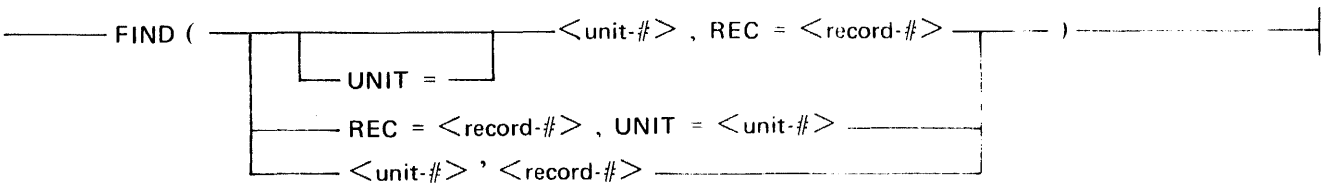
G50338

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

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

FIND STATEMENT

The FIND statement positions a direct access file at a specified record. The form of the FIND statement follows:



<unit-#> is an integer expression giving the unit number of the file on which the FIND operation is to be performed. <record-#> is an integer expression giving the number of the record at which the file is to be positioned.

Execution of a FIND statement positions the designated file immediately before the specified record.

Examples of FIND statements follow:

```
FIND (3,REC=1024)
FIND (REC=10000,UNIT=9)
FIND (2'1500)
FIND (IUNIT,REC=IREC)
FIND (IFILE+1'IPART*2-1)
```

INTERNAL FILES

Internal files permit the transfer and conversion of data from internal storage to internal storage. An internal file is a character variable, character array, character array element, or character substring. An internal file specifier (replacing the UNIT= <unit-#> in the control list) is the symbolic name of a character variable, character array, character array element, or character substring. No FILE statement is associated with an internal file.

A record of an internal file is a character variable, character array element, or character substring. An internal file that is a character variable, character array element, or character substring contains only one record. If the file is a character array, it is treated as a sequence of character array elements. Each array element is a record of the file. The ordering of the records of the file is the same as the ordering of the array elements in the array (refer to appendix D). Every record of the file has the same length, which is the length of an array element in the array.

A character storage location in an internal file becomes defined with a value when a WRITE to the record containing that storage location is performed; when the storage location is contained in the input list of a READ statement accessing another file; or when a character assignment is performed on a character entity (character variable, character array element, or character substring) that contains the character storage location. A READ that accesses an internal file can only be performed on records whose character storage locations are all defined.

Only sequential access formatted input/output statements are permitted with internal files. After an input/output statement accesses an internal file, the file pointer is repositioned at the beginning of the file. Slash (/) characters (refer to section 12) can be contained in the format of an input/output statement that accesses an internal file, in which case, the file pointer is positioned at the beginning of the next record and the input/output statement continues. If a WRITE statement accesses only part of a record because slash (/) characters occur in the format or because the end of the input/output statement occurs before the end of a record, then the remainder of the record is filled with blanks.

Examples of input/output statements that reference internal files follow:

```
READ (A,100) B,C,D
READ (A,'(I4///I4)') E,F
WRITE (A,200) H,(I(J),J = 1,14)
READ (A(3)(2:5),300) K
```

The first and second READ statements cause array A to be read as though it were a file. The character values are converted from EBCDIC to the types specified by the format associated with the READ. The second READ statement converts the first four character locations of the first element of A to an integer and assigns the integer to variable E. The first four characters of the fourth element are converted to an integer and assigned to variable F.

The third statement, a WRITE statement, demonstrates the manner in which data can be written to an internal file. The variables in the output list are converted to EBCDIC representation using the associated format and assigned to an element or elements of array A.

The fourth statement, a READ statement, shows how an array element substring can be read as a file. For more information on data assignment in input/output statements, refer to section 12.

UNFORMATTED I/O

Data can also be transferred between a file and entities specified within the program by means of unformatted I/O. Unformatted I/O statements take the form of the READ and WRITE statements previously described in this section. A READ or WRITE statement is unformatted if there is no format specifier; otherwise, it is a formatted READ or WRITE statement. There is no editing or conversion of transferred data associated with unformatted I/O. Refer to appendix C, Description of Unformatted I/O Records, for additional information.

Execution of an unformatted READ statement fetches one record from the file indicated by the unit number. If an I/O variable list is specified as part of the statement, data is transferred to the specified locations. Transfer occurs as full storage units, and the record accessed should have been generated by unformatted WRITE statements. If no I/O variable list is specified on an unformatted READ, one record is skipped in the file indicated by the unit number.

If the I/O variable list for an unformatted direct-access READ specifies more data to be transferred than is present in the record, an error occurs. Refer to ERR in this section. The I/O variable list for an unformatted sequential READ can transfer more than one record.

Execution of an unformatted WRITE statement writes one or more records to the file indicated by the unit number. The mandatory I/O variable list denotes the sequence of values to be contained in the record. The contents of the indicated storage locations are placed unchanged in the generated record as full storage units intended to be read by the unformatted READ statement.

The unformatted WRITE statement depends on the declared or default size of records within the affected file. When an unformatted direct-access WRITE statement attempts to transfer more values than can be contained in one record, the program is terminated unless the statement contains an ERR action specifier. An unformatted sequential WRITE can transfer more than one record.

LIST-DIRECTED I/O

An asterisk (*) character used as a format specifier indicates that list-directed I/O is to be performed. List-directed I/O is described in more detail in section 12. Examples follow:

```
READ (5,*)
PRINT *
WRITE (8,FMT=*)
```

NAMELIST I/O

A namelist name used as a format specifier indicates that namelist I/O is to be performed. Namelist I/O is described in more detail in section 12. Examples follow:

```
READ (5,NLNAME)
WRITE (UNIT=9,FMT=NLN)
```

SECTION 12
FORMAT SPECIFICATIONS

A format specification is used in conjunction with a formatted input/output statement to determine the editing to be performed between the internal representation and the characters of a record or sequence of records in a file. A format specification is used with an input statement to determine the manner in which the characters in the input record are to be converted and what characters of the input record to use. On output, the format specification is used to determine how the internal data items are to be converted and placed in the record(s).

FORMAT SPECIFICATION METHODS

There are two methods for specifying a format in an input/output statement control list. With the first method, an input/output statement <control-list> can contain a label specifying that the format specification is given in a FORMAT statement with the given label. With the second method, a <control-list> can specify a character array name, character variable, or other character expression which contains the format specification as (part of) the value of the specified character entity. The leftmost character positions of the specified entity must constitute a format specification when the statement is executed.

FORMAT Statement

The following is the proper form of the FORMAT statement:

G50339

A FORMAT statement is preceded by a label in columns 1 through 5 of the statement. This is the label that is specified when referring to the FORMAT statement in the control list of an input/output statement.

Character Format Specification

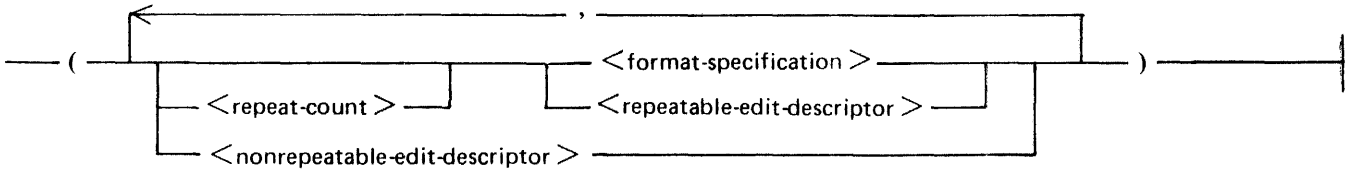
When a character array, character variable, or other character expression (not involving the concatenation of a character variable with an assumed length) appears as the format identifier in an input/output <control-list>, the leftmost character storage locations of the character entity specified must be in a defined state (contain character values) with character data that constitute a format specification when the input/output statement is executed.

A character format specification must be of the form described under Form of a Format Specification in this section. The format specification must begin with a left parenthesis and end with a right parenthesis. The format specification can optionally be preceded by blanks and the storage locations within the character entity that follow the format specification can contain other data without affecting the format specification.

If a format identifier is the symbolic name of a character array, the format specification can extend beyond the end of the first element of the array. The format identifier is considered to be the concatenation of all elements of the array in the order in which the array is stored (refer to appendix D). However, if the format identifier is a character array element, the format specification must be contained entirely within the specified element.

FORM OF A FORMAT SPECIFICATION

The following is the proper form of a format specification:



G50340

As shown in this diagram, a format specification can contain another <format-specification>. A <repeat-count> can precede a <repeatable-edit-descriptor> or a nested <format-specification> and is the same as expressing the modified item n times in succession, where n is the number replacing <repeat-count> in the diagram. <repeatable-edit-descriptor> specifies how to edit the data being transferred. <nonrepeatable-edit-descriptor> specifies a special editing function as described in Nonrepeatable Edit Descriptors in this section.

The comma used to separate edit descriptors can be omitted:

1. Between a P edit descriptor and an immediately following F, E, D, or G edit descriptor.
2. Before or after a slash edit descriptor.
3. Before or after a colon edit descriptor.

INTERACTION BETWEEN INPUT/OUTPUT LIST AND FORMAT

The transfer of data between internal storage and a record of a file in a formatted input/output statement is dependent on two factors:

1. The next edit descriptor contained in the corresponding format specification.
2. The next item in the input/output list, if one exists.

For each item in an input/output list there must be a corresponding repeatable edit descriptor in the format specification (or two F, E, D, or G edit descriptors if the item in the input/output list is of type COMPLEX).

When a formatted input/output statement occurs, the corresponding format specification is accessed and processed from left to right, except where a repeat specifier occurs beginning at the first left parenthesis. When a nonrepeatable edit descriptor is encountered, the appropriate action is taken as specified by that descriptor. When a repeatable edit descriptor is encountered, a search is made for an input/output list item, and if one exists, appropriately edited information is transferred between the input/output list item and the record of the file. Processing of the format specification continues from this point. If a repeatable edit descriptor is encountered and the input/output list has been exhausted, the input/output statement is terminated.

An embedded format specification or repeatable edit descriptor preceded by a repeat specification is processed as a list of n (where n is the repeat count) format specifications or repeatable edit descriptors identical to the format specification or edit descriptor without the repeat specification. An omitted repeat specification is treated the same as a repeat specification whose value is 1.

If format control encounters the rightmost parenthesis of a complete format specification and another input/output list item is not specified, format control terminates. However, if another input/output list item is specified, the file pointer is positioned at the beginning of the next record and format control reverts to the beginning of the format specification terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format specification. If such reversion occurs, the reused portion of the format specification must contain at least one repeatable edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat specification, the repeat specification is reused.

If an input/output list contains at least one list item, at least one repeatable edit descriptor must exist in the format specification. A format specification of ( ), or a format specification containing only nonrepeatable edit descriptors, can only be used with an input/output statement that does not contain an input/output list. If a format specification of ( ) is given, a record containing no characters is written or one input record is skipped.

During processing of a format specification, if a colon edit descriptor is encountered and the corresponding input/output list has been exhausted, the input/output statement is terminated.

EDIT DESCRIPTORS

Edit descriptors are used to specify the form of a record and direct the editing between the characters in a record and internal representations of data. The internal representation of a datum corresponds to the internal representation of a constant of the corresponding type (refer to appendix D).

A field is a part of a record that is read on input or written on output when one I, F, E, D, G, L, A, Z, H, apostrophe, or quote edit descriptor is processed. The field width is the size in characters of the field.

Edit descriptors are divided into two categories: repeatable and nonrepeatable.

The following is a list of repeatable edit descriptors available in FORTRAN 77:

Repeatable Edit Descriptor	Meaning
Iw	For integer data.
Iw.m	For integer data.
Ew.d	For real, double-precision, or complex data.
Ew.dEe	For real, double-precision, or complex data.
Fw.d	For real, double-precision, or complex data.
Dw.d	For real, double-precision, or complex data.
Gw.d	For real, double-precision, or complex data.
Gw.dEe	For real, double-precision, or complex data.
Lw	For logical data.
A	For character data.
Aw	For character data.
Zw	For any numeric or logical data.

The capital letters A, D, E, F, G, I, L, and Z refer to the type of edit descriptor. The lower-case letters w, d, e, and m are nonzero unsigned integer constants, where w is the field width, d is the number of significant digits to the right of the decimal point, e is the number of digits in the exponent, and m is the number of significant digits in the integer data item.

The following is a list of nonrepeatable edit descriptors:

Nonrepeatable Edit Descriptors	Meaning
'h1h2...hn'	Apostrophe editing.
"h1h2...hn"	Quote editing.
nHh1h2...hn	Hollerith editing.
Tc	Tab editing.
TLc	Tab editing.
TRc	Tab editing.
X	Tab editing.
nX	Tab editing.
/	Slash editing.
:	Colon editing.
S	Sign control.
SP	Sign control.
SS	Sign control.
kP	Scale factor.
BN	Blank control.
BZ	Blank control.

H, T, TL, TR, X, /, :, S, SP, SS, P, BN, BZ, ', and " specify a type of editing. The figures h1h2...hn represent characters representable by the processor, c is a nonzero unsigned integer constant, and k is an optionally signed integer constant giving the scale factor.

Repeatable Edit Descriptors

Repeatable edit descriptors determine the manner in which items in the input/output list are to be edited and transmitted to or from a file. Each item in an input/output list is associated with a repeatable edit descriptor in the corresponding format specification.

Table 12-1 summarizes the data item types that can be read using each type of repeatable edit descriptor. The letter A. indicates that the operation is allowed; the letters NA indicate the operation is not allowed.

Table 12-1. Input Data Item Types

Data Item Type	Format Specification							
	I	**F**	**E**	**G**	**D**	**A**	**L**	**Z**
INTEGER	A	A	A	A	A	A	NA	NA
REAL	NA	A	A	A	A	A	NA	NA
DOUBLE	NA	A	A	A	A	A	NA	NA
LOGICAL	NA	NA	NA	A	NA	A	A	NA
CHARACTER	NA	NA	NA	NA	NA	A	NA	NA
Hex	NA	NA	NA	NA	NA	A	NA	A

Table 12-2 summarizes which repeatable edit descriptors can be associated with each type of item in an input list. The letter A indicates that the operation is allowed; the letters NA indicate that the operation is not allowed.

Table 12-2. Input Variable Item Types

Variable Type	Format Specifications							
	I	**F**	**E**	**G**	**D**	**A**	**L**	**Z**
INTEGER	A	NA	NA	A	NA	NA	NA	A
REAL	NA	A	A	A	A	NA	NA	A
DOUBLE	NA	A	A	A	A	NA	NA	A
COMPLEX	NA	A	A	A	A	NA	NA	A
LOGICAL	NA	NA	NA	A	NA	NA	A	A
CHARACTER	NA	NA	NA	NA	NA	A	NA	NA

Table 12-3 summarizes which repeatable edit descriptors can be associated with each type of item in an output list (arithmetic expressions, variables, and so forth). The letter A indicates that the operation is allowed; the letters NA indicate that the operation is not allowed.

Table 12-3. Output List Item Types

List Item Type	Format Specification							
	I	**F**	**E**	**G**	**D**	**A**	**L**	**Z**
INTEGER	A	NA	NA	A	NA	NA	NA	A
REAL	NA	A	A	A	A	NA	NA	A
DOUBLE	NA	A	A	A	A	NA	NA	A
COMPLEX	NA	A	A	A	A	NA	NA	A
LOGICAL	NA	NA	NA	A	NA	NA	A	A
CHARACTER	NA	NA	NA	NA	NA	A	NA	NA

On input, if an exponent is used in the data item, it can have one of the following forms:

E[±] <integer-constant>
D[±] <integer-constant>
± <integer-constant>

If the exponent is preceded by an E or a D and is positive, the + sign is optional.

For numeric input, leading blanks are ignored but are counted in the length of the field w. The interpretation of blanks, other than leading blanks, is determined by a combination of any BLANK= specifier in the FILE declaration and any BN or BZ blank control that is currently in effect for the unit. BN and BZ are described later in this section. A field of all blanks is considered to be zero.

Format Specification I

The format specification I is used to transmit data as integer values between internal storage and a file (either external or internal). The I format specification has two forms: Iw and Iw.m. The .m portion of the second form has no effect on input.

Input Using Iw

On input, the integer format specification Iw causes the value of the integer data item in the input field to be assigned to the corresponding integer variable in the input list. The integer data item must be in the form of an optionally signed integer constant.

The magnitude of the value in the input field must not exceed the maximum magnitude permitted for an integer data item. Refer to Integer Constants in section 4 for additional information.

Examples:

Variable Type	Data Item	Specification	Internal Value
INTEGER	567	I3	+567
INTEGER	27	I2	+27
INTEGER	−11234	I6	−11234

Output Using Iw and Iw.m

On output, the integer format specification Iw causes the corresponding integer output list item to be written to the specified output file.

The field width (w) specifies the number of positions of the record that the list item is to occupy. Specifying Iw causes the integer number to be placed right-justified in the output field over a field of blanks. Specifying Iw.m causes at least m digits, including leading zeros if necessary, to be output. If .m is not specified, at least one digit is output.

Unless otherwise specified, the plus sign is omitted for positive numbers. If a value of the integer quantity to be written requires more than w digits, or if w cannot accommodate both the sign position and the value in the case of a negative quantity, the output field is filled with asterisks (*).

Examples:

List Item Type	List Item Value	Specification	Output Field
INTEGER	−79	I4	b−79
INTEGER	0	I3	bb0
INTEGER	−37216	I5	*****
INTEGER	+22	I4.3	b022
INTEGER	+2361	I4.2	2361

Examples of statements using I format follow:

```
      READ (5,100) I,J,K
      WRITE (6,200) I,J,K
100   FORMAT (3I3)
200   FORMAT (I3,I2,I5)
```

Assume the input record contains the following (BLANK = ZERO is in effect for the file and b denotes a blank):

1b23b456bbb789b10

The READ statement from this example assigns the following values to variables I, J, and K:

I = 102
J = 304
K = 560

Format Specification F

The F format specification is used to transfer real, double-precision, and complex values between internal storage and a file.

Input Using Fw.d

On input, the format specification Fw.d causes the value of the data item in the input field to be assigned to the corresponding real, double-precision, or complex variable in the input list. The data item must be in the form of an integer, real, or double-precision constant (refer to section 4 for the forms of constants).

The field width (w) specifies the number of positions that the input item is to occupy, including the decimal point and the exponent, if present, and the decimal digits. The input data item can contain more digits than FORTRAN 77 uses to approximate the value of the constant. If a decimal point appears in the input field, the actual decimal location in the input value overrides the decimal point placement specified by d. If there is no decimal point in the input field, a decimal point is assumed d places from either the right side of the input field or from the E, D, or signed integer constant denoting the exponent.

On input, the real format specifiers Fw.d, Ew.d, Gw.d, Dw.d, Ew.dEe, and Gw.dEe function in the same manner.

Examples:

Variable Type	Data Item	Specification	Internal Value
REAL	3.672593	F8.4	+3.672593
REAL	36725931	F8.4	+3672.593
REAL	−3672.E02	F8.4	−367200.
REAL	−3672+02	F8.4	−36.72
DOUBLE PRECISION	367259D−10	F10.4	+.00000000367259

Output Using Fw.d

On output, the format specification Fw.d causes the corresponding real, double-precision, or complex output list item to be written without an exponent on the specified output file.

The real number is placed, right-justified and rounded to d decimal places, in the output field superimposed over a field of blanks. The plus sign is optional for positive numbers (dependent upon the sign control in effect, described in this section). On output, the field width w must include enough positions to accommodate d decimal places, a decimal point, and the integral part of the value. The position for the sign is included in the field width w.

If the magnitude of the number exceeds the specified field width (w), the output field is filled with asterisk (*) characters.

Examples:

List Item Type	List Item Value	Specification	Output Field
REAL	+36.7929	F7.3	b36.793
REAL	+36.7934	F9.3	bbb36.793
REAL	−0.0316	F6.3	b−.032
REAL	0.0	F6.4	b.0000
DOUBLE PRECISION	37624.816952	F8.3	********
REAL	+579.645	F6.2	579.65
REAL	−579.645	F6.2	******
REAL	−0.895	F5.2	b−.90

Examples of statements using F format follow:

```
CHARACTER *12 A
READ (*,'(2F6.2,F3.2)') (X(I),I=1,3)
WRITE (A,'(3F4.1)') X(1), X(2), X(3)
```

Assume the input record contains the following (b denotes blank).

```
21E−0360.215b12
```

The variables would take on the following values:

```
X(1)= .21E−03 = .00021
X(2)= 60.215
X(3)= .12
```

The value transmitted to the character variable A, which is used as an internal file, would be the following:

```
A='     .060.2****'
```

Format Specification E

The E format specification is used to transfer real, double-precision, or complex values between a file and internal storage. The Ew.d, Dw.d, and Ew.dEe edit descriptors indicate that the external field occupies w positions, the fractional part of which consists of d digits (unless a scale factor greater than one is in effect), and the exponent part consists of e digits. The e has no effect on input.

Input Using Ew.d

On input, the format specifiers Ew.d, Fw.d, Gw.d, and Dw.d function in the same manner.

Output Using Ew.d

On output, the format specification Ew.d causes the corresponding real, double-precision, or complex output list item to be written with an exponent on the specified output file. The number is normalized (most significant digit placed immediately to the right of the decimal point) by multiplying the number by 10**e. The e becomes the exponent which is output with the number if the scale factor is zero.

The real number is placed right-justified and rounded to d digits, together with a 4-place exponent field, in the output field over a field of blanks. Since no significant digits are written to the left of the decimal point in the output field, d has a different interpretation with the Ew.d format. The position for the minus sign required for negative numbers is included in the field width w. For positive numbers, w has the value d + 5 + (the number of leading blanks desired). For negative numbers, w has the value d + 6 + (the number of leading blanks desired, and an optional plus sign). The decimal point must be counted when determining the field width.

Examples:

List Item Type	List Item Value	Specification	Output Field
REAL	+36.7929	E12.5	bb.36793E+02
REAL	−36.7929	E11.5	−.36793E+02
REAL	−36.7929	E10.5	**********
REAL	22.323	E8.1E3	b.2E+001
DOUBLE PRECISION	872568.394816897	E12.7	.8725684E+06

Examples of statements using E format follow:

```
      READ (*,100) X,Y,Z
      PRINT 100, X,Y,Z
100   FORMAT (E1.1,E9.3,E10.3E3)
```

Assume the input record contains the following (b denotes blank).

1b123bE−02b12.3bE+02

The variables would be assigned the following values:

X = .1
Y = .123E−02 = .00123
Z = 12.3E+02 = 1230

The values output to the file (line printer by default) would be the following:

* .123E−02 .123E+004

Format Specification D

The format specification Dw.d is identical to Ew.d, except that the exponent part of the output contains a D rather than an E.

Format Specification G

The G format specification is a multi-purpose format descriptor which can be used with variables of type INTEGER, REAL, DOUBLE PRECISION, or LOGICAL.

Input Using Gw.d and Gw.dEe

On input, the G format specification is interpreted as an I, F, E, D, or L format descriptor, depending upon the type of the variable in the input list. The Ee portion of the Gw.dEe form is ignored on input.

If the input variable is of type INTEGER or LOGICAL, the Gw.d format specification functions in the same manner as the Iw or Lw specification, respectively. The .d portion of the general form is still required, but is ignored.

If the input variable is REAL or DOUBLE PRECISION, the Gw.d and Gw.dEe format specifications functions in the same manner as the Fw.d, Ew.d, Ew.dEe, and Dw.d.

Examples:

Variable Type	Data Item	Specification	Internal Value
REAL	529.4	G5.1	+529.4
LOGICAL	T	G1	.TRUE.
INTEGER	45	G2	+45
REAL	−6.1E+04	G8.1	−61000.
DOUBLE PRECISION	5.3294D+02	G10.4	+532.94

Output Using Gw.d and Gw.dEe

On output, the general format specifications Gw.d and Gw.dEe cause the corresponding output list item to be written to the specified output file. The data item type is determined by the type of the output list item and can be either INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL.

The G format specification is interpreted as an I, F, E, D, or L format specification, depending upon the magnitude and the type of the output list item.

If the output list item is of type INTEGER or LOGICAL, the Gw.d format specification functions in the same manner as the Iw or Lw specification, respectively. The .d portion of the general form is still required, but is ignored.

If the output list item is REAL, DOUBLE PRECISION, or COMPLEX, the Gw.d and Gw.dEe specifications produce either an F, E, or D format representation according to the following criteria (N is the absolute value of the list item, n is 4 for Gw.d and e+2 for Gw.dEe):

If 0.1 <= N < 1 output format is F(w−n).d, nX

If 1 <= N < 10 output format is F(w−n).(d−1), nX

.
.
.

If 10**(d−2) <= N < 10**(d−1) output format is F(w−n).1, nX

If 10**(d−1) <= N < 10**d output format is F(w−n).0, nX

If none of the above conditions apply, the output format is Ew.d, Ew.dEe, or Dw.d depending on the type of the output list item and whether Gw.d or Gw.dEe is used as the format specification.

For example, if 5.7319 is the value represented internally and G10.3 is the format specified, the resulting format would be F6.2, 4X, which would output bb5.73bbbb.

If the format specified for the value 5731.9 were G10.3, the resulting format would be E10.3, with the corresponding output bb.573E+04. Since 5731 is greater than 10**3 (1000), the specification would produce an E-format representation.

On output of REAL or DOUBLE PRECISION, the field width (w) must include enough positions to accommodate an exponent, a decimal point, and a sign position if the quantity is negative, or a sign is specified in the format. Refer to Sign Control in this section for additional information.

Examples:

List Item Type	List Item Value	Specification	Output Field
REAL	+10.	G12.5=F8.3	bb10.000bbbb
REAL	+100000.	G12.5=E12.5	bb.10000E+06
INTEGER	−10	G5.0=I5	bb−10
LOGICAL	.TRUE.	G4.0=L4	bbbT
DOUBLE PRECISION	+123467890123.	G20.13=F16.0	bb123467890123.bbbb
REAL	+101010.	G10.5=E10.5	.10101E+06

Examples of statements using the G format follow:

```
      LOGICAL LOG
      READ (3,100) X, LOG, INTEG
100   FORMAT (3G9.3)
      WRITE (*,'(G9.2,G2.0,G10.0)') X, LOG, INTEG
```

Assume the input record contains the following (b denotes a blank):

−1234E−03b.TFILLERb12345678

The values of the variables would be:

X = −1.234E−03 = −.001234
LOG = .TRUE.
INTEG = 12345678

The values transmitted to file 6 would be:

−0.12E−02bTbb12345678

Complex Editing

A complex data item consists of a pair of separate real data items; therefore, the editing is specified by two successively interpreted F, E, D, or G edit descriptors. The first of the edit descriptors specifies the real part; the second specifies the imaginary part. The two edit descriptors need not be the same edit descriptor used twice. Nonrepeatable edit descriptors can appear between the two successive F, E, D, or G edit descriptors.

Format Specification L

The format specification L is used to transfer logical data items between internal storage and a file.

Input Using Lw

On input, Lw causes the value of the LOGICAL data item in the input field to be assigned to the corresponding variable or array element of type LOGICAL in the input list. The input field width (w) must be greater than or equal to 1. The input field consists of optional blanks, optionally followed by a decimal point, followed by a T or an F. The T or F can be followed by additional characters in the field. The logical constants .TRUE. and .FALSE. are acceptable input forms.

Examples:

Data Item	Specification	Internal Value
T	L1	.TRUE.
.F	L2	.FALSE.
TFILLERS	L9	.TRUE.
bbbF	L4	.FALSE.

Output Using Lw

On output, the format specification Lw causes the corresponding logical list item in the output list to be written to the specified output file. The logical value T for .TRUE. or F for .FALSE. is written adjacent to w − 1 blanks.

Examples:

List Item Value	Specification	Output Field
.FALSE.	L1	F
.FALSE.	L3	bbF
.TRUE.	L2	bT

The following are statements using L format:

```
LOGICAL LOG1, LOG2
CHARACTER * 7 A,B
A='(L5,L2)'
READ (5,A),LOG1,LOG2
WRITE (B,A) LOG1,LOG2
```

Assume the input record contains the following (b denotes a blank):

```
.TRUEbF
```

Execution of this program segment would cause the LOGICAL variables LOG1 and LOG2 to be assigned the following values:

```
LOG1=T
LOG2=F
```

The output to the internal file (character variable B) would be the following:

```
bbbbTbF
```

Format Specification A

The A format specification is used to transmit EBCDIC character data between internal storage and a file. Only CHARACTER type entities (variables, arrays, and so forth) in the input/output list can be associated with A edit descriptors in the corresponding format specification. If the field width w is not specified in the A edit descriptor, the width of the field is the declared length of the input/output list item.

Input Using Aw

On input, the alphanumeric format specification Aw causes the character string of width w in the input field to be assigned to the corresponding character variable, character array element, or character array substring. The number of characters that can be stored in a variable depends upon the length of the character entity in the input list.

If the field width (w) exceeds the maximum number of characters (m) that can be contained within the input variable, the first w − m characters are skipped and the remaining rightmost (m) characters are assigned to the variable. If the field width (w) is less than the maximum number of characters that can be contained within the input variable, the alphanumeric string is assigned left-justified, with trailing blanks, to the variable.

Examples:

Variable Type	Data Item	Specification	Internal Value
CHARACTER * 4	ABCDEFGH	A8	EFGH
CHARACTER * 7	ABCbEFG	A7	ABCbEFG
CHARACTER * 10	ABCD	A3	ABCbbbbbbb
CHARACTER * 1	AB	A2	B

Blanks are not ignored when using an A edit descriptor.

Output Using Aw

On output, the alphanumeric format specification Aw causes the corresponding list item in the output list to be written on the specified output file. If the field width (w) exceeds the maximum number of characters that can be contained in the output list item, the alphanumeric string is placed right-justified in the output field over a field of blanks. If the field width (w) is less than the number of characters in the output list item, the leftmost characters in the variable are written.

Examples:

List Item Type	List Item Value	Specification	Output Field
CHARACTER * 4	ABCD	A6	bbABCD
CHARACTER * 3	ABC	A8	bbbbbABC
CHARACTER * 5	ABCDb	A3	ABC
CHARACTER * 4	ABCD	A2	AB
CHARACTER * 12	ABbbbCDEFbbG	A9	ABbbbCDEF

Examples of statements using A format follow:

```
      CHARACTER *2 A(5)
      READ (5,'(5A2)') A
      READ (A,100) (I(J),J = 1,5)
      WRITE (A,100) (I(J),J = 5,1, - 1)
100   FORMAT (I2)
```

Assume the input record contains the following:

1234567890

The first READ statement assigns the data to an internal file (character array A). The second READ uses data in A to assign integer values to array I. The WRITE statement returns the values to array A in reverse order. The following are the values of A after the first READ operation is completed:

A(1) = '12' A(2) = '34' A(3) = '56' A(4) = '78' A(5) = '90'

After the second READ operation, elements of array I have the following values:

I(1) = 12 I(2) = 34 I(3) = 56 I(4) = 78 I(5) = 90

After the WRITE operation, the elements of A have the following values:

A(1) = '90' A(2) = '78' A(3) = '56' A(4) = '34' A(5) = '12'

Reversion (refer to Interaction Between Input/Output List and Format in this section) must be used when processing the FORMAT statement used in the program segment.

Format Specification Z

The Z format specification can be used to assign hexadecimal digits to variables or to transmit data in hexadecimal form to a variable or array of any data type other than type CHARACTER.

Input Using Zw

On input, the hexadecimal format specification Zw converts the EBCDIC representation of the digits 0 through 9 and the characters A through F in the input field to 4-bit hexadecimal digits and assigns them to the corresponding INTEGER, REAL, DOUBLE PRECISION, LOGICAL, or COMPLEX variable in the input list. Leading, embedded, and trailing blanks within the input field are interpreted as zeros. The hexadecimal digits in the input field are transmitted right-justified to the corresponding input variable.

If the width (w) of the input field is less than the length of the variable (in hexadecimal digits), leading zeros are supplied. If the field width (w) is greater than the length of the input variable (in hexadecimal digits), the leftmost digits are truncated. Refer to section 7, Hexadecimal Assignment in DATA Statement, for a description of storage requirements for the different types of variables.

Examples:

Variable Type	Data Item	Specification	Internal Representation
INTEGER	00BC614E	Z8	00BC614E
REAL	4BEBCE50	Z8	4BEBCE50
DOUBLE PRECISION	C140000BC614E009	Z16	C140000BC614E009
LOGICAL	00000001	Z8	00000001(.TRUE.)

Output Using Zw

On output, the hexadecimal format specification Zw causes the hexadecimal value of the corresponding INTEGER, REAL, DOUBLE PRECISION, LOGICAL, or COMPLEX output list item to be converted to the EBCDIC representation of the hex digits 0 through F and then to be written to the specified file.

The hexadecimal value is placed right-justified in the output field over a field of blanks. If the length of the output list item (in hexadecimal digits) is less than the field width (w), leading blanks are supplied. If the length of the output list item (in hexadecimal digits) is greater than the field width (w), the leftmost digits are truncated. Refer to appendix D for additional information.

Examples:

List Item Type	List Item Value	Specification	Output Field
INTEGER	000BC614	Z9	b000BC614
REAL	4BEBCE50	Z9	b4BEBCE50
INTEGER	000BC614	Z6	0BC614
REAL	4BEBCE50	Z6	EBCE50
DOUBLE PRECISION	C140000BC614E009	Z16	C140000BC614E009
LOGICAL	00000001(.TRUE.)	Z9	b00000001

Examples of statements using Z format follow:

```
      LOGICAL LA
      READ (5,100) LA, Y
      WRITE (6,200) LA, Y
100   FORMAT (Z1,Z8)
200   FORMAT (L2,F10.4)
```

Assume the input record contains the following (b denotes a blank):

```
1b0000000
```

The output record would contain the following after the program segment is executed:

```
bTbbbbb.0000
```

Nonrepeatable Edit Descriptors

Nonrepeatable edit descriptors permit the use of special editing functions within the format specification. Nonrepeatable edit descriptors are not associated with any items of the input/output list and cannot be preceded directly by a repeat specifier.

String Editing

The string (or Hollerith) format specification wHs, 's', or "s" allows character strings to be written without employing character variables as storage.

The letter w in the Hollerith form is the number of characters, including blanks, following the H that are part of the output string.

The 's' and "s" edit descriptors have the form of character constants.

Examples:

```
100   PRINT 200
200   FORMAT ('ISN"T THIS NICE?')
300   WRITE (6,400) A
400   FORMAT (30HTHE DIAMETER OF THE CIRCLE IS ,F5.2," INCHES.")
```

Positional Editing

The position within a record at which data transfer is to occur can be determined using the X, T, TL, and TR edit descriptors. Using these edit descriptors permits the processing of the same character positions within a record more than once or the capability of skipping character positions within a record.

When used with input, positional edit descriptors permit the same characters within a record to be read more than once, possibly with different format specifiers. Also, character positions within the input record can be skipped by positioning the pointer beyond the unwanted characters.

When used with output, positional edit descriptors can allow characters within the output file to be overwritten, or character positions within the record can be skipped. Any character positions within the record that are skipped and had not previously been filled are written as blanks.

X Editing

The nX edit descriptor causes the next data transfer to occur n character positions forward from the current position. The X edit descriptor does not cause any data to be transmitted. The unmodified edit descriptor X is the same as specifying 1X.

Examples:

```
100     READ (A,'(I2,2X,I3)') (J(I),I=3,4)
200     WRITE (9,300)
300     FORMAT ("12345",7X,"67890")
```

The READ statement in this example reads two integer fields into array positions J(3) and J(4) from an internal file (character variable or character array A). Two character positions are skipped between the first and second fields. The WRITE statement writes two strings that are each enclosed by quotation marks with seven blanks between them.

T Editing

The Tn edit descriptor indicates that the next datum to be transferred is to begin at character position n.

The TLn edit descriptor specifies that the next datum to be transferred is to begin n character positions to the left of the current position.

The TRn edit descriptor specifies that the next datum to be transferred is to begin n character positions to the right of the current position.

An example of T editing follows:

```
        READ (6,100) A,I
100     FORMAT (T8,F3.2,TL3,I3)
```

Both variables A and I receive values from the same location within the current record, bytes 8 through 10.

Slash Editing

The slash edit descriptor (/) indicates the end of data transfer on the current record. With both sequential and direct-access files, the rest of the current record is skipped and the file is positioned at the which becomes the current record of the file. On output to a file connected for sequential access, a new record is created and becomes the last and current record of the file. Processing of the format specification continues from this point.

A record containing only blanks can be written on output. If any character positions in a record are skipped due to slash editing, the record is written with blanks in these character positions. On input, an entire record can be skipped by using slash editing.

Examples:

```
100     READ (5,'(I2/I2/I2)') I,J,K
200     WRITE (*,600) (X(I),I=1,100)
300     READ (A,600) (X(I),I=1,100)
600     FORMAT (50(2I2/))
```

The READ statement at line 100 in the previous example reads three integer fields and skips to the beginning of the next record after reading each field. The WRITE statement at line 200 writes 100 elements of array X on each of 50 records of file 6, two elements per record. The READ statement at line 300 reads 100 elements of array X from character array A. Each element of A must have a length of at least four characters, since two I2 fields are read from each element. Array A must have at least 50 elements.

Colon Editing

A colon edit descriptor (:) indicates that the processing of the current format specification is to terminate if there are no further items in the corresponding input/output list.

Example:

```
        WRITE (6,100) 1,2
        WRITE (6,200) 1,2
100     FORMAT (" LENGTH=",I2," WIDTH=",I2," DEPTH=",I2)
200     FORMAT (" LENGTH=",I2:" WIDTH=",I2:" DEPTH=",I2)
```

The following is the result of the previous example:

```
LENGTH= 1 WIDTH=2 DEPTH=
LENGTH= 1 WIDTH=2
```

The colon, following the second I2 in line 200, caused the output list to be checked for more items. Since none were found, the WRITE was terminated. The first WRITE statement could only terminate and found to be exhausted.

Sign Control

The S, SP, and SS edit descriptors are used to control the printing of an optional plus sign that can be printed with values that are output using I, F, E, D, and G edit descriptors.

The S and SS edit descriptors in B 1000 FORTRAN 77 suppress the printing of a plus sign preceding positive numbers for the duration of the format specification from the point at which the S or SS occur. Sign suppression is the default condition at the beginning of format processing.

The SP edit descriptor forces the printing of a plus sign preceding positive numeric values for the duration of the format specification from the point at which the SP edit descriptor occurs. An adequate field width must be provided to contain the additional output character. If there is not enough space for the plus sign, asterisks are printed in the field.

Example:

```
      WRITE (6,100) 21.7,193.2,18,234
100   FORMAT (SP,2F5.1,2(I4,SS))
```

Execution of the program segment in this example results in the following output:

```
+21.7***** +18 234
```

Scale Factor

The scale factor is specified by the kP edit descriptor, where k is an optionally signed integer constant called the scale factor. The scale factor is zero at the beginning of each input/output statement. When a scale factor is established, it remains in effect until the next kP edit descriptor or the end of the input/output statement. The scale factor affects values interpreted by subsequent F, E, D, and G edit descriptors.

The scale factor affects editing in the following manner:

1. On input with F, E, D, and G editing (providing that no exponent exists in the field) and on output with F editing, the externally represented number equals the internally represented number multiplied by 10**k (EXTERNAL = INTERNAL*(10**k) or (INTERNAL=EXTERNAL/(10**k)).
2. On input with F, E, D, and G editing, the scale factor has no effect if there is an exponent in the field.
3. On output with E and D editing, the real part of the quantity is multiplied by 10**k and the corresponding exponent is reduced by k.
4. On output with G editing, the scale factor only has effect when the value of the list item requires that E editing is used, and then it has the same effect as with E output editing.

Examples:

```
READ (5,'(2PF4.1)') A
WRITE (6,'(-3P2E14.7)') A, 41174.
```

If the input record contains the value 1234, then A has the value 1.234 (123.4/(10**2)). The output record has the following value:

```
bb.0001234E+04bb.0004117E+08
```

Blank Control

The BN and BZ edit descriptors specify how blanks within a numeric input field are to be interpreted. During processing of a format specification, these descriptors override the BLANK= setting for the file as specified in the file declaration from the point of occurrence until the end of the input/output statement.

The BN edit descriptor specifies that all blanks in the numeric input field are to be ignored. The effect is the same as if all blanks were removed and the data right-justified in the field.

The BZ edit descriptor specifies that all blanks within the numeric input field are to be interpreted as zeros.

Example:

```
      CHARACTER *11 A
      A='1 3 5 6 01'
      READ (A,100) I,J,K
100   FORMAT (BZ,I2,I3,BN,I6)
```

After execution of the preceding program segment, the variables have the following values:

I = 10
J = 305
K = 601

The BN and BZ edit descriptors have no effect on output.

Positioning By Format Control

After each I, F, E, D, G, L, A, Z, H, apostrophe, or quotation mark edit descriptor is processed, the file pointer is positioned after the last character read or written in the current record.

After each T, TL, TR, X, or slash edit descriptor is processed, the file is positioned according to the manner described under Nonrepeatable Edit Descriptors in this section.

If reversion occurs (refer to Interaction Between Input/Output List and Format in this section), the file is positioned at the beginning of the next record before processing of the format specification is resumed.

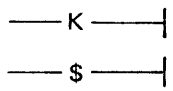
During a READ operation, any unprocessed characters of the record are skipped whenever the next record is read.

Format Modifiers

Format modifiers change the output fields of integer, real, double-precision, or complex data by inserting comma (,) or dollar sign ($) characters.

The format modifiers can be used with the Iw, Iw.m, Fw.d, Ew.d, Dw.d, Ew.dEe, and Ew.dDe edit descriptors. They can also be used with the Gw.d edit descriptor when the input/output list items are type integer, real, or double precision.

The two forms of format modifiers follow:



The modifier symbols K and $ indicate the type of format modification. The format modifiers appear to the left of the edit descriptor and to the right of any repeat specifier. Both can appear together. Format modifiers cannot be used on input.

K Modifier

The K modifier inserts comma (,) characters into the output field between digit triples to the left of the decimal point. The field width w must be wide enough to accommodate the comma (,) characters in addition to the other characters.

$ Modifier

The $ modifier inserts a dollar sign ($) character immediately to the left of the leftmost nonblank character in the output field. The field width w must be wide enough to accommodate the dollar sign ($) character in addition to the other characters.

An example of the use of both the K and $ format modifiers follows:

```
      SUM = 75250.
      WRITE (6,100) SUM
100   FORMAT ("TOTAL COST:",$KF15.2)
```

Execution of this program segment results in the following output:

```
TOTAL COST:     $75,250.00
```

Carriage Control

When a line printer is used for output, the first character of each line of print controls the spacing of the printer carriage. The control characters are as follows:

Character	Action
(blank)	Single-space before printing.
0 (zero)	Double-space before printing.
1 (one)	Skip to channel 1 of carriage control tape before printing. (Advance to first line of next page).
+ (plus sign)	No advance before printing.
n (any digit 2 through 9)	Advance to channel n before printing.

The first character of the print line is used only to control the action of the printer carriage. It is not printed.

LIST-DIRECTED FORMATTING

List-directed formatting provides a way to input and output records without explicit reference to a format specification. The format specifier used in list-directed input/output is the asterisk (*) character. The format of list-directed input is determined by the values and value separators that make up the input record(s). The format of list-directed output is determined by the values of the items in the output list.

The values that are written to a file in list-directed output and the values that are assigned to items of the input list in list-directed input are either constants, null values, or have one of the following forms:

r*c
r*

where r is an unsigned, nonzero, integer constant, and c is a constant. The r*c form represents r successive appearances of the constant c, and the r* form represents r successive null values. Neither of these forms can contain embedded blank characters, except where permitted within the constant c.

A value separator is one of the following:

1. A comma (,) character optionally preceded by one or more contiguous blank characters and optionally followed by one or more contiguous blank characters.
2. A slash (/) character optionally preceded by one or more contiguous blank characters and optionally followed by one or more contiguous blank characters.
3. One or more contiguous blank characters.

The end of a record has the same effect as a blank character. Any sequence of two or more consecutive blank characters is treated as a single blank character, unless it is within a character constant.

List-directed Input

List-directed input occurs when an asterisk (*) character is used as a format specifier in an input statement. The format of list-directed input is determined by the values and value separators that make up the input record(s). The general forms that these values and value separators can take are described in the preceding paragraphs. The form of the input value must be acceptable to the format specification of the corresponding input list item. All input forms that are normally acceptable to a corresponding format specification are acceptable in list-directed formatting except as noted below.

When the corresponding input list item is of type real or double precision, the input form must be a numeric input field. A numeric input field is a field suitable for F editing and is assumed to have no fractional part unless a decimal point (.) character appears within the field.

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

When the corresponding list item is of type logical, the input form cannot include either slash (/) or comma (,) characters among the optional characters permitted for L editing.

When the corresponding list item is of type character, the input form consists of a nonempty string of characters enclosed in apostrophe (') characters. Each apostrophe (') character within a character constant is represented by two consecutive apostrophe (') characters without an intervening blank character or end of record. Character constants can be continued from the end of one record to the beginning of the next record. The end of the record does not cause a blank character or any other character to become part of the constant. The constant can be continued on as many records as needed. The slash (/), comma (,), and blank characters can appear in character constants.

Assume that len is the length of the list item, and w is the length of the character constant. If len is less than or equal to w, the leftmost len characters of the constant are transmitted to the list item. If len is greater than w, the constant is transmitted to the leftmost w characters of the list item and the remaining len minus w characters of the list item are filled with blank characters. This occurs as though the constant were assigned to the list item in a character assignment statement.

A null value is specified in one of three ways: 1) having no characters between successive value separators, 2) having no characters preceding the first value separator in the first record read by each execution of a list-directed input statement, or 3) using the r* form. A null value has no effect on the definition status of the corresponding input list item. If the input list item is defined, it retains its previous value; if it is undefined, it remains undefined. A null value must not be used as either the real or imaginary part of a complex constant, but a single null value can represent an entire complex constant. The end of a record following any other separator, with or without separating blank characters, does not specify a null value.

A slash (/) character encountered as a value separator during execution of a list-directed input statement terminates execution of that input statement following the assignment of the previous value. If there are additional items in the input list, the effect is as if null values had been supplied for them.

All blank characters in a list-directed input record are considered to be part of some value separator except for the following:

1. Embedded blank characters in a character constant.
2. Embedded blank characters enclosing the real or imaginary part of a complex constant.
3. Leading blank characters in the first record read by each execution of a list-directed input statement, unless immediately followed by a slash (/) or comma (,) character.

An example of a program segment containing list-directed input follows:

```
DOUBLE PRECISION DP
COMPLEX COMPLX
LOGICAL LOGIC
CHARACTER*7 CHAR
REAL REAL
INTEGER INT(10)
READ (5,*) DP,COMPLX,CHAR, REAL,LOGIC,INT
```

Examples of input records that can be read by the preceding list-directed input statement follow:

```
1234567890.123D+29,(2,0.37),T,'ABCDEFG',9876.54321,10*1
1D+20,(1,0.25),F,'NEWCHAR',7.0,5,5,5,5,5,5,5,5,5,5
1D+20 /(1,0.25)  T/ 'NEW' ,  7.0, 5 5 5 5 5 5 5 5 5 5
```

List-directed Output

List-directed output occurs when an asterisk (*) character is used as a format specifier in an output statement. The format of list-directed output is determined by the values of the items in the output list. The form of the value separators is described in the introductory paragraphs with the exception that the slash (/) character is not an output separator. The general form of the output values is also as described in the introductory paragraphs and is the same as required for input except as noted in the following paragraphs.

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

Logical constants are output as T for the value TRUE and F for the value FALSE.

Integer constants are output with the effect of an Iw edit descriptor, with an appropriate value of w.

Real and double-precision constants are output with the effect of either an F edit descriptor or an E edit descriptor, depending on the magnitude x of the value. If x is within the range 10**d1 .LE. x .LT. 10**d2, where d1 and d2 are processor-dependent integer values, the constant is output with the format 0PFw.d; otherwise, the format 1PEw.dEe is used.

Complex constants are enclosed in parenthesis "( )" characters, with a comma (,) character separating the real and imaginary parts. The end of a record can occur between the comma (,) character and the imaginary part only if the entire constant is as long as, or longer than, an entire record. The only embedded blank characters permitted within a complex constant are between the comma (,) character and the end of a record and one blank character at the beginning of the next record.

Character constants that are output are not preceded or followed by a value separator. The constants are not delimited by apostrophe (') characters and each internal apostrophe (') character is represented externally by one apostrophe (') character. If any record begins with the continuation of a character constant from a preceding record, the processor inserts a blank character at the beginning of the record for carriage control.

If two or more successive values in an output record have identical values, the processor has the option of outputting a repeated constant of the form r*c instead of the sequence of identical values.

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

An example of a program segment containing list-directed output follows:

```
DOUBLE PRECISION DP/1234567890.123456D+29/
COMPLEX COMPLX/(2,0.37)/
LOGICAL LOGIC/.TRUE./
CHARACTER*7 CHAR/'ABCDEFG'/
REAL REAL/9876.54321/
INTEGER INT(10)/1,1,2,2,2,3,3,3,3,3/
WRITE (6,*) 'CONSTANTS: ',DP,COMPLX,CHAR,REAL,LOGIC,INT
```

Execution of this program segment results in the following list-directed output:

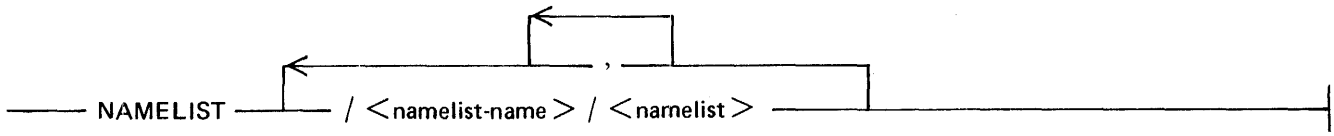
```
CONSTANTS:  .1234568E+39, (2.000000, .3700000)ABCDEFG 9876.542,
T, 2*1, 3*2, 5*3
```

NAMELIST FORMATTING

Namelist formatting provides a way to use a single name to input and output a list of variables, arrays, or array elements without reference to a format specification. The single name is called the namelist name and is declared in a NAMELIST statement. Namelist formatting is indicated by the use of a namelist name as a format specifier in an input/output statement.

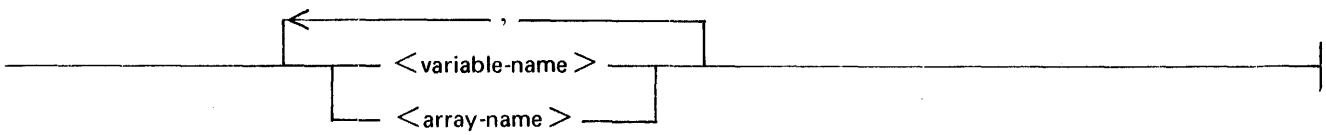
NAMELIST Statement

The form of a NAMELIST statement follows:



<namelist-name> follows the same naming conventions as variables.

<namelist> has the following form:



Names of dummy arguments cannot appear in <namelist>.

In a NAMELIST statement, the variables and arrays whose names appear in the <namelist> are declared to be named by <namelist-name>. A variable or array can be named in more than one namelist list.

Examples of NAMELIST statements follow:

```
NAMELIST NLIST/A, B, C, D
NAMELIST BBALL/RUNS,HITS,ERRORS,INNING,SCORE1,SCORE2
```

Form of Namelist Input/Output

An input statement specifying namelist formatting uses a namelist data group as input. An output statement specifying namelist formatting creates a namelist data group as output. A namelist data group consists of one or more records. The first character in each record of a data group must be a blank. The second character of the first record of a data group must be an ampersand (&) character followed immediately by a <namelist-name>. The <namelist-name> must contain no embedded blank characters and must be followed by one or more blank characters. Following these blank characters is a sequence of value assignments separated by comma (,) characters. The end of a data group is indicated by the appearance of the four characters &END. A comma (,) character can optionally appear before &END. The remainder of the record following &END is ignored.

A value assignment has one of the following forms:

1. variable name = constant.
2. array name = list of values separated by comma (,) characters.
3. array element name = list of values separated by comma (,) characters.

A value has one of the forms:

1. k*constant
2. constant

where k is an unsigned, nonzero, integer constant. The k*constant form is equivalent to a list of k successive appearances of the constant.

In the form:

array name = list of values

the list of values must contain no more constants than there are elements in the array.

In the form:

array element name = list of values

the list of values must contain no more constants than there are in the array block beginning with the named element and ending with the last element of the array.

The constants can be integer, real, double-precision, complex, logical, or character constants. If the constant is logical, the forms T and F can also be used for .TRUE. and .FALSE., respectively.

No embedded blank character can appear in the variable names, array names, array element names, or in the arithmetic or logical constants. Any trailing blank characters that follow integer constants or follow exponent parts of real or double-precision constants are treated as zeros. Any number of blank characters can appear: 1) between the variable name, array name, or array element name and the equal sign (=) character, 2) between the equal sign (=) character and the constant or list of values, 3) between a value and the preceding comma (,) character in a list of values, 4) between a value assignment and the preceding comma (,) character and 5) between &END and the preceding comma (,) character.

The end of a record in the data group can only occur in the following places: 1) between the namelist name and a value assignment, 2) between a value assignment and the preceding comma (,) character in the list of value assignments, 3) between the equal sign (=) character and the constant or list of values, 4) between a value and the preceding comma (,) character in a list of values, and 5) between &END and the preceding value assignment, comma (,) character, or namelist name.

A variable or array whose name appears in the list of value assignments in the data group for the <namelist-name> must be named in the namelist list for <namelist-name>. It is not necessary for all of the variables and arrays named in the namelist list to have their names appear in the list of value assignments in the data group for <namelist-name>. The order of the names in <namelist> is not significant. A name associated with a name appearing in a namelist list for <namelist-name> cannot be substituted for that name in the value assignment list in the data group for <namelist-name>.

Namelist Input

When an input statement specifying namelist formatting is executed, the file is positioned and accessed repeatedly until the data group for the specified namelist is found. For each value assignment, assignment to the indicated variable or array occurs as follows:

1. For the form:

   variable name = constant

   the constant is assigned to the variable.

2. For the form:

   array name = list of values

   each constant in the list of values is assigned to an element of the array. The elements of the array are assigned in the order specified by the array element ordering, beginning with the first element.

3. For the form:

   array element name = list of values

   each constant in the list of values is assigned to an element of the array. The elements of the array are assigned in the order specified by the array element ordering, beginning with the element specified in the value assignment.

Conversion is applied in the same manner as for arithmetic, logical, and character assignment statements.

A variable named in the namelist list for <namelist-name> but whose name does not appear in a value assignment in the data group for namelist <namelist-name> retains its current value if defined or remains undefined. Any element of an array named in the namelist list for <namelist-name> which is not assigned a value in a value assignment in the data group for namelist <namelist-name> retains its current value if defined or remains undefined.

An example of namelist input follows:

```
INTEGER INT(10)
NAMELIST/CONSTS/DP,COMPLX, LOGIC,CHAR,REAL,INT
READ (6,CONSTS)
```

Examples of data groups that can be read by the preceding namelist input statement follow:

```
&CONSTS LOGIC=F, REAL = 7.0 &END
&CONSTS DP =7777.77D+30, CHAR='NEWCHAR', INT(5)=100, &END
&CONSTS COMPLX = (33,3.3),INT=10*0, &END
&CONSTS &END
```

Namelist Output

When an output statement specifying namelist formatting is executed, the output is written to the file in a form that can be read by a READ statement specifying the same namelist name as a format specifier. The values of all of the variables and arrays whose names appear in the namelist list for the specified namelist are written. The values of integer, real, double-precision, complex, logical, and character variables or arrays are written as integer, real, double-precision, complex, logical, and character constants respectively. Fields for the data are made large enough to contain all significant digits.

The value of a variable is written in the form:

```
variable name = constant
```

The values of an array are written in the form:

```
array name = list of constants
```

where the list of constants contains one constant for each element in the array, written in the order specified by the array element ordering.

An example of a program segment containing namelist output follows:

```
DOUBLE PRECISION DP/1234567890.12345D+29/
COMPLEX COMPLX/(2,0.37)/
LOGICAL LOGIC/.TRUE./
CHARACTER*7 CHAR/'ABCDEFG'/
REAL REAL/9876.54321/
INTEGER INT(10)/1,1,2,2,2,3,3,3,3,3/
NAMELIST/CONSTS/DP, COMPLX,LOGIC, CHAR,REAL,INT
WRITE (6,CONSTS)
```

Execution of this program segment results in the following data group:

```
&CONSTS DP =.123456789012345D+39, COMPLX = (2.000000, .3700000),
LOGIC =T,CHAR ="ABCDEFG", REAL =9876.542, INT =2*1, 3*2,5*3,
&END
```

The number of records output and the location of the end of each record depend upon the output device.

SECTION 13
SUBPROGRAMS

Subprograms are program units which can be invoked in the main program or any other program segment as a separate executable procedure or which can be used for data initialization. There are three types of subprograms: functions, subroutine subprograms, and block data subprograms. These three types of subprograms and associated concepts and statements are described in this section.

FUNCTIONS

Functions are procedures which return a value to a calling program unit at the point at which the call was made.

Example:

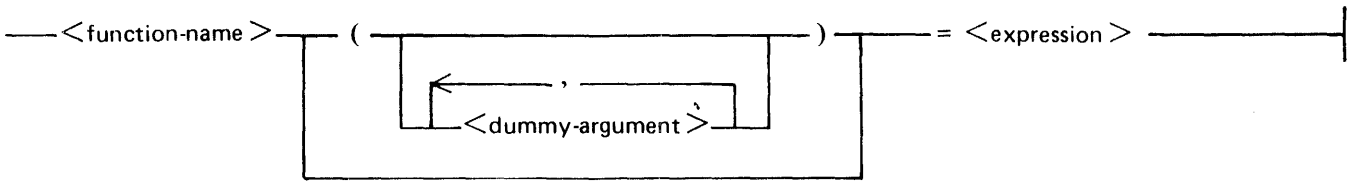
```
A = FUNC(B,C) + Y
```

In this example, FUNC is the name of a function. Control transfers to the function named FUNC during processing of the expression. A value is obtained from the function and used in the position occupied by the symbolic name FUNC in the expression. The value is then added to the value of variable Y and the result is assigned to A. Parentheses must always appear following the function reference even if there are no arguments (refer to ARGUMENTS in this section) being passed.

There are three types of function subprograms that can be used in FORTRAN 77. A function which is a single statement within the program unit that references it is a statement function. A function which is a separate subprogram that can be referenced by any other program unit (including another function) is a function subprogram. A function which is supplied by FORTRAN 77 for the user is an intrinsic function.

Statement Functions

A statement function is a function which can be expressed as one statement. It has the same general form as an assignment statement, except that the function name and dummy argument list appear to the left of the replacement operator. This statement is called a statement function declaration. The following is the proper form of a statement function declaration:



G50342

<function-name> is the name of the statement function and is constructed in the same manner as a variable name. <function-name> has the default type associated with the initial letter in its name unless another type is given in an IMPLICIT statement or explicit type statement (refer to Explicit Type Statement in section 6). <function-name>, if declared to be type CHARACTER, can have any length declaration, except an assumed length (CHARACTER * (*)).

Once a symbolic name is used as the name of a function in a program unit, that name cannot be used for any other purpose in that program unit except as the name of a common block.

<dummy-argument> is a dummy variable which is used by the statement function. A <dummy-argument> list in a statement function cannot contain array names or alternate return specifiers (refer to Alternate Return in this section). Dummy arguments are described later in this section.

<expression> is an arithmetic, logical, or character expression (depending on the type associated with the function name) employing any of the following:

1. A constant.
2. The symbolic name of a constant.
3. A variable reference.
4. An array element reference.
5. An intrinsic function reference.
6. A reference to a statement function for which the statement function statement appears in preceding lines of the program unit.
7. An external function reference.
8. An expression enclosed in parentheses that meets all of the requirements specified for the <expression>.

Each variable reference can be either a reference to a dummy argument of the statement function or a reference to a variable that appears within the same program unit as the statement function statement. If the symbolic name of a local variable or subprogram dummy variable is also the name of a dummy variable in the statement function, the dummy variable within the statement function is used without affecting the value of the local variable or subprogram dummy variable.

Referencing a Statement Function

A statement function is referenced by using its name in an expression.

Execution of a statement function reference results in the following:

1. Evaluation of actual arguments that are expressions.
2. Association of actual arguments with the corresponding dummy arguments.
3. Evaluation of the expression.
4. Conversion, if necessary, of an arithmetic expression value to the type of the statement function as explained in Arithmetic Assignment Statements in section 7, or a change, if necessary, in the length of a character expression value as described in Character Assignment Statements in section 7.

The resulting value is available to the expression that contains the function reference.

The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments.

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

When a statement function reference is executed, the actual arguments must be defined.

A statement function can reference other statement functions. However, the statement function(s) being referenced must be declared before the statement function making the reference and must also be de-

clared within the same program unit. A function reference in the <expression> of a statement function must not change the value of any dummy argument associated with the statement function.

An example of a statement function follows:

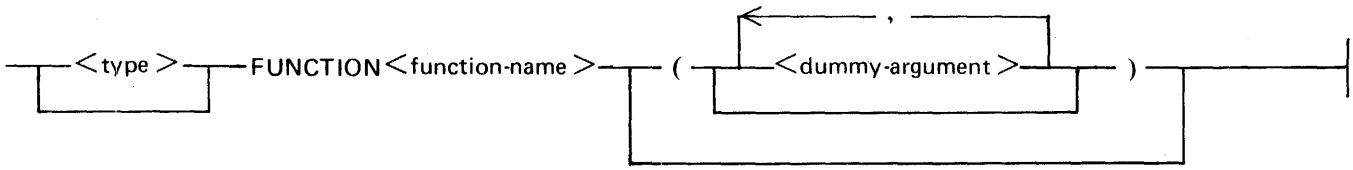
```
      REAL X,Y
      CIRCUM(Y)= SQRT(C-(Y**2))
            .
            .
            .
100   X=CIRCUM(Y)*2
            .
            .
            .
      END
```

In the example, the expression at line 100 temporarily transfers control to statement function CIRCUM. The value returned by CIRCUM (a real value) is multiplied by 2 and assigned to X. CIRCUM references an intrinsic function called SQRT within its expression.

Function Subprograms

A function subprogram is a separate program unit with local data and, optionally, other function subprograms. Any variable names, except dummy variable names and common variables, are local to the function subprogram. Any change in the value of a local variable has no effect on any variable with the same symbolic name occurring in any other program unit.

Function subprograms permit the function to be comprised of more than one statement. The beginning of a function subprogram is denoted by a FUNCTION statement. The following is the proper form of the FUNCTION statement:



G50343

A FUNCTION statement declares <function-name>, and optionally, <type> for the function. The naming convention for <function-name> is the same as that for a variable name. The function must assign a value to the <function-name> before the end of the function.

<type> can be declared in the FUNCTION statement, in an IMPLICIT statement, or in an explicit type statement following the function declaration (refer to Explicit Type Statement in section 6). <type> can be any data type, including CHARACTER *<expr> and CHARACTER *(*), where <expr> is an integer constant expression that does not contain the symbolic name of an integer. For a function declared as CHARACTER *(*), the length of the value returned is determined by the length declaration for the function name in the calling program unit. The length declaration in the calling program unit must be an integer constant expression. If no type is assigned, the default <type> associated with the first letter of <function-name> is assumed.

<dummy-argument> is a dummy variable name, dummy array name, or dummy procedure name. The <dummy-argument> list must not contain an alternate return specifier.

Referencing a Function Subprogram

A function subprogram is referenced by using the function name in an expression of an executable program.

Execution of an External Function Reference

Execution of a function subprogram results in the following:

1. Evaluation of actual arguments that are expressions.
2. Association of actual arguments with the corresponding dummy arguments.
3. The actions specified by the referenced function.

The type of the function name in the function reference must be the same as the type of the function name in the referenced function. The length of the character function in a character function reference must be the same as the length of the character function in the referenced function.

When a function subprogram is executed, the function must be one of the function subprograms in the executable program.

Actual Arguments for a Function Subprogram

The actual arguments in a function subprogram reference must agree in order, number, and type with the corresponding dummy arguments in the referenced function. The use of a subroutine name as an actual argument is an exception to the rule requiring agreement of type.

An actual argument in an external function reference must be one of the following:

1. Any expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses, unless the operand is the symbolic name of a constant.
2. An array name.
3. An intrinsic function name.
4. An external procedure name.
5. A dummy procedure name.

An actual argument in a function reference can be a dummy argument that appears in a dummy argument list within the subprogram containing the reference.

Example:

```
      COMPLEX X,I
      CHARACTER * 5 A, B * 3
            .
            .
            .
      A(3:4) = 'MN'
100   A = B('ABC') // A(3:4)
      X = I()
            .
            .
            .
      END
      FUNCTION B(C)
      CHARACTER * 3 B, C
            .
            .
            .
      B = C(1:2) // 'J'
      END
      COMPLEX FUNCTION I
      COMPLEX L
            .
            .
            .
      I = (4.5, 5.7) * L
      END
```

The two functions in the example return values to the expressions in which they are referenced at the position where the reference is made. In each case, an assignment is made within the body of the function to the function name. This value is converted to the type of the function and becomes the value that is returned. At line 100, for example, character variable A is assigned the value "ABCMN".

The function subprogram can contain any statements recognizable by FORTRAN 77, except FILE, BLOCK DATA, PROGRAM, or SUBROUTINE. As with the main program, all declaration and specification statements must precede the first executable statement in the subprogram. A function subprogram cannot contain a direct or indirect reference to itself.

A function subprogram is terminated by an END statement.

Intrinsic Functions

Intrinsic functions are subprograms supplied by FORTRAN 77 for use by the programmer. The values returned by intrinsic functions and their definitions are given in table 13-1 and are available for use within the expression in which they occur. Intrinsic functions are referenced either by a specific name or by a generic name. The manner in which intrinsic functions are referenced is described in the following paragraphs.

Specific Name and Generic Name

When an intrinsic function is referenced by a specific name (refer to table 13-1), the type of the value returned is the type associated with the specific name. The types of the arguments are fixed and must agree with the types given in table 13-1. When a generic name is used to refer to an intrinsic function, the type of the result (except for intrinsic functions performing type conversion, nearest integer, and absolute value of a complex argument) is the same as the type of the actual arguments. No explicit type is associated with the generic name of an intrinsic function.

For those intrinsic functions that have more than one argument, all arguments must be of the same type. If the specific or generic name of a function is used in a subprogram as a dummy argument or as a local variable or is specified in an EXTERNAL statement, then the intrinsic function becomes unavailable for use in that subprogram. Only the specific name of an intrinsic function can be used as an actual argument (refer to Intrinsic Statement in section 6).

All angles are expressed in radians. The result of a function of type COMPLEX is the principal value.

Table 13-1. Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Type Conversion	Conversion to integer int(arg) See Note 1	1	INT	- INT IFIX IDINT -	INTEGER REAL REAL DOUBLE COMPLEX	INTEGER INTEGER INTEGER INTEGER INTEGER
	Conversion to Real See Note 2	1	REAL	REAL FLOAT - SNGL -	INTEGER INTEGER REAL DOUBLE COMPLEX	REAL REAL REAL REAL REAL
	Conversion to Double See Note 3	1	DBLE	- - - -	INTEGER REAL DOUBLE COMPLEX	DOUBLE DOUBLE DOUBLE DOUBLE
	Conversion to Complex See Note 4	1 or 2	CMPLX	- - - -	INTEGER REAL DOUBLE COMPLEX	COMPLEX COMPLEX COMPLEX COMPLEX
	Conversion to Integer See Note 5	1	-	ICHAR	CHARACTER	INTEGER
	Conversion to Character See Note 5	1	-	CHAR	INTEGER	CHARACTER
Truncation	int (arg) See Note 1	1	AINT	AINT DINT	REAL DOUBLE	REAL DOUBLE
Nearest Whole Number	int (arg + .5) if arg $\geq$ 0 int (arg – .5) if arg $<$ 0	1	ANINT	ANINT DNINT	REAL DOUBLE	REAL DOUBLE
Nearest Integer	int (arg + .5) if arg $\geq$ 0 int (arg – .5) if arg $<$ 0	1	NINT	NINT IDNINT	REAL DOUBLE	INTEGER INTEGER
Absolute Value	| arg | or , if arg is Complex (arg** 2 + argi**2)**0.5 See Note 6	1	ABS	IABS ABS DABS CABS	INTEGER REAL DOUBLE COMPLEX	INTEGER REAL DOUBLE REAL

Table 13-1. Intrinsic Functions (Cont)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Remaindering	arg1-int(arg1/arg2)*arg2 See Note 7	2	MOD	MOD AMOD DMOD	INTEGER REAL DOUBLE	INTEGER REAL DOUBLE
Transfer of Sign	|arg1| if arg2 ≥ 0 -|arg1| if arg2 < 0 See Note 8	2	SIGN	ISIGN SIGN DSIGN	INTEGER REAL DOUBLE	INTEGER REAL DOUBLE
Positive Difference	arg1 – arg2 if arg1 > arg2 0 if arg1 ≤ arg2	2	DIM	IDIM DIM DDIM	INTEGER REAL DOUBLE	INTEGER REAL DOUBLE
Double-Precision Product	arg1 * arg2	2	-	DPROD	REAL	DOUBLE
Choosing Largest Value	max (arg1, arg2, . . .)	≥2	MAX	MAX0 AMAX1 DMAX1	INTEGER REAL DOUBLE	INTEGER REAL DOUBLE
			- -	AMAX0 MAX1	INTEGER REAL	REAL INTEGER
Choosing Smallest Value	min (arg1, arg2, . . .)	≥2	MIN	MIN0 AMIN1 DMIN1	INTEGER REAL DOUBLE	INTEGER REAL DOUBLE
			- -	AMIN0 MIN1	INTEGER REAL	REAL INTEGER
Length	Length of Character Entity. See Note 9	1	-	LEN	CHARACTER	INTEGER
Index of a Substring	Location of Substring arg2 in String arg1 See Note 10	2	-	INDEX	CHARACTER	INTEGER
Imaginary Part of Complex Argument	argi See Note 6	1	-	AIMAG	COMPLEX	REAL
Conjugate of a Complex Argument	(argr, -argi) See Note 6	1	-	CONJG	COMPLEX	COMPLEX
Square Root	arg**0.5 See Note 11	1	SQRT	SQRT DSQRT CSQRT	REAL DOUBLE COMPLEX	REAL DOUBLE COMPLEX
Exponential	e**arg	1	EXP	EXP DEXP CEXP	REAL DOUBLE COMPLEX	REAL DOUBLE COMPLEX
Natural Logarithm	log (arg) See Note 12	1	LOG	ALOG DLOG CLOG	REAL DOUBLE COMPLEX	REAL DOUBLE COMPLEX

Table 13-1. Intrinsic Functions (Cont)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Common Logarithm	log 10 (arg) See Note 12	1	LOG10	ALOG10 DLOG10	REAL DOUBLE	REAL DOUBLE
Sine	sin (arg) See Note 13	1	SIN	SIN DSIN CSIN	REAL DOUBLE COMPLEX	REAL DOUBLE COMPLEX
Cosine	cos (arg) See Note 13	1	COS	COS DCOS CCOS	REAL DOUBLE COMPLEX	REAL DOUBLE COMPLEX
Tangent	tan (arg) See Note 13	1	TAN	TAN DTAN	REAL DOUBLE	REAL DOUBLE
Arcsine	arcsin(arg) See Note 14	1	ASIN	ASIN DASIN	REAL DOUBLE	REAL DOUBLE
Arccosine	arccos (arg) See Note 15	1	ACOS	ACOS DACOS	REAL DOUBLE	REAL DOUBLE
Arctangent	arctan (arg) See Note 16	1	ATAN	ATAN DATAN	REAL DOUBLE	REAL DOUBLE
	arctan (arg1/arg2) See Note 16	2	ATAN2	ATAN2 DATAN2	REAL DOUBLE	REAL DOUBLE
Hyperbolic Sine	sinh (arg)	1	SINH	SINH DSINH	REAL DOUBLE	REAL DOUBLE
Hyperbolic Cosine	cosh (arg)	1	COSH	COSH DCOSH	REAL DOUBLE	REAL DOUBLE
Hyperbolic Tangent	tanh (arg)	1	TANH	TANH DTANH	REAL DOUBLE	REAL DOUBLE
Lexically Greater Than or Equal	arg1 $\geq$ arg2 See Note 17	2	-	LGE	CHARACTER	LOGICAL
Lexically Greater Than	arg1 $>$ arg2 See Note 17	2	-	LGT	CHARACTER	LOGICAL
Lexically Less Than or Equal	arg1 $\leq$ arg2 See Note 17	2	-	LLE	CHARACTER	LOGICAL
Lexically Less Than	arg1 $<$ arg2 See Note 17	2	-	LLT	CHARACTER	LOGICAL

Table 13-1. Intrinsic Functions (Cont)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of	
					Argument	Function
Time	See Note 18	1	-	TIME	CHARACTER	REAL
Date	See Note 19	1	-	DATE	CHARACTER	CHARACTER
Pseudo Random Number	See Note 20	1	-	RANDOM	INTEGER	REAL
Bit Manipulation	See Note 21	2	AND	-	INTEGER REAL LOGICAL	INTEGER REAL LOGICAL
	See Note 22	2	OR	-	INTEGER REAL LOGICAL	INTEGER REAL LOGICAL
	See Note 23	2	EQUIV	-	INTEGER REAL LOGICAL	INTEGER REAL LOGICAL
	See Note 24	1	COMPL	-	INTEGER REAL LOGICAL	INTEGER REAL LOGICAL

Notes for table 13-1:

1. For arg of type INTEGER, INT(arg) .EQ. arg. For arg of type REAL or DOUBLE PRECISION, there are two cases: if ABS(arg) .LT. 1, then INT(arg) .EQ. 0; if ABS(arg) .GE. 1, then INT(arg) is the integer with the largest magnitude that does not exceed the magnitude of arg and whose sign is the same as the sign of arg.

   Example:

   INT(−3.7) = −3

   For arg of type COMPLEX, INT(arg) is the value obtained by applying the above rule to the real part of arg.

   For arg of type REAL, IFIX(arg) is the same as INT(arg).

2. For arg of type REAL, REAL(arg) is arg. For arg of type INTEGER or DOUBLE PRECISION, REAL(arg) is an approximation of arg with only the amount of precision that a REAL datum can contain. For arg of type COMPLEX, REAL(arg) is the real part of arg.

   For arg of type INTEGER, FLOAT(arg) is the same as REAL(arg).

3. For arg of type DOUBLE PRECISION, DBLE(arg) is arg. For arg of type INTEGER or REAL, DBLE(arg) is an approximation of arg with the amount of precision that a DOUBLE PRECISION datum can contain. For arg of type COMPLEX, DBLE(arg) is an approximation of the real part of arg with the amount of precision that a DOUBLE PRECISION datum can contain.
4. CMPLX can have one or two arguments. If only one, the argument can be of type INTEGER, REAL, DOUBLE PRECISION, or COMPLEX. If there are two arguments, both must be of the same type and can be of type INTEGER, REAL, or DOUBLE PRECISION.

   For arg of type COMPLEX, CMPLX(arg) is arg. For arg of type INTEGER, REAL, or DOUBLE PRECISION, CMPLX(arg) is the complex value whose real part is REAL(arg) and whose imaginary part is zero.

   CMPLX(arg1,arg2) is the complex value whose real part is REAL(arg1) and whose imaginary part is REAL(arg2).

5. ICHAR provides a means of converting from type CHARACTER to type INTEGER, based on the position of the character in the processor collating sequence. The first character in the collating sequence corresponds to position 0 and the last to position n − 1, where n is the number of characters in the collating sequence. Refer to section 2 for more information on the Burroughs FORTRAN 77 collating sequence.

   The value of ICHAR(arg) is an integer in the range 0 .LE. ICHAR(arg) .LE. n − 1, where arg is an argument of type CHARACTER of length one. The value of arg must be a character which can be represented by the B 1000 processor. The position of that character in the collating sequence is the value of ICHAR.

   For any characters c1 and c2 which can be represented by the B 1000 processor, (c1 .LE. c2) is TRUE only if (ICHAR(c1) .LE. ICHAR(c2)) is TRUE, and (c1 .EQ. c2) is TRUE only if (ICHAR(c1) .EQ. ICHAR(c2)) is TRUE.

   CHAR(i) returns the character in position i of the B 1000 FORTRAN 77 collating sequence. The value is of type CHARACTER of length one. The expression i must be an integer expression whose value must be in the range 0 .LE. i .LE. n − 1.

ICHAR(CHAR(i)) = i (for 0 < = i < = n − 1)

CHAR(ICHAR(c)) = c (for any character c which can be represented by the B 1000 processor)

6. A complex value is expressed as an ordered pair of real numbers, (argr,argi), where argr is the real part and argi is the imaginary part.
7. The result for MOD, AMOD, and DMOD is undefined when the value of the second argument is zero.
8. If the value of the first argument of ISIGN, SIGN, or DSIGN is zero, then the result is zero, which is neither positive nor negative.
9. The value of the argument of the LEN function need not be defined at the time the function reference is executed.
10. INDEX(arg1,arg2) returns an integer value indicating the starting position within the character string arg1 of a substring identical to string arg2. If arg2 occurs more than once in arg1, the starting position of the first occurrence is returned. If arg2 does not occur in arg1, the value 0 is returned. The value 0 is also returned if LEN(arg1) .LT. LEN(arg2).
11. The value of the argument of SQRT and DSQRT must be greater than or equal to zero. The result of CSQRT is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.
12. The value of the argument of ALOG, DLOG, ALOG10, and DLOG10 must be greater than zero. The value of the argument of CLOG must not be (0.,0.). The range of the imaginary part of the result of CLOG is −pi .LT. imaginary part .LE. pi. The imaginary part of the result is pi only when the real part of the argument is less than zero and the imaginary part of the argument is zero.
13. The absolute value of the argument of SIN, DSIN, COS, DCOS, TAN, and DTAN is not restricted to be less than 2*pi.
14. The absolute value of the argument of ASIN and DASIN must be less than or equal to one. The range of the result is −pi/2 .LE. result .LE. pi/2.
15. The absolute value of the argument of ACOS and DACOS must be less than or equal to one. The range of the result is 0 .LE. result .LE. pi.
16. The range of the result of ATAN and DATAN is −pi/2 .LE. result .LE. pi/2. If the value of the first argument of ATAN2 and DATAN2 is positive, the result is positive. If the value of the first argument is zero and the second argument is positive, the result is zero. If the value of the first argument is zero and the second argument is negative, the result is pi. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is pi/2. The arguments must not both have the value zero. The range of the result for ATAN2 and DATAN2 is −pi .LT. result .LE. pi.
17. LGE, LGT, LLE, and LLT are used to compare CHARACTER strings according to the ASCII collating sequence described in American National Standard Code for Information Interchange, ANSI X3.4-1977 (ASCII). Table 13-2 describes the conditions that return TRUE and FALSE values for these comparators. There is no intrinsic function for "lexically equal to" since the normal comparator (.EQ.) can be used whether the collating sequence is ASCII or EBCDIC.

Table 13-2. Truth Table for Lexical Comparators

Comparator	.TRUE.	.FALSE.
LGE(a1,a2)	a1 equals a2 a1 follows a2	a1 precedes a2
LGT(a1,a2)	a1 follows a2	a1 equals a2 a1 precedes a2
LLE(a1,a2)	a1 equals a2 a1 precedes a2	a1 follows a2
LLT(a1,a2)	a1 precedes a2	a1 equals a2 a1 follows a2

If the operands for LGE, LGT, LLE, and LLT are of unequal length, the shorter operand is essentially extended to the length of the longer operand with blank characters in the extended portion.

If either of the character entities being compared contains a character that is not in the ASCII character set, the result is dependent upon the B 1000 processor.

18. TIME(arg) returns a real value in units of seconds according to the value of the character constant arg, as shown in table 13-3.

Table 13-3. Values Returned by the TIME Function

Value of Character Constant arg	Value Returned
DAY	Current time of day
ELAPSED	Elapsed clock time for the program
PROCESSOR	Total processor time for the program

The DATE function can also obtain the current time of day.

19. DATE(arg) returns a CHARACTER*6 value of the current date according to the value of the character constant arg, as shown in table 13-4.

Table 13-4. Values Returned by the DATE Function

Value of Character Constant arg	Value Returned
MMDDYY	Current date in the form MMDDYY
YYMMDD	Current date in the form YYMMDD
YYDDD	Current date in the form YYDDD
HHMMSS	Current time of day in the form HHMMSS

20. RANDOM(arg) returns a uniformly distributed psuedo random number between 0.0 and 1.0. The argument must be an integer variable or integer array element. The function uses the initial integer value as a seed to begin random number generation and modifies the integer argument for any subsequent generation. For this reason, it is suggested that the argument not be the DO variable of a DO loop. Each reference to the RANDOM function in an arithmetic expression must be evaluated whenever the expression itself is evaluated.
21. AND(arg1,arg2) returns the logical product of all data bits in arg1 with all data bits in arg2. Each argument must be defined with hexadecimal data.
22. OR(arg1,arg2) returns the logical sum of all data bits in arg1 with all data bits in arg2. Each argument must be defined with hexadecimal data.
23. EQUIV(arg1,arg2) returns the logical equivalence of all data bits in arg1 with all data bits in arg2. The bit pattern returned has a 1 for every bit position in which the arguments are the same and a 0 for every bit position in which they are different. Each argument must be defined with hexadecimal data.
24. COMPL(arg) returns the logical complement of all data bits in the argument. The argument must be defined with hexadecimal data.

SUBROUTINE SUBPROGRAMS

A subroutine subprogram is a procedure external to the main program containing one or more subroutines. A subroutine subprogram has a SUBROUTINE statement as the first statement and can contain more than one entry point by the use of ENTRY statements (refer to ENTRY STATEMENT in this section). A subroutine subprogram can contain any statements except a FILE, PROGRAM, or BLOCK DATA statement. As with the main program, all declaration and specification statements must precede the first executable statement. A subroutine subprogram is terminated by an END statement.

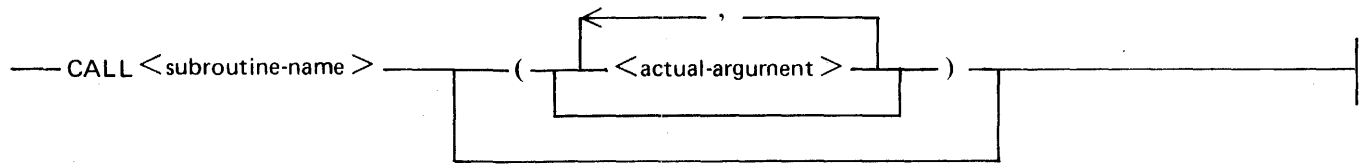
There are two types of subroutine subprograms. The first type is called a subroutine and is a subprogram provided by the user and is not supplied by FORTRAN 77. The second type is called an intrinsic subroutine and is a subprogram supplied by FORTRAN 77 for the user.

Subroutine

A subroutine is a program unit that can be referenced by the main program or any other program unit by using a CALL statement.

CALL Statement

The following is the proper form of the CALL statement:



G50344

<subroutine-name> is the symbolic name of a subroutine that exists and has been declared in a SUBROUTINE or ENTRY statement. <actual-argument> is described under Actual Arguments in this section.

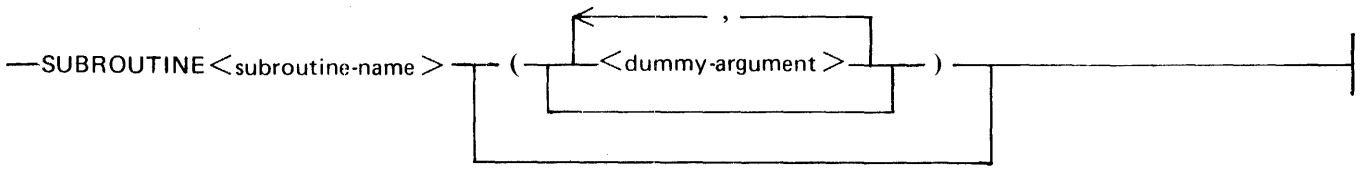
Execution of a CALL statement causes the following to occur:

1. Actual arguments that are expressions are evaluated.
2. Actual arguments are associated with the corresponding dummy arguments in the referenced subroutine.
3. Control is transferred to the specified subroutine.

A subroutine subprogram can call another subprogram, but cannot call itself either directly or indirectly.

SUBROUTINE Statement

The following is the proper form of the SUBROUTINE statement:



G50345

<subroutine-name> is the symbolic name of a subroutine conforming to the rules for variable names. There is no type associated with a subroutine name. <dummy-argument> is a dummy variable name, dummy array name, dummy procedure name, or an asterisk (*) character. Dummy arguments are described in this section. One or more dummy arguments of a subroutine can become defined or redefined to return results to the calling program unit.

Actual Arguments for a Subroutine

The actual arguments in a subroutine reference must agree in order, number, and type with the corresponding dummy arguments in the dummy argument list of the referenced subroutine. The use of a subroutine name or an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type.

An actual argument in a subroutine reference must be one of the following:

1. An expression (except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses, unless the operand is the symbolic name of a constant).
2. An array name.
3. An intrinsic function name.
4. An external procedure name.
5. A dummy procedure name.
6. An alternate return specifier of the form *s, where s is the statement label of an executable statement that appears in the same program unit as the CALL statement.

An actual argument in a subroutine reference can be a dummy argument name that appears in a dummy argument list within the subprogram containing the reference. An asterisk dummy argument must not be used as an actual argument in a subprogram reference.

Intrinsic Subroutines

Intrinsic subroutines are subroutines supplied by FORTRAN 77 for the user. These subroutines are Burroughs extensions to the guidelines of the American National Standards Institute committee for FORTRAN 77 (ANSI X3.9-1978).

Table 13-5 contains the intrinsic subroutines supplied by FORTRAN 77 and their proper form and semantics.

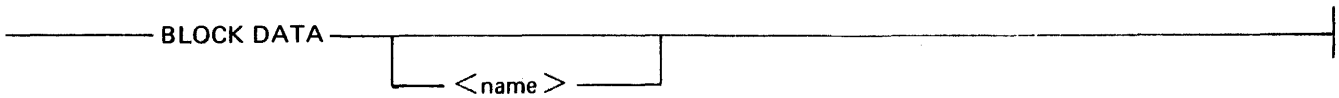
Table 13-5. Intrinsic Subroutines

Intrinsic Subroutine	Form	Semantics
Produce a dumpfile	CALL DUMP	Causes a dumpfile to be produced. After the dumpfile has been produced, execution continues with the statement following the CALL DUMP statement.
Terminate a program	CALL EXIT	Causes termination of a program as though a STOP statement had been executed.
Transfer bit values from one variable to another	CALL MVBITS (<a>,<b>,<i>, <j>,<k>)	<a> and <b> are variables or array elements of type integer, real, or logical. <i>, <j>, and <k> are integer expressions. The <k> bit subfield of <a> starting at the <i>th bit from the high order end of <a> is replaced by the <k> bit subfield of <b> starting at the <j>th bit from the high order end of <b>. The sums <i> plus <k> and <j> plus <k> must not be greater than the number of bits per storage unit plus 1.
Transfer a control string to the MCP	CALL ZIP (<string>)	<string> is an array or character expression containing a valid MCP control string. The CALL ZIP statement zips the control string to the MCP.

BLOCK DATA SUBPROGRAM

The block data subprogram is a nonexecutable program unit which has as the first statement a BLOCK DATA statement. The block data subprogram cannot contain any executable statements and initializes elements in labeled and unlabeled COMMON to predetermined values. More than one block data subprogram is allowed per executable program as long as an attempt is not made to initialize the same COMMON block twice.

The proper form of the BLOCK DATA statement is as follows:



<name> is the name of the block data subprogram, and its construction is governed by the same rules that apply to variable names (refer to Variable Names in section 5).

The block data subprogram is only a means whereby elements in COMMON storage can be initialized at compile time. These elements can be reassigned a value at any time during the execution of the program.

The statements following the BLOCK DATA statement define the elements in COMMON storage to be initialized, and only the explicit type COMMON, DIMENSION, EQUIVALENCE, IMPLICIT, PARAMETER, and DATA declaration statements can be used. An END statement must be the last statement in the subprogram.

The construction of block data subprograms is subject to the following restrictions:

1. A block data subprogram must contain at least one COMMON statement.
2. All elements of a COMMON block must appear in the COMMON statement list even though some of those elements are not to be initialized.
3. All type, dimension, initial values, or equivalent information associated with the variables or arrays in a COMMON block must be declared in the block data subprogram. A COMMON statement must also appear in the program unit referencing the COMMON block, as well as any declarations necessary to completely describe the entities referenced.
4. There can be only one unnamed block data subprogram in an executable program.
5. More than one COMMON block can be initialized by a block data subprogram.
6. Elements of any one COMMON block cannot be initialized by more than one block data subprogram.

Example:

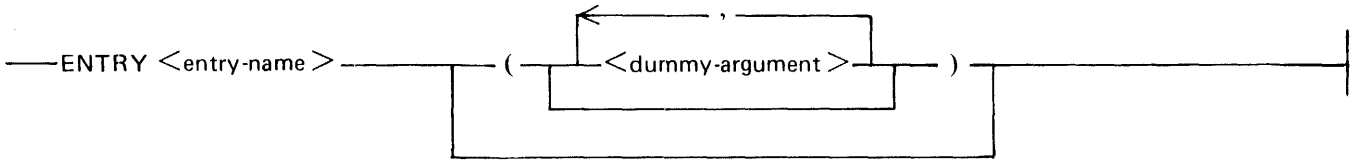
```
BLOCK DATA ALPHA
LOGICAL L1, L2
DOUBLE PRECISION D(2)
COMMON/BLOC1/I,R,L1/BLOC2/M,D,C,L2
DIMENSION R(3), M(2,2)
DATA D/2*1.92837465D0/
DATA I,R/456,2*1.56, 5.1/, L2/.TRUE./
END
```

In this example, elements in the COMMON blocks labeled BLOC1 and BLOC2 are to be initialized; therefore, all the elements in these blocks are listed in a COMMON statement. This is permissible, as more than one COMMON block can be initialized by a block data subprogram. All type and dimension information associated with the COMMON blocks is declared by the explicit type and DIMENSION statements. As required, the initial and last statement of the subprogram are, respectively, the BLOCK DATA and END statements.

ENTRY STATEMENT

The ENTRY statement permits a procedure to begin at a particular statement within the subprogram

The ENTRY statement permits a procedure to begin at a particular statement within the subprogram in which the ENTRY statement appears. The symbolic name of an entry in a subprogram, called the entry name, can be used in a calling program as the name of a subroutine entry if the entry declaration occurs within a subroutine subprogram, or as a function subprogram entry if the entry declaration occurs within a function subprogram. An ENTRY statement can occur anywhere following the SUBROUTINE or FUNCTION statement and preceding the END statement in the subprogram. The following is the proper form of the ENTRY statement:



G50346

<entry-name> is the symbolic name of the entry into the subprogram and it follows the same naming conventions as a variable. <dummy-argument> is the symbolic name of a variable, array, dummy procedure name, or (if the ENTRY occurs within a subroutine subprogram) an asterisk (*) character. An ENTRY statement in a function subprogram and the corresponding <dummy-argument> list follow the same rules as a FUNCTION declaration and is referenced in the same manner as a function subprogram. An ENTRY statement in a subroutine subprogram and the corresponding <dummy-argument> list follow the same rules as a SUBROUTINE declaration, and is referenced in the same manner as a subroutine subprogram.

There can be more than one ENTRY statement within a subprogram. The ENTRY statement can begin anywhere after the SUBROUTINE or FUNCTION declaration statement, except within a block IF statement or between a DO statement and the terminal statement of the DO loop. When control is passed to a subprogram by referencing the appropriate function name, subroutine name, or entry name, every ENTRY statement following the subprogram name referenced, within the same subprogram, is ignored for that reference of the subprogram.

A reference to an entry name in an expression (function entry) or in a CALL statement (subroutine entry) results in the following:

1. Evaluation of actual arguments that are expressions.
2. Association of actual and dummy arguments.
3. Transfer of control to the first executable statement after the specified <entry-name> within an existing function or subroutine subprogram.

An example of an ENTRY statement in a subroutine subprogram (subroutine entry) follows:

```
CALL CALC(A,B)
     .
     .
     .
END

SUBROUTINE XALG(X,Y,Z)
     .
     .
     .
ENTRY CALC(X,C)
     .
     .
     .
END
```

If an ENTRY statement occurs within a function subprogram, the symbolic name given in the ENTRY statement (the <entry-name>) is equivalenced to the FUNCTION name. An assignment to either name is equivalent and the names can be used interchangeably, if of the same type. The <entry-name> is not required to have the same type as the name of the function subprogram. The <entry-name> cannot appear in an executable statement preceding the ENTRY statement in which it is declared.

An example of an ENTRY statement in a function subprogram (function subprogram entry) follows:

```
      INTEGER EXPR1, EXPR2, X
           .
           .
           .
100   X=EXPR1 + EXPR2
           .
           .
           .
      END

      INTEGER FUNCTION EXPR1
      INTEGER EXPR2
200   I=2
      ENTRY EXPR2
300   I=I+1
      EXPR1=I
      END
```

In this example, the expression at line 100 makes two function calls. The first function call transfers control to line 200. The second function call transfers control to line 300. Variable X is finally assigned the value 7 (local variables retain values between subsequent references of the subprogram in which they occur).

ARGUMENTS AND COMMON BLOCKS

Arguments and common blocks provide means of communication between the referencing program unit and the referenced procedure.

Data can be communicated to a statement function or intrinsic function by an argument list. Data can be communicated to and from an external procedure by an argument list or common blocks. Procedure names can be communicated to an external procedure only by an argument list.

A dummy argument appears in the argument list of a procedure. An actual argument appears in the argument list of a procedure reference.

The number of actual arguments must be the same as the number of dummy arguments in the referenced procedure.

Dummy Arguments

Statement functions, function subprograms, and subroutine subprograms use dummy arguments to indicate the types of actual arguments and whether each argument is a single value, array of values, procedure, or statement label. A statement function dummy argument can be only a variable.

Each dummy argument is classified as a variable, array, dummy procedure, or asterisk (*) character. Dummy argument names can appear wherever an actual name of the same class and type can appear, except where explicitly prohibited.

Dummy argument names of type integer can appear in adjustable dimensions in dummy array declarators. Dummy argument names must not appear in EQUIVALENCE, DATA, PARAMETER, SAVE, INTRINSIC, or COMMON statements, except as common block names. A dummy argument name must not be the same as the procedure name appearing in a FUNCTION, SUBROUTINE, ENTRY, or statement function statement in the same program unit.

Actual Arguments

Actual arguments specify the entities that are to be associated with the dummy arguments for a particular reference of a subroutine or function. An actual argument must not be the name of a statement function in the program unit containing the reference. Actual arguments can be constants, symbolic names of constants, function references, expressions involving operators, and expressions enclosed in parentheses only if the associated dummy argument is a variable that is not defined during execution of the referenced function subprogram.

The type of each actual argument must agree with the type of the associated dummy argument, except when the actual argument is a subroutine name or an alternate return specifier.

Association of Dummy and Actual Arguments

At the execution of a function or subroutine reference, an association is established between the corresponding dummy and actual arguments. The first dummy argument becomes associated with the first actual argument; the second dummy argument becomes associated with the second actual argument, and so forth.

All appearances within a function or subroutine subprogram of a dummy argument, whose name appears in the dummy argument list of the procedure name referenced, become associated with the actual argument when a reference to the function or subroutine is executed.

A valid association occurs only if the type of the actual argument is the same as the type of the corresponding dummy argument. A subroutine name has no type and must be associated with a dummy procedure name. An alternate return specifier has no type and must be associated with an asterisk (*) character.

If an actual argument is an expression, it is evaluated just before the association of arguments takes place.

If an actual argument is an array element name, the subscript is evaluated just before the association of arguments takes place. The subscript value remains constant as long as that association of arguments persists, even if the subscript contains variables that are redefined during the association.

If an actual argument is a character substring name, the substring expressions are evaluated just before the association of arguments takes place. The value of each of the substring expressions remains constant as long as that association of arguments persists, even if the substring expression contains variables that are redefined during the association.

If an actual argument is a function subroutine name, the procedure must be available at the time it is referenced.

If an actual argument becomes associated with a dummy argument that appears in an adjustable dimension, the actual argument must be defined with an integer value at the time the procedure is referenced.

A dummy argument is undefined if not currently associated with an actual argument. An adjustable array is undefined if the dummy argument array is not currently associated with an actual argument array, or if any variable appearing in the adjustable array declarator is not currently associated with an actual argument and is not in a common block.

Argument association can be carried through more than one level of procedure reference. A valid association exists at the last level only if a valid association exists at all intermediate levels.

Argument association within a program unit terminates at the execution of a RETURN or END statement in the program unit. There is no retention of argument association between one reference and the next of a subprogram.

Length of Character Dummy and Actual Arguments

If a dummy argument is of type CHARACTER, the associated actual argument must be of type CHARACTER and the length of the dummy argument must be less than or equal to the length of the actual argument. If the length (len) of a dummy argument of type CHARACTER is less than the length of an associated actual argument, the leftmost len characters of the actual argument are associated with the dummy argument.

If a dummy argument of type CHARACTER is an array name, the restriction on length is for the entire array and not for each array element. The length of an array element in the dummy argument array can be different than the length of an array element in an associated actual argument array, array element, or array element substring, but the dummy argument array must not extend beyond the end of the associated actual argument array.

If an actual argument is a character substring, the length of the actual argument is the length of the substring. If an actual argument is the concatenation of two or more operands, the length is the sum of the lengths of the operands.

Table 13-6 shows the types of actual arguments that can be associated with a given dummy argument.

Table 13-6. Association of Actual and Dummy Arguments

Dummy Argument	Actual Argument
Simple Variable	A constant, variable array element, substring, or expression of the same type.
Array Name	Array name, array element, or array element substring.
Procedure Name	Intrinsic function, external procedure, or dummy procedure.
Asterisk (*) Character	*n (where n is a statement label).

Variables as Dummy Arguments

A dummy variable must be associated with an actual argument that is a constant, variable, array element, substring (simple or array element), or expression of the same type as the dummy variable. The actual argument and the associated dummy argument must always have the same type.

If the dummy argument is of type CHARACTER, the length of the dummy argument must be less than or equal to the length of the actual argument. If the length of the dummy argument is less than the length of the actual argument, the rightmost len − n characters of the actual argument are ignored, where len is the length of the actual argument and n is the length of the dummy argument. The length of a dummy character argument is constant for every invocation of the subprogram if the length specification is a constant or integer constant expression. If the length specification is an integer expression containing variables, the value of the expression is determined with each invocation of the subprogram. This value becomes the length of the character entity for the duration of the subprogram. If the length of the dummy character argument (or array) is assumed, the dummy argument (or array) assumes the length of the actual argument. If the actual argument is a constant, substring, or character expression, the number of characters in the constant, substring, or expression is the length of the actual argument.

An example of the use of actual arguments and dummy arguments follows:

```
                .
                .
                .
      X=  A(3.,Y,(Z),4.+FUNB(3))
                .
                .
                .
      REAL FUNCTION A(B,C,D,E)
                .
                .
                .
100   D=9.34
200   C=D-1.10
                .
                .
                .
      END
```

In this example, Y is the only actual argument that can have its value changed by the called subprogram: When D is assigned a value at line 100, the assignment is local to the function and has no effect on the calling program unit. When C is assigned a value at line 200, the value of Y is also affected in the calling program.

Arrays as Dummy Arguments

The actual argument associated with a dummy array must be an array name, an array element, or an array element substring. If the actual argument is an array element, it is an index into the actual array, and that portion of the actual array from the element specified to the end of the array is passed to the dummy array. In this case, the dummy array begins at the element specified by the actual parameter. When variables are used in the expression of the subscripts of the actual array element, the index into the actual array is determined at the time of the call and changes in value of these variables during execution of the subprogram have no effect on the association of the dummy and actual arrays. Any attempt to reference an element that exceeds the declared bounds for the dummy array causes program termination at the point where the reference occurs.

A dummy array need not have the same number of dimensions or the same number of elements as the corresponding actual array; however, a dummy array must not be associated with an actual array in such a way as to extend beyond the last element of the actual array.

Association of arrays of differing sizes and dimensions is done according to array element ordering, as described in appendix D. The following paragraphs describe the correspondence that is established between the actual argument and the dummy argument in the subprogram at the time the subprogram reference is executed.

Numeric Arrays

If the dummy array is a constant dimension array (refer to Types of Arrays in section 5), it must not have more elements than the actual array. If the actual parameter is an array element, the initial element of the dummy array is associated with the element specified by the actual parameter. If, for instance, the actual array is dimensioned A(4,5), the dummy array is dimensioned B(−3:1,2), and the CALL appears as:

```
CALL EXAMP(A(2,3))
     .
     .
     .
END

SUBROUTINE EXAMP(B)
DIMENSION B(-2:1,2)
     .
     .
     .
END
```

The following list shows the storage sequence for the portions of the two arrays which are associated and their correspondence:

A(2,3)	A(3,3)	A(4,3)	A(1,4)	A(2,4)	A(3,4)	A(4,4)	A(1,5)
B(-2,1)	B(-1,1)	B(0,1)	B(1,1)	B(-2,2)	B(-1,2)	B(0,2)	B(1,2)

As can be seen from this list, if the dummy array is dimensioned to have less elements than the actual array, references into the actual array that exceed the bounds of the dummy array are not permitted. A(2,5) in the above example can not be referenced by dummy array B.

If the dummy array is an adjustable array, the variables given in the dimension bounds for that array must appear in a common block in the subprogram containing the adjustable array, or in the same dummy parameter list as the adjustable array in which they are used. When a reference is made to the subprogram containing the adjustable array in the parameter list, the values of the variables used in the bounds declaration of the adjustable array are evaluated. These values are then used to determine bounds for the adjustable array for that reference of the subprogram. When the bounds of the adjustable array are determined, the number of elements in the adjustable array must not exceed the number of elements of the actual array or that portion of the actual array which is passed. Every time the adjustable array is referenced, the bounds of each subscript are checked. If any of the dimension bounds are exceeded, an error occurs.

Example:

```
DIMENSION A(-5:0,0:9)
DATA I/6/
CALL JUST(A(-2,8),I)
.
.
I=15
CALL ADJUST(A(-1,0),I)
END

100 SUBROUTINE JUST(B,J)
DIMENSION B(J)
.
.
200 ENTRY ADJUST(B,J)
.
.
END
```

When the subprogram in this example is entered through statement 100 (CALL JUST), array B has 6 elements and is associated with array A beginning at element A(−2,8) (then A(−1,8), A(0,8), A(−5,9), and so forth). When the program is entered at statement 200 (CALL ADJUST), array B is associated with 15 elements in array A beginning with element A(−1,0) (then A(0,0), A(−5,1), and so on), and ending with element A(−4,3).

If the dummy array is an assumed size array, the declaration contains an asterisk (*) character as the declaration for the upper bound of the final dimension. The dimension bounds that precede the final dimension can contain constants or variables as in a constant array or an adjustable array. The product of the sizes of the dimensions (dimension size = upper bound − lower bound + 1) that precede the dimension containing the asterisk must not exceed the size of the actual array or that portion of the actual array which is passed.

Example:

```
DIMENSION A(15)
X=ASSUME(A(3))
.
.
END

FUNCTION ASSUME(B)
DIMENSION B(4,4,-3:*)
.
.
END
```

This example generates an error because the product of the sizes of the dimensions in B that precede the dimension containing the asterisk (*) character is greater than the number of elements passed in the call (4*4>13).

The assumed-size array can be dimensioned differently from the actual array with which it is associated. The upper bound for the final dimension is the smallest positive number that enables the dummy array to contain every element of the actual array that is passed.

Example:

```
DIMENSION A(3,4)
CALL ASSUME(A)
      .
      .
      .
END

SUBROUTINE ASSUME(B)
DIMENSION B(3,3,*)
      .
      .
      .
END
```

In this example, the smallest number replacing the asterisk (*) character that enables B to contain all the elements of A is 2. This does not infer that all elements of B that contain a 1 or a 2 in the final

dimension subscript can be referenced. Only those elements of B that are associated with an element of A can be referenced during execution of the subprogram. The association between arrays A and B can be illustrated as follows:

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

Array elements A(1,1) and B(1,1,1) reference the same storage location, array elements A(2,1) and B(2,1,1) reference the same storage location, and so on. Referencing B(3,2,2) is invalid in the previous example. When a reference is made to an assumed-size array, the bounds of each subscript are checked, and the final displacement into the dummy array is also checked to determine whether a reference is being made beyond the end of the actual array.

The following is an example of a constant dimension, adjustable dimension, and assumed-size dimension in a single array:

```
      DIMENSION A(-10:0,3,-1:1,-4:-3)
      I=2
100   CALL JUMBLE(A(-7,2,-1,-4),I)  Pass 184 of 198 element array
        .
        .
        .
      END

      SUBROUTINE JUMBLE(B,J)
      DIMENSION B(4,-3:J,0:*)        Upper bound of last dimension is 7. B(4,0,7)
                                     is the last element of B that can be referenced.
```

In this example, the value of the last dimension of B is 7. However, only the following elements of B with 7 as the last dimension can be referenced:

```
B(1,-3,7) B(2,-3,7) B(3,-3,7) B(4,-3,7) B(1,-2,7)
B(2,-2,7) B(3,-2,7) B(4,-2,7) B(1,-1,7) B(2,-1,7)
B(3,-1,7) B(4,-1,7) B(1,0,7) B(2,0,7) B(3,0,7) B(4,0,7)
```

This list gives the storage sequence for the largest subscript in the final dimension. B(4,0,7) is the final storage location of array B when referenced at line 100.

Character Arrays

A dummy array can have a character type as well as one of the numeric types. A dummy array can be a constant array, adjustable array, or an assumed-size array. In addition, the length of each element can be fixed or assumed (refer to Character Type Statements in section 6). If the length of the elements in the actual and dummy arrays are not the same length, the dummy and actual array elements do not consist of the same characters, but an association still exists.

When a subprogram is invoked, the length of an element of the dummy array is first determined. If the length contains an expression, the value of the expression is determined and becomes the length of an element of the array during execution of the subprogram. If the length is assumed, the length of an element becomes the length of the corresponding actual array element or array element substring if the actual argument is an array element substring.

Next, the number of elements in the array is determined by inserting values for any expressions in adjustable dimensions. For a type CHARACTER, the number of elements in the dummy array can exceed the number of elements in the actual array. However, the number of character storage locations in the dummy array can not exceed the number of character storage locations in the actual array. The formula for determining the number of character storage locations in a character dummy array is similar to the formula for determining the number of elements in a numeric array. It is the product of the sizes of the dimensions multiplied by the length of an individual element.

Example:

```
CHARACTER * 4 A(6,5)
```

The size of the dummy array in this example is 120 characters. The actual array with which it is associated must have at least 120 characters.

In an assumed-size dummy character array (upper bound of the final dimension contains an asterisk (*) character), the product of the sizes of the dimensions before the dimension containing the asterisk, multiplied by the length of an individual character, must not exceed the number of character storage locations passed. The number of elements in an assumed-size dummy character array is determined by the following formula:

n= INT(<c>/<len>)

<c> is the number of character storage locations passed and <len> is the length of an element of the dummy array.

If the actual parameter is an array element, the subprogram reference passes the memory address of that element, the element length, and the number of bytes from the element to the end of the array. If the actual parameter is an array element substring, the subprogram reference (as in this case) also passes the address of the substring, along with the length of the substring, and the number of bytes from the beginning of the substring to the end of the array.

Example:

```
CHARACTER * 5 A(3)
CALL REMAP1(A(1)(2:4))
CALL REMAP2(A(2)(1:3))
CALL REMAP3(A,5)
      .
      .
      .
END

SUBROUTINE REMAP1(B)
CHARACTER * 3 B(2,2)
CHARACTER * (*) C(2,*)
CHARACTER * (I) D(2)
      .
      .
      .
ENTRY REMAP2(C)
      .
      .
      .
```

```
      ENTRY REMAP3(D,I)
            .
            .
            .
      END
```

When this subroutine is referenced through REMAP1, actual array A is associated with dummy array B. The storage locations in array A, starting with storage location A(1)(2:2), are associated in groups of three (the length of an element in B) with the elements of array B. The following list shows the association between the elements of array B and the storage locations in array A:

B(1,1) with A(1)(2:4)
B(2,1) with A(1)(5:5) // A(2)(1:2)
B(1,2) with A(2)(3:5)
B(2,2) with A(3)(1:3)

Any change made to an element of array B during this reference of the subroutine is also a change to the corresponding locations in array A.

When the subroutine is referenced through REMAP2, dummy array C is associated with actual array A. Array C is an assumed-size array and also has an assumed length for its elements. In this case, the elements of C take on the length of the substring passed (3). The substring in the actual parameter list is also an index into array A, and the initial storage location of C is the storage location of A containing the value F before the subroutine reference. The value of the final dimension of C must be large enough to allow C to contain all character storage locations in A (there are 10 passed) but not allow C to extend beyond the end of A. Therefore, the value of the final dimension of C must be 2, since this is the smallest number that allows C to contain all of the locations passed in the CALL. However, not all of the subscripts within these bounds can be referenced. The character storage locations in A are associated in groups of three with the elements of C in the following manner:

C(1,1) with A(2)(1:3)
C(2,1) with A(2)(4:5) // A(3)(1:1)
C(1,2) with A(3)(2:4)

The element C(2,2) cannot be referenced because C would extend beyond the displacement bound of A by two character storage locations. If, in the previous example, the first dimension bound of C is 4, a run-time error would occur because 4 (number of elements preceding the asterisk (*) character) multiplied by 3 (the length of an element) equals 12, which is greater than 10 (the number of characters passed).

When the subroutine is referenced through REMAP3, the storage locations in array A are associated with array D. The element length of array D is also passed when REMAP3 is invoked. In the previous example, D obtains a length of five characters for each element in D. Element D(1) points to the same memory locations as A(1), and D(2) points to the same memory locations as A(2) when REMAP3 is invoked.

Procedures as Dummy Arguments

If the actual argument associated with a dummy argument is the name of an external procedure, the dummy argument name can be used to reference the procedure (subprogram) during the subprogram. Refer to the EXTERNAL Statement in section 6.

Dummy Arguments in ENTRY Subprograms

The same rules apply to the dummy arguments in the ENTRY statement as to the dummy arguments in a SUBROUTINE or FUNCTION declaration statement. In addition, a dummy argument appearing in the dummy argument list of an ENTRY statement cannot appear in a preceding executable statement unless it also appears in a dummy argument list in an ENTRY, SUBROUTINE, or FUNCTION statement prior to the executable statement in question. All declarations associated with dummy arguments, including dummy arguments appearing in ENTRY statements only, must be given before the first executable statement of the subprogram unit.

Example:

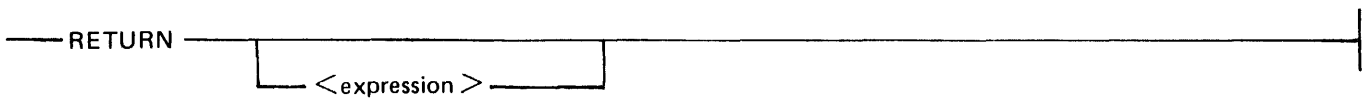
```
      SUBROUTINE ALPHA
            .
            .
            .
100   A=5.2
            .
            .
            .
      ENTRY OMEGA(A)
            .
            .
            .
      END
```

In this example, the variable A is a dummy argument of the ENTRY statement OMEGA. The use of A at statement 100 is not valid since it has not yet been declared in a dummy argument list. When A is referenced at statement 100, it is considered to be a local variable. The later use of A as a dummy argument in the ENTRY statement is ambiguous, since A cannot be both a local variable and a dummy argument within the same subprogram.

RETURN STATEMENT

The RETURN statement is a control statement provided to specify the manner in which control is returned to the calling program unit following the execution of a subprogram.

The proper form of the RETURN statement follows:



G50347

<expression> is an integer expression specifying an alternate return.

A function or subroutine subprogram can cause termination of program execution by the execution of a STOP statement, or return of control to the calling program unit by execution of a RETURN statement or END statement. Use of a RETURN statement in a main program is prohibited. The point in the calling program at which execution resumes is determined by the form of the RETURN statement employed. The RETURN statement has two forms: standard return, and alternate return.

Standard Return

The execution of a RETURN statement of the first form causes a standard return to the calling program unit. For a subroutine, the first executable statement following the CALL statement which invoked the subroutine is executed next. For a function, FORTRAN 77 resumes evaluating the rest of the expression that contains the function reference in the calling program unit.

If the END statement of the subprogram is encountered before a RETURN statement is executed, a standard return is performed.

Alternate Return

The second form of the RETURN statement allows control to be returned to a specified, labeled, executable statement in the calling program unit. An alternate return from a function is not allowed.

An alternate return is indicated in a subprogram by a RETURN statement followed by an arithmetic expression. The value of this expression must be an integer. The integer value, n, is used to select the n-th asterisk (*) character in the dummy argument list of the SUBROUTINE. If n is greater than the number of asterisk (*) characters in the list, or if n is less than or equal to zero, a standard return is performed. These asterisks in the dummy argument list are referred to as alternate return specifiers.

The statement label corresponding to the selected asterisk (*) character is specified in the actual argument list, preceded by an asterisk and is used to identify the statement to which control is to be returned. The actual argument list of the CALL statement must contain such a statement label in each position where the dummy argument list of a SUBROUTINE statement or ENTRY statement contains an asterisk.

An example of alternate returns follows:

```
      CALL SUBA(A1,*33,A2,*20,II)
15    Z=A1 + II
20    Z=Z + A2
33    Z=Z+A1
```

The called subroutine could be the following:

```
      SUBROUTINE SUBA (A,*,B,*,J)
      IF (A) 5,6,7
5     RETURN 1
6     J=J+B
      RETURN INT(A+2)
7     RETURN
      END
```

In this example, suppose that when the IF statement in the subroutine is reached, A is positive. A branch is made to statement number 7, which is a normal return. The subroutine then returns to the calling program unit at the statement following the CALL, statement number 15.

If A is less than zero when the IF statement in the subroutine is reached, there is a branch to statement number 5. This is an alternate return to the first label in the actual argument list, since RETURN 1 selects the first asterisk (*) character in the dummy argument list. The return in the calling program unit is to statement number 33.

If A is equal to zero when the IF statement in subroutine SUBA is executed, there is a branch to statement number 6. Since A = 0 when the arithmetic expression is evaluated, the statement becomes RETURN 2. This causes a return to statement number 20 in the calling program unit.

A CALL statement that results in an alternate RETURN can be regarded as a CALL followed by a computed GO TO, shown as follows:

```
CALL SUBA(A1,*33,A2,*20,II)
```

This statement can be treated as an equivalent replacement for:

```
CALL SUBA(A1,A2,II,JUMP)
GO TO(33,20),JUMP
```

JUMP is assigned a value of 1 or 2 (or some other value if return is to be standard) by the subroutine.

SECTION 14
COMPILER CONTROL IMAGES

This section describes Compiler Control Images (CCIs): instructions with which the user controls the options provided by the FORTRAN 77 compiler. This section also describes the function of the CCI-controlled files used by the FORTRAN 77 compiler.

A CCI consists of a dollar sign ($) character in column 1 and one or more options with the associated parameters, if any, separated by blanks in the next 71 columns of the line (columns 2 through 72). A second but optional dollar sign ($) character in column 2 of the line specifies that the line upon which a CCI appears is written to a CCI-controlled file labeled NEWSOURCE. The use of this file is explained more fully in the explanation of the NEW option in this section. A CCI can appear on any line within the source code of a FORTRAN 77 program and affects the compilation only after the point where encountered.

TYPES OF OPTIONS

Compiler Control Images consist of three types of options: boolean, immediate, and value. These types of options are described next.

A boolean option is one which is either SET (enabled or TRUE) or RESET (disabled or FALSE). When SET, a boolean option causes the FORTRAN 77 compiler to apply an associated function to all subsequent processing until disabled. The following commands specify the condition of a boolean option:

SET
Enables or sets the option (TRUE). The use of SET in a CCI is optional.

RESET
Disables or resets the option (FALSE).

POP
Each time an option is SET or RESET, its value (TRUE or FALSE) is placed on a stack associated with the option. All options have a default setting which is the first value on the stack. A POP command discards the current setting of the boolean and the value (TRUE or FALSE) on the top of the stack associated with the specified option becomes the new setting for the option.

Examples of the SET, RESET, and POP commands follow:

```
$SET AUTOBIND OPT
$RESET OMIT
$POP DELETE OMIT RESET SEQCHECK
```

In the examples, SET enables AUTOBIND and OPT (stack value TRUE). RESET disables OMIT (stack value FALSE). POP discards the current setting for DELETE and OMIT and uses the top entry on the stacks associated with each to obtain a new setting. For example, if OMIT was SET before RESET in the second example, OMIT would be SET again. The RESET in the third example affects only SEQCHECK.

A SET, RESET, or POP command affects every boolean option on the same card image until the next SET, RESET, or POP command on the same record. A SET, RESET, or POP appearing on the same record as a nonboolean option has no effect on that option.

Immediate options cause the compiler to perform the specified action when the option is encountered. These options can have parameters. Immediate options are only in effect during the period required to perform the desired task. For example, the PAGE option immediately advances the paper to the beginning of the next page of the printer listing.

Example:

```
$PAGE
```

A value option causes the FORTRAN 77 compiler to store a value associated with a given function. A value option must be followed by a space or an equal sign (=) character and then by the quantity desired.

Examples:

```
$DYNAMIC=520
$REMOVEICM MYICM
```

The options described in the remainder of this section are organized according to function, not according to the type of option. The type of option is designated in each option description. The groupings for these options are Limiting Options, Source Input Options, Source Output Options, Intermediate Code Module Options, and Miscellaneous Options.

LIMITING OPTIONS

The following CCI options specify various limits for the current compilation.

DYNAMIC

DYNAMIC is a value option that specifies the size, in words, to be assigned to the dynamic memory of an object program. Each data page of dynamic memory contains 256 words and the total number of data pages allowed for an object program is 1024. By default, the compiler assigns an amount of dynamic memory equal to the sum of all the data pages. Therefore, DYNAMIC can only be used to reduce the size of dynamic memory assigned to the object program. Reducing the size of dynamic memory can be used with an object program that cannot obtain enough system memory to be executed. Reducing the size of dynamic memory can also be accomplished by modifying the MEMORY attribute of the object file (refer to the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982). When dynamic memory is reduced from the default size, the need for memory overlays can increase the execution time of the program. Refer to Data Allocation Information in appendix D for a description of how data is allocated to dynamic memory.

Example:

```
$DYNAMIC = 512
```

This example, although assigning a value equal to two data pages, cannot contain two full pages at the same time because a word from the total dynamic memory of the program is used to link memory pages together.

ERRORLIMIT

ERRORLIMIT is a value option that specifies the number of syntax errors the compiler can encounter in the source program before terminating the compile. If the ERRORLIMIT option is not present, the quantity of syntax errors does not cause the compile to terminate.

Example:

```
$ERRORLIMIT 5
```

STACKSIZE

STACKSIZE is a value option that specifies the size of the subprogram parameter stack. The parameter stack contains the addresses or values of all actual arguments being passed to subprograms, as well as the subprogram return addresses. Arguments are placed on the stack when a subprogram is referenced and popped off the stack when the subprogram is exited. Only those arguments that were placed on the stack when the subprogram was entered are popped off when the subprogram is exited. By default, the value of STACKSIZE is set to the maximum possibly required.

Example:

```
$STACKSIZE 20
```

This example would allow the stack to contain a maximum of 20 entries. Memory space is conserved by making the stack as small as possible.

SOURCE INPUT OPTIONS

The following options describe the form the source input to the compiler is to take. These options, except OMIT and SEQCHECK, are only recognized by the compiler if MERGE is set.

DELETE

DELETE is a boolean option that inhibits all source images from the file named SOURCE beginning at the specified line number from being compiled into the object file until the option is reset. All source images that are deleted are not written to the file named NEWSOURCE if NEW is set. DELETE is ignored if MERGE is disabled. The DELETE option is reset by default.

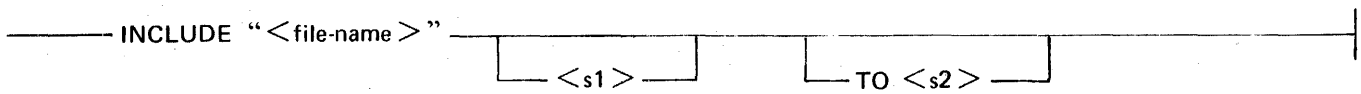
Example:

```
$SET DELETE                    00001000
$RESET DELETE                  00002000
```

If a line number appears in columns 73-80 of the record, the compiler begins deleting source images at the line where DELETE is SET, and stops deleting where the DELETE option is RESET. If no line number appears on the card image that sets the DELETE option, the action begins at the current line.

INCLUDE

INCLUDE is an immediate option that specifies a range of lines from another source file that is to be compiled and inserted at the point where the INCLUDE option occurs. The INCLUDE option has the following form:



<s1> is the beginning line number and <s2> is the ending line number in the range of lines from the file <file-name> to be included in the compilation at the point where the INCLUDE option occurs. If a range is not specified, all records from the file are included. If <s1> is omitted, the range of lines from the start of the file to <s2> are included. If TO <s2> is omitted, the range of lines from <s1> to the end of the file are included.

Example:

```
$INCLUDE "(MY)/LIBRARY" 1200 TO 1500
```

MERGE

MERGE is a boolean option that causes the compiler to combine a secondary source input file with the primary source input file during compilation. The MERGE option has the following form:

G50349

<file-name> is the name of the file to be merged with the primary input file. If <file-name> is not specified, a file named SOURCE is searched for unless otherwise pre-empted by a label equation. The MERGE option must be specified in the primary source file named CARD. The MERGE procedure is as follows:

1. The compiler reads a source card image from each file (CARD and SOURCE) examining the line numbers in columns 73 – 80.
2. The card image with the lower line number is processed by the compiler.
3. The compiler continues by reading the next card image from the file that previously had the lower line number, and compilation continues with step 2.
4. If a card image from the primary file (CARD) does not have a line number, it is immediately processed without comparison with the current line in SOURCE.

Example:

```
?DATA CARD
$SET MERGE
$SET DELETE                                    00001400
      IF (A.LE.7.) GO TO 10                    00001500
$RESET DELETE                                  00001600
?END
```

This example ignores all input from file SOURCE from line 1400 through 1600. When the line immediately preceding line 1400 in file SOURCE has been compiled, the next statement to be compiled is line 1500 in file CARD.

If the line numbers of the two source card images are equal, the primary source card image is used and the secondary source card image is discarded. The MERGE option is reset by default.

OMIT

OMIT is a boolean option that causes all source input from file SOURCE to be ignored beginning with the line number of the OMIT card image or with the next card image from the file SOURCE if the OMIT card does not contain a line number. Card images from file SOURCE are ignored until the OMIT option is reset. When the OMIT option is reset, card images are again used from file SOURCE beginning at the next line number greater than the line number containing the RESET OMIT. The OMIT option can be on a card image in the primary source file or in the secondary source file. Unlike the DELETE option, the source images that were ignored are written to the file NEWSOURCE if the NEW option is set.

```
$SET OMIT                                    00001000
      X=X+2                                  00001200
      X=X**2                                 00001300
$RESET OMIT                                  00002200
```

The OMIT option is reset by default.

SEQCHECK

SEQCHECK is a boolean option that causes the compiler to verify that line numbers of source input to the compiler are in ascending order. When merging, the compiler looks at the line number of the next source card image chosen for compilation and compares it with the line number of the last card image that was compiled. If the input is out of sequence, a sequence error is generated.

Example:

```
$SET SEQCHECK
```

SEQUENCE

SEQUENCE (or SEQ) is a boolean option that causes the compiler to assign line numbers to source card images written to file NEWSOURCE. This includes those card images that were omitted because the OMIT option was set. The values assigned to the line numbers are dependent on the setting of the sequence range options. The SEQUENCE option is reset by default.

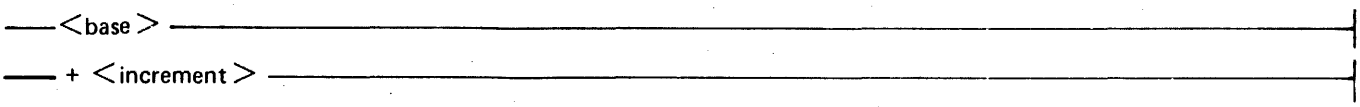
Example:

```
$SET SEQ
```

SEQUENCE Range Options

The SEQUENCE range options are value options that specify the values of the line numbers assigned

The SEQUENCE range options are value options that specify the values of the line numbers assigned to the source output file NEWSOURCE. The SEQUENCE range options can appear anywhere in the primary or secondary source file, before or after the SEQUENCE option and can appear more than once. The following are the proper forms of the SEQUENCE range options:



G50350

<base> is an integer constant which is the line number of the next record to be written to file NEWSOURCE. The base is 1000 by default when the SEQUENCE option is encountered. The + <increment> specifies the amount to add to the base line number after each line is written to NEWSOURCE. <increment> is 1000 by default.

Examples:

1. $10 +10

   $SET SEQ

2. $SEQ 10 +10 00001000

These two examples are equivalent. The SEQUENCE range options must appear on a CCI so as not to be associated with any other option requiring an integer or plus sign (+) integer as a parameter or a value.

VOID

VOID is an immediate option that causes the compiler to discard all source input from file SOURCE starting at the line following $VOID and continuing until the line number in file SOURCE exceeds the specified line number on the VOID option. VOID is ignored if MERGE is disabled.

Example:

$VOID 1000 00000010

This example discards lines 11-1000 of file SOURCE.

SOURCE OUTPUT OPTIONS

The following options affect the format of the printer listing of the source output generated by the compiler.

DOUBLE

DOUBLE is a boolean option that produces a double-spaced listing of the source program from the point where the option is encountered. This option is reset by default.

Examples:

```
$SET DOUBLE
$PAGE RESET DOUBLE
```

The second example demonstrates that two options can appear on the same line.

INCLNEW

INCLNEW is a boolean option that causes the compiler to write any source language statements following the INCLUDE option to the file NEWSOURCE when NEW is enabled. INCLNEW is reset by default.

Example:

```
$SET INCLNEW
```

LIST

The LIST option causes the compiler to produce a listing of the source program being compiled. This is a boolean option which is set by default.

Example:

```
$SET LIST
```

LISTDELETED

LISTDELETED is a boolean option that causes the compiler to list all source language input deleted by enabling the DELETE or VOID options. This option is reset by default.

Example:

```
$SET LISTDELETED
```

LISTINCL

LISTINCL is a boolean option that causes the compiler to list all source language input which was accepted for compilation as a result of enabling the INCLUDE option. This option is reset by default.

Example:

```
$SET LISTINCL
```

LISTOMITTED

LISTOMITTED is a boolean option that causes the compiler to list all source language input which was omitted by enabling the OMIT option. This option is reset by default.

Example:

```
$SET LISTOMITTED
```

LISTP

LISTP is a boolean option that causes the compiler to list those source images that are input from the file CARD. If LIST is set, this option has no effect. LISTP is reset by default.

Example:

```
$SET LISTP
```

LISTDOLLAR

LISTDOLLAR is a boolean option that causes the compiler to list all Compiler Control Images during compilation. LISTDOLLAR is reset by default.

Example:

```
$SET LISTDOLLAR
```

MAP

MAP is a boolean option that causes the compiler to include, as part of the output listing, information concerning the allocation of variables within the object code produced by the compilation process. MAP is reset by default.

Example:

```
$SET MAP
```

NEW

NEW is a boolean option that causes the compiler to output the source created by the MERGE process to a file named NEWSOURCE or optionally, to the file named after the keyword NEW. This option is ignored if MERGE is disabled. NEW is reset by default.

Example:

```
$SET NEW
$SET NEW FILEA
```

PAGE

PAGE is an immediate option that causes the listing to skip to the beginning of the next page.

Example: $PAGE

SUMMARY

SUMMARY is a boolean option that causes the compiler to produce a summary of appropriate information about the compilation on the output listing. SUMMARY is reset by default.

Example:

```
$SET SUMMARY
```

XREF

XREF is a boolean option that causes the compiler to print a cross-reference listing of symbolic names and statement labels of the source input. Cross-referenced names are associated by line number unless the XSEQ option is used.

Example:

```
$XREF
```

XSEQ

XSEQ is a boolean option used in conjunction with the XREF option that causes the cross-referenced names to be associated by sequence number.

Example:

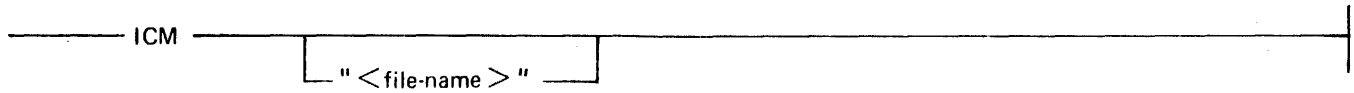
```
$XSEQ
```

INTERMEDIATE CODE MODULE OPTIONS

An Intermediate Code Module (ICM) is a code module generated by the compiler from the compilation of the main program or any subprogram. The code generated from the compilation of each syntactically correct program unit is written to a temporary file named ICM. The code from each program unit is contained within a code module within file ICM. The following options permit the user to save and reuse selected ICMs that are compiled without syntax errors: ICM, REMOVEICM, and USEICM.

ICM

ICM specifies that file ICM is to be saved, and optionally, gives a name for the file in which it is to be saved. The ICM option has the following form:



G50351

If <file-name> is specified, the ICM option creates a library file with the name <file-name>; otherwise, the ICM option creates a library file with the name .SBRTN. The library file contains all syntactically correct program units from the file ICM. The names of the ICMs in the library file are the same as the names of the subprograms. The names of any main programs are the names given in a PROGRAM statement, or OMAIN by default. If specified, ICM must appear before any FORTRAN 77 language statements and any USEICM Compiler Control Images.

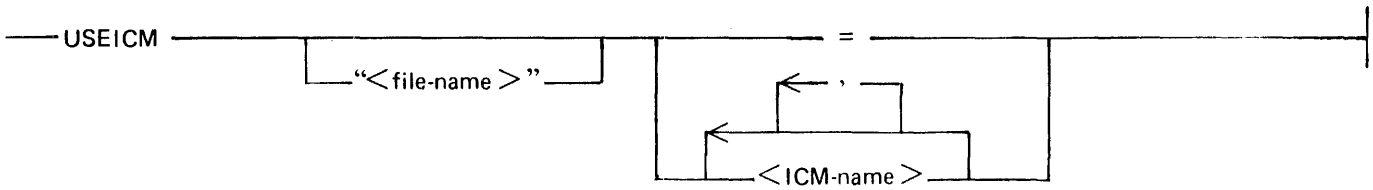
An example of the ICM option follows:

```
$ICM    "SUBCODE"
        PROGRAM CALCUL
              .
              .
              .
        SUBROUTINE GEOM (I)
              .
              .
              .
        REAL FUNCTION TRIG
              .
              .
```

This example creates an ICM library file named SUBCODE which contains three ICMs (if the entire program is syntax-error free): CALCUL, GEOM, and TRIG. Since AUTOBIND is set by default, the ICMs are also bound together into an executable code file. If AUTOBIND is reset, no attempt is made to bind the program modules together and the source file need not contain a main program (CALCUL in the above example).

USEICM

USEICM specifies an ICM library code file containing subprograms to be used (all or part) by the current compile. The USEICM option has the following form:



G50352

<file-name> is the name of the ICM library file on disk. Each <ICM-name> specified is regarded as a syntax-free code module, which is written to file ICM being created by the current compile. The equal sign (=) character specifies that all code modules in <file-name> are to be used by the current compile. No subprogram or main program being compiled can have the same name as a code module brought in by the USEICM option.

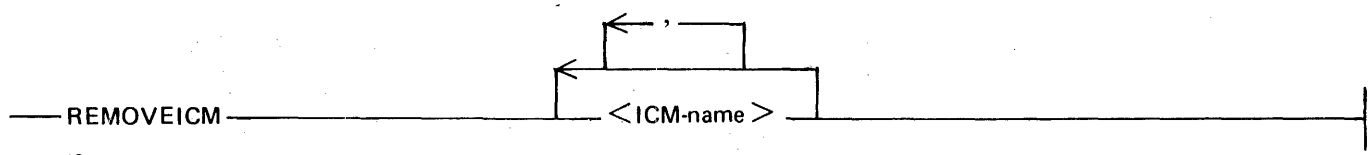
An example of the USEICM option follows:

```
$ICM "CONTROL"
$USEICM "SUBCODE" CALCUL, GEOM
INTEGER FUNCTION TRIG
      .
      .
      .
```

In this example, CALCUL (from the example under ICM) is used as the main program when the program is bound together.

REMOVEICM

REMOVEICM removes code modules from file ICM being created. This option is especially convenient when the user attempts to bring in an ICM library code file with a large number of modules and replace only a few modules with new ones with the same name from the current compile. The REMOVEICM option has the following form:



G50353

<ICM-name> is the name of an intermediate code module within file ICM being created. The REMOVEICM and USEICM options provide the capability to selectively recompile portions of a previously compiled program which had syntax errors or logical errors in some of the program units. This concept is illustrated by the following example:

The source program name FORTPROG was compiled as follows:

```
$ICM "MYMODS"
      PROGRAM VAL                              00001000
            .
            .
            .
      SUBROUTINE FREQ(EX, VERIF)               00012000
            .
            .
            .
      SUBROUTINE CONV                          00024000
            .
            .
            .
      INTEGER FUNCTION COMPUT                  00033000
            .
            .
            .
```

Suppose the user wants to recompile subroutine FREQ to update it or correct a syntax error, and bind the entire program into an executable FORTRAN 77 code file. The following card images would provide the desired results:

```
?CO CODEFILE FORTRAN77 LI
?FI SOURCE NAM FORTPROC
?DATA CARD
$SET MERGE NEW
$USEICM "MYMODS" =
$REMOVEICM FREQ
$SET OMIT
$RESET OMIT                                    00011999
      IF (I.NE.J*2) CALL CONV                  00017100
$SET OMIT                                      00023999
```

This example recompiles only subroutine FREQ, and obtains the rest of the program from ICM library code file MYMODS. The FORTRAN 77 statement that was added could also have been added by using CANDE or some other means, and the above sequence performed without the card image containing the FORTRAN 77 statement.

MISCELLANEOUS OPTIONS

The following miscellaneous options are provided by FORTRAN 77.

AUTOBIND

AUTOBIND is a boolean option that causes the compiler to combine the ICMs in file ICM into an executable code file. ICM contains intermediate code modules created from the compilation of subprograms, main programs, and any modules added by a USEICM control option. The AUTOBIND option has the following form:

G50354

<main-program-name> is the name of a main program in ICM to be used as the main program of the executable code file. A program name is specified by a PROGRAM statement or is 0MAIN if no PROGRAM statement is present. This permits a source file to contain more than one main program. However, no two main programs or subprograms can have the same name. This causes a duplicate file situation in ICM.

If no <main-program-name> is specified, the binder uses either:

1. The last compiled syntactically correct main program explicitly named in a PROGRAM statement.
2. 0MAIN, if there is no explicitly named main program in ICM.

An example of the use of the AUTOBIND compiler control option follows:

```
$SET AUTOBIND "MAIN2"
```

The AUTOBIND option is set by default. If AUTOBIND is reset, no attempt is made to create an executable code file from ICM.

CLEAR

CLEAR is an immediate option that causes all boolean options except MERGE and NEW to be reset.

Example:

```
$CLEAR
```

END

END is an immediate option that causes the compiler to terminate compilation and close and save all currently open disk files. This permits the user to specify a premature end to compilation anywhere within the program being compiled.

Example:

```
$END                    00066000
```

ERRORLIST

ERRORLIST is a boolean option which, when set, causes the compiler to list all syntax errors encountered during compilation in a separate file called ERRORS. ERRORLIST is set by default if running through CANDE; otherwise, it is reset.

Example:

```
$RESET ERRORLIST
```

INTERPRETER

INTERPRETER is a value option that specifies the name of the interpreter to be used with the executable code file being generated. INTERPRETER has the following form:

G50355

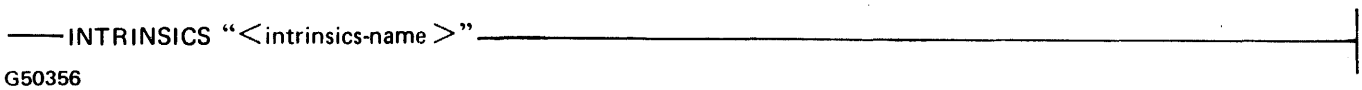
<interpreter-name> is the name of a disk file that is an interpreter for FORTRAN 77. The default interpreter name is FORTRAN77/INTERP3M.

Example:

```
$INTERPRETER "INTDEBG"
```

INTRINSICS

INTRINSICS is a value option which specifies the name of the intrinsics file to be used when compiling the FORTRAN 77 program. The INTRINSICS option has the following form:



G50356

<intrinsics-name> is the name of a disk file which contains all intrinsics necessary to compile the program. The default intrinsic file name is FORTRAN77/INTRINSICS.

Example:

```
$INTRINSICS "USER/(JONES)/INTRINALT"
```

NOBOUNDS

NOBOUNDS is a boolean option that inhibits the compiler and interpreter from checking bounds when arrays are referenced. Program execution time can be enhanced by setting this option once it is determined that all array references are correct. This option is set by default.

Example:

```
$NOBOUNDS
```

APPENDIX A
B 1000 FORTRAN 77 LANGUAGE SYSTEM

The purpose of this appendix is to provide an outline of the features of the B 1000 FORTRAN 77 language system. This includes the following:

1. A summary of system requirements.
2. A digest of user-oriented compiler information.
3. A complete description of control records and the structure of the FORTRAN 77 compilation source file.

The FORTRAN 77 compiler described in this appendix and the object programs generated by it are designed to operate under control of the B 1000 Master Control Program (MCP).

SYSTEM REQUIREMENTS

The following is a description of the system hardware and software required for the B 1000 FORTRAN 77 language system.

Required Hardware

The following hardware devices must be provided for the FORTRAN 77 system to operate: B 1000 processor (except B 1825, B 1830, and B 1710 series of processors), and disk.

Required System Software

The FORTRAN 77 system file requirements are as follows:

1. The FORTRAN 77 compiler (which includes a binding phase).
2. The intrinsic file (which contains various subprograms supplied with the compiler).
3. The FORTRAN 77 interpreter (which executes the object code).

The FORTRAN 77 compiler, interpreter, and intrinsic files must all reside on the same disk, unless otherwise specified in a Compiler Control Image. Refer to INTERPRETER and INTRINSICS in section 14. If these files are on a user disk cartridge or pack, they are referenced by prefacing their names with the disk cartridge or pack name.

USER/COMPILER INTERFACE

The purpose of the B 1000 FORTRAN 77 compiler is to accept application programs written in the FORTRAN 77 language and to produce from these programs object code which can be executed on the B 1000 system.

Concurrent to the production of object code, the user is provided with compile-time debugging and diagnostic facilities and the ability, to a limited extent, to control the functions performed by the compiler, such as in the area of compiler file handling. Compiler file handling is available using the FORTRAN 77 Compiler Control Images. Refer to section 14 for additional information.

The debugging and diagnostic facilities provided by the compiler are compile-time additions to the compiler-provided printer listing of input source statements. The following items are provided as diagnostic aids by the compiler:

1. Syntax-error messages, which are placed on the printer listing generally following the line of text bearing the questionable statement.
2. Messages denoting warnings are placed on the printer listing following the line bearing the incorrect statement.
3. A special character is printed above the syntax-error or warning message to give the approximate location of the error.
4. Various compiler information messages.

All user communication with the compiler and all compiler output is handled using compiler files. A description of the interface between the user and the FORTRAN 77 compiler is, therefore, an examination of the features of these compiler files. Figure A-1 contains the names and characteristics of the compiler files.

Intermediate Code Files

Depending on the Compiler Control Images used, intermediate code files and/or a single executable file is produced by the compiler. Each subprogram is compiled into a separate file in an intermediate nonexecutable form. An executable file is produced by the binding part of the compiler using the ?COMPILE record specification or through the appropriate Compiler Control Image. For information on the use of Compiler Control Images, refer to section 14.

Compiler Files

Compiler communication is handled through various input and output files.

The compiler has the capability of merging input from two files on the basis of sequence numbers. When inputs are being merged, indications of text insertions or replacements are made to appear on the output listing. In addition to the output listing, the FORTRAN 77 compiler can also generate an updated symbolic output file. These files can be created in addition to the compiler-generated output code file. Compiler input and output files are described in detail in the following text.

Input Files

The primary compiler input file is a file with the internal name CARD; the secondary input file is a serial disk file with the internal name SOURCE. The presence of the primary file CARD is required for each compilation; the presence of the secondary file SOURCE is optional for each compilation. File CARD is coded with 80-character records and is unblocked. File SOURCE is coded with 80-character records and uses input blocking. Both the CARD file and the SOURCE file can be label-equated (using label equation records) to change the file's external file name and hardware device. Refer to the description of the FILE statement in the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982.

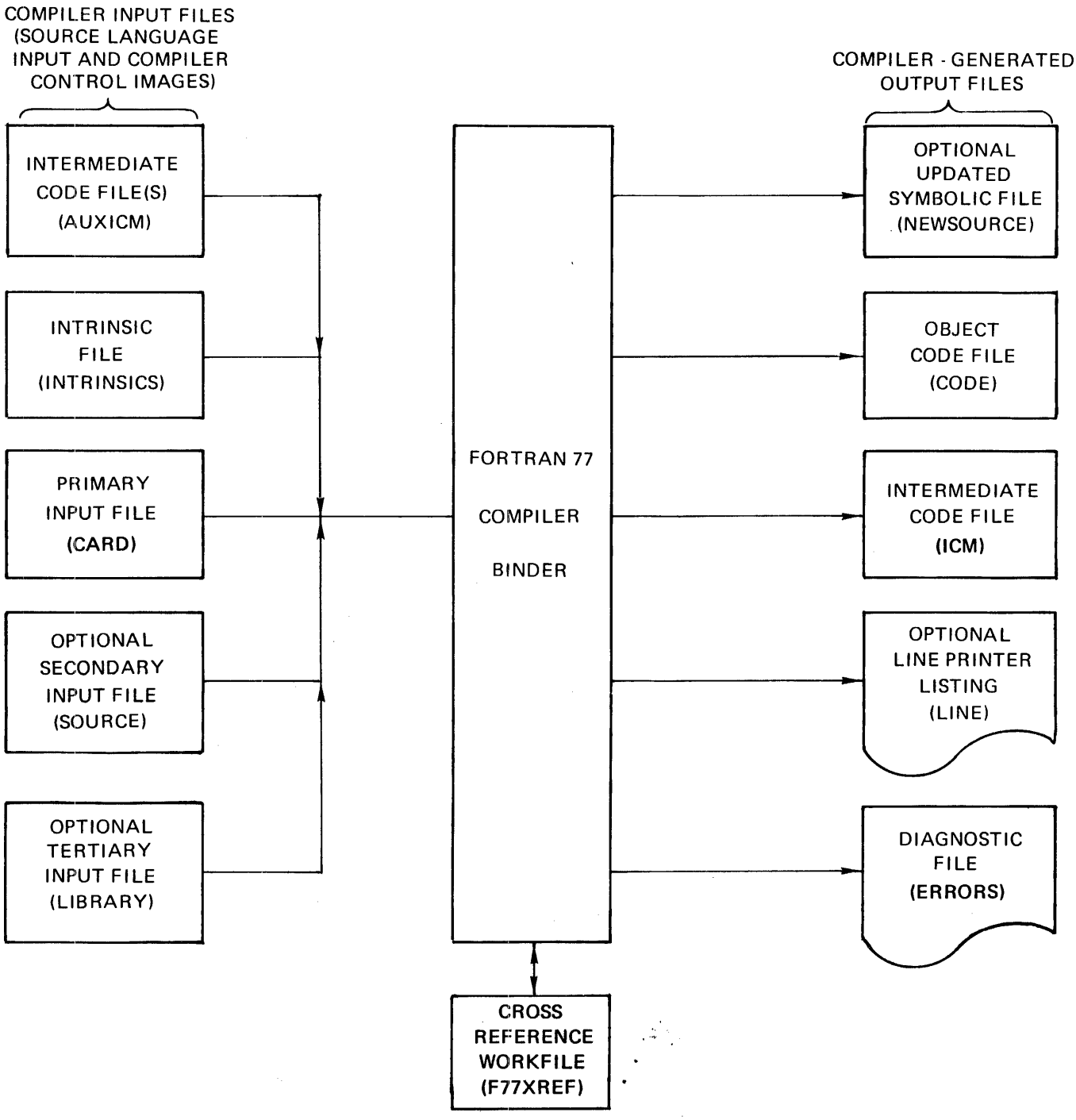


Figure A-1. FORTRAN 77 Compilation System

Output Files

Output files produced by the compiler include intermediate and object code files, an updated symbolic file, a syntax-error file, and a line printer listing. The intermediate code file has the internal name ICM.

The object code file has the internal name CODE and is saved on disk after the compilation unless the COMPILE system control record specifies otherwise. The external file name of the saved code file is identical to the program-name appearing on the COMPILE record. Refer to the subsection entitled MCP Control Cards in this appendix.

The compiled program is logically segmented within the resultant code file by program unit. The code for each program unit begins at a physical disk segment boundary and fills as many disk segments as required within the limits of the system. The updated symbolic file is, by default, a disk file (NEW-SOURCE) generated only if the compiler control option NEW is set. This file contains the compilation source input or a selected portion of this input as specified by the compiler control option NEW and can be used as the SOURCE file for a succeeding compilation.

The printer listing is an optional print file that is created unless the compiler control option LIST is reset. The LIST option is set by default. The file has the internal name LINE, and contains the following information:

1. Source and Compiler Control Images input to the compiler.
2. Code segmentation information.
3. Error messages and error count.
4. Processor compilation time and elapsed compilation per subprogram unit.
5. Timing breakdown of major compilation activities for all program units.
6. Estimated space needed for the program files.
7. Total number of bits of object code generated for each subprogram.
8. Number of disk segments required for the program code file.
9. Estimated memory required to run the object program.

Depending upon the specified setting of the LIST and MAP compiler control options, the printer listing can contain more or less information than the basic items listed above.

Compiler File Names and Defaults

The FORTRAN 77 input and output files and information concerning the configuration of each file are listed in table A-1. Table A-1 lists the internal name of the file (the name used when the file is declared within the FORTRAN 77 compiler), the purpose served by the file, the default hardware device of the file, the default record size (RSZ) and records per block (RECORDS.BLOCK) of the file, and a brief commentary on the file.

Table A-1. FORTRAN 77 Compiler File Names and Characteristics

Internal Name	Purpose	Default Hardware	RSZ/ Block	Comments
CARD	Input card file	CARD READER	80/1	Required for each compilation. Primary compiler input file.
SOURCE	Input disk file	DISK	180/1	Optional file; not necessary for compilation. Secondary compiler input file, selected by setting MERGE CCI.
LIBRARY	Input disk file	DISK	180/1	Optional file; not necessary for compilation. Tertiary compiler input file, selected by setting INCLUDE CCI.
F77XREF	Cross reference work file	DISK	25/60	Work file used by cross reference.
CODE	Executable object code file	DISK	180/1	Generated object code file. Saved or discarded and assigned the program-name.
NEWSOURCE	Updated symbolic output file	DISK	90/2	Optional output file produced when NEW CCI is set.
LINE	Line printer listing	LINE PRINTER	132/1	Output from the compile.
ERRORS	Diagnostic file	REMOTE	132/1	Default error file if running from CANDE.
INTRINSICS	Intrinsics and intrinsic functions	DISK	180/1	The intrinsics and intrinsic functions file.
ICM	Intermediate code file	DISK	180/1	As output: destination of intermediate code modules from the compiler. Saved if ICM set. As input: source of intermediate code modules to the binder.
AUXICM	Intermediate code file	DISK	180/1	Optional input. Source of previously compiled intermediate code modules to be copied into file GIF for transmission to the binder if USEICM is set.

The attributes of any of these files can be changed through use of label equation records directed to the compiler. Refer to the discussion on the FILE card in the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982.

Large FORTRAN 77 Program Code Files

Code files on the B 1000 system must be contained in one disk area. If the FORTRAN 77 compiler terminates because the file space was exceeded for the code file, the BLOCKS.PER.AREA file attribute of the CODE file of the FORTRAN 77 compiler must be increased in the following way.

```
COMPILE <code-file-name> FORTRAN77 LI
FILE CODE BLOCKS.PER.AREA = <integer>;
```

The value of <integer> must be greater than the default value of 700.

MCP CONTROL RECORDS

When a FORTRAN 77 source program is compiled, the actions to be performed are specified by control records. Control records included in a compilation source file are of two types: MCP control records (? records), and Compiler Control Images ($ records). The structure of the FORTRAN 77 compilation source file is explained in the text that follows.

Compilation of a FORTRAN 77 source program is achieved by presenting the compilation source file to the MCP. The entities comprising the structure of the FORTRAN 77 compilation source file and the order of occurrence follow.

Compilation Source File

1. ? COMPILE record.
2. Label equation records. (FILE statement) (optional).
3. ? DATA CARD record. (Necessary only if the source program is to immediately follow, as with punch cards. Refer to the DATA statement in the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982.)
4. Source input file CARD. (Compiler Control Images can be inserted here.)
5. ? END (end of file) record. (Necessary only if the DATA statement is used.)

MCP control cards are distinguishable from other cards by an invalid character in column 1 for 80-column cards or a question mark (?) character for 96-column cards. An invalid character is represented by a question mark (?) character for clarity in this manual. If the program is compiled from the ODT, the question mark is deleted. MCP control information is entered in a free-form format in columns 2 through 72.

? COMPILE Record

The ? COMPILE record instructs the MCP to compile the indicated program-name with FORTRAN 77 using one of the following options:

1. ? COMPILE program-name FORTRAN77

   This option causes the source program to be compiled, bound, and executed (compile and go). The resultant object program is not entered in the disk directory. The resultant intermediate code files are removed from the disk directory upon binding unless the Compiler Control Image $ICM is specified.

2. ? COMPILE program-name FORTRAN77 LIBRARY

   This option causes the source program to be compiled and bound, but not executed. The resultant object program is entered in the disk directory. The resultant intermediate code files are removed from the disk directory upon binding unless the Compiler Control Image $ICM is specified. Execution is specified by the execution statement, ? EXECUTE <program-name>, placed after the ? END record, if present. Additional information on the ? EXECUTE statement can be found in the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982.

3. ? COMPILE program-name FORTRAN77 SAVE

   This option causes the source program to be compiled, bound, and executed, and the resultant object program to be entered in the disk directory. The resultant intermediate code files are removed from the disk directory upon binding unless the Compiler Control Image $ICM is specified.

4. ? COMPILE program-name FORTRAN77 SYNTAX

   This option causes the source program to be compiled only for a syntax check.

For compile card options 1, 2, and 3, the intermediate code files are created and left in the disk directory after compilation until binding occurs. If the program is terminated before binding, or the compiler control option NO AUTOBIND is specified, the intermediate code files remain in the directory. Refer to section 14 in this manual, for additional information. If any errors result during compilation, the error-free intermediate code files remain in the directory and no binding occurs. The error-free intermediate code files remaining in the directory do not have to be recompiled.

If the required intermediate code files are not on disk or a subprogram is referenced which was not compiled, a message stating that the file or subprogram unit is missing is given during binding. Subprograms can be compiled independently or with the main program which references them.

Program-name

The program-name can consist of one, two, or three identifiers of up to 10 characters each, separated by slashes. Following are the four forms a program-name can take:

1. family-name (an identifier which is a single file name).
2. family-name/file-identifier (an identifier which can be a single file name or a file with subprogram entries).
3. dp-id/family-name/file-identifier (the disk-pack-identifier is specified when a removable disk pack is used).
4. dp-id/family-name/ (a single file name residing on a removable disk pack).

An executable code file has the program-name specified on the COMPILE statement. Intermediate code files have the program-name on the COMPILE statement, except that the file-identifier is replaced by the following:

1. The subprogram name in a SUBROUTINE or FUNCTION statement.
2. Zero (0) followed by the file-identifier (if any) for an unbound main program.
3. BLOCK. for the BLOCK DATA subprogram.

The disk-pack-identifier must be included in the program-name if the intermediate code files are in other than the system disk directory.

Table A-2 shows the four forms an intermediate code file program-name can take given the family-name, file-identifier, and disk-pack identifier.

Table A-2. ICM Name Conversions

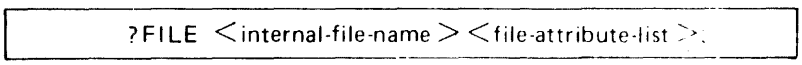
Type of ICM	Program-names on COMPILE Statement			
	MAIN	**MAIN/SUB**	**FORTRAN77 MAIN/SUB**	**FORTRAN77 MAIN/**
Unbound Main Program	MAIN/0	MAIN/0SUB	FORTRAN77/ MAIN/0SUB	FORTRAN77/ MAIN/0
SUBROUTINE X	MAIN/X	MAIN/X	FORTRAN77/ MAIN/X	FORTRAN77/ MAIN/X
BLOCK DATA Subprogram	MAIN/BLOCK.	MAIN/BLOCK.	FORTRAN77/ MAIN/BLOCK.	FORTRAN77/ MAIN/BLOCK.

MAIN is the family-name.
SUB is the file-identifier.
FORTRAN77 is the disk-pack identifier.

Label Equations (FILE statement)

Label equations can optionally be included in the compilation source file and can be used to modify the original attributes of the FORTRAN 77 system files. Label equations are specified with the FILE statement.

If used, the FILE statement immediately follows the COMPILE record and precedes ?DATA CARD, if present. The general form of the FILE statement follows.



G50358

A list of file-attributes and uses can be found in the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982, under the FILE statement.

The FORTRAN 77 compiler's internal file names for use with the FILE statement are shown in table A-1.

? DATA CARD Record

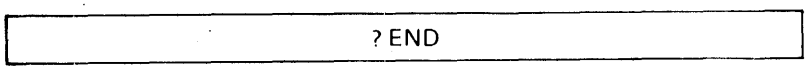
A record of the following form is required to label the source file if the source file is to follow the ? COMPILE record and any label equations.

Source Input File CARD

These records are the FORTRAN 77 statements comprising the source program.

? END Record

The ? END record designates the end of file for the compilation source file if the DATA statement is used. The ? END record is coded as follows:



G50360

The ? END card is the last card in the compilation source file.

The examples that follow show seven ways a program named JOB, containing two subprograms (SUB1 and SUB2), can be compiled and executed using the MCP control statements ? COMPILE, ? DATA, and ? END.

Example 1 – Compile and go:

```
?COMPILE JOB FORTRAN77
?DATA CARD
    PROGRAM JOB
    .
    .
    .
    END

    SUBROUTINE SUB1
    .
    .
    END

    FUNCTION SUB2
    .
    .
    END
?END
```

Example 2 – Compile, execute, save ICMs:

```
?COMPILE JOB FORTRAN77          or   ?COMPILE JOB FORTRAN77 SAVE
?DATA CARD                           ?DATA CARD
$ICM                                 $ICM
    PROGRAM JOB                          PROGRAM JOB
    .                                    .
    .                                    .
    .                                    .
    END                                  END

    SUBROUTINE SUB1                      SUBROUTINE SUB1
    .                                    .
    .                                    .
    END                                  END

    FUNCTION SUB2                        FUNCTION SUB2
    .                                    .
    .                                    .
    END                                  END
?END                                 ?END
```

When the SAVE option is used, the object program is also entered in the disk directory.

Example 3 – Compile and execute in three steps:

```
?COMPILE SUB FORTRAN77 LIBRARY
?DATA CARD
$ICM
$NO AUTOBIND
    SUBROUTINE SUB1                          (An ICM named SUB/SUB1 is on disk.)
    .
    .
    END
?END

?COMPILE JOB FORTRAN77 LIBRARY
?DATA CARD
$ICM
$NO AUTOBIND
C MAIN PROGRAM JOB                           (An ICM named JOB/0 is on disk.)
    .
    .
    .
    END
?END

?COMPILE SUB FORTRAN77 LIBRARY      or   ?COMPILE SUB FORTRAN77 SAVE
?DATA CARD                               ?DATA CARD
$ICM                                     $ICM
    FUNCTION SUB2                            FUNCTION SUB2
    .                                        .
    .                                        .
    END                                      END
?END                                     ?END
?EXECUTE JOB
```

APPENDIX B
OPTIMIZING PROGRAM COMPILATION

When compiling on B 1000 systems with sufficiently large memory configurations, FORTRAN 77 compilation times can be enhanced in the following ways.

1. The size of the compiler's dynamic memory is a factor that affects compilation time. Dynamic memory size can be increased from the default of 200,000 bits by the following control statement:

   ?MODIFY FORTRAN77 MEMORY = integer

   Increasing dynamic memory size for the compiler results in increased compilation speed unless there is not enough memory for code segments, MCP overlays, and other programs.

2. Compiler files can be assigned to different disk drives, enhancing compilation time by relieving disk arm contention. When a pack-id precedes the program-name on a COMPILE card, each intermediate code file and the object code file are written to the designated user pack. When no pack-id precedes the program-name, those files are written to system disk. Likewise, when a pack-id precedes the compiler name on the COMPILE card, the FORTRAN 77 compiler, the INTRINSICS built-in function and intrinsic file, and the FORTRAN 77 interpreter are expected by the MCP to reside on the specified disk.
3. Only one buffer is associated with each of the compiler files. Associating two buffers with each of the compiler files generally speeds compilation time. This is accomplished by using the following statements:

   ?MODIFY FORTRAN77
   ?FILE <file-name> BUFFERS = 2

The file names of each of the compiler files can be found in table A-1 in appendix A. For additional information concerning the FILE statement, refer to the B 1000 Systems System Software Operation Guide, Volume 1, form number 1108982.

APPENDIX C
DESCRIPTION OF UNFORMATTED I/O RECORDS

Each unformatted record is written as 32 bits of control information, followed by the values of the variables in the I/O variable list. Each variable generates an 8-bit, 32-bit, or 64-bit grouping depending on whether the item is of type CHARACTER, INTEGER, REAL, LOGICAL, DOUBLE PRECISION, or COMPLEX.

For example, if three single-precision variables (A,B,C) were written unformatted, the record on disk would appear as 32 bits of control information, followed by 32 bits containing the machine representation of A, followed by another 32 bits containing the machine representation of B, followed by a third 32-bit group containing the machine representation of C. The 32 bits of control information have the following format:

Bit	Meaning
1-16	Always set for unformatted I/O.
17	When set, indicates that the unformatted record begins in this logical record.
18	When set, indicates that the unformatted record ends in this logical record.
19-32	Length field. This is the total number of bytes in the logical record, including the control word. To calculate the length field: 1. Add four bytes for each REAL, INTEGER, and LOGICAL variable, eight bytes for each DOUBLE PRECISION and COMPLEX variable, and one byte for each character variable. 2. To the sum in item 1, add four bytes for the control word.

In order to read an unformatted record, the I/O variable list of the read statement must agree in order with the list of the write statement originally used to write the unformatted record.

The default logical record size for disk is 180 bytes (one segment), which is sufficient for the control information and 44 single-precision variables. If the list size on a sequential WRITE exceeds the declared or default logical record size, the filled logical record is written and a second logical record is built, containing additional control information followed by those items which did not fit the first logical record. This process continues until the list is exhausted. If the list size for a direct-access WRITE exceeds the logical record size, a data error results.

APPENDIX D
STORAGE ALLOCATION

Each B 1000 FORTRAN 77 data item is allocated one or more units of storage, depending upon the type of value(s) the item represents. The primary unit of storage involved is the 32-bit word. Word, in this appendix, refers to 32 contiguous bits unless otherwise stated.

This appendix describes: 1) the allocation of storage to two groups of data items: simple variables and arrays, and 2) the compiler and binder listings describing the storage allocation of both data and code. Simple variables include the following types: INTEGER, REAL, LOGICAL, DOUBLE PRECISION, COMPLEX, and CHARACTER. An array is a grouping of data words in contiguous memory locations. Arrays can be of the same types as simple variables; an array consists of a group of contiguous data words of type CHARACTER, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL.

The notation [m:n] is used in this appendix to describe data word fields. The bits in a data word are numbered 0 through 31. In the notation used here, m denotes the number of the leftmost bit of the field being described, and n denotes the number of bits in the field. For example, in the byte field shown in figure D-1, bits 3 through 6 are described by [3:4].

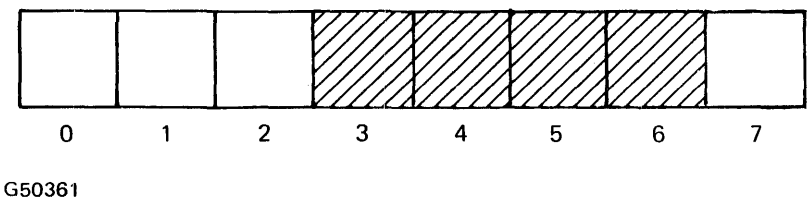


Figure D-1. Representation of [3:4]

Hexadecimal constants are used extensively in this appendix to indicate word contents. Such constants are particularly suited to describing the value of a data word, since each digit in a hexadecimal constant indicates the contents of a 4-bit field.

SIMPLE VARIABLES

Each simple variable generally requires one or two words of storage, depending upon the type of the variable. CHARACTER variables are an exception to this rule. Storage is allocated to CHARACTER variables in groups of 8-bit bytes, depending upon the declared length.

INTEGER Variables

An INTEGER variable requires one word of storage. The data word corresponding to an INTEGER variable is partitioned as follows:

Field	Contents
[0:32]	Integer (2's-complement)

Integer values are represented internally in 2's-complement notation.

An example of the internal representation of the integer 10 (represented in hexadecimal constant notation) follows:

Z0000000A

An example of the internal representation of −10 follows:

ZFFFFFFF6 (Twos-complement notation)

Integer values −2,147,483,648 through 2,147,483,647 can be stored with accuracy. A larger range requires more than 32 bits.

REAL Variables

A REAL variable requires one word of storage. The data word corresponding to a REAL variable is partitioned as follows:

Field	Contents
[0:1]	Mantissa sign bit (1 = negative, 0 = positive)
[1:7]	Hexadecimal exponent (in excess-64 notation)
[8:24]	Mantissa field (normalized hexadecimal number)

Real (floating-point) values are represented internally in two parts: the mantissa in signed-magnitude form and the exponent, which is a hexadecimal exponent in excess-64 notation. The sign of the value is denoted by the mantissa sign bit of the data word. This bit is 0 for positive or zero values and 1 for negative values. The magnitude of the mantissa is stored left-normalized within the data word, with the radix point assumed to the left of the mantissa field. The exponent, expressed in excess-64 notation, represents the power of 16 by which the mantissa (in hexadecimal form) must be multiplied to determine the actual position of the radix point. In other words, the excess-64 exponent represents the number of hexadecimal positions that the radix point must be moved to represent the actual hexadecimal value. In excess-64 notation, an exponent equal to 64 represents an exponent value of 0.

Examples of the internal representations of the indicated real numbers follow:

Real Value	Hex Constant
17.5	Z42118000
16.8	Z4210CCCC
−65.1	ZC2411999
10000.5	Z44271080
.000583	Z3E26351D
−.000583	ZBE26351D

The range that can be stored in real form is approximately .539761E−78 through .7237006E+76. Up to seven decimal digits can be stored with accuracy.

DOUBLE PRECISION Variables

A DOUBLE PRECISION variable is allocated two words of storage. The first word is identical to the data word of a REAL variable, with the second word considered as an extension to the right of the mantissa of the first word. The two data words corresponding to a DOUBLE PRECISION variable are partitioned as follows:

Field	Contents
[0:1]	Mantissa sign bit (0 = positive, 1 = negative)
[1:7]	Hexadecimal exponent (in excess-64 notation)
[8:56]	Mantissa (normalized hexadecimal number)

Double-precision values are represented internally in two parts: the mantissa in signed-magnitude form, and the exponent as a hexadecimal exponent in excess-64 notation, identical to the representation of REAL variables as described in this appendix. The magnitude of the mantissa is stored left normalized within the 56-bit mantissa field.

The maximum range that can be stored in double-precision form is the same as for a REAL variable. Up to 18 decimal digits can be stored with accuracy.

LOGICAL Variables

A LOGICAL variable requires one word of storage. The data word corresponding to a LOGICAL variable is partitioned as follows:

Field	Contents
[0:31]	Limited use
[31:1]	Value bit

Bits 0 through 30 are only used when .TRUE. is assigned to a LOGICAL variable in the FORTRAN 77 source code. In this case, bits 0 through 31 (all bits) are set to 1. Bit 31 is the only bit actually tested in a logical comparison. If its value is 1, the value of the variable is TRUE; if its value is 0, the value of the variable is FALSE. The value of the leftmost 31 bits is ignored. For example, if a LOGICAL variable is equivalenced to any odd−valued (−5,−3,−1,1,3,5, and so on) INTEGER variable, the value of the LOGICAL variable is TRUE. When equivalenced to any even−valued (−4,−2,0,2,4, and so on) INTEGER variable, the LOGICAL variable is FALSE.

COMPLEX Variables

A COMPLEX variable is allocated 64 bits of storage. The first of these two data words contains the REAL part of the variable, while the remaining data word contains the imaginary part of the variable. Each of these two data words is identical to the data word of a REAL variable.

The two data words corresponding to a COMPLEX variable are partitioned as follows:

Field	Contents
First data word (real part)	
[0:1]	Mantissa sign bit (1 = negative, 0 = positive)
[1:7]	Hexadecimal exponent (in excess-64 notation)
[8:24]	Mantissa (normalized hexadecimal number)
Second data word (imaginary part)	
[0:1]	Mantissa sign bit (1 = negative, 0 = positive)
[1:7]	Hexadecimal exponent (in excess-64 notation)
[8:24]	Mantissa (normalized hexadecimal number)

The real and imaginary values are represented internally in the identical manner as described above for REAL variables.

The maximum magnitude that can be stored in each data word is the same as for a REAL variable.

ARRAYS

FORTRAN 77 arrays are provided to allow the user to organize program storage locations into a structure convenient to the user. Internally, an array is stored as a group of one or more contiguous data words. A description of the correspondence between the array elements and the group of internal storage words follows.

A FORTRAN 77 array of any legal number of declared dimensions ($1 <= n <= 7$) is represented internally by a 1-dimensional array (a vector) of storage locations. Each element of the array has storage requirements identical to that of a simple variable of the same type as the array. For example, each element of a REAL array requires one word of storage, whereas each element of a DOUBLE PRECISION array requires two words of storage. The partitioning of each storage word is identical to that of the storage word(s) corresponding to a simple variable of the same type as the array element.

Each INTEGER, REAL, and LOGICAL array is allocated a series of internal data words exactly equal in number to the elements of the array. DOUBLE PRECISION arrays are allocated twice as many internal data words as array elements. CHARACTER arrays are allocated space in 8-bit bytes. Word size for a CHARACTER array depends on the declared size in the corresponding specification statement: 1 byte (8 bits) for each byte of declared length. Refer to Character Type Statements in section 6 for additional information. For 1-dimensional arrays, the internal data word, word pair, or n-byte word corresponding to each array element occurs in the same position in the internal array as the element occurs in the array.

For example, arrays A1 and A2 are declared using the following statements:

```
DIMENSION A1(1:20)
DOUBLE PRECISION A2(20)
```

The REAL array A1 is allocated 20 words of storage (one word per element), and the DOUBLE PRECISION array A2 is allocated 40 words of storage (two words per element). Therefore, the array element A1(2) is assigned the second word of the internal array corresponding to A1, and A2(2) is assigned the third and fourth words of the internal array corresponding to A2.

An example of a declaration of a CHARACTER array follows:

```
CHARACTER * 20 A3(5:15)
```

Array A3 is allocated 11 words of storage. Words for this array are 20 bytes long.

Arrays of more than one dimension are stored by columns into one continuous internal vector of storage words. This storage process occurs in the manner displayed in figure D-2.

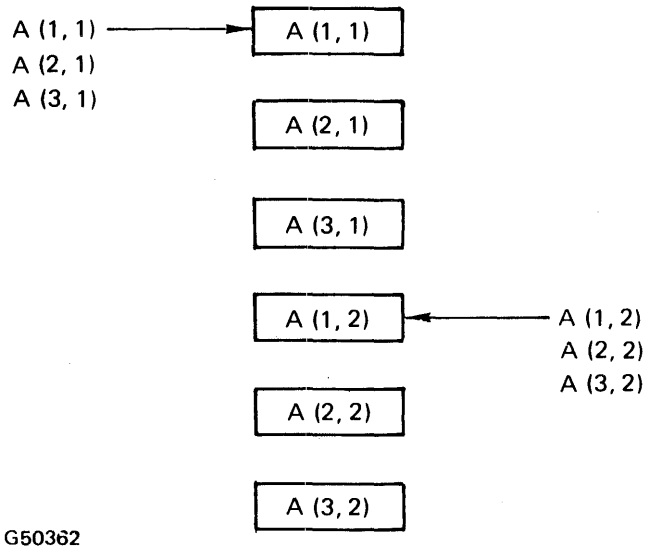


Figure D-2. Storage of a Multi-Dimensional Array

The word(s) corresponding to each element of the 2-dimensional array in this diagram are located in the internal array in the order shown. Beginning with element A(1,1), each successive element in the array is stored in the order of occurrence: proceeding down each column of the array, from the leftmost columns to the rightmost columns. For an array of three dimensions or greater, this process is repeated for each successive layer of the array (each value of the third subscript, fourth subscript, fifth subscript, and so forth).

For any n-dimensional array, the elements of the array correspond (in a set order) to successive storage locations in the 1-dimensional internal array associated with the array. The appropriate order is identical to that obtained when one lists all of the array elements by varying the first subscript most rapidly, the second subscript next, and so forth. As an example, the elements of the 5-dimensional array A(2,1,2,2,1) are stored in this order:

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

Element A(1,1,1,1,1) in this example corresponds to the first element of the internal array, element A(2,1,1,1,1) corresponds to the second element of the internal array, and so forth.

NOTE

For clarity, the dimensions in the preceding two examples begin with the value 1, but FORTRAN 77 allows negative subscripts also. Refer to Array Declarations in section 5.

An array A, having n dimensions, can be declared by means of an array declaration of the type:

```
DIMENSION A(L1:U1,L2:U2, L3:U3, ... ,Ln:Un)
```

Ln and Un represent the lower and upper bounds of dimension n, respectively, where n is an integer between 1 and 7, inclusive. (If the lower bound is omitted, it defaults to 1.)

Array A contains elements of the form:

```
A(S1,S2,S3,...,Sn)
```

The position, P, of the storage unit (word, word pair, or n-byte word) assigned to this array element within the corresponding internal array, can be found by means of the following formula:

```
P = (S1 - L1 + 1) + ((U1 - L1 + 1) * (S2 - L2))
```

A multidimensional array can be equivalenced to a 1-dimensional array and the elements of the 1-dimensional array will then correspond in order to the storage units assigned to the multidimensional array. This order is also the order in which array elements are considered when an array name appears without a subscript list within the variable list in a I/O statement, or as the item to be initialized in a DATA statement.

DATA ALLOCATION INFORMATION

Storage allocated for data in a program can consist of two components: dynamic memory and static memory. Dynamic memory is segmented into pages of 1024 bytes each which can be overlayed and recalled as necessary during the execution of a program. Static memory is not segmented.

The compiler groups data in a program in two categories: local data and common blocks. Each group of data is assigned a location in either dynamic or static memory depending on the size of the data. If the size of a local data group is 2048 bytes or greater, the data is placed in dynamic memory; otherwise, it is placed in static memory. If the size of a common block is 4096 bytes or greater, the common block is placed in dynamic memory; otherwise, it is placed in static memory.

If the size of a common block is less than 4096 bytes in one routine of a program and is 4096 bytes or more in another routine of the program, the following compiler error message results:

```
PLEASE RECOMPILE FORCING ALL REFERENCES TO
'<common-block-name>' TO DYNAMIC MEMORY
```

This error is corrected by defining the size of the common block to be 4096 bytes or more in each routine it appears. Dummy data items can be added to the common block so that it contains the necessary 4096 bytes.

For each subprogram, data is assigned addresses relative to zero in both static and dynamic memory. The relative address for each variable is listed by the compiler (when the MAP option is set) in the SYMBOLIC REFERENCE INFORMATION table. In this table, each variable that is assigned an address in dynamic memory has a page number associated with its address; each variable that is assigned an address in static memory does not have a page number associated with its address.

In order to determine the actual address assigned to a variable in static memory, it is necessary to use the allocation information printed by the binder portion of the compiler. This information is listed by the compiler by default in the MODULE ALLOCATION INFORMATION table. This table lists the actual starting address in static memory of the local data for each subprogram. This starting address must be added to the relative address listed in the SYMBOLIC REFERENCE INFORMATION table that follows the subprogram containing the variable.

The binder lists by default the starting address of each block of data in the BLOCK DATA ALLOCATION INFORMATION table. If the local data of a subprogram is large enough to be placed in dynamic memory, the block of data is given a name consisting of a dollar sign ($) character followed by the first five characters of the subprogram name. This table gives the name of the block of data, its starting address in either static or dynamic memory, and its size in bytes.

Any block of data allocated to dynamic memory is given an integral number of pages. For example, if a block is 2500 bytes long, it is allocated three 1024 byte pages. The next block is allocated space starting with the fourth page.

CODE SEGMENTATION INFORMATION

The object code produced by the compiler is broken into segments that are overlayed in memory as necessary. A mapping of the beginning code address for each source statement is provided by default in the CODE ADDRESSES listing that follows each subprogram. This listing consists of pairs of numbers, where the first number is the source line number and the second number is the code address.

The CODE ADDRESSES listing contains addresses relative to zero. To determine the actual code address for a particular source statement, it is necessary to use the final code mapping given by the binder in the MODULE ALLOCATION INFORMATION table. The name of each subprogram is listed in the table along with the segment number where the code for that subprogram starts. This starting segment number must be added to the relative code address given in the CODE ADDRESSES listing to compute the actual code address. Code for a subprogram is allocated an integral number of segments, therefore, displacements within a segment do not change from those in the CODE ADDRESSES listings, only the segment numbers change.

APPENDIX E
FORTRAN77/ANALYZER

The FORTRAN77/ANALYZER program is a utility program which analyzes either a FORTRAN 77 code file, a FORTRAN 77 intermediate code file, or both.

A code file is generated from the compile operation and is used for the execution of the program. It contains information concerning the program parameter block, file parameter block, code segment dictionary, data page dictionary, data pages, layout table, and the code to be executed.

An intermediate code file is generated by the Compiler Control Image ICM. This code file consists of a directory and one or more Intermediate Code Modules (ICMs). The directory records the number of ICMs, the ICM names, and the ICM locations in the file.

An ICM consists of a header describing information pertinent to the module, as well as the address and size information of the following parts of the module:

1. Data (preinitialized data, code addresses to be resolved and symbol table information)
2. File Parameter Block (if any)
3. Code (subroutine, function, or main program)
4. Reference Table (where subroutines, functions, and common blocks are referenced in the code)

The following entities generate an Intermediate Code Module (ICM): the main program, a subroutine, a function, an entry point, a block data subprogram, and one or more file declarations.

PROGRAM EXECUTION

The FORTRAN77/ANALYZER program is executed with the following statement:

```
EXECUTE FORTRAN77/ANALYZER
```

Various switch settings can modify the mode of operation of the FORTRAN77/ANALYZER program. These switch settings are described in table E-1.

Table E-1. Switch Settings for the FORTRAN77/ANALYZER Program

Switch	Value	Description
0	0	Default setting. All input is from a remote terminal. All output is to a remote terminal except when directed to a line printer by a command preceded by P or PRINT.
	1	All input is through accept messages from the ODT. All output is to a line printer.
	3	All input is from a card reader and all output is to a line printer.
1	0	Default setting. Page ejects are inserted in the output to a line printer.
	1	Page ejects are suppressed.

The user provides input options through a remote terminal, the ODT, or a card reader, depending on the selected switches. In response to the command prompt ENTER FILE NAME, the user enters the following:

<code file name or ICM file name>

This statement must be preceded by the mix number if SWITCH 0 = 1. The program verifies the file name and either gives an error message or responds with the command prompt:

ENTER OPTIONS, NULL TO STOP, OR HELP

If the user enters HELP, the program provides a list of appropriate options according to the file type the user has indicated. These options determine the part of the code or ICM file to be analyzed. The list of options displayed for a code file follows:

Option	Definition
PPB	Program Parameter Block
FPB	File Parameter Block
SD	Code Segment Dictionary
DD	Data Page Dictionary
DP [page number]	Data Page (one or all, default is all)
LT	Layout table
CODE [number, name or ALL]	
ALL	All of the above
GET <file-name>	Get another file

For an ICM file, the list of options displayed follows:

Option	Definition
ALL	Directory and all ICMs
DIRECTORY	Directory listing
an ICM name	ICM name to be analyzed
HEADER	ICM Header
DATA	ICM Data
FPB	ICM File Parameter Block
CODE	ICM Code
REF	ICM Reference Table
GET <file-name>	Get another file

Options are entered individually or serially. If entered serially, they are separated by comma (,) or blank characters.

If CODE is specified, it can be followed by either a code segment number, a code segment name, or the word ALL indicating that all code segments are to be analyzed. If CODE is entered without qualifications, the program prompts the user to enter more information.

If ICM is specified, only one name can be specified at a time. If qualifying options are not specified, the default is an analysis of the entire ICM.

PROGRAM TERMINATION

The FORTRAN77/ANALYZER program is terminated by transmitting a BYE or a null entry.

ERROR MESSAGES

The following is a list of possible error messages.

<segment name> : INVALID SEGMENT NAME. TRY AGAIN.

<option name> : INVALID OPTION. TRY AGAIN.

<page number> : PAGE NUMBER OUT OF BOUND. TRY AGAIN.

<segment number> : SEGMENT OUT OF BOUND. TRY AGAIN.

<token entered> : INVALID OPTION.
or
INVALID ICM NAME.
or
MORE THAN ONE ICM SPECIFIED.

<ICM name> : ICM NOT FOUND

FILE MISSING : <file name entered>

INVALID FILE NAME

FORTRAN77 ICM OR CODE FILE REQUIRED

APPENDIX F
JOB SPAWNING

The following sample program illustrates the use of job spawning using the B 1000 FORTRAN 77 language. This program zips a sort and then waits for data to be returned before continuing.

```
FILE  4(KIND=DISK,RECL=80,BLOCKSIZE=320,STATUS=NEW,MYUSE=IO,FILE="CQ")
C    MODIFY FILE4 TO TYPE QUEUE AND QMX=10 AFTER COMPILATION.
FILE  5(KIND=DISK,RECL=40,BLOCKSIZE=4000,STATUS=NEW,FILE="SINP")
FILE  7(KIND=DISK,RECL=80,BLOCKSIZE=160,STATUS=NEW,FILE="SORTIN")
FILE  8(KIND=DISK,RECL=40,BLOCKSIZE=4000,STATUS=OLD,FILE="SOUT")
      CHARACTER *80 CQUEUE
      WRITE(7,100)"FILE IN SINP (DISK DEFAULT) OUT SOUT (DISK DEFAULT)"
      WRITE(7,100) "KEY (1 5)"
  100 FORMAT(A)
      CLOSE 7
      DO 200 I = 1, 100
  200 WRITE(5,300) INT(RANDOM(J) * 10000)
  300 FORMAT(I5)
      CLOSE 5
      WRITE(4,*(" ")*)
      BACKSPACE 4
      CALL ZIP("QU CQ EX SORT;FI CARDS NAM SORTIN DSK DEF%")
      I = 0
  310 READ(4,100) CQUEUE
      IF (CQUEUE(1:2) .EQ. "06") THEN
          I = I + 1
          IF (CQUEUE(51:52) .EQ. "00") THEN
              IF (I .LT. 4) GO TO 310
              GO TO 360
          ELSE
              IF (I .EQ. 3) THEN
                  PRINT 100, " SORT INTRINSIC GOT AN ABNORMAL EOJ."
              ELSE
                  PRINT 100, " SORT PROGRAM GOT AN ABNORMAL EOJ."
              ENDIF
              STOP
          ENDIF
      ENDIF
      IF (CQUEUE(1:2) .EQ. "05") I = I + 1
      GO TO 310
  360 DO 400 I = 1, 100
          READ(8,500) J
  400 PRINT 500, J
  500 FORMAT(" ",I5)
      CALL ZIP("RE SORTIN%")
      END
```

APPENDIX G
FORTRAN 77 S-LANGUAGE

INTRODUCTION

The B 1000 FORTRAN 77 S-Language provides the virtual machine interface between the code generated by the FORTRAN 77 compiler and the FORTRAN 77 interpreter. This appendix describes the format of FORTRAN 77 S-instructions and explains each operator as a member of one of the following classes:

ARITHMETIC
LOGICAL REPLACEMENT AND IF STATEMENT
BRANCHING
TYPE AND SIGN CONVERSION
SUBSCRIPT VALUE COMPUTATION
DO-LOOP MAINTENANCE
CHARACTER TYPE
SUBROUTINE LINKAGE
SPECIAL FUNCTION
PRIVILEGED USER
TRIGONOMETRIC AND OTHER FUNCTIONS

All FORTRAN 77 S-Language programs have associated with them a base register and a limit register. The area between the base and the limit is to be used as data space only. All program code, organized in segment form, is stored at any available location in memory according to the memory management algorithms used by the B 1000 operating system.

Various parameters necessary for the running of the S-Language object code and maintained by the operating system are stored beyond the Limit Register in the Run Structure Nucleus (RSN).

BASE-LIMIT MEMORY LAYOUT

Static memory contains:

1. Intrinsic common blocks, consisting of:

   - Overflow/divide-by-zero mask
   - Statement number
   - Debug interface table address
   - Environment nucleus block address
   - Segment dictionary address
   - Data dictionary address
   - Code segment number
   - I/O buffer
   - I/O pointers and variables known to the interpreter
   - Common intrinsic variables

2. Local data blocks, consisting of:

   - Return address
   - Descriptors of dummy arguments
   - Other local data, but not including paged arrays

3. Other common blocks as declared, not including paged blocks.

Dynamic memory contains paged local arrays and/or equivalence groups and paged common blocks. The placement of these elements in dynamic memory is determined by size. Since data pages are byte addressable, a data item can begin at any byte address. A numeric item occupies four or eight bytes of memory. The size of most data pages is 1024 (2**10) bytes, except for the last page of a data block. A data block consists of consecutive data pages as described by the data dictionary.

A code segment contains a transfer vector of variable length followed by executable code consisting of S-operators. Refer to the subsection entitled Formats for the format of a transfer vector.

INSTRUCTION SET

This subsection contains two lists of the instruction set of the FORTRAN 77 S-language. The first is an alphabetical list of mnemonics and the second is a numeric list of operation codes grouped according to function.

Alphabetical List of Mnemonics

Mnemonic	Numeric Operation Code	Function
ABS	68	Absolutize
ACOS	9C	Arccosine
ADDR	6A	Base-relative address
AGO	67	Assigned GOTO
AIF	65	Arithmetic IF
AINT	9E	Floor
ALOG	9F	Natural log
ALOG10	A0	Log to base 10
AMOD	94	Remainder
ASIN	9B	Arcsine
ATAN	9D	Arctangent
BAT	8B	Build array table
BAAT	8C	Build assumed size array table
BNRY	81	Get binary input
BUMP	A3	Bump
CALL	70	Subroutine call
CAT	7B	Character concatenation
CATD	79	Character concatenation with descriptor
CGO	66	Computed GOTO
COMM	7C	Communicate
COS	96	Cosine
COSH	99	Hyperbolic cosine
CREL	5E	Character relation
CRIF	5F	Character relational IF
CS	8D	Compute subscript value
CSB	8E	CS and check bounds
CSV	8F	CS with array table, also checks bounds
DADD	50	Double add

(continued)

Mnemonic	Numeric Operation Code	Function
DBL	57	Double precision
DCAL	71	Dynamic subroutine call
DDIV	53	Double divide
DESC	6B	Make descriptor
DMOVE	02	Move double word
CMUL	52	Double multiply
DO.UP	90	DO loop update
DREL	5C	Double relation
DRIF	5D	Double relational IF
DS	7F	Discontinue job
DSUB	51	Double subtract
EXP	A2	Exponential
FANC	87	Fetch and clear error condition
FADD	0C	Real add
FAMMM	37	Real add – mem,mem,mem
FAMMR	36	Real add – mem,mem,reg
FAMRM	35	Real add – mem,reg,mem
FAMRR	34	Real add – mem,reg,reg
FARMM	33	Real add – reg,mem,mem
FARMR	32	Real add – reg,mem,reg
FARRM	31	Real add – reg,reg,mem
FARRR	30	Real add – reg,reg,reg
FDIV	0F	Real divide
FDMMM	4F	Real divide – mem,mem,mem
FDMMR	4E	Real divide – mem,mem,reg
FDMRM	4D	Real divide – mem,reg,mem
FDMRR	4C	Real divide – mem,reg,reg
FDRMM	4B	Real divide – reg,mem,mem
FDRMR	4A	Real divide – reg,mem,reg
FDRRM	49	Real divide – reg,reg,mem
FDRRR	48	Real divide – reg,reg,reg
FLOAT	54	Convert from integer to floating point
FMMMM	47	Real multiply – mem,mem,mem
FMMMR	46	Real multiply – mem,mem,reg
FMMRM	45	Real multiply – mem,reg,mem
FMMRR	44	Real multiply – mem,reg,reg
FMRMM	43	Real multiply – reg,mem,mem
FMRMR	42	Real multiply – reg,mem,reg
FMRRM	41	Real multiply – reg,reg,mem
FMRRR	40	Real multiply – reg,reg,reg
FMUL	0E	Real multiply
FREL	5A	Real relation
FRIF	5B	Real relational IF
FSMMM	3F	Real subtract – mem,mem,mem
FSMMR	3E	Real subtract – mem,mem,reg
FSMRM	3D	Real subtract – mem,reg,mem
FSMRR	3C	Real subtract – mem,reg,reg

(continued)

Mnemonic	Numeric Operation Code	Function
FSRMM	3B	Real subtract – reg,mem,mem
FSRMR	3A	Real subtract – reg,mem,reg
FSRRM	39	Real subtract – reg,reg,mem
FSRRR	38	Real subtract – reg,reg,reg
FSUB	0D	Real subtract
GOTO	64	Unconditional branch
HMON	92	Hardware monitor (not implemented)
IADD	08	Integer add
IAMMM	17	Integer add – mem,mem,mem
IAMMR	16	Integer add – mem,mem,reg
IAMRM	15	Integer add – mem,reg,mem
IAMRR	14	Integer add – mem,reg,reg
IARMM	13	Integer add – reg,mem,mem
IARMR	12	Integer add – reg,mem,reg
IARRM	11	Integer add – reg,reg,mem
IARRR	10	Integer add – reg,reg,reg
IDIV	0B	Integer divide
IDMMM	2F	Integer divide – mem,mem,mem
IDMMR	2E	Integer divide – mem,mem,reg
IDMRM	2D	Integer divide – mem,reg,mem
IDMRR	2C	Integer divide – mem,reg,reg
IDRMM	2B	Integer divide – reg,mem,mem
IDRMR	2A	Integer divide – reg,mem,reg
IDRRM	29	Integer divide – reg,reg,mem
IDRRR	28	Integer divide – reg,reg,reg
IFIX	55	Convert from floating to integer
IMMMM	27	Integer multiply – mem,mem,mem
IMMMR	26	Integer multiply – mem,mem,reg
IMMRM	25	Integer multiply – mem,reg,mem
IMMRR	24	Integer multiply – mem,reg,reg
IMRMM	23	Integer multiply – reg,mem,mem
IMRMR	22	Integer multiply – reg,mem,reg
IMRRM	21	Integer multiply – reg,reg,mem
IMRRR	20	Integer multiply – reg,reg,reg
IMUL	0A	Integer multiply
INSERT	89	Insert bits
IREL	58	Integer relation
IRIF	59	Integer relational IF
ISMMM	1F	Integer subtract – mem,mem,mem
ISMMR	1E	Integer subtract – mem,mem,reg
ISMRM	1D	Integer subtract – mem,reg,mem
ISMRR	1C	Integer subtract – mem,reg,reg
ISRMM	1B	Integer subtract – reg,mem,mem
ISRMR	1A	Integer subtract – reg,mem,reg
ISRRM	19	Integer subtract – reg,reg,mem
ISRRR	18	Integer subtract – reg,reg,reg

(continued)

Mnemonic	Numeric Operation Code	Function
ISUB	09	Integer subtract
LB	6E	Local base
LCR	7D	Load communicate reply
LEN	6C	Character length
LIF1	63	Logical IF,1-operand
LIF2	61	Logical IF,2-operands
LNOT	62	Logical NOT
LOAD	06	Load register
LOG	60	Logical relation
MMOVE	07	Move memory
MOVE	01	Move single word
MVC	7A	Move character
NEG	69	Change sign
NEXT	84	Examine next character
PASS	6F	Pass descriptor
REAL	83	Get real value
RMOVE	04	Move register
RTN	72	Subroutine return
SAVE	6D	Save registers
SFCL	73	Statement function call
SFRTN	74	Statement function return
SIGN	82	Search for optional sign
SIN	95	Sine
SINH	98	Hyperbolic sine
SNGL	56	Convert from double to single precision
SPAM	75	Scramble and provide arguments
SQRT	A1	Square root
SSTL	77	Substring move
SSTR	76	Substring descriptor
STC	78	Store characters
STMN	91	FORTRAN statement number
STORE	05	Store register
TAN	97	Tangent
TANH	9A	Hyperbolic tangent
TIME	7E	Processor time
VD	8A	Validate descriptor
WEF	85	Write E-format
WFF	86	Write F-format
WID	80	Write integer digits
WIF	93	Write I-format
XTRACT	88	Extract bits

Numeric List of Operation Codes

This subsection contains a numeric list of operation codes grouped according to function.

Arithmetic Replacement S-Operators

Numeric Operation Code	Mnemonic	Function
00		Invalid
01	MOVE	Move single word
02	DMOVE	Move double word
03		Invalid
04	RMOVE	Move register
05	STORE	Store register
06	LOAD	Load register
07	MMOVE	Move memory
08	IADD	Integer add – ss,ss,sd
09	ISUB	Integer subtract – ss,ss,sd
0A	IMUL	Integer multiply – ss,ss,sd
0B	IDIV	Integer divide – ss,ss,sd
0C	FADD	Floating add – ss,ss,sd
0D	FSUB	Floating subtract – ss,ss,sd
0E	FMUL	Floating multiply – ss,ss,sd
0F	FDIV	Floating divide – ss,ss,sd
10	IARRR	Integer add – reg,reg,reg
11	IARRM	Integer add – reg,reg,mem
12	IARMR	Integer add – reg,mem,reg
13	IARMM	Integer add – reg,mem,mem
14	IAMRR	Integer add – mem,reg,reg
15	IAMRM	Integer add – mem,reg,mem
16	IAMMR	Integer add – mem,mem,reg
17	IAMMM	Integer add – mem,mem,mem
18	ISRRR	Integer subtract – reg,reg,reg
19	ISRRM	Integer subtract – reg,reg,mem
1A	ISRMR	Integer subtract – reg,mem,reg
1B	ISRMM	Integer subtract – reg,mem,mem
1C	ISMRR	Integer subtract – mem,reg,reg
1D	ISMRM	Integer subtract – mem,reg,mem
1E	ISMMR	Integer subtract – mem,mem,reg
1F	ISMMM	Integer subtract – mem,mem,mem
20	IMRRR	Integer multiply – reg,reg,reg

(continued)

Numeric Operation Code	Mnemonic	Function
21	IMRRM	Integer multiply – reg,reg,mem
22	IMRMR	Integer multiply – reg,mem,reg
23	IMRMM	Integer multiply – reg,mem,mem
24	IMMRR	Integer multiply – mem,reg,reg
25	IMMRM	Integer multiply – mem,reg,mem
26	IMMMR	Integer multiply – mem,mem,reg
27	IMMMM	Integer multiply – mem,mem,mem
28	IDRRR	Integer divide – reg,reg,reg
29	IDRRM	Integer divide – reg,reg,mem
2A	IDRMR	Integer divide – reg,mem,reg
2B	IDRMM	Integer divide – reg,mem,mem
2C	IDMRR	Integer divide – mem,reg,reg
2D	IDMRM	Integer divide – mem,reg,mem
2E	IDMMR	Integer divide – mem,mem,reg
2F	IDMMM	Integer divide – mem,mem,mem
30	FARRR	Floating add – reg,reg,reg
31	FARRM	Floating add – reg,reg,mem
32	FARMR	Floating add – reg,mem,reg
33	FARMM	Floating add – reg,mem,mem
34	FAMRR	Floating add – mem,reg,reg
35	FAMRM	Floating add – mem,reg,mem
36	FAMMR	Floating add – mem,mem,reg
37	FAMMM	Floating add – mem,mem,mem
38	FSRRR	Floating subtract – reg,reg,reg
39	FSRRM	Floating subtract – reg,reg,mem
3A	FSRMR	Floating subtract – reg,mem,reg
3B	FSRMM	Floating subtract – reg,mem,mem
3C	FSMRR	Floating subtract – mem,reg,reg
3D	FSMRM	Floating subtract – mem,reg,mem
3E	FSMMR	Floating subtract – mem,mem,reg
3F	FSMMM	Floating subtract – mem,mem,mem
40	FMRRR	Floating multiply – reg,reg,reg
41	FMRRM	Floating multiply – reg,reg,mem
42	FMRMR	Floating multiply – reg,mem,reg
43	FMRMM	Floating multiply – reg,mem,mem
44	FMMRR	Floating multiply – mem,reg,reg
45	FMMRM	Floating multiply – mem,reg,mem
46	FMMMR	Floating multiply – mem,mem,reg
47	FMMMM	Floating multiply – mem,mem,mem
48	FDRRR	Floating divide – reg,reg,reg
49	FDRRM	Floating divide – reg,reg,mem
4A	FDRMR	Floating divide – reg,mem,reg
4B	FDRMM	Floating divide – reg,mem,mem
4C	FDMRR	Floating divide – mem,reg,reg

(continued)

Numeric Operation Code	Mnemonic	Function
4D	FDMRM	Floating divide – mem,reg,mem
4E	FDMMR	Floating divide – mem,mem,reg
4F	FDMMM	Floating divide – mem,mem,mem
50	DADD	Double precision add
51	DSUB	Double precision subtract
52	DMUL	Double precision multiply
53	DDIV	Double precision divide
A3	BUMP	Bump

Logical Replacement and IF Statement S-Operators

Numeric Operation Code	Mnemonic	Function
58	IREL	Integer relation
59	IRIF	Integer relational IF
5A	FREL	Floating relation
5B	FRIF	Floating relational IF
5C	DREL	Double relation
5D	DRIF	Double relational IF
60	LOG	Logical relation
61	LIF2	Logical IF – 2 operands
62	LNOT	Logical NOT
63	LIF1	Logical IF – 1 operand

Branch S-Operators

Numeric Operation Code	Mnemonic	Function
64	GOTO	Branch unconditional
65	AIF	Arithmetic IF
66	CGO	Computed GOTO
67	AGO	Assigned GOTO

Type and Sign Conversion S-Operators

Numeric Operation Code	Mnemonic	Function
54	FLOAT	Convert from integer to floating
55	IFIX	Convert from floating to integer
56	SNGL	Convert from double to single precision
57	DBL	Convert from single to double precision
68	ABS	Absolute value
69	NEG	Change sign

Subscript Value Computation S-Operators

Numeric Operation Code	Mnemonic	Function
8A	VD	Validate descriptor
8B	BAT	Build array table
8C	BAAT	Build assumed size array table
8D	CS	Compute subscript value
8E	CSB	CS with bounds checking
8F	CSV	CS with array table, also checks bounds

Do Loop Maintenance

Numeric Operation Code	Mnemonic	Function
90	DO.UP	Update DO loop

Character Type S-Operators

Numeric Operation Code	Mnemonic	Function
5E	CREL	Character relational
5F	CRIF	Character relational IF
6C	LEN	Character length
76	SSTR	Substring descriptor
77	SSTL	Substring move
78	STC	Store characters
79	CATD	Concatenation with descriptor
7A	MVC	Move character
7B	CAT	Concatenation

Subroutine Linkage S-Operators

Numeric Operation Code	Mnemonic	Function
6D	SAVE	Save/restore registers
6E	LB	Local base
6F	PASS	Pass descriptor
70	CALL	Subroutine call
71	DCAL	Dynamic subroutine call
72	RTN	Subroutine return
73	SFCL	Statement function call
74	SFRTN	Statement function return
75	SPAM	Scramble and provide arguments

Special Function S-Operators

Numeric Operation Code	Mnemonic	Function
91	STMN	FORTRAN statement number
92	HMON	Hardware monitor (not implemented)

Privileged User S-Operators

Numeric Operation Code	Mnemonic	Function
6A	ADDR	Base-relative address
6B	DESC	Make descriptor
78	STC	Store characters
7C	COMM	Communicate
7D	LCR	Load communicate reply
7E	TIME	Processor time
7F	DS	Discontinue job
80	WID	Write integer digits
81	BNRY	Get integer input
82	SIGN	Get optional sign
83	REAL	Get floating input
84	NEXT	Examine next input
85	WEF	Write E-format
86	WFF	Write F-format
87	FANC	Fetch and clear error condition
88	XTRACT	Extract bits
89	INSERT	Insert bits
93	WIF	Write I-format

Operators @94@ through @A2@ can also be used as privileged operators with the format: PRIV <desired function> STD_SOURCE ... STD_DESTINATION. These operators are described under Trigonometric and Other Functions.

Trigonometric and Other Functions

All of these operators can be used as privileged operators.

Numeric Operation Code	Mnemonic	Function
94	AMOD	Remainder
95	SIN	Sine
96	COS	Cosine
97	TAN	Tangent
98	SINH	Hyperbolic sine
99	COSH	Hyperbolic cosine
9A	TANH	Hyperbolic tangent
9B	ASIN	Arcsine
9C	ACOS	Arccosine

(continued)

Numeric Operation Code	Mnemonic	Function
9D	ATAN	Arctangent
9E	AINT	Floor
9F	ALOG	Natural log
A0	ALOG10	Log to base 10
A1	SQRT	Square root
A2	EXP	Exponential
@A4@ through @FF@		Invalid

FORMATS

This subsection contains the formats of the elements that make up the S-instructions.

Registers

SCRATCH.PAD ASSIGNMENTS

	A	B
15	TEMP.HI	TEMP.LO
14	ITEMP	SCRATCH.3
13	SCRATCH.5	SCRATCH.6
12	CODE.SEQ.NUMBER	SCRATCH.4
11	SCRATCH	IX-REG 11
10	DATA.DICT.BASE	IX-REG 10
9	BR.REG	IX-REG 9
8	LOCAL.DATA.BLOCK	IX-REG 8
7	TRANSFER.VECTOR / CSB	IX-REG 7
6	NEXT.INST.PTR	IX-REG 6
5	ACC.5 (Hi)	IX-REG 5 (ACC.5 Lo)
4	ACC.4 (Hi)	IX-REG 4 (ACC.4 Lo)
3	ACC.3 (Hi)	IX-REG 3 (ACC.3 Lo)
2	ACC.2 (Hi)	IX-REG 2 (ACC.2 Lo)
1	ACC.1 (Hi)	IX-REG 1 (ACC.1 Lo)
0	SCRATCH.1	SCRATCH.2

Error Condition Information

Six bits that contain error information are located between the base and the limit (at a base-relative address). The six bits are located at byte address 27, following a filler of BIT(2), and are set as follows:

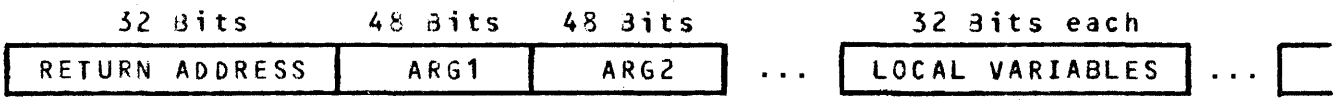
Bit	Bit is set if:
First	An overflow conditions occurs
Second	An exponent underflow condition occurs
Third	A divide-by-zero condition occurs
Fourth	Overflow is permitted
Fifth	Underflow is permitted
Sixth	Divide by zero is permitted

Values

Format:

Integer	
2's complement integer	BIT(32)
Logical	
FALSE = 0, TRUE = not 0	BIT(32)
Real	
sign	BIT(1)
exponent (excess 64; radix 16)	BIT(7)
mantissa	BIT(24)
Double Precision	
sign	BIT(1)
exponent (excess 64; radix 16)	BIT(7)
mantissa	BIT(56)
Character	
byte	BIT(8)
EXTERNAL_CODE_ADDRESS	
filler	BIT(4)
SEGMENT_NUMBER	BIT(10)
bit displacement into segment	BIT(18)

Local Data Block



Subroutine Linkage Mechanism

A 48-bit argument descriptor is built using the PASS S-operator for each actual argument passed.

Execution of the CALL S-operator in the calling subroutine updates the return-address field of the local data block associated with the called subroutine, and control is passed to the called subroutine.

Execution of the RTN S-operator in the called subroutine uses the return address information in its local data block to get back to the calling subroutine. The return-address field is cleared before control is returned to the calling subroutine.

Layout Table

The layout table contains the names and types of variables.

For each subroutine and common block (in a block data subprogram only), there is an entry in this format:

Format:

```
Name of subprogram                          CHARACTER (6)
#TV_ENTRIES                                 BIT ( 8)
#SYMBOL_ENTRIES                             BIT (20)
Address of local data block                 BIT (20)
TV'S(#TV_ENTRIES)                           CHARACTER (6)
SYMBOLS(#SYMBOL_ENTRIES) with the format:

        NAME                                CHARACTER (6)
        LENGTH                              BIT(8)
        ARRAY_VARIANT                       BIT(1)
        TYPE                                BIT(3)
        INDIRECT_VARIANT                    BIT(1)
        BLOCK-RELATIVE ADDRESS              BIT(19)


        where LENGTH has a value of zero for a
        non-character symbol and TYPE has one of
        the following values:

          1 = INTEGER
          2 = REAL
          3 = DOUBLE
          4 = COMPLEX
          5 = LOGICAL
          6 = CHARACTER
          7 = LABEL


        SYMBOLS(#SYMBOL_ENTRIES) can also have the following
        format if the preceding entry describes an actual
        array (ARRAY_VARIANT ON and INDIRECT_VARIANT OFF):

        Number of dimensions                BIT(3)
        Number of elements                  BIT(17)
        Lower bound of first dimension      BIT(8)
        Upper bound of first dimension      BIT(12)
        Lower bound of second dimension     BIT(8)
        Upper bound of second dimension     BIT(12)
        Lower bound of third dimension      BIT(8)
        Upper bound of third dimension      BIT(12)

        The field containing the number of dimensions contains
        the value zero if there are more than three dimensions
        or if the values of the lower or upper bounds exceed
        the sizes of their respective fields.
```

TV'S is an array of transfer vector items that has the number of elements specified by #TV__ENTRIES.

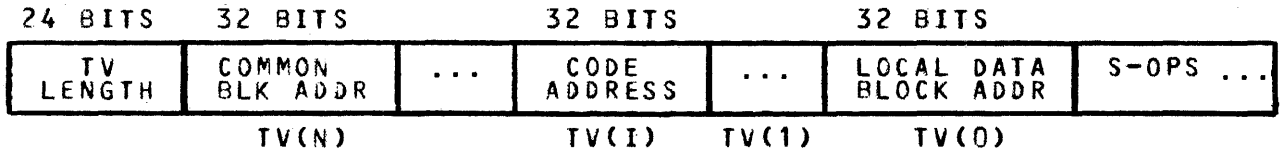
SYMBOLS is an array of symbols that has the number of elements specified by #SYMBOL__ENTRIES and is used by the dump analyzer.

Transfer Vector

Format:

```
Length of transfer vector                                  BIT(24)
SEGMENT DISPLACEMENT CODE ADDRESS(ES)
  [Number of code addresses varies, 0 included]
      Filler (with value of @0@)                           BIT(4)
      CODE SEGMENT NUMBER                                  BIT(10)
      Bit displacement into segment                        BIT(18)

Block address (can be local data block and/or
        common block address)
      Filler (with value of @300@)                         BIT(12)
      Base-relative block address                          BIT(20)
        (byte offset)
```

The transfer vector is zero-relative, beginning at the right where the first S-operator starts. Entry zero of the transfer vector is always the local data block address. Other entries can be either code addresses or common block addresses, which can be intermixed in the transfer vector.



Assigned GOTO and Format Table

The assigned GOTO and FORMAT table is an array of N elements in which (N – 1) is the number of unique labels assigned in a program unit.

Executable statement label entry format:

Filler	BIT(8)
Code address (see description under Addresses in this appendix)	BIT(24)

FORMAT statement label entry format:

Byte offset into local data block	BIT(32)

Table G-1 is a sample assigned GOTO and FORMAT table constructed by the compiler. A sample of FORTRAN 77 code which could generate the table is given in the text following.

Table G-1. Sample Assigned GOTO and FORMAT Table

Column 1 = 32 bits	Column 2 = 32 bits
4	< row 1 unused >
01100000000000000000000000000000	code address for label 5
10000000000000000000000000000000	data address for FORMAT 15
00100000000000000000000000000000	code address for label 10
10000000000000000000000000000000	data address for FORMAT 20

The value in the first row of column 1 is the total number of code addresses and FORMAT label addresses in column 2.

The first bit of each row after the first is 0 for code addresses or 1 for FORMAT label addresses.

For FORMAT label addresses, all bits in column 1 are zero except the first. This zero area is the area pointed to by OP.3 in the assigned GOTO operator, but OP.3 does not exist for VARIANT 6 of the PASS operator.

Each bit in column 1 (except bit 0) specifies whether a branch can be made to the code address in column 2 for a particular assigned GOTO statement. Bit 1 is for the first assigned GOTO statement in the program unit, bit 2 is for the second assigned GOTO statement in the program unit, and so forth.

In table G-1, for example, the first assigned GOTO can only branch to the first code address, and the second assigned GOTO can branch to either code address. The FORTRAN 77 code to generate such a table might look like:

```
      ASSIGN 5 TO N
      ASSIGN 15 TO I
      GO TO N(5)
 5    CONTINUE
10    ASSIGN 10 TO N
      GO TO N(10,5)
      ASSIGN 20 TO J
15    FORMAT( ... )
20    FORMAT( ... )
```

Standard Index

Format:

STD_INDEX

Variant	BIT(1)
Index	Bit varying

Variant	Index	
0	REGISTER_NUMBER	BIT(4)
1	Tag	BIT(3) followed by
	Address	Bit varying

Tag	Address
0	DIRECT_ADDR
1	INDIRECT_ADDR
2	PO_ADDR
3-7	Undefined

Addresses

Format:

```
DIRECT_ADDR
  Local block versus non-paged common block variant        BIT(1)
  If VARIANT=0, byte offset                                BIT(19)
     The 19 bits of offset are in 2's complement.
     (if leading bit is on, the value is negative)
  If VARIANT=1, TV_INDEX followed by                       BIT(10)
     Byte offset.                                          BIT(19)

INDEXED_DIRECT_ADDR
  DIRECT_ADDR                                              Bit varying
  STD_INDEX      (content contains element subscript)      Bit varying

CHARACTER_DIRECT_ADDR
  DIRECT_ADDR                                              Bit varying
  LENGTH         (element length in bytes)                 BIT(8)

INDEXED_CHARACTER_DIRECT_ADDR
  DIRECT_ADDR                                              Bit varying
  STD_INDEX      (content contains element subscript)      Bit varying
  LENGTH         (element length in bytes)                 BIT(8)

PO_ADDR
  Page number                                              BIT(10)
  Byte offset into page                                    BIT(10)

INDEXED_PO_ADDR
  PO_ADDR                                                  BIT(20)
  STD_INDEX      (content contains element subscript)      Bit varying

CHARACTER_PO_ADDR
  PO_ADDR                                                  BIT(20)
  LENGTH         (element length in bytes)                 BIT(8)

INDEXED_CHARACTER_PO_ADDR
  PO_ADDR                                                  BIT(20)
  STD_INDEX      (content contains element subscript)      Bit varying
  LENGTH         (element length in bytes)                 BIT(8)

INDIRECT_ADDR
  Local block versus non-paged common block variant        BIT(1)
  If VARIANT=0, byte offset.                               BIT(19)
     The 19 bits of offset are in 2's complement.
     (if the leading bit is on, the value is negative)
  If VARIANT=1, TV_INDEX followed by                       BIT(10)
     Byte offset                                           BIT(19)

INDEXED_INDIRECT_ADDR
  INDIRECT_ADDR                                            Bit varying
  STD_INDEX      (content of element subscript             Bit varying
                  applies to address passed)

CHARACTER_INDIRECT_ADDR
  INDIRECT_ADDR                                            Bit varying

INDEXED_CHARACTER_INDIRECT_ADDR
  INDIRECT_ADDR                                            Bit varying
  STD_INDEX      (content of element subscript             Bit varying
                  applies to address passed)

CODE_ADDRESS
  Filler         (reserved for further development)        BIT(6)
  Bit displacement into code segment                       BIT(18)

EXTERNAL_CODE_ADDRESS (PASSED)
  Code segment number                                      BIT(14)
  Bit displacement into code segment                       BIT(18)
  Filler                                                   BIT(16)
```

```
REGISTER_NUMBER                                          BIT(4)

MEM_ADDR          (byte offset into local data block)    BIT(20)

INDIRECT_DESCRIPTOR
  Non-indexed versus indexed variant                     BIT(1)
  REGISTER_NUMBER                                        BIT(4)
  If VARIANT=1, STD_INDEX                                Bit varying

Basic descriptor
  Static versus dynamic memory variant                   BIT(1)
  If VARIANT=0, base-relative address                    BIT(19)
  If VARIANT=1, page/offset address                      BIT(19)
  LENGTH                                                 BIT(8)

ARRAY_DESC_PASSED
  Basic descriptor                                       BIT(28)
  NUMBER_ELEMENTS_PASSED                                 BIT(20)

CHAR_DESC_PASSED
  Basic descriptor                                       BIT(28)
  Filler                                                 BIT(20)

PO_ADDR_PASSED
  Basic descriptor                                       BIT(28)
  Filler                                                 BIT(20)
```

Standard Source

Format:

STD_SOURCE

Variant BIT(4)
Source Bit varying

Variant	SOURCE_LOCATION	Contents (SOURCE_LOCATION)
0	Interpreter creates	Integer value = 0
1	Interpreter creates	Integer value = 1
2	Interpreter creates	Integer value = 2
3	Interpreter creates	Integer value = 3
4	In line	4 bit in-line literal
5	In line	10 bit in-line literal
6	In line	32 bit in-line literal
7	In line	64 bit in-line literal
8	DIRECT_ADDR	32/64 bit variable
9	INDIRECT_ADDR	Basic descriptor
A	PO_ADDR	32/64 bit variable
B	Unused	
C	INDEXED_DIRECT_ADDR	Same as 8
D	INDEXED_INDIRECT_ADDR	Same as 9
E	INDEXED_PO_ADDR	Same as A
F	REGISTER_NUMBER	32/64 bit variable

Standard Destination

Format:

STD_DESTINATION

Variant BIT(3)
Destination Bit varying

Variant	Destination
0	DIRECT_ADDR
1	INDIRECT_ADDR
2	PO_ADDR
3	Unused
4	INDEXED_DIRECT_ADDR
5	INDEXED_INDIRECT_ADDR
6	INDEXED_PO_ADDR
7	REGISTER_NUMBER

Standard Character Source

Format:

STD__CHAR__SOURCE

Variant BIT(4)
Source Bit varying

Variant	SOURCE__ADDRESS	Contents (SOURCE__ADDRESS)
0	Invalid	
1	Invalid	
2	Invalid	
3	Invalid	
4	Invalid	
5	Invalid	
6	Invalid	
7	LENGTH(BIT(8)) // IN__LINE	BYTE__LENGTH in-line literal
8	CHARACTER__DIRECT__ADDR	Character
9	CHARACTER__INDIRECT__ADDR	CHARACTER__DESCRIPTOR
A	CHARACTER__PO__ADDR	Character
B	Unused	
C	INDEXED__CHARACTER__DIRECT__ADDR	Character
D	INDEXED__CHARACTER__INDIRECT__ADDR	CHARACTER__DESCRIPTOR
E	INDEXED__CHARACTER__PO__ADDR	Character
F	INDIRECT__DESCRIPTOR	CHARACTER__DESCRIPTOR

Standard Character Destination

Format:

STD__CHAR__DESTINATION

Variant BIT(3)
Destination Bit varying

Variant	Destination
0	CHARACTER__DIRECT__ADDR
1	CHARACTER__INDIRECT__ADDR
2	CHARACTER__PO__ADDR
3	Unused
4	INDEXED__CHARACTER__DIRECT__ADDR
5	INDEXED__CHARACTER__INDIRECT__ADDR
6	INDEXED__CHARACTER__PO__ADDR
7	INDIRECT__DESCRIPTOR

Run-Time Dimension Table

An example of a run-time dimension table constructed by the BAT/BAAT S-operators for array A(J,N,2) is shown in figure G-1.

Scale BIT(24)		BASE_OFFSET BIT(24)
LENGTH	Lower bound BIT(24)	Upper bound BIT(24)
<J>		
<N*J>		
<N*J*2>		

: ..... Assumed size

Last upper bound = maximum possible subscript value.

Figure G-1. Example of Run-Time Dimension Table

ARITHMETIC REPLACEMENT S-OPERATORS

The following is a list of arithmetic replacement S-operators ordered by operation code.

MOVE SINGLE WORD (MOVE)

Operation Code: 01

Format:

8 BITS	VARIES	VARIES
@01@	STD_SOURCE	STD_DESTINATION

STD__SOURCE ::= REAL | INTEGER

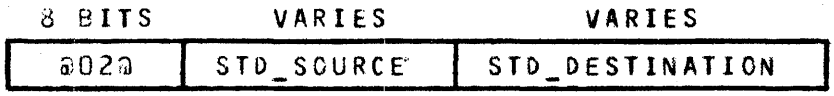
Operation:

STD__DESTINATION := STD__SOURCE

MOVE DOUBLE WORD (DMOVE)

Operation Code: 02

Format:



The standard source is a double-precision value.

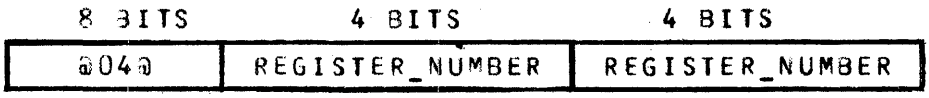
Operation:

STD__DESTINATION := STD__SOURCE

MOVE REGISTER (RMOVE)

Operation Code: 04

Format:



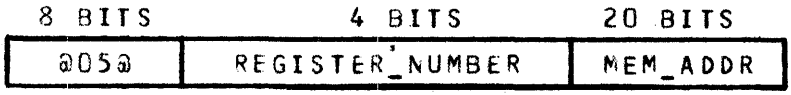
Operation:

Contents of OP.2 get contents of OP.1.

STORE REGISTER (STORE)

Operation Code: 05

Format:



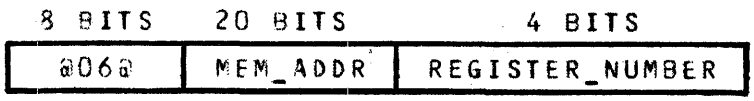
Operation:

Contents of specified register stored at specified MEM__ADDR.

LOAD REGISTER (LOAD)

Operation Code: 06

Format:



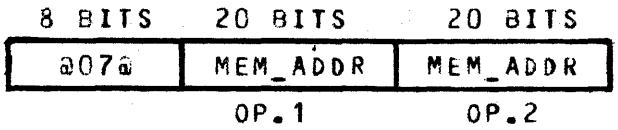
Operation:

Value at MEM__ADDR loaded to specified register.

MOVE MEMORY (MMOVE)

Operation Code: 07

Format:



Operation:

Value at OP.1 moved to OP.2.

ADD (ADD)
SUBTRACT (SUB)
MULTIPLY (MUL)
DIVIDE (DIV)

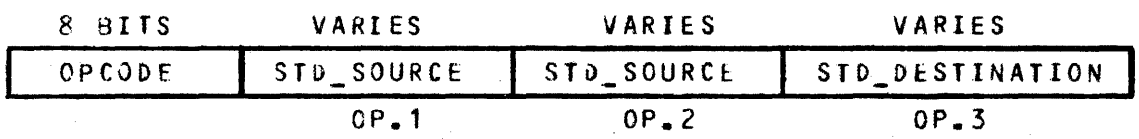
With two operands of standard source and one operand of standard destination.

Operation Codes: shown in table G-2

Table G-2. Operation Codes for ADD, SUBTRACT, MULTIPLY, and DIVIDE

	ADD	SUBTRACT	MULTIPLY	DIVIDE
DOUBLE PRECISION	DADD	DSUB	DMUL	DDIV
	@50@	@51@	@52@	@53@
REAL	FADD	FSUB	FMUL	FDIV
	@0C@	@0D@	@0E@	@0F@
INTEGER	IADD	ISUB	IMUL	IDIV
	@08@	@09@	@0A@	@0B@

Format:



Operation:

STD_DESTINATION := OP.1 [+,−,*,/] OP.2

ADD (ADD)
SUBTRACT (SUB)
MULTIPLY (MUL)
DIVIDE (DIV)

With operands as shown below.

The operands are of the form:

reg,reg,reg	mem,mem,mem
reg,reg,mem	mem,reg,mem
reg,mem,mem	mem,mem,reg
reg,mem,reg	mem,reg,reg

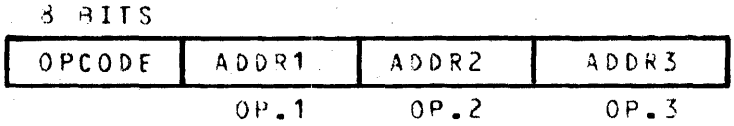
where reg is a REGISTER_NUMBER and mem is a MEM_ADDR.

Operation Codes: shown in table G-3

Table G-3. Operation Codes for ADD, SUBTRACT, MULTIPLY, and DIVIDE

	ADD	SUBTRACT	MULTIPLY	DIVIDE
mem,mem,mem	IAMMM @17@ FAMMM @37@	ISMMM @1F@ FSMMM @3F@	IMMMM @27@ FMMMM @47@	IDMMM @2F@ FDMMM @4F@
mem,mem,reg	IAMMR @16@ FAMMR @36@	ISMMR @1E@ FSMMR @3E@	IMMMR @26@ FMMMR @46@	IDMMR @2E@ FDMMR @4E@
mem,reg,mem	IAMRM @15@ FAMRM @35@	ISMRM @1D@ FSMRM @3D@	IMMRM @25@ FMMRM @45@	IDMRM @2D@ FDMRM @4D@
mem,reg,reg	IAMRR @14@ FAMRR @34@	ISMRR @1C@ FSMRR @3C@	IMMRR @24@ FMMRR @44@	IDMRR @2C@ FDMRR @4C@
reg,mem,mem	IARMM @13@ FARMM @33@	ISRMM @1B@ FSRMM @3B@	IMRMM @23@ FMRMM @43@	IDRMM @2B@ FDRMM @4B@
reg,mem,reg	IARMR @12@ FARMR @32@	ISRMR @1A@ FSRMR @3A@	IMRMR @22@ FMRMR @42@	IDRMR @2A@ FDRMR @4A@
reg,reg,mem	IARRM @11@ FARRM @31@	ISRRM @19@ FSRRM @39@	IMRRM @21@ FMRRM @41@	IDRRM @29@ FDRRM @49@
reg,reg,reg	IARRR @10@ FARRR @30@	ISRRR @18@ FSRRR @38@	IMRRR @20@ FMRRR @40@	IDRRR @28@ FDRRR @48@

Format:



Operation:

Contents (ADDR3) := contents (ADDR1) [+,−,+,/] contents (ADDR2)

BUMP (BUMP)

Operation Code: A3

Format:

Operation:

STD__DESTINATION := STD__DESTINATION + literal

where the addition performed is an integer add.

LOGICAL REPLACEMENT AND IF STATEMENT S-OPERATORS

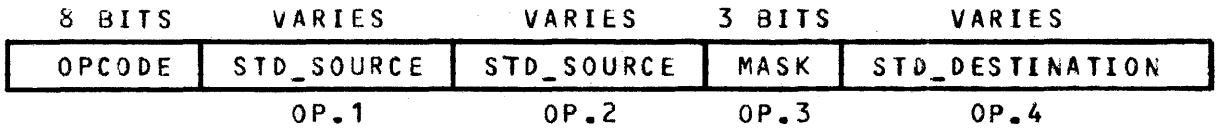
The following is a list of logical replacement and IF statement S-operators ordered by operation code.

RELATION (IREL, STD.SOURCES are of type INTEGER)
(FREL, STD.SOURCES are of type REAL)
(DREL, STD.SOURCES are of type DOUBLE)

Operation Codes:

58 – IREL
5A – FREL
5C – DREL

Format:



Operation:

For interpretation of mask, see DRIF (Op @5D@).

STD__DESTINATION := TRUE if OP.1 relates to OP.2 in any of the ways specified by the mask, otherwise FALSE.

TRUE @FFFFFFFF@ and FALSE @00000000@.

RELATIONAL IF (IRIF, integer relational IF)
(FRIF, floating point relational IF)
(DRIF, double precision relational IF)

Operation Codes:

59 – IRIF
5B – FRIF
5D – DRIF

Format:

8 BITS	VARIES	VARIES	3 BITS	24 BITS
OPCODE	STD_SOURCE	STD_SOURCE	MASK	CODE_ADDRESS
OPCODE	OP.1	OP.2	OP.3	OP.4

Operation:

For a mask of:	Then branch to CODE.ADDRESS if:
1	OP.1 .LE. OP.2
2	OP.1 .GE. OP.2
3	OP.1 .EQ. OP.2
4	OP.1 .NE. OP.2
5	OP.1 .LT. OP.2
6	OP.1 .GT. OP.2

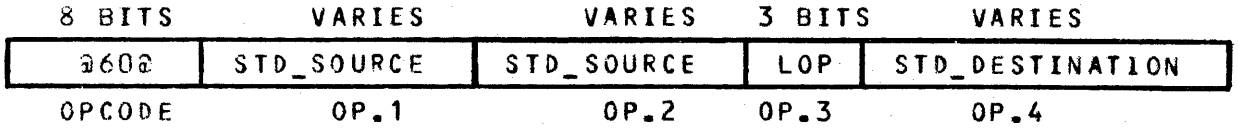
where OP.1 AND OP.2 are of type:

INTEGER	if OP-CODE is IRIF
REAL	if OP-CODE is FRIF
DOUBLE PRECISION	if OP-CODE is DRIF

LOGICAL RELATION (LOG)

Operation Code: 60

Format:



Operation:

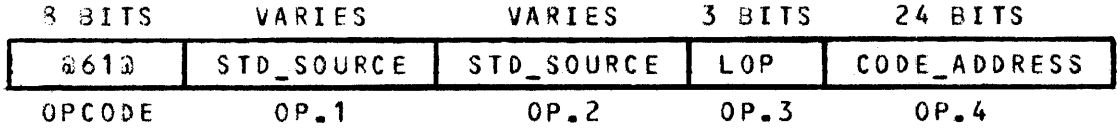
STD__DESTINATION := OP.1 LOP OP.2 where LOP is:

0	→	NEQV	(EXOR)
1	→	AND	
2	→	OR	
3			(not used)
4	→	EQV	(NEOR)
5	→	NAND	
6	→	NOR	
7			(not used)

LOGICAL IF – 2 OPERANDS (LIF2)

Operation Code: 61

Format:



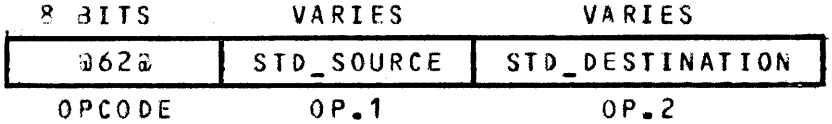
Operation:

For interpretation of the mask, see Logical Relation (Op @60@).
Branch to CODE__ADDRESS if the result of (OP.1 MASK OP.2) is TRUE.

LOGICAL NOT (LNOT)

Operation Code: 62

Format:



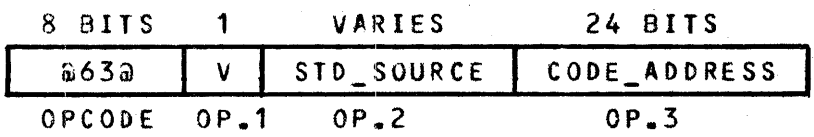
Operation:

STD__DESTINATION := COMPLEMENT(STD__SOURCE)

LOGICAL IF – 1 OPERAND (LIF1)

Operation Code: 63

Format:



Operation:

V=0 → if STD_SOURCE TRUE then branch to CODE_ADDRESS.
V=1 → if STD_SOURCE FALSE then branch to CODE_ADDRESS.

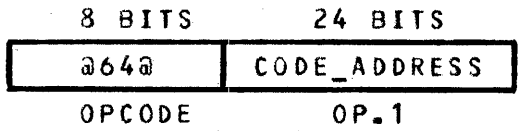
BRANCH S-OPERATORS

The following is a list of branch S-operators ordered by operation code.

UNCONDITIONAL BRANCH (GOTO)

Operation Code: 64

Format:



Operation:

Branch to the CODE_ADDRESS (displacement into code segment).

ARITHMETIC IF (AIF)

Operation Code: 65

Format:

8 BITS	1	VARIES	24 BITS	24 BITS	24 BITS
@65@	VAR	STD_SOURCE	CODE_ADDRESS	CODE_ADDRESS	CODE_ADDRESS
OPCODE	OP.1	OP.2	OP.3	OP.4	OP.5

Operation:

If STD__SOURCE < 0 then branch to CODE__ADDRESS in OP.2
If STD__SOURCE = 0 then branch to CODE__ADDRESS in OP.3
If STD__SOURCE > 0 then branch to CODE__ADDRESS in OP.4

VARIANT = 1 → STD__SOURCE is double precision
VARIANT = 0 → STD__SOURCE is real or integer

The variant only exists so that if the following standard source is a literal, NIP can be aligned correctly at code address as the operation is stepped through.

COMPUTED GOTO (CGO)

Operation Code: 66

Format:

8 BITS	VARIES	8 BITS	24 BITS		24 BITS
@66@	STD_SOURCE	COUNT	CODE_ADDRESS	...	CODE_ADDRESS
OPCODE	OP.1	OP.2	OP3	...	OP.COUNT

STD__SOURCE ::= contains an index into the list of code addresses following.

COUNT ::= the number of CODE__ADDRESS following.

CODE__ADDRESS := the address to branch to.

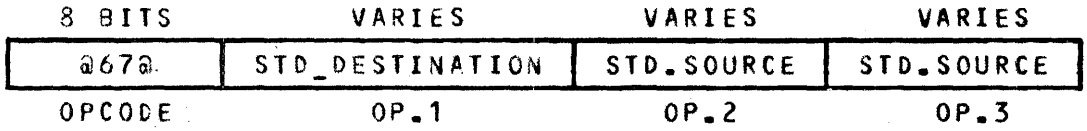
Operation:

Branch to the code address pointed to by the index in STD__SOURCE.

ASSIGNED GOTO (AGO)

Operation Code: 67

Format:



OP.1 points to a table with N rows and 2 columns where (N − 1) is the number of unique labels assigned in a program unit.

Operation:

If the most significant bit of OP.2 is zero (SUBBIT(OP.2,0,1) = 0), then the variable is not assigned a label and an error is given.

If the value of SUBBIT(OP.2,1) is greater than the value of the first entry in the table (pointed to by OP.1), the assigned label is not in the table and an error is given.

If the OP.3 bit of the table entry whose row equals OP.2 in the first column of the table equals 0 (SUBBIT (OP.1(OP.2,1),OP.3,1) = 0), the label is out of scope and an error is given.

Otherwise, if none of the above conditions is TRUE, branch to the code address found at the table entry whose row equals OP.2 in the second column of the table (OP.1(OP.2,2)).

Refer to the format section at the beginning of this appendix for the description of the assigned GOTO table.

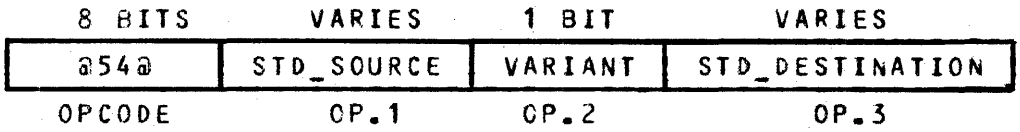
TYPE AND SIGN CONVERSION S-OPERATORS

The following is a list of type and sign conversion S-operators ordered by operation code.

CONVERT FROM INTEGER TO REAL OR DOUBLE PRECISION (FLOAT)

Operation Code: 54

Format:



Operation:

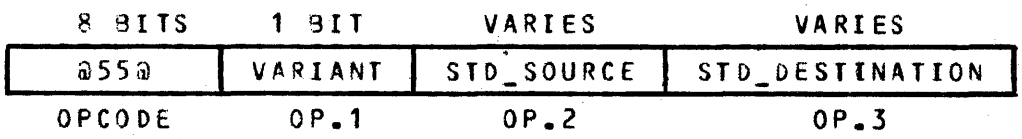
STD__DESTINATION := FLOAT(STD__SOURCE)

If VARIANT = 1 , STD__DESTINATION is double precision.
If VARIANT = 0 , STD__DESTINATION is real.

CONVERT FROM REAL OR DOUBLE PRECISION TO INTEGER (IFIX)

Operation Code: 55

Format:



Operation:

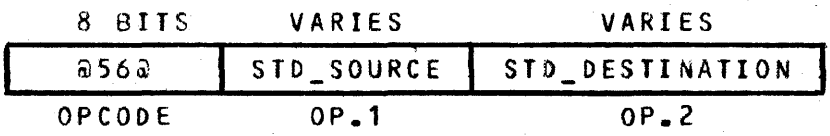
STD__DESTINATION := IFIX(STD__SOURCE)

If VARIANT = 1, STD__SOURCE is double precision.
If VARIANT = 0, STD__SOURCE is real.

CONVERT FROM DOUBLE TO REAL (SNGL)

Operation Code: 56

Format:



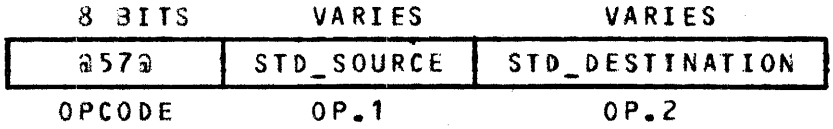
Operation:

STD__DESTINATION := SNGL(STD__SOURCE)

CONVERT FROM REAL TO DOUBLE PRECISION (DBL)

Operation Code: 57

Format:



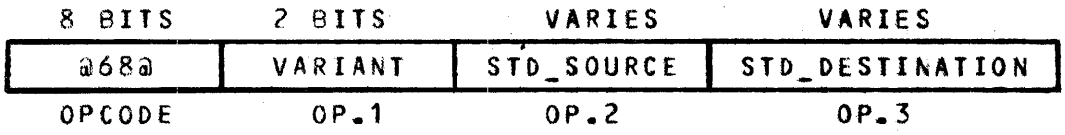
Operation:

STD__DESTINATION := DBL(STD__SOURCE)

ABSOLUTE VALUE (ABS)

Operation Code: 68

Format:



Variant	STD__SOURCE
00	Floating point number
01	Double-precision number
10	Integer
11	Undefined

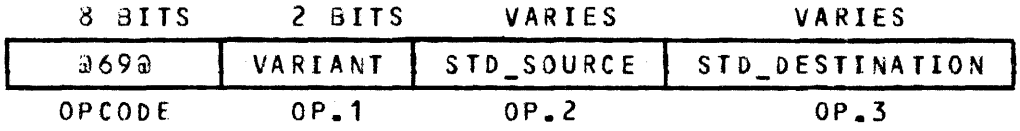
Operation:

STD__DESTINATION := absolute value (STD__SOURCE)

CHANGE SIGN (NEG)

Operation Code: 69

Format:



Variant	STD_SOURCE
00	Real
01	Double precision
10	Integer
11	Unused

Operation:

STD_DESTINATION := – (STD_SOURCE)

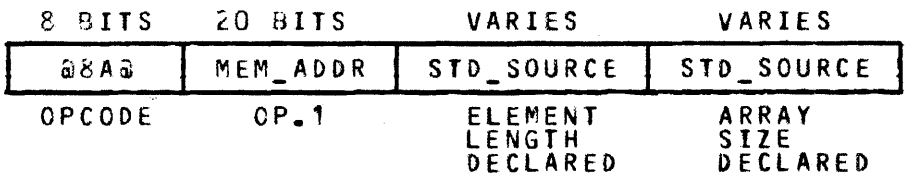
SUBSCRIPT VALUE COMPUTATION S-OPERATORS

The following is a list of subscript value computation S-operators ordered by operation code.

VALIDATE DESCRIPTOR (VD)

Operation Code: 8A

Format:



MEM_ADDR is the block-relative address of the descriptor of the argument passed. This descriptor describes where the actual argument is, its element length passed, and the actual size passed.

ELEMENT LENGTH DECLARED EQL 0 implies assumed-length.

ARRAY SIZE DECLARED is the number of elements passed if the array is a numeric array, or the number of bytes passed if the array is a character array, or zero if the array is a character variable. It points to the size field of the run-time dimension table built by the BAT/BAAT S-operators if it is assumed size or adjustable size. Otherwise, it is a literal.

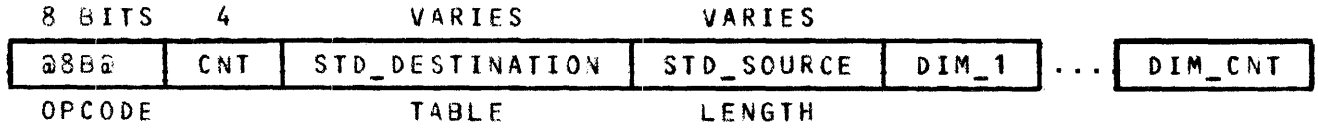
Operation:

1. Verify that the size of the array declared is less than or equal to the size of the array passed.
2. Modify the descriptor at MEM_ADDR to reflect the declared element length.

BUILD ARRAY TABLE (BAT)

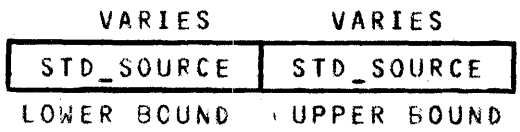
Operation Code: 8B

Format:



CNT ::= number of dimensions

Format of DIM__#:



Operation:

Construct an array descriptor using the remaining information of the format:

```
OFFSET          BIT(32)  see item 1 below
DIM_INFO_1      RECORD1
DIM_INFO_2      RECORD1
    ...
DIM_INFO_CNT    RECORD1
SIZE            BIT(32)  see item 2 below
```

where

```
A_1 = 1
A_# = A_(#-1)*D_(#-1)

D_# = (1 + UPPER_BOUND_#
       - LOWER_BOUND_#)
```

RECORD1 format:

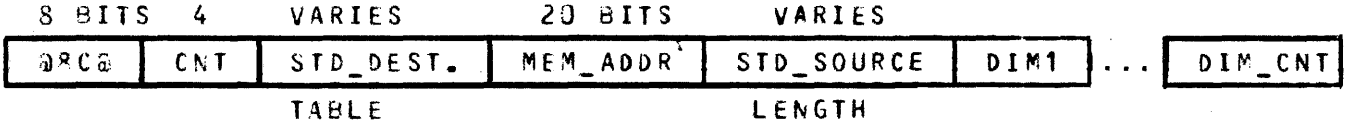
```
A_#             BIT(32)
LOWER_BOUND     BIT(32)
UPPER_BOUND     BIT(32)
```

1. OFFSET = −(LOWER__BOUND__1 * A__1 + ... + LOWER__BOUND__CNT * A__CNT)
2. SIZE = A__(CNT+1)*LENGTH (or what is passed in if assumed-size)
3. UPPER__BOUND__CNT is replaced by SIZE/LENGTH

BUILD ASSUMED SIZE ARRAY TABLE (BAAT)

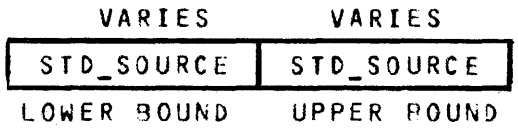
Operation Code: 8C

Format:



CNT = number of dimensions

Format of dimension information:



Operation:

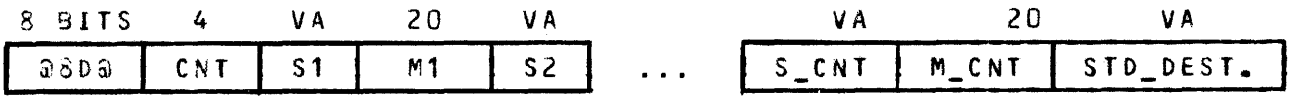
Same as the BAT S-operator except:

1. MEM__ADDR is used to fetch either assumed length and/or assumed size.
2. If LENGTH = 0, use assumed length.
3. If dimension count of upper bound = 0, use assumed size.

COMPUTE SUBSCRIPT VALUE (CS)

Operation Code: 8D

Format:



CNT ::= number of subscripts present

S ::= STD__SOURCE (subscripts)

M# ::= scale factor

Operation:

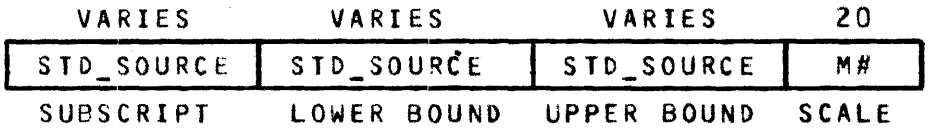
STD__DESTINATION := S1 * M1 + S2 * M2 + ... + S__CNT * M__CNT

CS WITH BOUNDS CHECKING (CSB)

Operation Code: 8E

Format:

S# Format:



Operation:

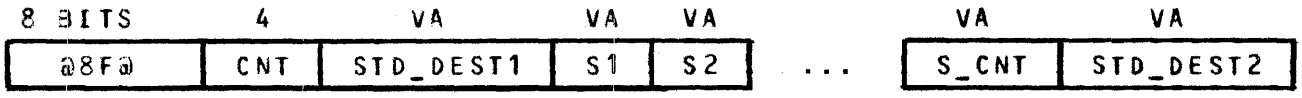
Same as for CS except that the subscript is checked to verify that lower bound is LEQ subscript LEQ upper BOUND before the index value is computed.

STD__DESTINATION := S1 * M1 + ... + S__CNT * M__CNT

COMPUTE SUBSCRIPT VALUE WITH ARRAY TABLE, CHECK BOUNDS (CSV)

Operation Code: 8F

Format:



CNT ::= 4-bit literal indicating number of dimensions

S ::= STD__SOURCE (subscripts)

STD__DEST1 is address of array descriptor as described by BAT.

Operation:

STD__DEST2 := OFFSET + (S1 * A__1 + S2 * A__2 + ... + S__CNT * A__CNT

Subscript bounds checking is always performed.

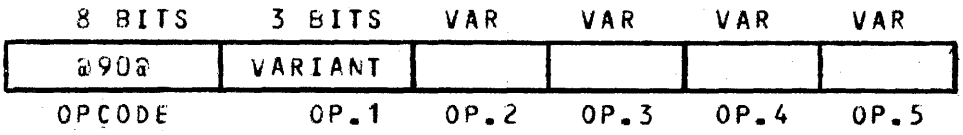
DO-LOOP MAINTENANCE

The following is a list of DO-loop maintenance S-operators ordered by operation code.

DO LOOP UPDATE (DO.UP)

Operation Code: 90

Format:



Variant	OP.2	OP.3	OP.4	OP.5
0	STD_INDEX	STD_DEST	STD_SOURCE	CODE_ADDRESS
1	STD_INDEX	STD_SOURCE	CODE_ADDR	STD_DESTINATION
2	STD_INDEX	STD_DEST	STD_SOURCE	CODE_ADDRESS
3	STD_INDEX	STD_DEST1	CODE_ADDR	STD_DEST2
4	–	STD_SOURCE	CODE_ADDR	–
5	–	STD_DEST	CODE_ADDR	–

Operation:

Variant	DO-type	Loc	Function

0	DO 2 I = N,M	Top	IF (STD__INDEX GT STD__SOURCE) THEN
			STD__DESTINATION = STD__INDEX
			GO TO CODE__ADDRESS
			END IF

1	DO 2 I = N,M	Btm	STD__INDEX = STD__INDEX + 1 IF (STD__INDEX LE STD__SOURCE) * GO TO CODE__ADDRESS STD__DESTINATION = STD__INDEX

2	Do 2 I=N,M,L
	% STD__SOURCE = ITER__CNT
	% REG = CTR__VAR
		Top	If (STD_SOURCE LE O) THEN
			STD__DESTINATION = STD__INDEX
			GO TO CODE__ADDRESS
			END IF

3	DO 2 I = N,M,L %STD__INDEX = CTR__VAR %STD__DEST1(0) = INCR %STD__DEST1(1) = ITER__CNT %STD__DEST2 = CTR__VAR	Btm	STD__INDEX = STD__INDEX + STD__DEST1(0) STD__DEST1(1) = STD__DEST1(1) - 1 IF (STD__DEST1(1) GT 0) * GO TO CODE__ADDRESS STD__DEST2 = STD__INDEX
4	DO 2 X = A,B,C %STD__SOURCE = ITER__CNT	Top	IF (STD__SOURCE LE 0) * GO TO CODE__ADDRESS
5	DO 2 X = A,B,C %STD__DEST. = ITER__CNT	Btm	%CTR__VAR incremented separately STD__DESTINATION = STD__DEST. - 1 IF (STD__DEST. GT 0) GO TO CODE__ADDR.

NOTE

The top DO.UP S-operator is generated only if the compiler cannot determine whether or not the DO loop is executed at least once.

STD__DEST1 in the form of VARIANT 3 is a 2-element array.

CHARACTER TYPE S-OPERATORS

The following is a list of character type S-operators ordered by operation code.

CHARACTER RELATION (CREL)

Operation Code: 5E

Format:

8 BITS	VARIES	VARIES	3 BIT	VARIES
@5E@	STD_CHAR_SOURCE	STD_CHAR_SOURCE	MSK	STD_DESTINATION
OPCODE	OP.1	OP.2	OP.3	OP.4

Operation:

STD_DESTINATION := TRUE if OP.1 relates to OP.2 in any of the ways specified by the mask; otherwise, FALSE.

For interpretation of the mask, see Character Relational If (Op @5F@).

Right blank fill is assumed.

CHARACTER RELATIONAL IF (CRIF)

Operation Code: 5F

Format:

8 BITS	VARIES	VARIES	3 BIT	24 BITS
@5F@	STD_CHAR_SOURCE	STD_CHAR_SOURCE	MSK	CODE_ADDRESS
OPCODE	OP.1	OP.2	OP.3	OP.4

Operation:

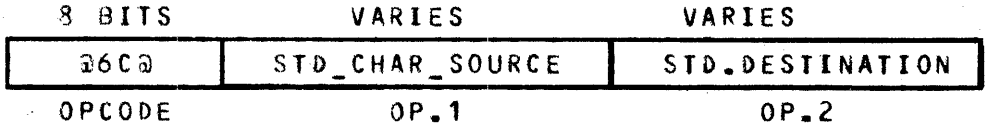
For a mask of:	then branch to CODE.ADDRESS if:
1	OP.1 .GT. OP.2
2	OP.1 .LT. OP.2
3	OP.1 .NE. OP.2
4	OP.1 .EQ. OP.2
5	OP.1 .GE. OP.2
6	OP.1 .LE. OP.2

Right blank fill is assumed.

LENGTH (LEN)

Operation Code: 6C

Format:



Operation:

STD__DESTINATION := CHAR__LENGTH (STD__CHAR__SOURCE)

Length of STD__CHAR__SOURCE is returned in bytes.

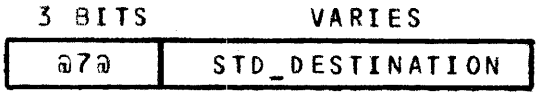
SUBSTRING DESCRIPTOR (SSTR)

Operation Code: 76

Format:

where LAST__OP can be either:

or



Operation:

If LAST__OP is of STD__CHAR__DESTINATION form, then

STD__CHAR__DEST2 = STD__CHAR__DEST1 (STD__SOURCE1 : STD__SOURCE2),

otherwise,

STD__DESTINATION = CHARACTER__DESCRIPTOR of
(STD__CHAR__DEST1 (STD__SOURCE1 : STD__SOURCE2)).

Right truncation or blank fill occurs.

SUBSTRING MOVE (SSTL)

Operation Code: 77

Format:

8 BITS	VARIES	VARIES	VARIES	VARIES
@77@	STD_CHAR_SOURCE	STD_CHAR_DEST.	STD_SOURCE1	STD_SOURCE2

Operation:

STD__CHAR__DEST.(STD__SOURCE1 : STD__SOURCE2) = STD__CHAR__SOURCE

Right truncation or blank fill occurs.

STORE CHARACTER (STC)

Operation Code: 78

Format:

8 BITS	VARIES	VARIES	VARIES	VARIES
@78@	STD_CHAR_SOURCE	STD_CHAR_DEST.	STD_DEST.	STD_SOURCE
OPCODE	OP.1	OP.2	OP.3	OP.4

Operation:

STD__CHAR__DEST(STD__DEST : STD__DEST+STD__SOURCE) =
STD__CHAR__SOURCE//STD__CHAR__SOURCE//STD__CHAR__SOURCE//...

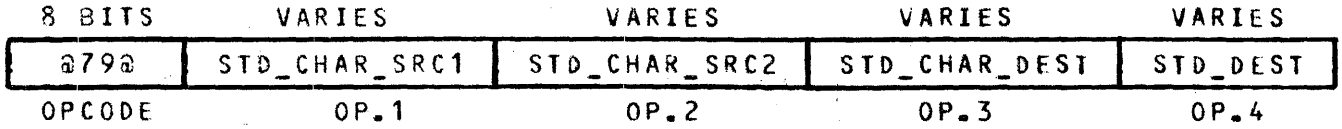
STD__DESTINATION := STD__DESTINATION + STD__SOURCE

Action automatically stops if LEN(STD__CHAR__DESTINATION) is reached.

CHARACTER CONCATENATION WITH DESCRIPTOR (CATD)

Operation Code: 79

Format:



Operation:

STD__CHAR__DESTINATION = STD__CHAR__SRC1 // STD__CHAR__SRC2

and

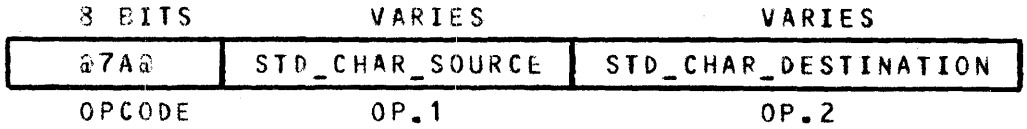
STD__DESTINATION = CHARACTER__PO__ADDR descriptor of
(STD__CHAR__DEST, MIN (LEN(STD__CHAR__DEST),
LEN(STD__CHAR__SRC1) + LEN(STD__CHAR__SRC2)))

Right truncation or blank fill occurs.

MOVE CHARACTERS (MVC)

Operation Code: 7A

Format:



Operation:

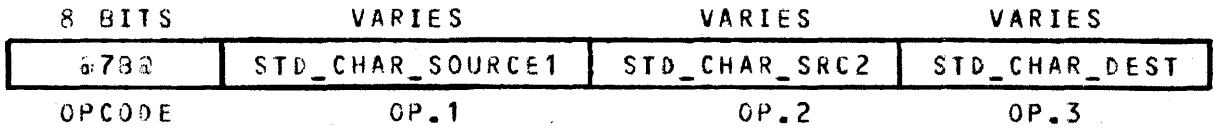
STD__CHAR__DESTINATION := STD__CHAR__SOURCE

Right truncation or blank fill as needed.

CHARACTER CONCATENATION (CAT)

Operation Code: 7B

Format:



Operation:

STD__CHAR__DESTINATION = STD__CHAR__SOURCE1 // STD__CHAR__SRC2

Right truncation or blank fill occurs.

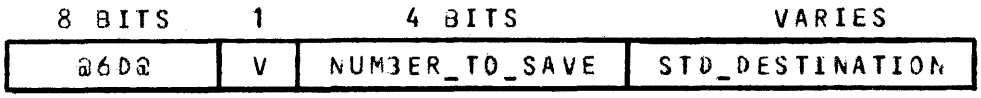
SUBROUTINE LINKAGE S-OPERATORS

The following is a list of subroutine linkage S-operators ordered by operation code.

SAVE REGISTERS (SAVE)

Operation Code: 6D

Format:



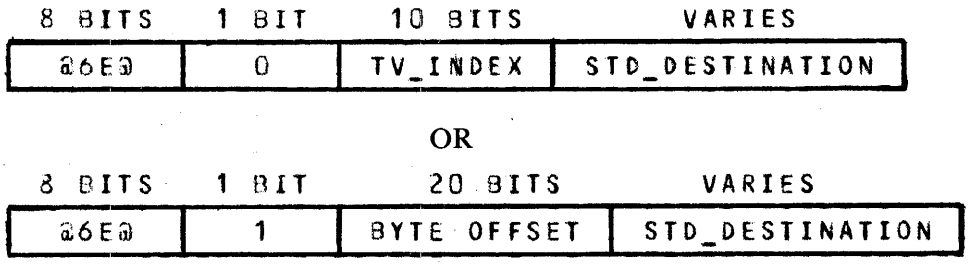
Operation:

If V = 0 save NDX__REGS #6 thru (1 + NUMBER_TO_SAVE) at STD_DEST
IF V = 1 restore NDX__REGS #6 thru (1 + NUMBER__TO_SAVE) from STD_DEST

LOCAL BASE (LB)

Operation Code: 6E

Format:



Operation:

If VARIANT = 0 then,

STD__DESTINATION := base-relative address of local data block for ENTRY(TV__INDEX).

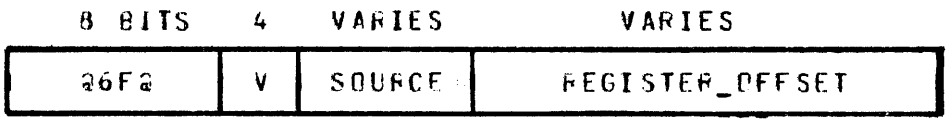
If VARIANT = 1 then,

STD__DESTINATION := base-relative address of local data block pointed at by code address located at given byte offset in current local data block.

PASS ACTUAL ARGUMENT (PASS)

Operation Code: 6F

Format:

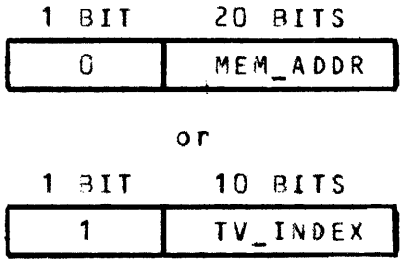


VARIANT	Source	Value Passed
0	STD__DESTINATION	word address
1	STD__CHAR__DESTINATION	CHARACTER__DESCRIPTOR
2	CODE__ADDRESS	code address (see below)
3	STD__SRC, STD__DEST	ARRAY__DESCRIPTOR (single word)
4	STD__SRC, STD__DEST	ARRAY__DESCRIPTOR (double word)
5	STD__SRC, STD__CHAR__DEST	character array descriptor
6	STD__DEST,STD__SRC	assigned format (ARRAY__DESC__PASSED)
7	STD__SRC, STD__CHAR__DEST, STD__SRC, STD__SRC	character array descriptor (used for substring)
8-F	unused	

Operation:

A descriptor is constructed and written to the location described by the REGISTER__OFFSET. Further information about this operation follows:

1. When the VARIANT equals 2, the specified code address is of the form:



2. When the VARIANT equals 3 or 4, the standard source operand is the array size and the standard destination operand is the first array element.

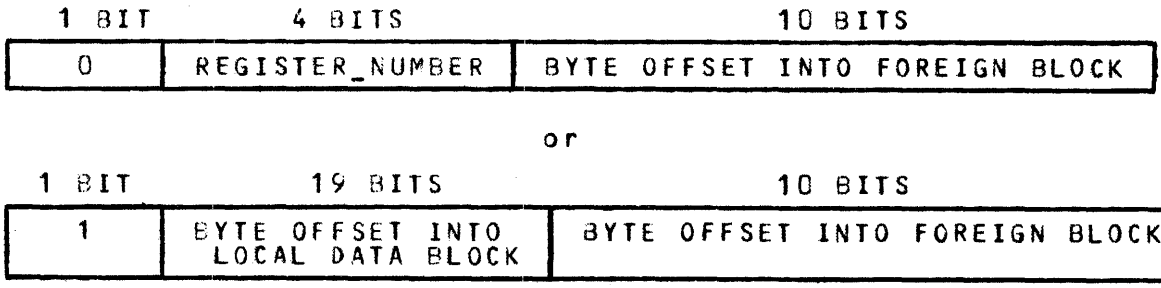
3. When the VARIANT equals 6, the standard destination operand points to the assigned FORMAT table (consisting of N rows and 2 columns). Refer to the Formats subsection of this appendix for a description of the assigned FORMAT table.

   The following error checking is done:

   1) If the most significant bit of the standard source operand is zero, then the variable is not assigned a label and an error is given.
   2) If the value of SUBBIT(STD__SRC,1) is greater than the value of the first table entry, then the assigned label is not in the table and an error is given.
   3) If the zero bit of the table entry whose row equals the STD__SRC in the first column of the table equals 0, then the label is not a FORMAT label and an error is given.

   If no error is detected, the array descriptor passed is constructed which points to the FORMAT address found in the row equal to the standard source operand in column 2 of the assigned FORMAT table. The descriptor passed reflects @FFFF@ elements and a length equal to zero. The FORMAT address pointed to by the descriptor is the local base-relative byte address of the first array element.

4. When the VARIANT equals 7, the first standard source operand is the array size. STD__CHAR__DEST(SECOND STD__SRC:THIRD STD__SRC) is the first array element. The second standard source operand is the first character and the third standard source operand is the last character of the standard character destination.
5. The REGISTER__OFFSET has one of two formats:



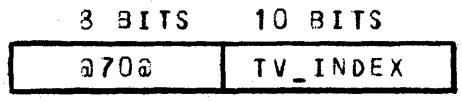
   The base-relative bit address (the index) added to the byte offset (DIRECT__ADDR) is the address of the destination of the 48-bit descriptor.

   All arguments are passed by address, except code addresses. If an argument is passed in the user program by value, a copy is made and the address of the copy is passed as the argument.

SUBROUTINE CALL (CALL)

Operation Code: 70

Format:



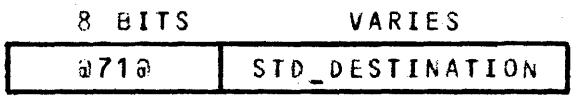
Operation:

Branch to external code address. The subprogram address is a 10-bit transfer vector index.

DYNAMIC SUBROUTINE CALL (DCAL)

Operation Code: 71

Format:



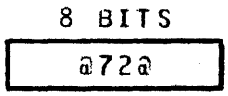
Operation:

Same as CALL except branch to CODE_SEGMENT_DISPLACEMENT found at address contained in the standard destination.

RETURN (RTN)

Operation Code: 72

Format:



Operation:

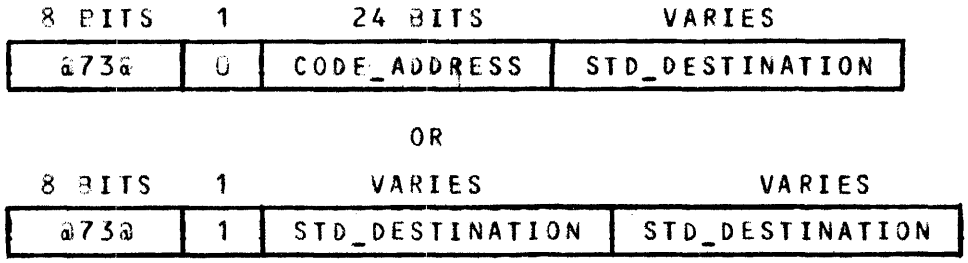
NEXT_INST_PTR := return address (from memory location 0 in local data block).

Returns to address specified in local data block.

STATEMENT FUNCTION CALL (SPFCL)

Operation Code: 73

Format:



When the one-bit VARIANT = 1, the first standard destination contains an external code address.

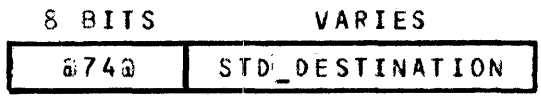
Operation:

Store NEXT__INST__PTR as EXTERNAL__CODE__ADDRESS at STD__DESTINATION (the first operand following the variant), then branch to specified code address (when VARIANT = 0) or to external code address contained in first STD__DESTINATION (when VARIANT = 1).

STATEMENT FUNCTION RETURN (SFRTN)

Operation Code: 74

Format:



Operation:

Branch to EXTERNAL__CODE__ADDRESS stored in STD__DESTINATION.

SCRAMBLE AND PROVIDE ARGUMENTS (SPAM)

Operation Code: 75

Format:

Operation:

#PARMS__PASSED are copied to an interpreter implied location. (#PARMS__PASSED + #PARMS__TO__ADD) 48-bit descriptors are zeroed out. #PARMS__PASSED are copied back as specified by ORDER__INFO.

ORDER__INFO is an array of #PARMS__PASSED elements; each element is BIT(20) and contains addresses which are the locations where the 48-bit argument descriptors are to be written. Length of ORDER__INFO is (#PARMS__PASSED * 20).

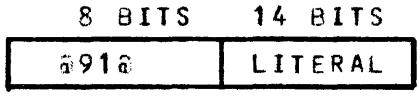
SPECIAL FUNCTION S-OPERATORS

The following is a special function S-operator.

FORTRAN STATEMENT NUMBER (STMN)

Operation Code: 91

Format:



Operation:

Move the 14-bit literal to the FORTRAN statement number field in the base-limit area.

Debug interpreters also check for breakpoint matching.

PRIVILEGED USER S-OPERATORS

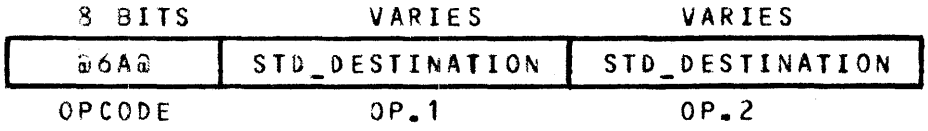
The following is a list of privileged user S-operators ordered by operation code.

NOTE
All references to BUFF, OFFSET, SIZE, BZFLG, and so forth (common references used in intrinsics) indicate predefined locations in memory. These values are not passed in the operator itself.

BASE-RELATIVE ADDRESS (ADDR)

Operation Code: 6A

Format:



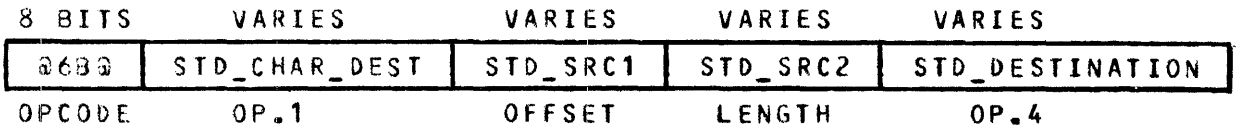
Operation:

OP.2 := ADDRESS(OP.1) – BASE_REGISTER

INDIRECT DESCRIPTOR OF NUMERIC ARRAYS (DESC)

Operation Code: 6B

Format:



Operation:

STD_DESTINATION := (SEGMENT, DISPLACEMENT, LENGTH)
Descriptor of SUBSTR (STD_CHAR_DEST, OFFSET, LENGTH)

NOTE
STD_CHAR_DEST describes the first byte of a numeric field. No checks are made for valid values of OFFSET or LENGTH.

STORE CHARACTER (STC)

Operation Code: 78

STC is also a character type S-operator.

COMMUNICATE (COMM)

Operation Code: 7C

Format:

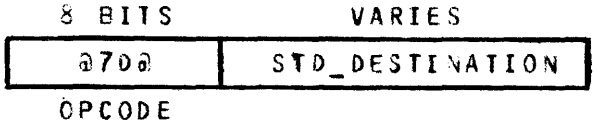
Operation:

Communicate (to the MCP) the information which is left-justified in standard destination. STD_SOURCE indicates the number of bits to be communicated.

LOAD COMMUNICATE REPLY (LCR)

Operation Code: 7D

Format:



Operation:

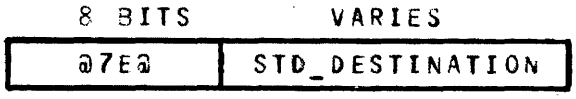
STD__DESTINATION := last 24 bits of the communicate reply.

It is expected that STD__DESTINATION is of type INTEGER.

PROCESSOR TIME (TIME)

Operation Code: 7E

Format:



Operation:

STD__DESTINATION := PROCESSOR__TIME.

DISCONTINUE JOB (DS)

Operation Code: 7F

Format:

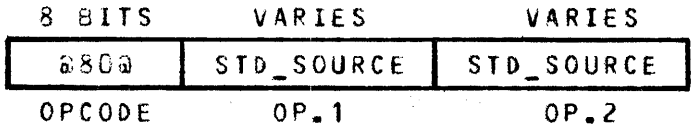
Operation:

Causes host to be aborted giving the message contained at STD__CHAR__DESTINATION.

WRITE INTEGER DIGITS (WID)

Operation Code: 80

Format:



Operation:

Writes OP.1 to predefined memory location the value, DECIMAL(OP.1, for the length specified in OP.2 (in bytes).

Left truncation of zero fill occurs as needed.

BINARY CONVERSION (BNRY)

Operation Code: 81

Format:

Operation:

Write to the location specified by the standard destination the value which is the SUBSTR of BUFF, offset by BFRPTR, for the length specified by size.

Binary conversion stops when a non-numeric/non-blank is encountered. The offset and size locations are updated to reflect the index and length of the remaining substring.

If BZFLG is TRUE, blank characters are treated as zeroes; otherwise, blank characters are treated as nulls.

SEARCH FOR SIGN (SIGN)

Operation Code: 82

Format:

Operation:

Reduce SUBSTR (BUFF, OFFSET, SIZE) until first NEQ " " (by adjusting offset and size). On EOS DO; STD_DESTINATION := .FALSE.; RETURN; END;

STD_DESTINATION := (first char) .EQ. "−" ;

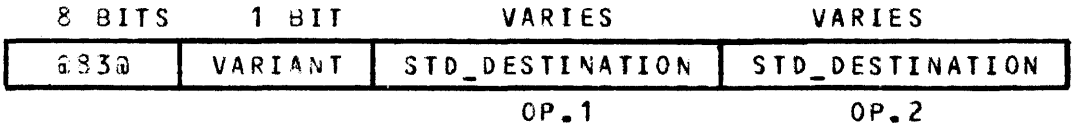
If (first char) = + or −, reduce the adjusted substring to skip the sign.

STD_DESTINATION reflects whether or not a minus sign (-) character is found.

GET REAL VALUE (REAL)

Operation Code: 83

Format:



Operation:

OP.1 := float the binary value of SUBSTR (BUFF, OFFSET, SIZE)

OP.2 := number of decimal digits found after the decimal point

Real value conversion stops when it encounters a character which is none of the following: a decimal digit, a blank, a decimal point. Offset and size are updated to reflect the index and length of the remaining substring.

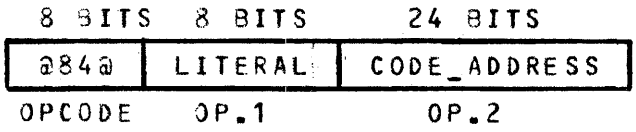
If BZFLG is TRUE, blank characters are treated as zeroes; otherwise, blanks are treated as nulls.

If VARIANT @(1)1@ then STD_DESTINATION is double precision else STD_DESTINATION is single precision

EXAMINE NEXT CHARACTER (NEXT)

Operation Code: 84

Format:



Operation:

```
If (SIZE .EQ. 0) THEN RETURN ;
IF SUBSTR (BUFF,OFFSET,1) .EQ. OP.1
   THEN DO ;
     BUMP OFFSET BY 1 ;
     DECREMENT SIZE BY 1 ;
     BRANCH TO CODE_ADDRESS ;
   END ;
```

SIZE is the length of the substring represented by BUFF.

WRITE E-FORMAT (WEF)

Operation Code: 85

Format:

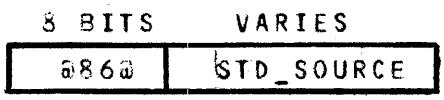
Operation:

Write to a predefined memory location in E-format the value contained in the standard source.

WRITE F-FORMAT (WFF)

Operation Code: 86

Format:



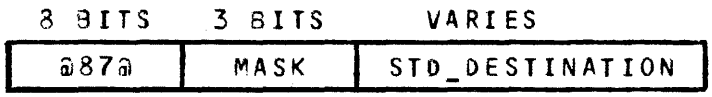
Operation:

Write to a predefined memory location in F-format the value contained in the standard source.

FETCH AND CLEAR ERROR CONDITION (FANC)

Operation Code: 87

Format:



Operation:

Write to STD__DESTINATION the value:
MASK .AND. ERROR__CONDITION__INFO

ERROR__CONDITION__INFO := (.NOT. MASK) .AND. ERROR__CONDITION__INF

where ERROR__CONDITION__INFO is three bits of information located at a base-relative address. It has the same format as the mask:

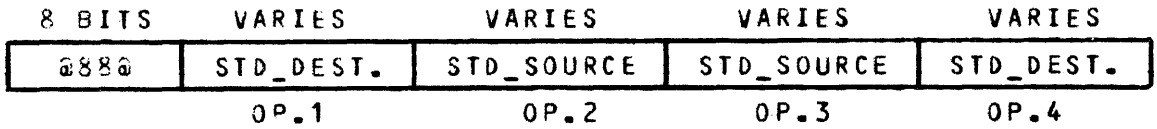
First bit	:	overflow condition
Second bit	:	exponent underflow condition
Third bit	:	divide-by-zero condition

The .AND. operation is done bit by bit.

EXTRACT BITS (XTRACT)

Operation Code: 88

Format:



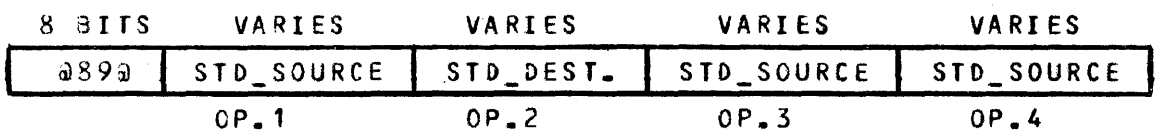
Operation:

OP.4 := SUBBIT (OP.1, OP.2, OP.3)

INSERT BITS (INSERT)

Operation Code: 89

Format:



Operation:

SUBBIT(OP.2, OP.3, OP.4) := OP.1

WRITE I-FORMAT (WIF)

Operation Code: 93

Format:

Operation:

Write to a predefined memory location in I-format the value contained in the standard source.

NOTE
Operators @94@ through @A2@ can also be used as privileged operators. They are described under the section entitled Trigonometric and Other Functions.

TRIGONOMETRIC AND OTHER FUNCTIONS

The following is a list of S-operators ordered by operation code for trigonometric and other functions.

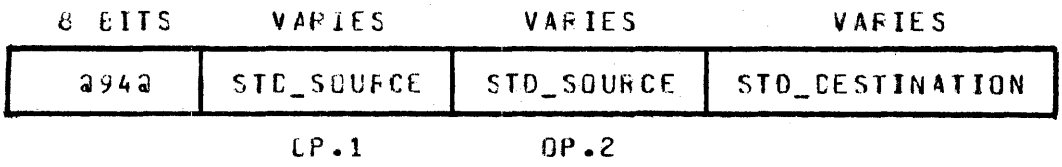
NOTE
All STD__SOURCES and STD__DESTINATIONS are real.

Any of these operators can also be used as privileged operators.

REMAINDER (AMOD)

Operation Code: 94

Format:



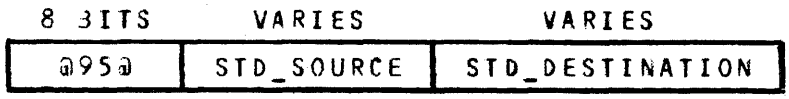
Operation:

STD__DESTINATION := OP1 − (AINT(OP1/OP2) * OP2)

SINE (SIN)

Operation Code: 95

Format:



Operation:

STD__DESTINATION := sine of STD__SOURCE

COSINE (COS)

Operation Code: 96

Format:

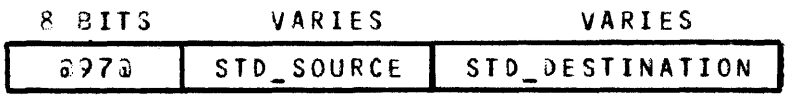
Operation:

STD__DESTINATION := cosine of STD__SOURCE

TANGENT (TAN)

Operation Code: 97

Format:



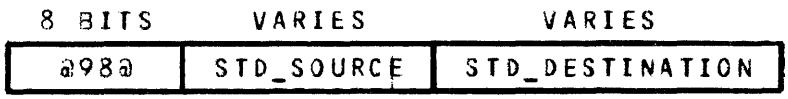
Operation:

STD__DESTINATION := tangent of STD__SOURCE

HYPERBOLIC SINE (SINH)

Operation Code: 98

Format:



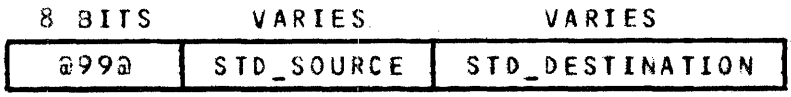
Operation:

STD__DESTINATION := hyperbolic sine of STD__SOURCE

HYPERBOLIC COSINE (COSH)

Operation Code: 99

Format:



Operation:

STD__DESTINATION := hyperbolic cosine of STD__SOURCE

HYPERBOLIC TANGENT (TANH)

Operation Code: 9A

Format:

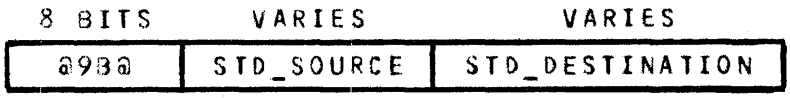
Operation:

STD__DESTINATION := hyperbolic tangent of STD__SOURCE

ARCSINE (ASIN)

Operation Code: 9B

Format:



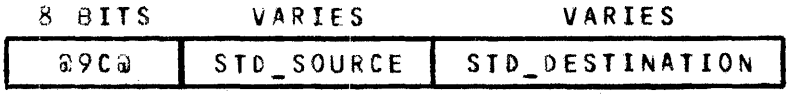
Operation:

STD__DESTINATION := arcsine of STD__SOURCE

ARCCOSINE (ACOS)

Operation Code: 9C

Format:



Operation:

STD__DESTINATION := arccosine of STD__SOURCE

ARCTANGENT (ATAN)

Operation Code: 9D

Format:

Operation:

STD__DESTINATION := arctangent of STD__SOURCE

FLOOR (AINT)

Operation Code: 9E

Format:

Operation:

STD__DESTINATION := The integer whose magnitude is the largest integer which does not exceed the magnitude of STD__SOURCE and whose sign is the same as that of STD__SOURCE.

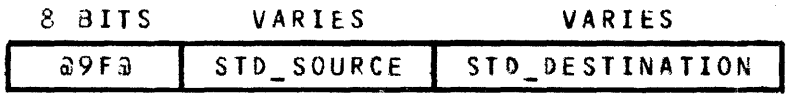
If $-1 <$ STD__SOURCE < 1, then STD__DESTINATION := 0.0

For example, AINT(-3.7) = -3.0

NATURAL LOG (ALOG)

Operation Code: 9F

Format:



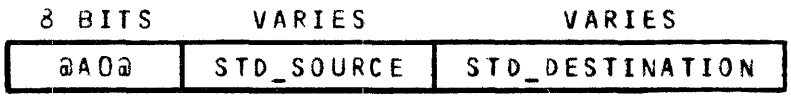
Operation:

STD__DESTINATION := natural log of STD__SOURCE

LOG TO BASE 10 (ALOG10)

Operation Code: A0

Format:



Operation:

STD__DESTINATION := log to base 10 of STD__SOURCE

SQUARE ROOT (SQRT)

Operation Code: A1

Format:

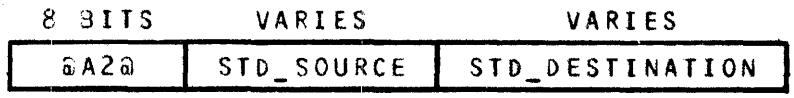
Operation:

STD__DESTINATION := square root of STD__SOURCE

EXPONENTIAL (EXP)

Operation Code: A2

Format:



Operation:

STD__DESTINATION := e ** STD__SOURCE

INDEX

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

Documentation Evaluation Form

Title: B 1000 Systems FORTRAN 77 Reference Manual

Form No: 1108867

Date: March 1983

Burroughs Corporation is interested in receiving your comments and suggestions, regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments:

From:

Name _____

Title _____

Company _____

Address _____

_____

Phone Number _____ Date _____

Remove form and mail to:

Burroughs Corporation
Corporate Documentation – West
1300 John Reed Court
City of Industry, CA 91745
U.S.A.