Burroughs

B 1000 Systems

FORTRAN

REFERENCE MANUAL

(RELATIVE TO MARK 10.0 SYSTEM SOFTWARE RELEASE)

Copyright © 1982 Burroughs Corporation, Detroit, Michigan 48232

PRICED ITEM

Printed in U.S.A.

Cover Revised
December 1982

1081882

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 TIO West Documentation, Burroughs Corporation, 1300 John Reed Court, City of Industry, California 91745, U.S.A.

Burroughs

PUBLICATION CHANGE NOTICE

PCN No.: 1081882-001 Date: December 1982

Publication Title: B 1000 Systems FORTRAN Reference Manual (April 1978)

Other Affected Publications: None

Supersedes: N/A

Description:

This PCN incorporates the MARK 10.0 System Software Release. Revisions to the text are indicated by a vertical black bar in the margin.

Replace these pages	Add these pages
Title	iiA
v thru vii	11-6A thru 11-6C
10-3	K-1 thru K-3
11-1 thru 11-5	
12-13 thru 12-17	
13-3 thru 13-5	
J-1	
Index-3 thru Index-9	

Copyright © 1982, Burroughs Corporation, Detroit, Michigan 48232

LIST OF EFFECTIVE PAGES

Page	Issue	Page	Issue
Title	PCN-001	13-1 thru 13-2	Original
ii	PCN-001	13-3 thru 13-6	PCN-001
iiA thru iiB	PCN-001	14-1 thru 14-2	Original
iii thru iv	Original	A-1 thru A-10	Original
v thru viii	PCN-001	B-1 thru B-3	Original
ix thru x	Original	B-4	Blank
1-1 thru 1-2	Original	C-1 thru C-7	Original
2-1 thru 2-8	Original	C-8	Blank
3-1 thru 3-4	Original	D-1 thru D-10	Original
4-1 thru 4-5	Original	E-1 thru E-10	Original
4-6	Blank	F-1	Original
5-1 thru 5-7	Original	F-2	Blank
5-8	Blank	G-1	Original
6-1 thru 6-4	Original	G-2	Blank
7-1 thru 7-18	Original	H-1	Original
8-1 thru 8-2	Original	H-2	Blank
9-1 thru 9-16	Original	I-1 thru I-2	Original
10-1 thru 10-2	Original	J-1	PCN-001
10-3 thru 10-4	PCN-001	J-2	Blank
11-1 thru 11-6	PCN-001	K-1 thru K-3	PCN-001
11-6A thru 11-6C	PCN-001	K-4	Blank
11-6D	Blank	Index-1 thru Index-2	Original
11-7 thru 11-14	Original	Index-3 thru Index-9	PCN-001
12-1 thru 12-12	Original	Index-10	Blank
12-13 thru 12-18	PCN-001		

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

LIST OF ILLUSTRATIONS

LIST OF TABLES

INTRODUCTION

GUIDE TO EFFICIENT USE

The purpose of this document is to present information directly connected with the FORTRAN[1] programming language as implemented on the B 1800/B 1700 data processing systems. Specifically, this document describes both the equation-oriented programming language accepted by the B 1800/B 1700 FORTRAN compiler and the various features of this compiler.

This manual is designed to provide the FORTRAN programmer with a source of reference information and is not a primer in the language and should not be used as such. The chapters in this manual proceed from basic language elements to general FORTRAN program structures, as follows:

- Sections 1 through 3 discuss characters, literals (i.e., constants and strings), and variables, the primary units of which the language is constructed.
- Sections 4 through 9 discuss FORTRAN expressions and statements, the computation-directed elements of the language.
- Sections 10 and 11 discuss the FILE declaration statement and I/O statements.
- Section 12 describes subprograms, intrinsic functions, and intrinsics.
- Sections 13 and 14 discuss compiler control cards and program structure.

In addition, a number of appendices are provided at the end of this manual which consist of a number of reference aids for the programmer.

The programmer with questions concerning the portion of the B 1800/B 1700 system which processes FORTRAN programs (i.e., the FORTRAN compiler) is directed to the portion of appendix A of this document which discusses the compiler feature in question.

B 1800/B 1700 FORTRAN LANGUAGE

B 1800/B 1700 FORTRAN is designed for compatibility with ANSI standard FORTRAN except for the exceptions and extensions listed in appendix B.

LANGUAGE DESCRIPTION CONVENTIONS

Interspersed throughout the discussion of the FORTRAN language contained in this document are brief indications of the general formats of various of the language constructs. These portions of text follow certain conventions to be explained here.

The method employed in this manual is to present the general format of the construct with the optional portions of the construct represented by lower-case letters and defined following the specification of the format. The remaining characters in the specification are required portions of the construct being defined.

[1] FORTRAN is an acronym for FORmula TRANslation, and was originally developed for International Business Machine equipment.

The proper format for an Example Item is:

EXAMPLE ITEM
EX(s)MP,LE
where s is a string of from one to three characters, each of which is the character A.

This example consists of the presentation of the general format of a fictitious language construct called an "Example Item". The portion of the construct over which the programmer has control is denoted by the lowercase letter s and is defined below the format specification. The balance of the characters in the example format (i.e., all uppercase letters, the parentheses, and the comma) are required portions of the construct. As defined, valid examples of this sample construct would be:

```
EX(A)MP,LE
EX(A A)MP,LE
EX(     AAA)MP,L E        Blank characters are ignored in FORTRAN except in string literals.
```

BASIC FORTRAN CONCEPTS

Certain basic concepts concerning the FORTRAN language are presented here as preliminary to the discussion of the B 1800/B 1700 implementation of this language. These concepts are discussed in detail in the following sections.

A problem-solving system written in the FORTRAN language is called a source program; a program which constitutes a self-contained processing structure is called an executable source program. Every executable FORTRAN 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 program 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, printing the results of computations, etc.), or describe program data items, or provide other program information without directly producing any actions during program execution.

Each program statement is constructed of a string of appropriate characters which is contained on one or more physical records (i.e., punched cards). The set of the physical records containing an executable source program constitutes a source deck. This deck may 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 the source program into FORTRAN S-CODE. The S-CODE generated by the compiler can then be executed on the B 1800/B 1700 using the FORTRAN 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 1700 System Software Operational Guide, form number 1068731.

The B 1800/B 1700 FORTRAN 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.

1. CHARACTER SET

Characters are the elements of which a language is constructed. The B 1800/B 1700 FORTRAN language is based upon a prescribed character set which is described in the present section. Each type of character within this FORTRAN character set is discussed here.

B 1800/B 1700 FORTRAN CHARACTER SET

For source program input, the B 1800/B 1700 FORTRAN character set may be described as consisting of these types of characters:

a. Digits.

   1. Decimal digits.

   2. Hexadecimal digits.

b. Letters.

c. Special characters.

Digits

Two types of digits are employed in B 1800/B 1700 FORTRAN: 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, and when the term "digit" is used in this manual it will refer 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, etc.

These two digit types are used to represent numerical values in the B 1800/B 1700 FORTRAN language.

Letters

For the B 1800/B 1700 FORTRAN language, letters are defined as consisting of these 27 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 $

The dollar sign ($) is included in the letter set (which is referred to as the set of alphabetic characters) since it may be employed anywhere any other letter may be used.

Special Characters

Special characters for the B 1800/B 1700 consist of the following 13 special characters:

blank . , = + - * /) (& ' "

The blank character has no meaning, except in string literals, and can be used throughout the program to improve readability.

Table 1-1 shows the corresponding card codes for each special character.

Table 1-1. Card Codes for the B 1800/B 1700 FORTRAN Characters

Basic FORTRAN Character	Character Name	EBCDIC Card Rows Punched
=	Replacement Operator	8, 6
+	Plus	12, 8, 6
-	Minus	11
*	Asterisk	11, 8, 4
/	Slash	1, 0
(	Left Parenthesis	12, 8, 5
)	Right Parenthesis	11, 8, 5
,	Comma	8, 3, 0
.	Decimal Point (period)	12, 8, 3
'	Apostrophe	8, 5
"	Quote Mark	8, 7
&	Ampersand	12
$	Dollar Sign	11, 8, 3
blank	Blank	none

2. LITERALS

The next level of language complexity for consideration consists of the literal (numeric constants and strings) and identifier FORTRAN constructs. These items are formed from the basic FORTRAN characters according to prescribed rules. Literals are discussed in this section; the various types of identifiers are discussed in appropriate sections. For example, variables are discussed in the next section. Other identifier types include the function name, subroutine name, and COMMON block name, each of which is discussed in the section concerned with the language feature with which that type of identifier is associated.

Literals function as FORTRAN value data-items used in problem solving and related operations such as input/output (I/O). The rules governing their use are discussed here.

Literals may be divided into numeric literals (constants) and string literals (strings) classifications.

NUMERIC LITERALS

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

a. Integer Constants.

b. Real (Floating-Point) Constants.

c. Double Precision Constants.

d. Complex Constants.

e. Hexadecimal Constants.

f. Logical Constants.

These six constant data constructs are discussed in the following paragraphs; their internal storage requirements are discussed in appendix E.

Integer Constants

An integer constant consists of a string of decimal digit characters which may be preceded by a sign character (+ or –). If the constant is non-zero and unsigned, it is interpreted as representing a positive value. A zero has the same value whether signed or unsigned.

One to ten decimal digit characters are permitted and accuracy is ensured providing its value does not exceed ± 8589934591. If this limit is exceeded, a syntax error will be given.

The proper format of an integer constant is:

INTEGER CONSTANT
s n
where s is an optional sign character (+ or –) and n is a string of decimal digit characters.

The following are examples of valid integer constants:

0	999999999
+0	03770
-0	8589934591
17711	-5708

The following are examples of invalid integer constants:

1.0	Decimal point not permitted; interpreted as a real constant.
3,000	No commas or other punctuation permitted.
9999999999	Exceeds the largest integer value allowed.

Real Constants

A real constant consists of a string of decimal digit characters, a decimal point character (.) and an optional sign character (+ or -), or it may consist of a representation written in scientific notation. If the real constant is written in scientific notation, it must consist of a string of decimal digit characters, an optional decimal point character, an optional sign character, and a trailing E followed by a one- or two-digit signed or unsigned integer constant which is the exponent. In all cases, if the real constant is non-zero and unsigned, it is interpreted as representing a positive value. A zero has the same value whether signed or unsigned. Real floating-point constants are stored in internal form in such a way that at least seven, and sometimes eight, digits of significance are retained. Constants longer than eight digits will generate a warning and only the most significant digits will be retained.

The range for a real non-zero constant is approximately 0.863616856E-77 (2**(-256)) $\leq$ m.n $\leq$ 0.57896041E+77 ((2**255)-(2**231)). If this limit is exceeded, a syntax error will be given. (For more information, see appendix E.)

The proper format of a real constant is:

REAL CONSTANT
sm.n or smpnEx
where s is an optional sign character (+ or -), m and n are strings of decimal digit characters (with a combined total not exceeding eight characters, either one (and only one) of which may be omitted. P is an optional decimal point (.) which may be omitted only if n is omitted, and x is a one- or two-digit signed or unsigned integer constant.

In FORTRAN scientific notation, the E portion of the real constant denotes that the value being represented is the number preceding the E multiplied by (ten raised to the power denoted by the integer constant following the E). Thus, the real constant 2E2 represents the value of 200 (i.e., 2 multiplied by 10 raised to the second power).

If scientific notation is used and the decimal point is omitted, the decimal location is assumed to immediately precede the E.

The following are examples of valid real constants:

3.141592		2.5E+07 (same as 25000000)
0.		1.023342E+8
0.0		00000000000007.
200E1 2E3 2.E3 2E+3	(equivalent)	-253. -.075 2.9979E08
.075		
6.02E23		

The following are examples of invalid real constants:

-1597	No decimal point or E portion; interpreted as an integer constant.
6.2E+78	Exceeds maximum size limit.
6.2E-78	Smaller than minimum size limit.
2.5E007	Three-digit integer in E portion.
E22	Exponent part alone not permitted; interpreted as a variable name.
2.7E.1.2	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.

Double Precision Constants

A double precision constant must be written in scientific notation. It must consist of a string of decimal digit characters, an optional decimal point character, an optional sign character, and a trailing D (instead of an E) followed by a one- to two-digit signed or unsigned integer constant which is the exponent. In all cases, if the double precision constant is non-zero and unsigned, it is interpreted as representing a positive value. A zero has the same value whether signed or unsigned.

Double precision floating point constants are stored in internal form in such a way that at least 18, and sometimes 19, decimal digits of significance are retained. Constants longer than 19 digits will generate a warning and only the most significant digits will be retained.

The range for a non-zero double-precision constant is approximately 0.863616855509444462 5D-77 $\leq$m.n$\leq$0.5789604461865809766D+77 or precisely 2**(-256) through (2**255) - (2**195). (For more information, see appendix E.)

The following is the proper format of a double-precision constant:

DOUBLE PRECISION CONSTANT
smpnDx
where s is an optional sign character (+ or -), m and n are strings of decimal digit characters (with a combined total not exceeding 19 characters), either one (and only one) of which may be omitted. P is an optional decimal point character which may be omitted only if n is omitted and x is a one- to two-digit signed or unsigned integer constant.

A double precision constant differs from a real constant only in the number of digits it may contain and the use of a D to indicate the exponent. If the decimal point is omitted, the decimal location is assumed to immediately precede the D.

The following are examples of valid double precision constants:

```
3.1415926535897932D0  }
3.1415926535897932D-0 }   (equivalent)
1D3     }
+1D+03  }   (equivalent)
+1.D+3  }
1234567890.12345678D+29
6.63D-03
9.80665D+0
```

The following are examples of invalid double precision constants:

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.
3.D 0007	Too many digits in exponent part.
1,234,567,890,123.	Commas not permitted; no D portion.
1.3E45	No D in exponent part.
123456789.12345678901	Exceeds maximum character limit; no D portion.

The internal machine representation of a double precision constant requires two consecutive storage locations. Thus, such constants may not be used interchangeably with real or integer constants, but obey rules discussed later in this manual.

Complex Constants

Complex values may be represented in FORTRAN through the use of complex constant literals. A complex constant consists of a real part and an imaginary part (in that order) separated by a comma and enclosed within parentheses. Each of these parts may be either an integer or real constant and must obey the particular rules governing that literal type (such as size limits, etc.). The proper format of a complex constant is:

COMPLEX CONSTANT
(m,n)
where m and n are the real and imaginary parts, respectively, each of which may be an integer or real constant.

The complex constant (m,n) represents the quantity m + nI, where I is the square root of -1.

The following are examples of valid complex constants and their equivalent mathematical expressions:

(123,456)	123 + 456I
(3.14159265,1597	3.14159265 + 1597I
(0,0.05)	0 + 0.05I (or 0.05I)
(100.,100.000)	100 + 100I
(1E2,0)	100 + 0I (or 100)
(0,-1)	0 + (-I) (or -I)

The following are examples of invalid complex constants:

(3,)	Missing imaginary part.
(12345678901234.D0,0)	Neither part may be a double precision constant.
(12.0,72E78)	Imaginary part exceeds size limit for real constants.
(3,4,5)	Excessive number of parts - more than two.

The internal machine representation of a complex constant requires two consecutive words of storage; the real part of the literal is stored in the first of the two words, the imaginary part is stored in the following word. (For more information, see appendix E.)

Hexadecimal Constants

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

A hexadecimal constant consists of the letter Z followed by either 9 or 18 hexadecimal digit characters. The hexadecimal constant assigns a value to the entire word of the variable, including the type bits (see appendix E).

The proper format for a hexadecimal constant is:

HEXADECIMAL CONSTANT
Zn
where n is a string of 9 or 18 hexadecimal characters.

The hexadecimal notation employed on this machine 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 is given here 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

The following are examples of valid hexadecimal constants and their internal types:

Z501800000	real
ZC00000000000000000	double precision
Z0ABCDEFA0	integer
ZFABZFAC4D567000000	double precision

The following are examples of invalid hexadecimal constants:

FFF60	Missing Z and an incorrect number of hex digits are given.
Z-1	Negative hexadecimal constants not permitted and an incorrect number of hex digits are given.
Z0ABCDEFGF	Contains the character G, which is not a hexadecimal digit character.
Z333.330033	Contains a decimal point which is illegal for a hexadecimal constant.
Z123456789ABCDEF012	Contains more than 18 hex characters.

Hexadecimal constants may only be used as data initialization values in a DATA statement to initialize simple variables, array elements, or arrays to specific configurations.

NOTE

Routines in a FORTRAN program that manipulate data in a manner dependent upon the known bit configurations of data types in memory are hardware dependent. Such routines do not use FORTRAN language as a machine-independent, problem-oriented language and must be converted when going from one system to another.

Logical Constants

FORTRAN allows the use of logical operations through the medium of the logical expression. Thus, two logical constant literals are provided to represent the logical values "true" and "false."

These two logical constants have the form:

LOGICAL CONSTANT
.TRUE. or .FALSE.

The use of these logical constants is restricted to certain types of expressions. See the section on logical expressions in section 4. The internal machine representation of these two literals is such that the data words corresponding to the constant .TRUE. and the integer constant 1 are identical and the data words corresponding to the constant .FALSE. and the integer constant 0 are identical.

STRING LITERALS

Constant string data items may be represented in B 1800/B 1700 FORTRAN through the use of the string literal. Any character valid to the B 1800/B 1700 processor may be used. This literal construct has two configurations: the Hollerith string and the proper string (quoted string). These two configurations may be used interchangeably and are distinguishable only by the differences in their formats.

The following is the general format of a Hollerith string:

HOLLERITH STRING
wHs
where w is a positive non-zero constant denoting the width of the string in characters and s is the string in question, being a string of one to 255 characters in length and containing any character valid to the processor.

The following is the general format of a proper string:

PROPER STRING
's' or "s"
where s is a string of one to 255 characters in length and containing any character valid to the processor.

If an apostrophe is desired within a proper string and the form 's' is used, two adjacent apostrophes embedded in the string are interpreted as a single apostrophe (e.g., 'DONT' 'T' represents the string of characters DON'T). Also, an apostrophe can be used as a valid character in a string delimited by quotation marks (e.g., "DON'T" represents the string of characters DON'T).

If a quotation mark is desired within a proper string and the form "s" is used, two adjacent quotation marks embedded in the string are interpreted as a single quotation mark (e.g., "DON""T" represents the string of characters DON"T). Also, a quotation mark can be used as a valid character in a string delimited by apostrophes (e.g., 'DON"T' represents the string of characters DON"T).

In all cases, any blank characters within the character string will be considered part of the string (e.g., embedded blanks are not ignored in this case). The following are examples of valid strings (b represents a blank character):

```
2HbQ
001Hb  }
"b"    }      (equivalent)
'bSIDUR1bb5'
'bb' 'ABCDEFGHJKL'   }
0014Hbb'ABCDEFGHJKL' }      (equivalent)
'b'
""b" IS A BLANK'
""b' IS A BLANK"
```

The following are examples of invalid strings:

3HbbbAB	Character string may not be longer than integer preceding the H indicates.
-HABCDEF	String width specification may not be a negative integer.
0HA	String width specification may not be zero.
'POIU'b'YT'	This construct will be interpreted as two separate strings, rather than as one string containing an apostrophe, since the two inner apostrophes are not immediately adjacent.
"ABC"EFGH"	Two adjacent quotes are needed if quotation marks are used as delimiters.

String literals are used to initialize variables in DATA statements, and as input/output (I/O) specifications in a FORMAT statement. String literals may be passed as arguments in subroutines and function subprograms. In no case may a string literal exceed 255 characters in size, and only four characters of a string literal can be contained in a data word. These restrictions are due to the fact that in the B 1800/B 1700 a data word contains 36 bits and each character requires 8 bits.

3. VARIABLES

FORTRAN variables, like literals, are symbolic names which are constructed from the FORTRAN character set according to appropriate rules and which represent unique values. Variables, however, represent values which may be altered during program execution.

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

This section contains a discussion of variable name construction which extends to array names and array elements.

The internal handling of variables is discussed in appendix E.

VARIABLE NAMES

A FORTRAN variable name is an identifier which consists of a string of one to six alphanumeric characters (i.e., letter or digit characters), with the leading character being a letter (including $). Special characters may not be used in variable names.

If the variable name is more than six characters long, a syntax error will be given. Embedded blanks are acceptable and removed by the system.

Variables may be classified into five fundamental types:

a. Integer.

b. Real.

c. Double Precision.

d. Complex.

e. Logical.

There is no variable of data type hexadecimal. Hexadecimal constants may only be used as data initialization values in DATA statements. The value represented by a variable of each of these types may be expressed by a constant literal of the same type. Thus, the value represented by an integer variable may be expressed by an integer constant, the value represented by a real variable may be expressed by a real constant, and so forth. The values represented by each variable type must, therefore, obey the magnitude and significant-digit restrictions governing the corresponding type of constant literal.

Unless declared otherwise in an explicit type statement or an IMPLICIT statement the identifier will be assigned a type according to its initial character. If this initial character is the letter I, J, K, L, M, or N, then the variable, by default, will be of INTEGER type. If this initial character is any other letter (including $), then the variable, by default, will be of REAL type. No such defaults exist for double precision or logical variables; variables of these types must be declared as such by explicit type statements.

VARIABLES

The following are examples of valid variable names with the type assigned to them when they are not affected by explicit type statements in the program in which the identifiers appear:

AMK599	Type REAL.
IF	Type INTEGER; note that this variable name is valid, since there are no reserved words in B 1800/B 1700 FORTRAN.
$64000	Type REAL.
END$Q	Type REAL.
OF TEN	Type REAL; interpreted as OFTEN (blank ignored).
LOOP3	Type INTEGER.

The following are examples of invalid variable names:

3LOOP	Variable name may not begin with a digit character.
BE-GIN	Characters other than letters, digits, or blanks are not allowed in a variable name.
$6,000	Comma character illegally embedded in variable name.
REALNUMBER	Too many characters: only six are permitted.

ARRAY ELEMENTS

FORTRAN variables may be divided into simple variables, which are denoted by a variable name only, and array elements, which are denoted by an array name followed by a subscript list enclosed in parentheses. The latter variable form indicates membership in a data grouping called an array; this form is discussed here.

An array is an ordered data set corresponding to an n-dimensional organization such that each member may be referenced by an array element, with each of the n subscripts in the element denoting location in the appropriate dimension. In B 1800/B 1700 FORTRAN a data array is limited to a maximum of 15 such dimensions. A maximum of 131,071 elements in an array is permitted.

An array is referenced by a single identifier (the array name) which is defined identically to a variable name. This identifier represents the data items contained in the array, all of which are identical in type. This type is indicated by the array name following the same rules for type assignment governing variable names.

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

ARRAY ELEMENT
a (s)
where a is a variable name, formed in accordance with rules governing such constructs, and s is the subscript list which consists of as many arithmetic expressions (subscripts) as there are array dimensions separated by commas.

A variable name is designated as an array name by means of an appropriate array declaration in a DIMENSION, explicit type, or COMMON statement.

This declaration is used to declare the maximum number and size of dimensions allowable for the array and must precede the first appearance of the array name in either an executable statement or DATA statement. In a given program unit an identifier may be used as a simple variable name or an array name, but not both. 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:

a. The dummy argument list of a subprogram.

b. The actual argument list of a subprogram reference.

c. The variable list of an I/O statement.

d. A COMMON, DATA, EQUIVALENCE, or explicit type statement.

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 INTEGER type only. The expression may contain any of the arithmetic operators, integer function references, or subscripted integer variables.

The minimum value any subscript may represent is 1, and the maximum value is the maximum value specified for that subscript in the array declaration. For information concerning exceptions to this rule, see appendix E.

An array element can never contain fewer subscripts than the array declaration except in an EQUIVALENCE statement.

The following are examples of valid array elements:

B(I)	
AMK 599(6)	Interpreted as AMK599(6).
I5(IT(3))	The subscript is itself an array element.
ARRAY2 (1,1,1,1)	
A(M*N)	

The following are examples of invalid array elements:

I(I)	A subscript must be a valid arithmetic expression; an array name does not constitute such an expression.
ARRAY3(0)	The minimum subscript value is 1.
AR RAY3(-1)	The value of the subscript of a singly-subscripted variable may not be 0 or less.
3ARRAYS(6)	An array name may not violate the rules governing variable names.
ARRAY(3.6)	Subscript must be INTEGER type only.

A detailed discussion of the internal handling of FORTRAN arrays is contained in appendix E.

4. EXPRESSIONS AND STATEMENTS

This section discusses the manner in which expressions are constructed and the general features of the statements that form the basis of the FORTRAN language.

EXPRESSIONS

The purpose of expressions is to specify equation-oriented rules whereby a unique data value may be obtained, possibly 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 may be divided into two basic types according to the type of literal which may represent the value of the expression: arithmetic expressions and logical expressions.

Operators

The operators which may be employed by a FORTRAN expression are listed in table 4-1, with the relative precedence assigned to each operator by the compiler. Eight is considered the highest precedence.

The presence of these operators in an expression indicates that an arithmetic or logical operation or a logical 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 may be used to override operator precedence.

Arithmetic Expressions

An arithmetic expression is a rule for computing a value representable by any type of numeric literal except a logical constant.

An arithmetic expression may contain only arithmetic operators and non-logical constants (excluding hexadecimal constants), variables, function references, and grouping parentheses. Logical operands of any sort are not permissible in arithmetic expressions. In general, mixed arithmetic operand types are permissible.

Immediately adjacent operators are not permissible and parentheses should be used to avoid adjacent operators. For example: A**(−2).

Expression Types

The types of the 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 is complex and at least one is double precision, the result is of DOUBLE PRECISION type. If none of the operands in an arithmetic operation is COMPLEX or DOUBLE PRECISION and at least one of them is real, the result is of REAL type. Only if both of the operands in an arithmetic operation are integer is the result of INTEGER type.

Table 4-1. Operators Used in FORTRAN 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
.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
.NOT.	Logical	3	Logical negation
.AND.	Logical	2	Logical conjunction
.OR.	Logical	1	Logical disjunction

Tables 4-2 and 4-3 illustrate the resultant types of arithmetic operations depending upon the types of the 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 4-2. 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 4-3. Resultant Types for Exponentiation

Type of Base	Type of Exponent			
	Integer	Real	Double	Complex
INTEGER	INTEGER	REAL	DOUBLE	ILLEGAL
REAL	REAL	REAL	DOUBLE	ILLEGAL
DOUBLE	DOUBLE	DOUBLE	DOUBLE	ILLEGAL
COMPLEX	COMPLEX	COMPLEX	COMPLEX	ILLEGAL

In the case of a division 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.

The following are examples of valid arithmetic expressions (all variables are non-logical):

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

Logical Expressions

Logical negation is expressed by the operator .NOT.. Given the value of a logical primary, the operator .NOT. will change the value to its complement. For example, if A is .TRUE., then .NOT.A is .FALSE..

The .AND. operator is used to express the logical product of two logical expressions. Given A.AND.B, the operation will yield the value .TRUE. if and only if both A and B are .TRUE.. If either or both A and B are .FALSE., then A.AND.B is .FALSE..

The .OR. operator is used to find the logical sum of any two logical expressions. Given A.OR.B, the operation will yield the value .TRUE. if either A or B or both are .TRUE.. Only if both A and B are .FALSE. will A.OR.B be .FALSE..

Table 4-4. Logical Expression Constructs

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

Except when appearing adjacent to a relational operator (i.e., in a relational expression), elements of arithmetic expressions (i.e., non-logical constants, variables, function references, and operators) may not appear in a logical expression.

Parentheses may be used to override operator precedence.

Examples:

The following are examples of valid logical expressions (variables A, B, C, D and array L are LOGICAL):

```
A
B2.GT.B3
A.OR.L(3)
A.AND.Q.LE.SIN(R)+3.7
C.AND..NOT.D
I+1.EQ.  FCN(1, 3, R, A)
```

The following are examples of invalid logical expressions:

A+B	Illegal operator.
SIN(Q)	The SIN built-in function returns a real value.
I+1.OR.R+6	An arithmetic expression is not a logical primary.

STATEMENTS

Every executable FORTRAN program consists of a sequence of statements. These statements may be classified into executable and nonexecutable statements.

Executable Statements

An executable statement is an instruction that causes action to be taken at that point in the program when the program is executed.

The B 1800/B 1700 executable statements that are discussed in this document are as follows:

Assignment statement (including ASSIGN statement).
BACKSPACE statement.
CALL statement.
CALL CHANGE.
CALL EXIT.
CALL ZIP.
CLOSE statement.
CONTINUE statement.
DO statement.
ENDFILE statement.
GO TO statement.
IF statement.
LOCK statement.
PAUSE statement.
PURGE statement.
READ statement.
RETURN statement.
REWIND statement.
STOP statement.
WRITE statement.
ZIP statement.

Non-Executable Statements

A non-executable statement is an instruction which gives information to the compiler regarding storage allocation, data initialization, I/O editing specifications, and program units.

The B 1800/B 1700 FORTRAN nonexecutable statements that are discussed in this document are as indicated:

BLOCK DATA statement.
COMMON statement.
DATA statement.
DIMENSION statement.
END statement;
EQUIVALENCE statement.
Explicit type statement.
EXTERNAL statement.
FORMAT statement.
FUNCTION statement.
IMPLICIT statement.
INTRINSIC statement.
Statement function definition.
SUBROUTINE statement.

Statement Labels

A statement may be optionally labeled so that it may be referred to in a GO TO statement, an IF statement, a READ or WRITE statement, a DO statement, or an actual argument list.

A statement label consists of a one- to five-digit unsigned integer constant. No statement label may appear on more than one statement, in the same program unit. A statement label may appear anywhere in the first through fifth character positions of the initial card image of a statement.

Statement labels on any nonexecutable statement except a FORMAT statement are flagged by a warning message by the compiler and are ignored.

5. SPECIFICATION STATEMENTS

The nonexecutable 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 available specification statements include the following:

Explicit type statements.
COMMON statement.
DIMENSION statement.
EQUIVALENCE statement.
EXTERNAL statement.
IMPLICIT statement.
INTRINSIC statement.

These statements are discussed 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. Such type specifications override any default specifications (i.e., due to the initial character in the symbolic name of the variable).

Program variables may be assigned the following types:

DOUBLE PRECISION or REAL*8
INTEGER
LOGICAL
REAL

The proper form of a type statement is as follows:

EXPLICIT TYPE STATEMENT
m. . .
where t is one of the type names just listed, m is a variable, array name, dummy argument name, array declaration, or FUNCTION name, and the ellipsis (. . .) indicates as many repetitions of m as desired, with each repetition preceded by a comma.

A variable is assigned a default type according to the initial character of its name. If the first letter of the variable name is I, J, K, L, M, or N, the variable will be of type INTEGER by default. All other variables will be of the type REAL by default.

Type statements must precede the use of affected variables in executable statements.

Array Declarations

A type statement may also be used to declare the size and number of dimensions of a program array by means of an array declaration in the list of data items. An array declaration has the form:

ARRAY DECLARATION
a (i)
where a is an array name and i is a list of dimensions which are unsigned integer constants and/or dummy variables separated by commas.

The array declaration must have integer constant dimensions if the declaration appears in a specification statement in a main program.

The array declaration may contain dimensions which are dummy arguments or elements of common storage, in addition to constant dimensions if the declaration appears in a specification statement in a subprogram. Such dimensions are called adjustable dimensions, since the size of the dummy array may be specified at the time the subroutine is invoked. The discussion of the CALL statement in section 9 explains array handling in subprograms.

The number of dimensions in an array declaration specifies the number of subscripts an element of that array must have. The product of the dimensions of the array declaration in a main program determines the number of internal storage locations assigned to the array. The number of these locations is equal to the product for REAL, INTEGER, and LOGICAL arrays. Twice this number of locations are assigned to DOUBLE PRECISION (or REAL*8) arrays. (See appendix E.)

Array declarations may also be specified in COMMON or DIMENSION statements.

Optional Size Specification

The optional length specifier may be used immediately following the type name and/or any element within the type statement. When the length specifier is used immediately following the type name, the list of elements specified by the type name will be of that length unless an individual element is specified otherwise by its own length specifier. The length specifier for an individual element will override the length specifier for the type name.

If the value of the length specifier is 8 and the type t is REAL, type DOUBLE PRECISION is assigned to the elements indicated in the type statement. All other length specifiers cause syntax errors.

An example of an explicit type statement using the optional length specifier is: REAL*8 SBC, TBC.

Examples of Type Statements:

```
REAL I, J, JTEST, D
LOGICAL L1, L2
DOUBLE PRECISION D
INTEGER A(20, 20, 4)
REAL*8 DOUBLE, DFCN1, $FCN
```

COMMON STATEMENT

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

The proper form of the COMMON statement is:

COMMON STATEMENT
COMMON /n/v . . .
where n is a COMMON name or blank, v is a list of variables, array names, and/or array declarations, and the ellipsis (. . .) indicates as many repetitions of the form /n/v as desired.

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 may access the block of storage associated with this name via a COMMON statement employing this name. COMMON storage associated with a COMMON name is called labeled 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 ".BLNK.". If the specification for blank COMMON is the first specification in a COMMON statement, the two slashes enclosing the COMMON name may 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 may be reused as another element within the program unit (i.e., simple variable name, array name, etc.).

Use of Array Declarations

Array declarations may be used in COMMON statements to declare the dimensionality of arrays in the same manner as type statements or DIMENSION statements. Array declarations are discussed in the section on the explicit type statement in this section.

Storage Assignments

Variables and arrays are assigned contiguous locations in COMMON storage in the order in which they appear in a COMMON statement. No dummy arguments may appear in a COMMON statement.

Entire program arrays but not individual array elements may be assigned storage locations in COMMON storage. A maximum of 30 unique COMMON block names may be defined in a program unit. If the same COMMON name appears more than once in a program unit, the common elements associated with one appearance are considered as extensions to the list of the previous appearance.

A discussion of the manner in which locations are allocated to elements of COMMON storage is presented in appendix E. Data initialization may be performed by means of a BLOCK DATA subprogram unit. The BLOCK DATA program unit is discussed in section 12. 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 may not be duplicated in COMMON statements. One variable may not be assigned to more than one block of COMMON storage within a program unit.

Examples of COMMON Statements:

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

DIMENSION STATEMENT

The nonexecutable DIMENSION statement specifies the size and number of dimensions of a program array.

The following is the proper form of the DIMENSION statement:

DIMENSION STATEMENT
DIMENSION d
where d is a list of array declarations separated by commas.

Each array referenced in a program unit must have its array bounds specified exactly once in that program unit. This specification may 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. See the explicit type statement section in this section.

Only an array declaration appearing in a subprogram may have dimensions which are variables, and such variables must be dummy arguments or elements of common storage. Array storage is discussed in appendix E.

Examples of DIMENSION Statements:

```
DIMENSION A(10)
DIMENSION B(N,2), C(6)
DIMENSION Q(J)
```

EQUIVALENCE STATEMENT

The nonexecutable EQUIVALENCE statement causes two or more variables or arrays referenced in a program unit to share the same memory locations.

The following is the proper form of the EQUIVALENCE statement:

EQUIVALENCE STATEMENT
EQUIVALENCE (k) . . .
where k is a list of two or more simple variables, array elements, or array names separated by commas, and the ellipsis (. . .) indicates as many repetitions of the form (k) as desired, with each repetition preceded by a comma.

No dummy argument may appear in an EQUIVALENCE statement list.

The subscripts of array elements in the list must be integer constants and must correspond in number to the number of dimensions declared for the array or be single-subscripted where the subscripted variable's linear position in the array is equated to the single subscript. The EQUIVALENCE statement is the only statement in the FORTRAN syntax where an element in a multi-dimensioned array may be referred to by means of a single subscript. Thus, the following two sets of statements are equivalent:

```
DIMENSION X(5), R(5,5)
EQUIVALENCE (X(1), R(1,5))
```

```
DIMENSION X(5),R(5,5)
EQUIVALENCE (X(1),R(21))
```

Since arrays are stored in column-wise order, element R(1,5) is the 21st element in the two-dimensioned array named R.

Each data item grouping in the EQUIVALENCE statement is enclosed in parentheses. Each such grouping is assigned storage locations to share. When arrays are involved, the indicated array element denotes the EQUIVALENCEd arrays are to overlap and be aligned in such a manner that the indicated elements share a storage location(s). No group may contain different elements of the same array. Thus, EQUIVALENCE (A(3),B,A(6)) is invalid.

The EQUIVALENCE statement may be used to associate additional elements with a COMMON block. This may extend the block beyond its former terminal point, thus increasing the size of the COMMON block. An EQUIVALENCE statement may not associate an array with a COMMON block in such a manner as to expand that block backwards to locations preceding the initial location of the block.

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

Appropriate storage considerations are discussed in appendix E.

Examples of EQUIVALENCE Statements:

```
EQUIVALENCE (A, B, C), (A(3),R(9),G)
EQUIVALENCE (D,E)
```

EXTERNAL STATEMENT

The non-executable EXTERNAL statement is used to identify a subprogram name as representing an external procedure and to permit the subprogram name to be used as an actual parameter. If a subprogram name is used as an actual parameter, then it must appear in an EXTERNAL statement.

The subprograms specified in the EXTERNAL statement are searched for first in the user intermediate code files, and if not found there then the FOR.INTRIN intrinsic file is searched.

The proper form of the EXTERNAL statement is:

EXTERNAL STATEMENT
EXTERNAL s
where s is one or more subprogram names separated by commas.

Any FORTRAN intrinsic function that cannot be redefined (see section 12) specified in an EXTERNAL statement is expanded in the code file. Such names are assumed to denote subprograms supplied by the user.

IMPLICIT STATEMENT

The nonexecutable IMPLICIT statement is an auxiliary statement which allows the default types assigned to variables due to their initial characters to be altered.

The following is the proper form of the IMPLICIT statement:

IMPLICIT STATEMENT
IMPLICIT t(c) . . .
where t is a type name, c is an initial character list, and the ellipsis (. . .) indicates as many repetitions of the form t(c) as desired, with a comma preceding each repetition.

Only one IMPLICIT statement is allowed in a program unit, and any program unit may contain an IMPLICIT statement. If used, the IMPLICIT statement must be the first statement following optional file declaration statements of the main program or the second statement of a sub-program (except for comment cards). The IMPLICIT statement applies only to symbolic names in the program unit in which the statement appears, including function and dummy arguments.

The list of initial characters appearing in an IMPLICIT statement is constructed as follows:

> 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 may be a letter or two letters separated by a hyphen (i.e., the minus sign, "-"). 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, then this letter pair

indicates a range of initial characters with which the default type is associated. The second of the letters in a hyphenated pair must follow the first in the alphabet. The "$" follows Z in the collating sequence.

The following are valid implicit statements:

```
IMPLICIT REAL (I-N)
IMPLICIT REAL (A-$)
IMPLICIT DOUBLE PRECISION (D)
IMPLICIT LOGICAL (A-C,L), REAL*8(D-F), COMPLEX (X)
```

IMPLICIT ranges that overlap (e.g., REAL (A-K) INTEGER (I-M) generate a warning message. The latest specification is used to determine the variable type.

INTRINSIC STATEMENT

The non-executable INTRINSIC statement specifies a symbolic name representing an intrinsic function from the intrinsic file is to be used as an actual argument. If an intrinsic name is used as an actual parameter, then it must appear in an INTRINSIC statement.

The intrinsic functions specified in an INTRINSIC statement are searched for first in the intrinsic file, and if not found there then the user intermediate code files are searched.

The proper form of the INTRINSIC statement is:

INTRINSIC s
where s is one or more intrinsic names separated by commas.

Any FORTRAN intrinsic function that cannot be redefined (see section 12) and is specified in an INTRINSIC statement is not expanded in the code file. Such intrinsic names are assumed to denote subprograms supplied by the user.

6. DATA STATEMENTS

The non-executable DATA statement is provided to allow compile-time initialization of program variables.

All variables are initialized unless the $NO INITIAL compiler control card is used. A run-time error will occur if an attempt is made to access the value contained in a variable not previously given a value by a DATA statement, assignment statement, or input statement.

The proper form of the DATA statement is:

DATA STATEMENT
DATA k/d/ . . .
where k is a list of variables to be initialized, d is an initial value list (see text), and the ellipsis (. . .) indicates as many repetitions of the form k/d/ as desired, with each repetition preceded by a comma.

STATEMENT USE

A DATA statement may appear after all specification statements and before the END statement in a program unit. The DATA statement has effect only at compilation time. Elements of a COMMON block may appear in DATA statements only in a BLOCK DATA subprogram.

Variable Lists

A variable list may contain variables, arrays, or array elements. Each element of the variable list should occur only once.

The following is an example of a variable list:

```
K,M, A(3), B(2,4,11)
```

Initial Value Lists

The list of values to which the elements of the variable list are to be initialized consists of a list of constants and strings separated by commas.

The constants may optionally be preceded by a repeat count of the form n*, where n is an unsigned non-zero integer constant. This repeat count indicates the number of times the immediately following constant is to be interpreted.

Constant values are assigned to elements of the variable list in the order in which they occur. For example, the DATA statement:

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

initializes the variables A and B to the values 2 and 3, respectively, and initializes C and D to 4.

The initial value list may contain Hollerith or proper strings of up to 255 characters long. (A syntax error message is given during compilation if a longer string is encountered.) A "primary" string is defined to be a string consisting of four or fewer characters that initializes a single-precision variable, or a string consisting of eight or fewer characters that initializes a double precision variable. Similarly, a "long" string is defined to be a string consisting of more than four characters that initializes a single precision variable, or a string of more than eight characters that initializes a double-precision variable.

A primary string is stored left-justified within the variable, with blank fill to the right if necessary. For example, if D is a DOUBLE PRECISION variable, the DATA statement:

```
DATA D/"bABCD"/
```

(where b represents a blank) initializes D to the value:

```
bABCDbbb
```

Any long string initializing a simple variable or array element is truncated to a primary string, with the leftmost characters of the string retained. A repeat count is not allowed to precede a string value.

A long string initializing an array name is transferred to the array, beginning with the indicated element, until either the end of the array or the end of the string is encountered. Character assignments are made to the array elements in the order in which these elements are stored internally. (See appendix E.) If the string terminates before the entire array is initialized, the last initialized array element will be filled on the right with blanks, if necessary.

For example, this program excerpt:

```
REAL A(6)
DOUBLE PRECISION D(3)
DATA D,A/'ABCDEFGHIJKLMNOPQRSTUVWXYZ' , 1,2,3,7HABCDEFG/
```

will cause these variables to be initialized to the indicated values:

```
D(1) contains 'ABCDEFGH'
D(2) contains 'IJKLMNOP'
D(3) contains 'QRSTUVWX'
A(1) contains 1
A(2) contains 2
A(3) contains 3
A(4) contains 'ABCD'
A(5) contains 'EFGb'
```

Variable List of IMPLIED DO in Data Statement

The list of variables to be initialized is constructed in the same manner as the IMPLIED DO of an I/O variable list (see section 11) except that:

a. No arithmetic expressions may appear in the variable list.

b. The initial parameter, terminal parameter, and incrementation parameter in an IMPLIED DO must be unsigned integer constants.

c. An array element may be subscripted only by an unsigned integer literal or the integer control variable of an IMPLIED DO.

d. Only array elements may appear within an IMPLIED DO list, and all control variables must be referenced.

Initial Value List of IMPLIED DO in Data Statement

The list of values to which the elements of the variable list are to be initialized consists of a list of constants and strings separated by commas.

The constants optionally may be preceded by a repeat count of the form n*, where n is an unsigned non-zero integer constant. This repeat count indicates the number of times the immediately following constant is to be repeated. A repeat count is not allowed to precede a string variable.

All elements of the variable list must be matched to the constants, and all constants must be used. A warning is issued and the remaining array is blank or zero filled if an entire array is specified in the variable list but there are not enough constants to completely initialize it. Long strings are truncated when assigned to array elements using an implied DO in the DATA statement. The initialization value for each element of an array must be specified separately when using an implied DO in a DATA statement.

Example of a DATA statement with an IMPLIED DO:

```
Data K,M,(A(I,2),B(I),I=1,2)/1,2,1,1,2*2/
```

At compile time:

K	is assigned 1.
M	is assigned 2.
A(1,2)	is assigned 1.0.
B(1)	is assigned 1.0.
A(2,2)	is assigned 2.0.
B(2)	is assigned 2.0.

HEXADECIMAL CONSTANTS

Hexadecimal constants may be used to initialize variables. The hexadecimal constant must be exactly nine hexadecimal digits, if initializing an integer, logical, or real variable, and 18 hexadecimal digits if the variable is double precision. (See appendix E.)

Conversion During Assignment

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

The following notation is used in this table:

ok	no conversion
real	round the first word of the double precision value
ext	extend the single precision value to a double precision value
i	invalid combination resulting in syntax error
conv. real	convert to real
conv. DP	convert to double precision

Table 6-1. DATA Statement Type Conversions

Constant Type	Variable Type			
	Integer	**Real**	**Double Precision**	**Logical**
Integer	ok	conv. real	conv. DP	i
Real	i	ok	ext	i
Double Precision	i	real	ok	i
Logical	i	i	i	ok
String	ok	ok	ok	ok
Single Precision Hexadecimal	ok	ok	i	ok
Double Precision Hexadecimal	i	i	ok	i

7. FORMAT SPECIFICATIONS AND FORMAT SPECIFIERS

This section discusses the format specifications provided to indicate data conversion during input and output and the method of associating these specifications with input and output statements using the FORMAT statement.

FORMAT SPECIFICATIONS

The following format specifications are provided to allow conversion of data between the representation of such data as characters in peripheral storage and the configuration of this data in internal storage.

Aw
Dw.d
Ew.d
Fw.d
Gw.d
wHs, 's', "s"
Iw
Lw
Tt
X or wX
Zw

In this list of specifications and in the following paragraphs, the items w, d, and t represent unsigned nonzero integer literals denoting the width of the field in the external character string, the number of digits in the fractional part of the external string, and a character position relative to the beginning of an external record, respectively. The item s represents a string of one to 255 characters.

Tables 7-1 and 7-2 summarize what type of data items may be read into what type of variable, using each of the format specifiers.

Table 7-3 summarizes the type of list item(s), i.e., (arithmetic expressions, variables) which may be written under each format specifier.

An A indicates that the conversion is allowed; NA indicates the conversion is not allowed.

Table 7-1. Input Data Item Types

Data Item Type	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
Alpha	NA	NA	NA	NA	NA	A	NA	NA
Hex	NA	NA	NA	NA	NA	A	NA	A

Table 7-2. Input Variable Item Types

Variable Type	I	F	E	G	D	A	L	Z
Integer	A	A	A	A	A	A	NA	A
Real	A	A	A	A	A	A	NA	A
Double	A	A	A	A	A	A	NA	A
Logical	NA	NA	NA	A	NA	A	A	A

Table 7-3. Output List Item Types

List Item Type	I	F	E	G	D	A	L	Z
Integer	A	A	A	A	A	A	NA	A
Real	A	A	A	A	A	A	NA	A
Double	A	A	A	A	A	A	NA	A
Logical	NA	NA	NA	A	NA	A	A	A

On input, the value of the data item may be changed if read into a variable of a type other than the type of the data item. On output, the value of the data item contained in the variable may be changed if the format specification does not correspond with the type of the variable being written.

On input, if an exponent is used, it may be of 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.

Format Specification A

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 integer, real, double-precision or logical variable in the input list.

Format specification A is used to assign alphanumeric characters to a variable or array of any data type. The alphanumeric characters to a variable or array of any data type. The alphanumeric input field is stored in memory in character form and thus should not be used in any computation.

The number of characters that may be stored in a variable depends upon the variable type. A maximum of four alphanumeric characters may be stored in an integer, real, or logical variable, and a maximum of eight characters may be stored in a double precision variable.

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
INTEGER	ABCDEFGH	A8	EFGH
REAL	ABCDEFG	A7	DEFG
DOUBLE PRECISION	ABCDEFGH	A8	ABCDEFGH
REAL	AB	A2	ABbb
INTEGER	54C2X	A5	4C2X

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
INTEGER	ABCD	A6	bbABCD
REAL	ABCD	A8	bbbbABCD
INTEGER	ABCD	A3	ABC
REAL	ABCD	A2	AB
DOUBLE PRECISION	ABCDEFGH	A12	bbbbABCDEFGH
LOGICAL	ABCD	A4	ABCD

Format Specification D

Input Using Dw.d

On input, the double precision format specification Dw.d causes the value of the data item in the input field to be assigned to the corresponding integer, real, or double precision variable in the input list.

The double precision format specification Dw.d functions in the same manner as Ew.d, Fw.d, or Gw.d.

Output Using Dw.d

On output, the double precision format specification Dw.d causes the corresponding integer, real, or double precision output list item to be written with an exponent on the specified output file.

The double precision format specification Dw.d is identical to Ew.d, with the following exceptions:

a. The Dw.d format specifier permits up to 18 significant digits to be output.

b. The exponent part of the output contains a D rather than an E.

Format Specification E

Input Using Ew.d

On input, the real format specification Ew.d causes the value of the data item in the input field to be assigned to the corresponding integer, real, or double-precision variable in the input list.

The data item must be in the form of an integer, real, or double-precision constant and is considered to be right-justified within the input data field. Leading, embedded, and trailing blanks within the field are read as zeros.

The magnitude of the value in the input field must not exceed the maximum magnitude for the corresponding variable.

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 Ew.d, Fw.d, Gw.d and Dw.d function in the same manner.

Examples:

Variable Type	Data Item	Specification	Internal Value
REAL	bbbbbb25046	E11.4	+2.5046
INTEGER	bbbbb25.046	E11.4	+25
REAL	-bb25046E-2	E11.4	-0.025046
DOUBLE PRECISION	b250.46D-10	E11.0	+0.000000025046
REAL	b-b25.04678	E11.4	-25.04678

Output Using Ew.d

On output, the real format specification Ew.d causes the corresponding integer, real, or double precision output list item to be written with an exponent on the specified output file.

The real number is placed right-justified and rounded to d digits, together with a four-place exponent field, in the output field over a field of blanks. Note that with the Ew.d format specification, d takes on a slightly different interpretation, since no significant digits are written to the left of the decimal point in the output field. The position for the minus sign required for negative numbers is included in the field width w. For positive numbers, W = d + 5 + (the number of leading blanks desired). For negative numbers, w = d + 6 + (the number of leading blanks desired).

Examples:

List Item Type	List Item Value	Specification	Output Field
REAL	+36.7929	E12.5	b0.36793E+02
REAL	-36.7929	E11.5	-.36793E+02
REAL	-36.7929	E10.5	**********
INTEGER	2589	E9.3	0.259E+03
DOUBLE PRECISION	872568.394816897	E12.7	.8725684E+06

Format Specification F

Input Using Fw.d

On input, the real format specification Fw.d causes the value of the data item in the input field to be assigned to the corresponding integer, real, or double precision variable in the input list.

The data item must be in the form of an integer, real, or double-precision constant and is considered to be right-justified within the input data field. Leading, embedded, and trailing blanks within the field are read as zeros.

The magnitude of the value in the input field must not exceed the maximum magnitude permitted for the corresponding variable.

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, and Dw.d 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
INTEGER	-367259.	F8.4	-367259
REAL	-3672.E2	F8.4	-367200.
REAL	-3672+02	F8.4	-367200.
DOUBLE PRECISION	367259D-10	F10.4	+.00000000367259
INTEGER	3.6272E1	F8.4	+36

Output Using Fw.d

On output, the real format specification Fw.d causes the corresponding integer, real, or double precision 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 over a field of blanks.

The plus sign is omitted for positive numbers. 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 minus sign required for negative numbers 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 asterisks (*).

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	-0.032
INTEGER	2567	F7.2	2567.00
REAL	0.0	F6.4	0.0000
DOUBLE PRECISION	37624.816952	F8.3	********
REAL	+579.645	F6.2	579.65
REAL	-579.645	F6.2	******
REAL	-0.895	F5.2	-0.90

Format Specification G

Input Using Gw.d

The G-format descriptor is a multi-purpose format descriptor which may be used with input variables of type integer, real, double precision, or logical. It is interpreted as an I, F, E, D, or L format descriptor, depending upon the type of the variable in the input list.

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 not required and if specified, is ignored.

If the input variable is real or double precision, the .d portion of the general form must be included and the Gw.d format specification functions in the same manner as Fw.d, Ew.d, and Dw.d.

Examples:

Variable Type	Data Item	Specification	Internal Value
REAL	b529.4	G6.1	+529.4
LOGICAL	bbT	G3	.TRUE.
INTEGER	bbb45	G5	+45
REAL	b-6.1E+04	G9.1	-61000.
DOUBLE PRECISION	bb5.3294D+02	G12.4	+532.94

Output Using Gw.d

On output, the general format specification Gw.d causes 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 may be either integer, real, double precision, or logical.

The G format specification is a multi-purpose format specification which may be used with output list items of type integer, real, double precision, or logical. It 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 not required and if specified, is ignored.

If the output list item is of the type real, the Gw.d specification produces either an F or E format representation of its value according to the following criteria.

If N is the absolute value of the list item, then:

IF 0.1	$\leq$	N<1	output is F(w-4).d, 4X
IF 1	$\leq$	N<10	output is F(w-4).(d-1), 4X
		.	
		.	
		.	
IF 10^{d-2}	$\leq$	N<10^{d-1}	output is F(w-4).1, 4X
IF 10^{d-1}	$\leq$	N<10d	output is F(w-4).0, 4X
Otherwise			output is Ew.d

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, with the corresponding 573.19bbbb.

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 $10.^{d-1}$ (100), the specification would produce an E-format representation.

If the output list item is of type double precision, the Gw.d format specification functions in the same manner as Dw.d.

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.

Examples:

List Item Type	List Item Value	Specification	Output Field
REAL	+10.	G12.5	bb10.000bbbb
REAL	+100000.	G12.5	b0.10000E+06
INTEGER	-10	G5	bb-10
LOGICAL	.TRUE.	G4	bbbT
DOUBLE PRECISION	+123467890123.	G20.12	bb0.123456789012D+13
REAL	+1010101010.	G12.7	**********

Format Specification H (Strings)

The string (or Hollerith) format specification wHs, 's' or "s" allows character strings to be input without employing program storage locations. The item s in this specification is a string of one to 255 characters. An apostrophe or a quote mark within a proper string may be represented by two adjacent apostrophes or quote marks, respectively. The same result can be achieved if a quote mark appears in a string delimited by apostrophes or an apostrophe appears in a string delimited by quote marks.

When employed as a format specification connected with a READ statement, the wHs, 's', or "s" specification causes the character string to be replaced by the next w characters on the input record. The format specification is thus modified at the time of the program execution; use of this specification for a subsequent output action will result in the new string being output.

Example:

```
      READ(5,15)I
15    FORMAT(2X, 'DUMMYbSTRING' ,I3)
      WRITE(6,15)I
```

Using the above code, if input starting in column 1 were bbMASTERbCOPYb303, the resulting output would be bbMASTERbCOPYb303.

The string 'DUMMYbSTRING' would be replaced by the characters MASTERbCOPYb.

Format Specification I

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, real or double precision variable in the input list.

The integer data item must be in the form of an integer constant and is considered to be right-justified within the input data field. Leading, embedded, and trailing blanks within the field are read as zeros.

The magnitude of the value in the input field must not exceed the maximum magnitude permitted for an integer data item.

Examples:

Variable Type	Data Item	Specification	Internal Value
INTEGER	567	I3	+567
REAL	bb-329	I6	-329.
DOUBLE PRECISION	-26bbbb	I7	-260000.
INTEGER	bbb27	I5	+27
INTEGER	-bb234	I6	-234

Output Using Iw

On output, the integer format specification Iw causes the corresponding integer, real or double precision output list item to be written on the specified output file.

The integer number is placed right-justified in the output field over a field of blanks.

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
REAL	+23.92	I4	bb24
INTEGER	-79	I4	b-79
DOUBLE PRECISION	+67486.942678	I5	67487
REAL	-67.486	I5	bb-67
INTEGER	0	I3	bb0
INTEGER	-37216	I5	*****

Format Specification L

Input Using Lw

On input, the logical format specification Lw causes the value of the logical data item in the input field to be assigned to the corresponding variable of type logical in the input list. The input field width w must be greater than or equal to 1. There may be leading blanks. The first character encountered in the field exclusive of leading blanks must be either T or 1 for .TRUE. or F or 0 for .FALSE.. Any characters following the T, 1, F, or 0 are ignored.

Examples:

Data Item	Specification	Internal Value
T	L1	.TRUE.
bbF	L3	.FALSE.
bbbTRU	L6	.TRUE.

Output Using Lw

On output, the logical format specification Lw causes the corresponding logical list item in the output list to be written on the specified output file.

The logical value is placed right-justified in the output field over a field of blanks as a T or F, for a .TRUE. or .FALSE., respectively.

Examples:

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

Format Specification T

The use of Tt in a format list will cause the next item of data to be transferred to or from character position t.

Examples:

```
      WRITE   (6,10)
10    FORMAT(T4,  5HABCDE)
```

The execution of this WRITE statement will cause ABCDE to be written starting in column 4.

Format Specification X

On input or output, the editing specification wX or X will cause w characters to be skipped in the record. The specification X will cause one character to be skipped.

Format Specification Z

Input Using Zw

On input, the hexadecimal format specification Zw causes the value represented by the hexadecimal digits in the input field to be assigned to the corresponding integer, real, double precision, or logical variable in the input list. Leading, embedded, and trailing blanks within the input field are interpreted as zeros.

The Z format specification may be used to assign hexadecimal digits to a variable or array of any data type. The hexadecimal digits in the input field are transmitted right-justified to the corresponding input variable. If the type bits (refer to appendix E) are included in the external input field, they must correspond to the type bits in the internal representation of the variable, or an error message will be given.

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 and the remaining hexadecimal digits must contain the correct type bits.

Examples:

Variable Type	Data Item	Specification	Internal Value
INTEGER	000BC614E	Z9	000BC614E
REAL	4BEBCE508	Z9	4BEBCE508
DOUBLE PRECISION	C140000BC614E009EF	Z18	C140000BC614E009EF
LOGICAL	000000001	Z9	000000001 (.TRUE.)

Output Using Zw

On output, the hexadecimal format specification Zw causes the hexadecimal value of the corresponding integer, real, double precision, or logical output list item to be written on the specified output 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 "STORAGE ALLOCATION" in appendix E for additional information.)

Examples:

List Item Type	List Item Value	Specification	Output Field
INTEGER	000BC614E	Z10	b000BC614E
REAL	4BEBCE508	Z10	b4BEBCE508
INTEGER	000BC614E	Z6	BC614E
REAL	4BEBCE508	Z6	BCE508
DOUBLE PRECISION	C140000BC614E009EF	Z18	C140000BC614E009EF
LOGICAL	000000001 (.TRUE.)	Z10	b000000001

FORMAT STATEMENT

The nonexecutable FORMAT statement specifies what type of data conversion and editing is to be performed during input and output between the characters in the records in program files and the data words in internal storage.

The proper form of the FORMAT statement is as follows:

FORMAT STATEMENT
n FORMAT(s)
where n is a statement label and s is a specification list which consists of appropriate format specifications modified as desired by parentheses, repeat counts, and scale factor designators, separated by commas and/or one or more slashes.

The FORMAT statement is associated with a formatted READ or a formatted WRITE statement by means of the statement label n.

Correspondence is established between I/O list elements and the elements of the format item list in the standard FORTRAN fashion. Repetition of a portion of the format items is accomplished by enclosing that format portion in parentheses and optionally preceding the left parenthesis by a repeat count indicating the number of times the format portion is to be repeated before using the succeeding item.

If there is an I/O list on the I/O statement referencing the FORMAT statement, the specification list must contain at least one specification besides wHs, 's', "s" Tt, wX, X, /, or parentheses.

Record Fields

A field in a formatted record contains a character string representation of the value of a single I/O list element. The field width of a format specification specifies the number of characters in the field. A field begins with the first character position of a record for the first I/O list element, or the first character after the previous field. The Tt specification may be used to change the character pointer for subsequent editing.

In numeric input fields, all blank characters are interpreted as zeros. Plus signs are optional on input and may be omitted. When input data is entered under a real format specification (i.e., D, E, F, or G), a decimal point appearing in the input field overrides the decimal place count, d.

Format Field Separators

Two fields in a format item list are separated by a comma, a slash, or a series of slashes. A slash is used to indicate the end of a record. On input, any remaining characters in the current record are ignored when a slash is encountered in the specification list and the next record is read. On output, a slash causes the current record to be written, and any subsequent output is placed in the next output record(s). Multiple slashes may be used to skip several records on input or generate several blank records on output. The final right parenthesis of a format item list also causes the current record to be written.

Repeat Counts

Format specifications and format list portions enclosed in parentheses may optionally be immediately preceded by an unsigned integer constant. This constant indicates the number of times that portion of the specification list is to be interpreted. If no such repeat count is indicated, a repeat count of 1 is assumed. A repeat count is not allowed to precede a string variable.

If the outer right parenthesis of the format specification list is encountered before the I/O list is exhausted, control reverts to the repeat count of the repeat specification group terminated by the last preceding right parenthesis. If no other right parenthesis exists in the specification list, then control reverts to the first left parenthesis of the specification list.

The following are examples of the use of repeat counts. In each case, the repeat count is 3:

```
3F10.4
3(A6/)
3(3A6,3(/I12)/)
2P3E20.5
```

Scale Factor Designator

A scale factor designator may optionally precede a D, E, F, or G format specification to modify the scaling of a number on input or output.

A scale factor designator has the following form:

SCALE FACTOR DESIGNATOR
nP
where n is an integer constant which is the scale factor, and may be preceded by an optional + or – sign.

When FORMAT control is initiated, a scale of zero is automatically established and applies until a scale factor is encountered in the FORMAT statement. Once a scale factor is encountered, it applies to all subsequently encountered F, E, G and D conversions until a different scale factor or the end of the FORMAT statement is encountered.

The effect of the scale factor on input and output for each of the format specifiers is discussed in the following paragraphs.

The effect of the scale factor on F, E, G, and D input is as follows:

If there is an exponent in the external field, the scale factor has no effect.

If no exponent exists in the external field, the effect of the scale factor is as follows:

Internal value = external string * 10^{-n}

Examples:

External String	Specification	Internal Value
367.92930	2PF9.5	+3.679293
0.0369	-2PF6.4	+3.690000

The effect of the scale factor on F output is as follows:

The external string will equal the internal value * 10^{n}

Examples:

Internal Value	Specification	External String
+3.679293	2PF12.5	bbb367.92930
+3.679293	-2PF12.5	bbbbb0.03679

The effect of the scale factor on E output is as follows:

When the scale factor, n, is greater than or equal to zero, then n significant digits will be placed to the left of the decimal point and d-n+1 significant digits will be placed to the right of the decimal point.

When the scale factor, n, is negative, -n leading zeros will be placed immediately to the right of the decimal point with d+n significant digits following the zeros.

In each case, the exponent is adjusted so the value of the output quantity remains unchanged.

Examples:

Internal Value	Specification	External String
+3.679293	2PE12.5	b36.7929E-01
+3.679293	-2PE12.5	bb.00368E+03

The effect of the scale factor on D output is as follows:

The effect of the scale factor on D output is the same as for E output, except that a D is used instead of an E in the exponent.

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	One space before printing
Zero	Double space before printing
Minus sign	Triple space before printing
1	Skip to channel 1 of carriage control tape before printing
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 to control the action of the carriage and is not printed.

FORMAT SPECIFICATIONS IN ARRAYS

A formatted READ, WRITE, PRINT, or PUNCH statement may employ an array name as a format item. This array name must contain an EBCDIC string of characters representing a parenthesized format specification list. This specification list is defined identically as the specification list portion of a FORMAT statement.

The specification list may be stored in the array using a DATA statement, a READ statement, or an assignment statement. Employing a READ statement allows the format specification list to be unspecified until execution time.

Example:

The following program illustrates the use of format specifications contained in arrays.

```
? COMPILE FORMAT/ARRAY WITH FORTRAN;
? DA CARDS
      DIMENSION IFORM(2), DATA(6)
      DATA IFORM/'(X,F6.3)'/, DATA/1.2,3.4,5.6,7.8,9.2,4.5/
      DO 1 I-1,3
1     WRITE(6, IFORM) DATA(I)
      READ(5,2) IFORM
2     FORMAT(2A4)
      DO 3 I=4,6
3     WRITE(6, IFORM) DATA(I)
      STOP
      END
? END
? DATA FILE5
(X,F6.1)
? END
```

The program produces the following output:

```
 1.200
 3.400
 5.600
   7.8
   9.2
   4.5
```

NAMELIST STATEMENT AND NAMELIST I/O

The NAMELIST statement associates a list of scalar and/or array variable names with a unique symbolic name called a NAMELIST NAME. A NAMELIST NAME may only be used as a format specifier in a READ, WRITE, PRINT, or PUNCH statement. When a NAMELIST NAME is used for input, any or all of the variables associated with the NAMELIST NAME may be assigned values. A variable associated with a NAMELIST NAME retains its current value if no value is assigned to it. When the NAMELIST NAME is used for output, all of the associated variables are generated in NAMELIST FORMAT (format specified later).

The syntax for associating a NAMELIST NAME with a set of variable names is:

NAMELIST STATEMENT
NAMELIST /n/p/n/p . . .
where n is a unique NAMELIST NAME which is constructed in the same manner as a variable name; p is a list of scalar and array variable names separated by commas.

A scalar or array variable name may be associated with more than one NAMELIST NAME. For example:

```
NAMELIST /N1/I,J,K,/N2/I,L,K,J
```

associates I, J, and K with the NAMELIST NAME N1 and I, L, K, and J with the NAMELIST NAME N2.

A total of 511 variable names may be specified in each subprogram in NAMELIST statements.

A NAMELIST NAME must be declared in a NAMELIST statement in the program unit in which it is used. Neither a NAMELIST NAME nor a variable associated with the NAMELIST NAME may be a dummy argument of a subprogram (see section 12).

An I/O statement that contains a NAMELIST NAME may not also have an I/O list of variables. An I/O statement that contains a NAMELIST NAME may only write to a sequential file.

A NAMELIST NAME may be used as the name of a COMMON BLOCK.

NAMELIST Record Format

The format of the NAMELIST record must be as follows:

a. The first character must be a blank.

b. The second character must be an "&" character.

c. The NAMELIST NAME must immediately follow the "&" (no embedded blanks).

d. The NAMELIST NAME must be followed by at least one blank.

e. Following the blank, a list of VALUE ASSIGNMENTS separated by commas for any or all of the variables associated with the NAMELIST NAME must occur. If this list continues onto another record, the next record must have a blank as its first character.

f. The list of VALUE ASSIGNMENTS must be terminated by &END, and the remainder of the record is ignored.

Example of NAMELIST input and/or output record:

```
&OTH A=1, 0, B=3, 5678&END
```

If there is a run-time NAMELIST error, the line printed by the intrinsic before the program stack dump is the last record of the namelist input or output list in which the error occurred. This may not be the record on which the error occurred. Also, there are two characters in the line that are not the characters that were actually read. If the NAMELIST read missed the &END (perhaps it was mispunched or "&" occurred in column 1), the read will continue to the next &END and give an error there. Since anything after the &END is ignored, it is suggested that after each &END an identifying word or number be included.

NAMELIST Record Restrictions

No embedded blanks may appear in the NAMELIST NAME or in the variable names or assignments.

Value Assignments

A VALUE ASSIGNMENT on a NAMELIST input record may be:

a. In any order, but only those that are associated with the NAMELIST NAME may appear.

b. A variable followed by an "=" sign and a literal value (no embedded blanks).

c. An array name or an array element followed by an "=" sign and one or more literal values separated by commas, with or without repeat specifiers, a literal value may be either a constant, a quoted string of length less than or equal to four characters, or a Hollerith string of length less than or equal to four characters.

Example:

```
&NAME I=1, J=1,2,3,4,2*5,3*"ABCD"&END
```

where I is an integer scalar and J is an integer array dimensioned J(3,3).

d. If an assignment is made to an array name or an array element, the constants may be preceded by a repeat specifier of the form n*, where n is an unsigned integer constant.

e. If an assignment is made to an array name, up to as many elements as are in the array may appear in the list of value assignments. If assignment is made to an array element, up to as many constants as there are succeeding elements in the array may appear in the list of value assignments.

Values are assigned to the elements of the array in the order in which they are stored internally, beginning with the element indicated or if only the array name is specified, with the first element.

Input Using NAMELIST

When a READ statement referencing a NAMELIST NAME is executed, records are read until a NAMELIST NAME is found. If the NAMELIST NAME is misspelled, the read action continues to the end of the file.

Only those variables associated with the NAMELIST NAME may be assigned values and the variable assignments may appear in any order. After the &END has terminated the value assignments, the file is positioned at the beginning of the next record.

On NAMELIST input, no more than 4 characters may be assigned, even to a double-precision variable.

The following trailing blanks on NAMELIST input are treated as zeros, where b indicates a blank. Notice that trailing blanks can cause overflow.

E01b	E010
10b	100
10bH- - -	100H - - -
10b. - - -	100. - - -
10bE	100E - - -

The program in the following example uses the NAMELIST statement.

Example:

```
      DIMENSION A(2,2), B(3)
      NAMELIST /NTEST/A,B,C,D
      DATA B(2)/6.0/
      READ 2,C
2     FORMAT (F3.1)
      READ NTEST
      READ 2, B(3)
      PRINT 3,A,B,C,D
3     FORMAT (I2/2(1X,A4/),F4.1/3(I2/),F4.1/1X,A3)
      STOP
      END

      DATA:
     -9.6
      2.0
       &NTEST A=1, 2*'ABCD', 1, B=7,
       D = 'IJK'b&END              (END OF NTEST RECORD)
       5.0
```

When the READ NTEST statement is executed, the file is accessed repeatedly until the NTEST NAMELIST data group is located. The values specified are assigned to the elements of arrays A and B and to D as if these statements had been executed:

```
A(1,1)=1.0
A(2,1)='ABCD'
A(1,2)='ABCD'
A(2,2)=1.0
B(1)=7.0
D='IJK'
```

Note that "(END OF NTEST RECORD)" is ignored. Another READ is performed on the file, and the value 5 is read, converted to a real number, and assigned to B(3).

The output generated by the PRINT statement is:

```
1
ABCD
ABCD
1.0
7
6
5
9.6
IJK
```

Output Using NAMELIST

Values may be written with output statements using a NAMELIST NAME as a format specifier. When such an output statement is executed, the value of every variable and array element associated with the NAMELIST NAME is placed in a data group suitable for input by a READ statement using that NAMELIST

NAME. No repeat specifiers are used. The values of the array elements are given in the order in which the elements are stored internally.

Values are written according to the type of each data item. No facilities are available for writing strings using NAMELIST.

8. ASSIGNMENT STATEMENTS

The executable assignment statements allow an arithmetic, logical, or label value to be assigned to a program variable.

The two proper forms of the assignment statement are as follows:

ASSIGNMENT STATEMENT
1. v = e 2. ASSIGN n TO k
where v is either a non-logical variable or array element name and e is an arithmetic expression; or v is a logical variable or array element name and e is a logical expression; n is a statement label, and k is a simple integer variable.

ARITHMETIC ASSIGNMENT STATEMENT

An arithmetic assignment statement is represented by form 1 of the assignment statement, with v being a non-logical variable and e being an arithmetic expression.

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, v.

In general, the variable and the arithmetic expression need not be of the same type. If the types are different, the expression is first evaluated and automatic conversion is subsequently performed on the value obtained to 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 Expression		
Type of v	**Integer**	**Real**	**Double Precision**
INTEGER	assign	fix	fix
REAL	float	assign	rnd
DOUBLE PRECISION	float, ext	ext	assign

assign	Value is transferred without change.
fix	Indicates a conversion to integer with truncation.
float	Indicates conversion from integer internal form to real internal form.
rnd	Indicates conversion from double precision to real with rounding.
ext	Indicates conversion from real to double precision, with the second word set to zero.

The internal storage formats of the various data types are discussed in appendix E.

Examples:

```
G(IROW+2,JCOL) = IROW–N
A=I+2+B/3.6
C=6.2+(1/2.3)
```

LOGICAL ASSIGNMENT STATEMENT

A logical assignment statement is represented by form 1 of the assignment statement, with v being a logical variable and e being a logical expression. When such a statement is executed, the logical expression is evaluated, and the logical value obtained is placed into the storage word allocated to the variable, v.

Examples:

```
L(2,4)=L(1,1).AND.L(1,2)
L=G.GT.H.OR.B.EQ.C
L=.TRUE.
```

GO TO ASSIGNMENT STATEMENT

A GO TO assignment statement (or ASSIGN statement) is represented by form 2 of the assignment statement. This statement assigns the label of an executable statement to a simple variable of integer type in order that the variable may be used in an assigned GO TO statement. The ASSIGN statement is the only statement that may give a variable the value of a label.

The following is an example of the use of the ASSIGN statement:

```
ASSIGN 999 TO LABEL
GO TO LABEL, (1,2,999)
```

When the assigned GO TO statement is executed, control will be transferred to the executable statement bearing the label 999 in that program unit.

The referenced variable should not be used for purposes other than ASSIGN and assigned GO TO statements, since unpredictable results could occur.

9. CONTROL STATEMENTS

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

The control statements consist of the following:

CALL Statement.
CALL EXIT Statement.
CONTINUE Statement.
DO Statement.
GO TO Statement.
IF Statement.
PAUSE Statement.
RETURN Statement.
STOP Statement.

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

CALL STATEMENT

The executable CALL statement causes the specified subroutine to be executed. The proper forms of the CALL statement are:

CALL STATEMENT
CALL s or CALL s (a)
where s is the name of a SUBROUTINE subprogram and a is a list of actual arguments separated by commas.

The elements of the list of actual arguments must agree in number, order of appearance, and type with the dummy arguments appearing in the SUBROUTINE statement appearing at the beginning of the called subprogram. These actual arguments are evaluated in left-to-right order before the subroutine is entered.

Hexadecimal constants may not be used as actual arguments.

An actual argument in a subroutine reference may be one of the following items:

a. Any non-hexadecimal constant.

b. A simple variable.

c. An array element.

d. An array name.

e. An expression.

f. An item of the form: &n, where n is a statement label.

Execution of a CALL statement results in an association of actual arguments with all appearances of dummy arguments in executable statements in the subroutine body, and in an association of actual arguments with variable dimensions in the subroutine, if any.

If the actual argument is a simple variable, array element, or constant which is enclosed in parentheses or preceded by an unary + or -, or any other expression, the value of that argument is passed into the corresponding dummy argument, and any changes in that dummy argument will remain local to the subprogram, which referenced it. Therefore, any association the actual argument may have with a COMMON block is not shared by the dummy argument. A simple variable, array name, or array element which is not specified otherwise, is passed by name and any changes in the value of the corresponding dummy argument will also be made in the actual argument. Therefore, if the actual argument is an element of a COMMON block, any change incurred in the value of the corresponding variable within the subprogram will be simultaneously recorded in the corresponding location in the COMMON block.

Examples:

```
CALLX (A)        Passed by name.
CALLX ((A))
CALLX (A+B)      Passed by value.
CALLX (-A)
```

Certain actual arguments may only be passed to a dummy argument that is used in a prescribed manner within the subprogram. The correspondence in usage that must exist between actual and dummy argument of the same type is listed as follows:

Actual Argument	Dummy Argument
Hollerith constant; length $\leq$ 4 characters.	Simple integer variable.
Hollerith constant; length $>$ 4 characters.	Integer array.
Hollerith constant; in parentheses.	Double precision.
Any arithmetic constant.	Simple variable.
Simple variable.	Simple variable.
Array element.	Simple variable or array name.
Array name.	Array name.
An expression.	Simple variable.
&n (where n is a statement label).	Asterisk (*).

A simple variable in a dummy argument list may receive any constant or expression of the same type. If a dummy argument is assigned a value in the subprogram, then the corresponding actual argument must be a simple variable, an array element, or an array name.

Following these associations, control is transferred to the first executable statement following the appropriate SUBROUTINE statement. If an actual argument is an array element with an arithmetic expression as a subscript, then the arithmetic expression is evaluated at the time of the execution of the CALL statement, and the resulting array element is associated with the corresponding dummy argument in the subroutine.

Array Handling

If a dummy argument of a subroutine is an array name, the corresponding actual argument in the CALL statement must be an array name or an array element. The following correspondence is established between the actual array and the dummy array in the subroutine at the time the CALL statement is executed.

If the actual argument is an array name, the entire array may be accessed in the subroutine. The dummy array may be assigned as many subscripts as desired. The correspondence between the internal storage locations assigned the actual array and the elements of the dummy array is established as described in appendix E. The subscripts appearing in the declaration of the dummy array will only determine the correspondence between actual and dummy array elements and the number of array elements which will be referenced if the dummy array name alone appears in an I/O list in the subroutine.

For example, if the array A is declared by means of the statement:

```
REAL A(20)
```

and is passed as an actual argument to the dummy argument B, which is declared in the subroutine by means of the statement:

```
REAL B(5)
```

then all 20 elements of A may be referenced by means of B (i.e., B(18) is a valid array element), but the statement:

```
WRITE(6,1)B
```

will cause only the first five of these elements to be written.

If the actual argument is an array element, the array from that element on may be accessed in the subroutine. Considering the preceding example, if the array element A (11) had been passed to the subroutine instead of A, then the last 10 elements of this array (i.e., A(11) through A(20)) could be referenced by means of the dummy array B. The WRITE statement displayed would then cause the first five of these elements (i.e., A(11) through A(15)) to be written.

Subroutine Returns

If the subroutine does not cause termination of the program, the subprogram may return control to the next executable statement following the CALL statement or to the executable statement whose label appears in the actual argument list preceded by an ampersand (&). See the discussion of the RETURN statement in this section.

The following CALL statement examples illustrate the use of actual arguments.

```
CALL SUBA(A, B(2), 3HABC, 'TITLE', I+20)
CALL DATE (0)
CALL BRANCH (TVAR, &100, &200)
```

CALL EXIT STATEMENT

The executable CALL EXIT statement is provided to allow the termination of an executing program.

The following is the proper format of the CALL EXIT statement:

CALL EXIT STATEMENT
CALL EXIT
Execution of the CALL EXIT statement causes unconditional termination of the program.

The execution of this statement within a program which does not contain a subprogram named EXIT produces exactly the same result as the execution of a STOP statement.

This statement may appear at any point within any program unit where an executable statement may appear.

CALL DUMP STATEMENT

The executable CALL DUMP statement is provided primarily as a program debugging aid.

The following is the proper format of the CALL DUMP statement:

CALL DUMP STATEMENT
CALL DUMP
Execution of the CALL DUMP statement causes a dumpfile to be produced.

The execution of the CALL DUMP statement causes a dumpfile to be produced. After the dumpfile has been produced, execution continues to the statement following the CALL DUMP statement.

CALL SWITCH STATEMENT

The executable CALL SSWTCH (I,J) statement enables a program to determine the setting of the least significant bit of program switches 1 through 8. The program switches are set at run time by the MCP SWITCH control instruction attribute. To set a sense switch, assign an odd number to it. If zero or an even number is assigned to a program switch, the sense switch is not set.

CALL SSWTCH STATEMENT
CALL SSWTCH (I,J)
Execution of the CALL SSWTCH statement interrogates sense switch I. J is assigned a 2 if sense switch I is set. J is assigned a 1 if sense switch I is not set.

The execution of the CALL SSWTCH (I,J) statement interrogates sense switch I, which must be an integer between 1 and 8. If sense switch I is set, then J is assigned a value of 2; otherwise, J is assigned a value of 1.

CALL OVERFL STATEMENT

The CALL OVERFL statement enables programmatic recovery when an exponent overflow or an exponent underflow has occurred. The proper form of the CALL OVERFL statement follows:

CALL OVERFL STATEMENT
CALL OVERFL (I)
Execution of the CALL OVERFL statement sets I to a 1 if an exponent overflow has occurred since the last CALL OVERFL statement. I is set to a 2 if neither an exponent overflow nor an exponent underflow has occurred since the last CALL OVERFL statement. I is set to 3 if an exponent underflow has occurred since the last CALL OVERFL statement.

If execution of a CALL OVERFL occurs in the main program or in any subprogram, then any occurrence of an exponent overflow results in the maximum value being returned (5.7896041E76 for REAL; 5.78960446186580976D76 for DOUBLE-PRECISION). In addition, a CALL OVERFL will return a zero result if any exponent underflow occurs.

CALL DUCHK STATEMENT

The CALL DUCHK statement enables programmatic recovery when a divide by zero occurs. The proper form of the CALL DUCHK statement follows:

CALL DUCHK STATEMENT
CALL DUCHK (I)
Execution of the CALL DUCHK statement causes I to be set to 1 if a divide by zero has occurred since the last CALL DUCHK statement; otherwise, I is set to 2.

If a CALL DUCHK statement occurs in the main program or in any subprogram, then any occurrence of a divide by zero results in a zero being returned as the result of the arithmetic operation.

CALL GETCH STATEMENT

The CALL GETCH statement transfers any byte of an integer variable to the leftmost byte of another integer variable. The proper form of the CALL GETCH statement follows:

CALL GETCH STATEMENT CALL GETCH (I1, I2, I3)
Execution of the CALL GETCH statement will transfer any byte from I1 to the leftmost byte of I3. I2 specifies the byte position within I1.

The CALL GETCH statement requires three integer variables as operands. The first is the variable from which one of the four bytes is taken. The second variable is the byte position, which can be 1, 2, 3, or 4, where a byte position of 1 designates the leftmost byte). The third parameter is the receiving variable. The byte transferred is placed in the leftmost position of the receiving variable.

The CALL GETCH statement is used in the following example:

Example:

```
    IARG1 = ABCD
    IARG3 = FGHI
    DO 10 I = 1,4
    CALL GETCH (IARG1, I, IARG3)
10 CONTINUE
```

Execution of the routine in this example results in the following:

IARG1	I	IARG3
ABCD	1	AGHI
ABCD	2	BGHI
ABCD	3	CGHI
ABCD	4	DGHI

CALL PUTCH STATEMENT

The CALL PUTCH statement transfers the leftmost byte of an integer variable to any byte of another integer variable. The proper form of the CALL PUTCH statement follows:

CALL PUTCH STATEMENT
CALL PUTCH (I1, I2, I3)
Execution of the CALL PUTCH statement transfers the leftmost byte of I3 to any byte of I1. I2 specifies the byte position within I1.

The CALL PUTCH statement requires three integer variables as operands. The first is the receiving variable. The second is the byte position of the receiving variable into which the byte is to be placed, which can be either 1, 2, 3, or 4, where a byte position of 1 designates the leftmost byte. The third parameter is the variable from which the leftmost (first) byte is taken.

The CALL PUTCH statement is used in the following example:

Example:

```
    IARG1 = ABCD
    IARG3 = FGHI
    DO 10 I = 1,4
    CALL PUTCH (IARG1, I, IARG3)
 10 CONTINUE
```

Execution of the routine in this example results in the following:

IARG1	I	IARG3
FBCD	1	FGHI
FFCD	2	FGHI
FFFD	3	FGHI
FFFF	4	FGHI

CONTINUE STATEMENT

The executable CONTINUE statement is provided primarily as a program documentation aid, as the execution of this statement produces no action.

The following is the proper format of the CONTINUE statement:

CONTINUE STATEMENT
CONTINUE
Execution of the CONTINUE statement has no effect.

The CONTINUE statement is used primarily as 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.

Examples of the use of the CONTINUE statement are provided in the following sample portion of a FORTRAN program.

```
      .
      .
      .
      DO  1 I=2,10,2
      A(I) = I/M
      WRITE (6,10) A (I)
      IF (A(I)) 3,1,1
    1 CONTINUE
      M=–M
    3 CONTINUE
      .
      .
      .
```

In this sample of source code, the CONTINUE statement labeled 1 is used as the final statement of a DO loop, and the CONTINUE statement labeled 3 is used as a transfer point for an arithmetic IF statement. The use of the CONTINUE allows the end of the DO loop to be readily located within a listing of the program. The second CONTINUE statement allows transfer to an arbitrary point in the program, the location of which may be changed merely by changing the location of the CONTINUE card.

DO STATEMENT

The executable DO statement is a control statement provided to alter the order of the execution of program statements in such a manner that a series of statements will be repeatedly executed while the value of a specified program variable is varied between specified limits. The number of iterations is dependent upon this control variable.

The following are the proper forms of the DO statement:

DO STATEMENT
DO s v = m,n,i or DO s v = m,n
where s is a label on an executable statement following the DO statement, v is the integer or real control variable, and m, n, and i are the initial parameter, terminal parameter, and incrementation parameter, respectively, each of which is any arithmetic expression.

The DO statement causes repeated execution of the statements in its range. The range of a DO statement is defined to be the first executable statement following the DO up to and including the statement whose label is specified in the DO statement.

The initial parameter, terminal parameter, and incrementation parameter of a DO statement may be any arithmetic expression. If the optional incrementation parameter is omitted, a value of 1 is assumed by the compiler.

The range of a DO is executed with a new value assigned to the control variable prior to each repetition. The initial parameter is assigned prior to the first execution of the range. Prior to each subsequent repetition, the control variable is incremented by the incrementation parameter.

A test is made immediately after incrementing the control variable to determine whether or not the terminal parameter has been exceeded. If not, the range of the DO is executed; if the terminal parameter has been exceeded, the DO is considered satisfied. In any case, the range of a DO is executed at least once.

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

The control variable is available for use within its range, and may be modified as desired. The incrementation parameter and terminal parameter are reevaluated prior to their use and may also be modified by any statement within the range. There is no restriction on transfers out of or into the range of a DO. If a transfer is made into the range of a DO, the programmer is responsible for the appropriate assignment of a value to the control variable. When a DO statement has been satisfied, the value of its control variable is the value which failed the final test.

When several DO statements share the last statement of their ranges, the control variable of one DO statement is not reassigned and tested until each DO statement in its range is satisfied. When the outermost DO statement is satisfied, control continues with the next executable statement following its range.

The following is an example of a section of a program unit using a DO statement:

```
      J= 10
      DO 2 I= 1,J
      A(I)=0
    2 B(I,I)=0
```

In this DO loop, A (1) through A (10) inclusive, and B (1,1), B (2,2), . . ., B (10,10) will all be set to zero.

Nesting

DO loops may be placed inside DO loops. This procedure is called "nesting" DO loops. For example:

```
      DO 10 I=I,23
      DO 10 J=3,I-1
   10 C(I,J+1)= I*J
```

The range of a nested DO loop must be entirely inside the range of the next outer DO loop.

Parameter Alteration

DO loop parameters may be changed during the execution of the DO loop. For example:

```
    SUM=0
    K=100
    DO 24 INTGR=2,K,2
    SUM=SUM + INTGR
24  K=K-2
```

In this example, the variable K is diminished in value by 2 each time that the DO loop is executed. This will affect the indexing in such a manner as to cause termination of the DO loop when INTGR=50, rather than 100 as was initially declared.

The control variable INTGR is incremented by 2 each time the DO loop is executed because of the presence of the incrementation parameter 2.

GO TO STATEMENT

The executable 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 following are the proper forms of the GO TO statement:

GO TO STATEMENT
1. GO TO n 2. GO TO v,(m) 3. GO TO (m),a
where n is a label appearing on an executable statement appearing in the same program unit, v is an integer variable, m is a list of statement labels separated by commas, and a is an integer arithmetic expression.

Unconditional GO TO

The simplest form of the GO TO statement is displayed by form 1, which represents the unconditional GO TO statement.

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 statement 23.

Assigned GO TO

Form 2 represents the assigned GO TO statement. The execution of this statement causes control to be transferred to the statement whose label was last assigned to the variable v by an ASSIGN statement. (See section 8.) The statement labels (m) must appear in the same program unit as the ASSIGN statement and assigned GO TO statement. The value of v must not be changed by another ASSIGN statement, between execution of the ASSIGN statement and execution of the assigned GO TO statement.

```
ASSIGN 23 to K
GO TO K, (10, 23, 30, 45)
```

Failure to assign a statement label to the variable v which appears in the list with an ASSIGN statement will cause program termination when the assigned GO TO is executed.

Computed GO TO

Form 3 represents 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. It is evaluated and the result is used to select one of the labels in the list.

If the 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 on to the next statement.

An example of a computed GO TO statement is:

```
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 will pass to the n-th statement in the list. For example, if I + 1 = 4 (i.e., I = 3), then control will pass 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 will pass to statement number 1, since both the first and fifth elements of the label list are 1. In this example, if I + 1 is less than 1 or greater than 6 (I less than 0 or greater than 5), control will pass to the next executable statement after the computed GO TO.

Notice also that the statement:

```
GO TO (3), I
```

is not the same as:

```
GO TO 3
```

The latter case is unconditional, whereas in the former case, control will pass to statement number 3 only if I = 1. If I has any other value, control will pass to the next statement, which may or may not be statement number 3.

IF STATEMENT

The executable IF statement is a control statement provided to cause branching from one point in an executing program to another point, depending upon an arithmetic or logical value.

The following are the proper forms of the IF statement:

IF STATEMENT
1. IF(a) n,z,p 2. IF(e) s
where a is an arithmetic expression, n, z, and p are labels appearing on executable statements in the same program unit, e is a logical expression, and s is any executable statement except a DO statement or a form 2 IF statement.

Arithmetic IF

Form 1 of the IF statement represents an arithmetic IF statement. The arithmetic IF statement is a three-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 is:

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

If I-J is negative, control will be transferred to statement number 10; if zero, to statement number 20; and if positive, to statement number 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 numbers of an arithmetic IF need be different.

For example:

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

In this example, control will pass to statement number 3 only if (A-2)*(B-3) is greater than zero. Otherwise, control will pass to statement number 10.

Notice that if all three statement numbers in the arithmetic IF statement are identical, the result is the same as an unconditional GO TO, provided that execution is not terminated during evaluation of the expression.

The statement following an arithmetic IF generally will need a label. This is not a syntactical requirement, but since an arithmetic IF breaks the sequential flow of execution to a labeled statement, it will never be possible to return to execute the statement following the IF unless that subsequent statement has a label.

Logical IF

Form 2 of the IF statement represents a logical IF statement. The logical IF will conditionally execute a statement following a logical expression. The logical expression following the IF and enclosed in parentheses is evaluated. If this logical expression is .TRUE., then the statement following the logical expression is executed. If it is .FALSE., then the statement is ignored. In either case, control then

passes normally on to the next statement, unless the statement following the logical expression was executed and caused a branch to another point in the program.

The statement following the logical expression may be any executable statement except a DO statement or another logical IF.

The following are examples of logical IF statements:

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

   If A equals B or C equals D (or both), then G will be incremented by 1. Otherwise, G will remain unchanged. In any event, control will then pass on to the next statement.

b. IF (L1) GO TO 97

   If L1 (which must be declared to be a LOGICAL variable) is .TRUE., then control will pass to statement number 97. If L1 is .FALSE., then control will pass on to the next statement.

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

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

PAUSE STATEMENT

The executable PAUSE statement is provided to allow an executing program to be suspended indefinitely.

The proper format of the PAUSE statement is:

PAUSE STATEMENT
PAUSE or PAUSE p
where p is a one- to eight-digit unsigned integer constant or a proper string (quoted string) containing up to eight characters.

The execution of the PAUSE statement causes the unconditional suspension of the program being executed pending operator action. In addition to suspending the program, the execution of this statement causes the optional integer or string following the PAUSE to be displayed at the operators console.

The program may be resumed at the first executable statement following the PAUSE statement, by means of a system input message from the operator's console, of the form:

mOK

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

mDS

where m is the mix number of the job. The following are valid examples of the PAUSE statement:

```
PAUSE
PAUSE 2
PAUSE 'JUMP'
```

RETURN STATEMENT

The executable 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 forms of the RETURN statement are:

RETURN STATEMENT
1. RETURN 2. RETURN a
where a is an integer arithmetic expression.

A SUBROUTINE or FUNCTION subprogram may 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. The point in the calling program at which execution resumes is determined by the form of the RETURN employed.

A RETURN in a main program is prohibited.

Standard Return

The execution of a RETURN statement of the first type causes a standard return to the calling program unit. For a subroutine, this means the next executable statement following the CALL statement invoking the subroutine will be the next statement executed. For a function, standard processing of the function reference will be performed and the calling program unit will continue executing in the standard fashion.

If the END statement of the subprogram is encountered before a RETURN statement is executed, a standard return will be performed. If no RETURN statement is found in a subroutine or function during execution, a warning message will be generated.

Nonstandard Return

Form 2 of the RETURN statement allows control to be returned to a specified labeled executable statement in the calling program unit. A nonstandard return from a function is not allowed.

A nonstandard return is indicated in a subprogram by a RETURN followed by an arithmetic expression. The value of this expression must be an integer. This final value, n, is used to select the n-th asterisk (*) in the dummy argument list of the SUBROUTINE. If n is greater than the number of * items in this list or if n is less than or equal to zero, program execution is terminated.

The statement label corresponding to the selected asterisk and found in the actual argument list preceded by an ampersand (&) will be 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 the SUBROUTINE statement contains an asterisk. (See section 12.)

An example of a nonstandard return is provided by the following excerpt:

```
    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 A+2
 7  RETURN
    END
```

In this example, suppose that when the IF statement in the subroutine is reached, A is positive. Then there will be a branch to statement number 7, which is a normal return. The subroutine will then return to the calling program unit at the statement following the CALL, statement number 15.

Supposing that A is less than zero, then when the IF statement in the subroutine is reached, there will be a branch to statement number 5. This is a nonstandard return to the first label in the actual argument list, since RETURN 1 selects the first asterisk in the dummy argument list. Then the return in the calling program unit will be to statement number 33.

If A is equal to zero, when the IF statement in subroutine SUBA is executed, there will be a branch to statement number 6. Since A = 0, when the arithmetic expression is evaluated, it will read RETURN 2. This will cause a return to statement number 20 in the calling program unit.

Had a RETURN 3 or RETURN 0 been attempted in SUBA, execution would have been terminated.

A CALL statement that results in a nonstandard RETURN may be regarded most easily as a CALL followed by a computed GO TO. For example, the statement:

```
CALL SUBA(A1,&33,A2,&20,II)
```

may be treated as an effectively equivalent replacement for:

```
CALL SUBA(A1,II,JUMP)
GO TO (33,20),JUMP
```

where JUMP is assigned a value of 1 or 2 (or some other value if return is to be standard) by the subroutine.

STOP STATEMENT

The executable STOP statement is provided to allow the termination of an executing program.

The following is the proper format of the STOP statement:

STOP STATEMENT
STOP or STOP s
where s is a one- to eight-digit unsigned integer constant.

The execution of the STOP statement causes the unconditional termination of the execution of the program being executed. This statement may appear at any point within any program unit except a BLOCK DATA subprogram.

The execution of a STOP statement or a CALL EXIT statement (for a program with no subroutine named EXIT) is the generally accepted manner in which a program may reach error-free termination. The optional integer following the STOP is displayed on the operator's console.

The following are valid examples of the STOP statement:

```
STOP
STOP 99
```

10. FILE DECLARATION STATEMENT

The FILE declaration statement associates a unit number with a data file. A data file is referenced in a FORTRAN I/O statement with a unit number. By default, the B 1800/B 1700 FORTRAN compiler associates the unit numbers 5, 6, and 7 with the card reader, line printer, and card punch files, respectively. When other than these default associations are required, or additional files are desired, FILE declaration statements must be used to inform the compiler of the attributes of the files.

Any attempt to access a non-default file without a FILE declaration statement results in a run-time error.

The FILE declaration statement is coded as follows:

Card Columns	Contents
1 through 6	FILEbb (where b = blank)
7 through 72	Unit number = external file name. UNIT = hardware-type, attribute list ...
73 through 80	Sequence number or blank.

The FILE declaration statement is coded in free-form format, with the exception of card columns 1 through 6. Blanks appearing between specifications in columns 7 through 72 are ignored, and commas are used as delimiters. If the FILE declaration statement cannot be contained in one card, continuation cards are permitted as defined for the FORTRAN language.

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

FILE declaration statements must precede the first executable statement of the main program and any intervening cards must be blank or comment cards. The default file descriptions associated with unit numbers 5, 6, and 7 are shown in table 10-1.

Table 10-1. Unit Number/Hardware Type Default Associations

Unit Number	Hardware Type	Internal/ External File-Name	No. of Buffers	Blocking Factor	Record Length
5	Card Reader	FILE5	1	1	80 Characters
6	Line Printer	FILE6	1	1	120 Characters
7	Card Punch	FILE7	1	1	80 Characters

Individual file attributes listed for each hardware type in table 10-1 may be redefined in a FILE declaration statement. Those attributes not specifically redefined retain the default condition for that hardware type.

Example:

```
FILE 3=CARDIN, UNIT=READER
```

In the above example, compilation of a program in which unit number 3 has been used to represent the card reader requires the indicated FILE declaration statement to define unit 3 as the card reader to the compiler.

The internal file name is FILE3 and the external file name is CARDIN. All other attributes shown in table 10-1 for the card reader are assigned by default.

A FILE declaration statement is not required if only unit numbers 5, 6, or 7 are used and the default hardware type associations shown in table 10-1 are assumed.

Example:

```
      N=5

      READ(N,20)X,Y,Z
20    FORMAT (3F6.2)

      END
```

In the above example, a simple integer variable is used as a unit number for the card reader. Since the variable is programmatically assigned, the default unit number, i.e., 5, for the card reader, a FILE declaration statement is not required.

Example:

```
FILE 4=INPUT, UNIT=READER

      N=4

      READ (N,20) X,Y,Z
      FORMAT (3F6.2)

      END
```

In the above example, a FILE declaration statement is required to inform the compiler that unit number 4 is used to represent the card reader. All other attributes shown in table 10-1 for the card reader are assigned by default.

EXTERNAL FILE NAME

A file-name uses the same naming convention as a program-name; that is, it consists of:

a. family-name

b. family-name/ file-identifier

c. dp-id/ family-name/ file-identifier

d. dp-id/family-name

where dp-id is a user disk pack identifier.

HARDWARE TYPE

The hardware-type of a data file is specified in a FILE declaration statement by means of the following key words.

Key Word	Device Type
PRINTER	Line Printer.
READER	Card Reader.
PUNCH	Card Punch.
TAPE	7- or 9-track magnetic tape.
TAPE9	9-track magnetic tape.
TAPE7	7-track magnetic tape.
DISK	Head-per-track disk or disk pack.
DISKFILE	Head-per-track disk.
DISKPACK	Disk Pack.
SPO	Console Printer.
PTR	Paper Tape Reader.
PTP	Paper Tape Punch.
REMOTE	Remote Terminal.

The hardware-type must be specified if other than the default devices for unit numbers 5, 6, and 7 are used.

Attribute-List

The key words which may be used in the attribute-list, together with their meaning, are shown below:

Key Word	Meaning
UNLABELED (or UNLABELLED)	Unlabeled file (not valid for disk).
BLOCKING= unsigned integer	Number of logical records per block.
BUFFERS= unsigned integer	Number of buffers.
RECORD= unsigned integer	Record length, in characters. The maximum record size in a FILE declaration statement is 2047 characters. The maximum file size is also 2047 characters. After compilation, a file can be modified to contain fewer characters than originally declared; the file cannot be modified to contain more characters than originally declared.
LOCK	Close with LOCK at EOJ if file still open.
NEW	Create a new file even though an old one exists. If first access is input, a no file condition will result. Must be specified to create a new file.
SAVE= unsigned integer	Specifies save factor of the number of days to retain file.
RANDOM	Random access technique.
RECORDS.AREA= unsigned integer	Number of records per area on disk with default number of 100. This is rounded up to the next multiple of blocking factor.

FILE DECLARATION STATEMENT

Key Word	Meaning
AREAS= unsigned integer	The unsigned integer is the number of areas, with a default of 25.

A FILE declaration statement is required in the compilation deck whenever the source program utilizes disk or tape files or input/output to the console printer, because no default unit number and hardware type association is recognized for these peripheral devices. The compiler, however, does associate certain default file attributes with each of these hardware types.

The default file attributes associated with disk, tape, and the console printer are as follows:

Hardware Type	Default Attributes
DISK	Buffers = 1. Blocking = 1. Record = 180 characters. Records per area = 100. Number of areas = 25. Access mode = serial.
MAGNETIC TAPE	Buffers = 1. Blocking = 1. Record = 180 characters.
CONSOLE PRINTER	Buffers = 1. Blocking = 1. Record = 70 characters.
PAPER TAPE READER	Buffers = 1. Blocking = 1. Record = 80 characters.
PAPER TAPE PUNCH	Buffers = 1. Blocking = 1. Record = 80 characters.

Association of a unit number with tape or disk or the console printer (by means of a FILE declaration statement) associates that unit number with the default attributes for the indicated device as shown above. Individual attributes may, of course, be redefined on the FILE declaration statement. Those file attributes, however, not specifically redefined retain the default condition.

Example:

```
FILE 9=DSKFIL, UNIT=DISK, RANDOM, RECORDS.AREA=1000
```

In the above example, unit number 9 is associated with a disk file having 1000 records per area and a random access mode. All other attributes shown above for disk files are assigned by default.

11. INPUT/OUTPUT STATEMENTS

INPUT/OUTPUT statements are used to transfer data between external and internal storage. The I/O statement constructs used in these transfers will be discussed in this section in the following order.

READ statements.
WRITE statements.
PRINT statements.
PUNCH statement.
I/O variable lists.
Action specifiers.
REWIND statement.
BACKSPACE statement.
CLOSE statement.
ENDFILE statement.
LOCK statement.
PURGE statement.
CHANGE statement.
Multi-file tape handling.

In explanations presented in the portions on input and output in this section, the symbols u, r, f, k, l, m, /, and * have the following meanings unless otherwise specified.

Symbol	Meaning
u	Unit number or file specifier. The unit number is an integer constant or simple integer variable whose value identifies the file being used for input or output. Unless otherwise specified by a FILE declaration statement, it is assumed at object time that the unit number designates the default hardware type, in section 10. Unit numbers 5, 6, and 7 are assigned to card reader, line printer, and card punch files, respectively, by default.
r	Random record number. It is an integer constant or integer expression whose value represents a particular record within a random disk file.
f	Format specifier. It is the label of a FORMAT statement, the name of an array containing format specifications, or a NAMELIST NAME.
l	Action specifier. It specifies a statement label to which a branch is made if one of the following conditions are encountered during execution of a READ or WRITE statement: (1) a parity error, (2) end-of-file, or (3) data error.
m	Input/Output Variable list. It may be a blank or it may contain one or more variables and/or implied DO loops, in any combination. An output variable list may also contain an arithmetic expression.
	Free-format indicator.
*	Free-format output specifier.

READ STATEMENT

The READ statement allows data to be read from peripheral storage, converted, and assigned to internal storage locations indicated by the I/O variable list (see definition in this section) or format portion of the statement. The data may be converted during the transfer process as indicated by a formatted READ, or read without conversion by using an unformatted READ. Provision may also be made to handle errors incurred during the read.

The proper forms of the READ statement are:

READ STATEMENT
1. READ f,m 2. READ (u,f)m 3. READ (u=r,f)m 4. READ (u,f,l)m 5. READ (u=r,f,l)m 6. READ f 7. READ (u,f) 8. READ (u=r,f) 9. READ (u,f,l) 10. READ (u=r,f,l) 11. READ (u)m 12. READ (u=r)m 13. READ (u,l)m 14. READ (u=r,l)m 15. READ (u) 16. READ (u=r) 17. READ (u,l) 18. READ (u=r,l) 19. READ / , m 20. READ (u /) m 21. READ (u , / , l) m
where u and r are integer arithmetic expressions representing a unit number and record number, respectively; f is a format specifier; l is an action specifier list; m is an I/O variable list; / is a free-format indicator.

File Referenced

The file is indicated by the unit number, u, except in forms 1, 6, and 19, where u is implicitly declared to be unit number 5, the card reader.

Record Number for Random Read

A record number may follow the unit number as shown in READ statement forms 3, 5, 8, 10, 12, 14, 16 and 18 if the file indicated by the unit number was declared RANDOM in the FILE declaration statement (see section 10). The record number must have a value between 1 and the number of records in the designated file inclusive. The equal sign separating the unit number and the record number may be replaced by an apostrophe.

The record number is used to position the file at the specified record before the read is performed. After the READ statement is executed, the file is positioned at the record immediately after the record read.

If the READ statement has a record number, the file must have an accessing technique of RANDOM. Conversely, if the record number is not present, the file must be SEQUENTIAL.

Formatted READ Statement

A formatted READ statement is denoted by forms 1 through 10 of the READ statement. Such READ statements are always associated with a format specifier. When a format is associated with a READ statement, the data transferred is scanned and edited according to the designated format specifications. No I/O variable list may occur in a READ statement using a NAMELIST name.

The following are examples of valid formatted READ statements:

```
READ (5, 1) AB
READ (FILEN, 75) (A(I), I=3,10)
READ 12
READ (23, 12345, DATA=124) A,B,C
READ (23=12,AF)
READ 5, ARRAY
READ (5, ARRAY)
READ (5, B, ERR=2)Y,Z(6),(A(J),J=1,6)
READ (I=1R-1,NMLST)
```

Unformatted READ Statement

An unformatted READ statement is denoted by forms 11 through 18. No editing of transferred data is associated with these forms of the READ statement. (See appendix F for description of unformatted records).

Execution of an unformatted READ statement inputs one logical 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 random READ specifies more data to be transferred than is present in the record, a DATA error occurs. The I/O variable list for an unformatted sequential READ may transfer more than one record.

Free-Format READ Statement

A free-format READ statement is denoted by forms 19 through 21. A free-format READ statement reads values from the input file until the I/O variable list is exhausted.

Data values in the input file must be separated by a comma (,) character. Blank characters, including embedded blank characters, are ignored.

The end of a physical record terminates the current value; any subsequent value is read from the following physical record. Only certain format specifications are permitted for each variable type:

Variable Type	Format Specifications
INTEGER	I format
REAL	I, E, F, and D format
DOUBLE	I, E, F, and D format
COMPLEX	I, E, F, and D format
LOGICAL	L format

If a blank or null field is read, the value defaults to 0 or FALSE.

The following is an example of a free-format READ statement:

```
LOGICAL O
READ  (5, / , END=50)  T, I, O
```

The following data record can be read by the preceding example:

```
1234.56,227, 1
```

WRITE STATEMENT

The WRITE statement allows data to be written to peripheral storage from the internal storage locations indicated by the I/O variable list or format portion of the statement. The data may be converted during the transfer process and positioned within records of a file as indicated by a formatted WRITE or written without conversion by using an unformatted WRITE. Provision may also be made to handle errors incurred during the write operation.

The proper forms of the WRITE statement are:

WRITE STATEMENT
1. WRITE (u,f)m 2. WRITE (u=r,f)m 3. WRITE (u,f,1)m 4. WRITE (u=r,f,1)m 5. WRITE (u,f) 6. WRITE (u=r,f) 7. WRITE (u,f,1) 8. WRITE (u=r,f,1) 9. WRITE (u)m 10. WRITE (u=r)m 11. WRITE (u,1)m 12. WRITE (u=r,1)m 13. WRITE (u , /) m 14. WRITE (u , / /) m 15. WRITE (u , * /) m 16. WRITE (u , * / /) m
where u and r are integer arithmetic expressions representing a unit number and record number, respectively; f is a format specifier; 1 is an action specifier list; m is an I/O variable list; / is a free-format indicator; * is a free-format output specifier.

File Referenced

Execution of a WRITE statement writes data from internal storage to peripheral storage. The file is indicated by the unit number, u.

Record Number for Random WRITE

A record number may follow the unit number as shown in WRITE statement forms 2, 4, 6, 8, 10, and 12. The record number must have a value between 1 and the number of records in the file indicated by the unit number, inclusive. The equal sign separating the unit number and the record number may be replaced by an apostrophe.

The record number positions the file at the specified record before the write is performed. Otherwise, the file position is unchanged prior to the write. After the WRITE statement is executed, the file is positioned at the record immediately following the record written.

If the WRITE statement has a record number, the file must have an accessing technique of RANDOM. Conversely, if the record number is not present, the file must be SEQUENTIAL.

Formatted WRITE Statement

A formatted WRITE statement is denoted by forms 1 through 8 of the WRITE statement. Such statements are always associated with a format specifier. When a format is associated with a WRITE statement, the data transferred is converted and positioned within the record(s) according to the designated format specifier. No I/O variable list may occur in a WRITE statement using a NAMELIST name. The following are examples of valid formatted WRITE statements:

Examples:

```
WRITE (6, 1)AB
WRITE (FILEN,75) (A(I),I=3,10)
WRITE (23,12345,ERR=12400) 6,A,4,B,3,C
WRITE (23=12,AF)
WRITE (7,ARRAY)
WRITE (6,B,ERR=2)&,Z(6),(A(J),J=1,6)
WRITE (I=(IR*2)+1,NMLST)
```

Unformatted WRITE Statement

An unformatted WRITE statement is denoted by forms 9 through 12. No conversion of data is associated with these forms of the WRITE statement.

Execution of an unformatted WRITE statement writes one or more logical 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 (see appendix F, Description of Unformatted Records).

The unformatted WRITE statement depends on the declared or default size of records within the affected file. When an unformatted random WRITE statement attempts to transfer more storage units than may be contained on one record, the program is terminated unless the statement contains a DATA or ERR action specifier. An unformatted sequential WRITE may transfer more than one record.

Free-Format WRITE Statement

A free-format WRITE statement is denoted by forms 13 through 16. A free-format WRITE statement writes values to the output file until the I/O variable list is exhausted.

Values are written to the output file using the most appropriate format. Trailing zero (0) and blank characters are removed. If only one slash (/) character is present in the statement (forms 13 and 15), the values are separated by a comma (,) character and a blank character. If two slash (/) characters are present in the statement (forms 14 and 16), the values are separated by two blank characters.

NOTE

Since a comma (,) character is not used to separate values when using forms 14 and 16, the output cannot be read using free-format input.

If the optional asterisk (*) character is present (forms 15 and 16), the values are preceded by < variable-name >=. If the value is an expression or an array element, < EXP > is substituted for the name.

If a value (including the preceding blank character, the trailing comma (,) or blank character, and < variable-name >= or < EXP >=, if required) cannot fit on the remaining portion of a record, it is written to the following record.

Literal strings of 255 characters or less can be written with a free-format WRITE statement if the string fits within a single record.

The following program segment contains an example of a free-format WRITE statement:

```
DIMENSION ARRAY(10)
LOGICAL D
A = 27.2
I = 19
D = .FALSE.
ARRAY(7) = 1.23
WRITE (3,*//)  A,I,D,ARRAY(7),(FLOAT(I)+A)
```

Execution of this program segment results in the following output:

```
A=27.2      I=19      D=F    < EXP >=1.23     < EXP >=46.2
```

PRINT STATEMENT

The PRINT statement receives data from the internal storage locations indicated by the I/O variable list or format specifier of the PRINT statement, converts the data, and writes it to the line printer. No action specifier is allowed with this statement, and the unit number is not explicitly specified.

The proper forms of the PRINT statement are:

PRINT STATEMENT
1. PRINT f,m 2. PRINT / ,m 3. PRINT / / ,m 4. PRINT * / , m 5. PRINT * / / , m
where f is a format specifier; m is an I/O variable list; / is a free-format indicator; * is a free-format output specifier.

File Referenced

Execution of a PRINT statement writes data from internal storage to one or more records of the implied unit number 6, a line printer file.

Record Access

The position of the data file is unchanged prior to the execution of the PRINT statement. After such a statement is executed, the file is positioned at the record immediately after the record(s) written.

Formatted PRINT Statement

A formatted PRINT statement is denoted by form 1. A format item is associated with this statement, specifying the manner in which the transferred data is edited. This item may be the label of a FORMAT statement, the name of an array containing format specifications, or a NAMELIST name. When NAMELIST I/O is employed, the PRINT statement may not contain an I/O variable list.

Free-Format PRINT Statement

A free-format PRINT statement is denoted by forms 2 through 5. The free-format PRINT statement functions like the free-format WRITE statement, except that the unit number is not explicitly specified.

PUNCH STATEMENT

The PUNCH statement receives data from the internal storage locations indicated by the I/O variable list or format specifier of the statement. No action specifier is allowed with this statement, and the unit number is not explicitly specified and writes to the CARD PUNCH.

The proper forms of the PUNCH statement are:

PUNCH STATEMENT
1. PUNCH f , m 2. PUNCH / , m 3. PUNCH / / , m 4. PUNCH * / , m 5. PUNCH * / / , m
where f is a format specifier; m is an I/O variable list; / is a free-format indicator; * is a free-format output specifier.

The operation of this statement is identical to the operation of the PRINT statement, except for the unit number. Every PUNCH statement has the implied unit number 7, a card punch file.

I/O VARIABLE LISTS

An input variable list m in an input statement specifies the variables to which values are assigned on input. An output variable list m specifies the variables whose values are transmitted on output. The input and output variable lists are of the same form. An I/O variable list may be a simple I/O variable list, a DO-implied I/O variable list, or a combination of these, each separated by a comma character (,). A simple I/O variable list consists of a variable, an array element, an array name, or any combination of these elements. The following are examples of valid simple I/O variable lists:

```
A,B(3)
XARRAY, ARRAY(2,4)
RESULT
```

A DO-implied I/O variable list is an I/O variable list immediately followed by a comma and a DO-implied specification, all of which are enclosed within a set of parentheses. A DO-implied specification is constructed in the same manner as the DO-control portion of a DO statement. (See section 9.) Thus, the proper format of a DO-implied I/O variable list is:

(m, v=p,q,i)

or

(m, v=p,q)

where m is an I/O variable list, v is the single-precision control variable, and p, q, and i are the single precision variables or constants which are the initial parameter, terminal parameter, and incrementation para-

meter, respectively. The control variable need not appear in the I/O variable list part of this construct. The following are examples of valid DO-implied I/O variable lists:

```
(A(I),B,I,C(I,J),I=1,10,2)
(IPLOT,COUNT=1,60)
(X,(Y(I,J),I=1,10),Z,J=1,2)
```

An I/O variable list may consist of any combination of simple and DO-implied I/O variable lists separated by commas. Such a list may contain grouping parentheses. The following are examples of valid I/O variable list constructs:

```
XARRAY(3), XARRAY, (XARRAY(EL),EL=1,20)
(A(I), I=1,20), (J(I), I=1,5)
(((Z(I,J,K), K=1,10),J=1,10),I=1,10),7,G
```

Transfer of data to and from storage locations occurs in the order these locations are named in the I/O variable lists, from left to right. Items named in a DO-implied I/O variable list are referenced repeatedly until the implied DO is satisfied. The appearance in an I/O variable list of an array element specifies only that array element, and the appearance of an array name specifies every element declared for the array in that program unit. These array elements are transferred in the order in which they are stored in the internal array assigned to the program array. (See discussion of array storage in appendix E.)

An arithmetic expression may appear in the I/O variable list of a WRITE statement, in which case the value of the expression is used to provide a value for output. If the expression invokes a function, that function must not cause execution of an I/O statement which employs the same data file as the file employed by the statement using the I/O variable list which contains the expression. Thus, if this I/O statement is executed:

```
WRITE (4,1) A,F (20,X)
```

then F must not be a function which contains an I/O statement employing unit number 4, since this is the file employed by this WRITE statement.

ACTION SPECIFIERS

An action specifier is defined as one of the following constructs:

ACTION SPECIFIERS
DATA = s END = s ERR = s
where s is an unsigned integer constant corresponding to the label on an executable statement appearing in the same program unit as the I/O statement.

DATA Action Specifier

If the DATA action specifier appears in a READ or a WRITE statement, then control automatically passes to statement s on the following conditions:

a. For a read (formatted or unformatted random) when the I/O variable list requests more data than the logical record contains.

b. For random files or sequential formatted files the logical record size is greater than the declared record size. For sequential unformatted files, the I/O may request more data than the declared record size.

c. When the input data for a formatted read does not meet the requirements of the format specifier.

d. On a formatted write when the type of the variable does not match the format specifier.

e. When the random record key is less than 1.

f. When the format specification (see section 7) exceeds the record size.

If any of the above conditions occur and the DATA action specifier is not specified, program execution is terminated.

END Action Specifier

If the END action specifier appears in a READ or WRITE statement, then control will automatically pass to statement s when an end-of-file condition occurs. Such end-of-file conditions occur when the following actions are attempted by a READ or WRITE statement.

a. Attempting to read a card with an invalid character or question mark (?) in column 1.

b. Attempting to read beyond the last record written on a tape.

c. Attempting to read a record beyond the last record previously written on disk.

d. Reading the EOF mark.

e. Attempting to write beyond the end of the designated number of AREAS specified in the FILE declaration statement of a disk file.

ERR Action Specifier

If the ERR action specifier appears in a READ or WRITE statement, then control will automatically pass to statement s when a parity error occurs during the data transfer. The ERR action specifier will avoid termination of the program at this point, and the next block may be processed if no parity errors occur in it.

Examples:

The following are valid examples of action specifiers:

```
READ(3,END=99)A
WRITE(6=R,35,ERR=70,DATA=80)A
READ(11,85,END=77,ERR=78)J,X,U
WRITE(8'IREC,END=25) ((X(I,J), I=1,10),J=1,10)
```

REWIND STATEMENT

The REWIND statement causes a pointer for the tape or disk file specified by the unit number to be reset to the beginning of the file.

The general form is:

REWIND STATEMENT
REWIND u
where u is a unit number execution of the REWIND statement causes the file referenced by the indicated unit number, u, to be positioned to its first record. The next I/O statement does not cause an implied open.

If the last reference to the file indicated by the unit number, u, is a WRITE statement, the file is closed and an ending label is written (tape only) prior to positioning the file to its initial point.

The REWIND statement is undefined for other than single file tape or disk files.

Examples:

```
REWIND 5
REWIND IUNIT
```

BACKSPACE STATEMENT

The executable BACKSPACE statement is an auxiliary I/O statement which allows a data file to be repositioned before it is accessed by an input or output statement. When a BACKSPACE statement is executed, the file indicated by the unit number is positioned one record before its present position, unless the file is presently positioned at its initial point, in which case no action is initiated.

The proper form of the BACKSPACE statement is:

BACKSPACE STATEMENT
BACKSPACE u
where u is a unit number.

A possible use of the BACKSPACE statement is to allow the last record accessed by an input or output statement to be the next record accessed when such a statement is executed.

CLOSE STATEMENT

The executable CLOSE statement is an auxiliary I/O statement which allows a data file to be released after the program has performed the necessary operations on it. The proper form of the CLOSE statement is:

CLOSE STATEMENT
CLOSE u
where u is a unit number.

The execution of a CLOSE statement causes the file designated by the unit number to be returned to system control. The program may access the file again with a READ or a WRITE statement.

ENDFILE STATEMENT

The executable ENDFILE statement is an auxiliary I/O statement which allows an ENDFILE record to be written and the file to be closed. The ENDFILE record is written on the output file indicated by the unit number at the point at which that file is currently positioned. The proper form of the ENDFILE statement follows:

ENDFILE STATEMENT
ENDFILE u
where u is a unit number.

The ENDFILE statement is intended for use with multi-file tape. When an ENDFILE statement follows a WRITE statement on the file, an ENDFILE record is written and the tape is positioned such that the next record written will follow the ENDFILE record. When an ENDFILE follows a READ of the last record of a file, the tape is positioned to the next file on the tape. If the record was not the last record of the file, the next READ will read the next record of the file. When an ENDFILE follows a REWIND or another ENDFILE designating the same file, the ENDFILE is ignored. When a program employs multi-file tapes, each file on the tape must have been closed with an ENDFILE before the next file may be written on the tape. The tape is not rewound.

LOCK STATEMENT

The executable LOCK statement is an auxiliary I/O statement which allows a data file to be closed and saved after the program has performed the necessary operations on it.

The proper form of the LOCK statement is:

LOCK STATEMENT
LOCK u
where u is a unit number.

The execution of a LOCK statement removes the file indicated by the unit number from the control of the program and saves it.

If the file indicated by the unit number is a tape file, the file is closed and the tape is rewound.

If the file indicated by the unit number is a disk file, the file is removed from program control and is placed in the disk directory. A disk file generated by a FORTRAN program will not remain on disk unless that file is closed with a LOCK statement or LOCK is specified in the FILE declaration statement and the file is not explicitly closed. If the file indicated by the unit number is not a disk file, the file is made inaccessible to the system until it is manually reset. This applies to devices (such as tape) accessed directly by the program but not to devices (such as line printers and card readers) normally accessed with backup files.

The following are examples of valid LOCK statements:

```
LOCK TAPENO – 1
LOCK 22
LOCK I+J
```

PURGE STATEMENT

The executable PURGE statement is an auxiliary I/O statement which allows a data file to be closed, purged from the system, and its space to be released for other use.

PURGE STATEMENT
PURGE u
where u is a unit number.

The following are examples of valid PURGE statements:

```
PURGE 10
PURGE FILENO
PURGE 9 + KFILE
```

A purged tape file may be reused. Purging a card reader, card punch, line printer, or remote file simply closes the file.

CHANGE STATEMENT

The CHANGE statement is used to change the file name of a file on a multi-file tape. The proper syntax for the CHANGE statement is:

CHANGE STATEMENT
CALL CHANGE (unit,n)
where unit is an arithmetic expression designating a unit number, n may be either an integer array name of at least eight integer words or a string literal of thirty characters.

The format for the array contents or the literal must follow the following rules:

a. The first ten characters represent the pack-id (meaningless for tape, but still required). These ten characters must be blank.

b. The second ten characters represent the family-name. This name must be the same for each file going to the same tape.

c. The third ten characters represent the file-name.

Restrictions:

a. The slash character is not allowed.

b. If the array is dimensioned less than eight integer words, an INVALID SUBSCRIPT error will occur during execution.

c. There is no syntax for changing the record or block size.

MULTI-FILE TAPE HANDLING

The ENDFILE statement is used to close all but the final file on a multi-file tape. A CLOSE on the final file causes the tape to be rewound after the ENDFILE record is written. The following statements cause a close on the file: LOCK, REWIND, CLOSE, and ENDFILE. If a statement which causes a close is executed, then any statement following it which causes a close will be ignored unless an I/O which causes an implied open has occurred in the interim.

If an ENDFILE statement is followed by a REWIND statement, and both specify the same unit number for tape, there is no error and the tape is not rewound. In this case a CLOSE statement can be executed to write an ENDFILE record, CLOSE and rewind the tape.

The following sequence of events is used to create a multi-file tape:

a. WRITE each record of the first file.

b. Write an ENDFILE record.

c. CHANGE the file name.

d. WRITE each record of the current file.

e. Repeat evens b through d as desired.

f. CLOSE the file.

The following sequence of events is used to read a multi-file tape:

a. CHANGE the file name to the name of the first file desired.

b. READ each record in the first file.

c. Read the ENDFILE record.

d. CHANGE the file to the next file desired.

e. READ each record in the current file.

f. Repeat events c through e as desired.

g. CLOSE the file.

In event d, the CHANGE statement may not specify a file name that occurs before the last file read. In order to read a previous file, use a CLOSE instead of an ENDFILE statement. The CHANGE statement following a CLOSE statement may specify any file on the tape. The next READ operation reads the first record of that file.

After reading the records from the file, the ENDFILE statement is used only after an END branch has been taken. Otherwise, the tape will be in the wrong position for a subsequent READ operation.

Exceptions:

a. The ENDFILE statement can occur before the first record of a file has been read with no effect.

b. The ENDFILE statement can occur after all records have been read but before the END branch has been taken. The result is the same as if the END branch had been taken.

A READ of the file without changing the name to a following file will cause an I/O error as a result of reading beyond the end of the file. A WRITE without changing the file name will cause a new file of the same name to be written.

If a CHANGE statement is executed followed by a READ, the READ will cause the rest of the tape to be searched for the new file name and locked if the file is not found. A WRITE will cause a new file to be written.

ZIP STATEMENT

The proper syntax for the ZIP statement is:

ZIP STATEMENT
CALL ZIP(s)
where s is a quoted string, a Hollerith literal or an integer array name; it may be any valid control string.

The first character of the control string may be a "?". If an array name is specified, it must be dimensioned with 128 or fewer elements.

The maximum number of characters allowed in the ZIP statement is 512.

12. SUBPROGRAMS, INTRINSIC FUNCTIONS, AND INTRINSICS

Subprograms are program units which may be invoked in the main program as a separate executable procedure, or which may be used for data initialization.

SUBROUTINE SUBPROGRAM

A subroutine is a sequence of statements initiated by a SUBROUTINE statement and terminated by an END statement.

The nonexecutable SUBROUTINE statement indicates the beginning of a subroutine subprogram and may specify the dummy arguments employed in that subprogram.

The proper forms of the SUBROUTINE statement are as follows:

SUBROUTINE STATEMENT
SUBROUTINE s or SUBROUTINE s (d)
where s is a SUBROUTINE name, and d is a list of dummy arguments separated by single commas.

SUBROUTINE Names

A SUBROUTINE name is constructed in the same manner as a variable name. No type is associated with this name, and once a symbolic name is used as a SUBROUTINE name it may not be used for any other purpose in that program.

Dummy Argument Lists

Each element of a dummy argument list may be a simple variable name, an array name, or an asterisk (*). If a dummy argument is a simple variable name, the actual argument in the subroutine CALL must be a variable name, an array element, or an expression. The actual argument must be of the same type as the dummy argument.

If an array appears as a dummy argument, the corresponding actual argument may be an array or array element. In the latter case, the actual argument may be thought of as an array whose first element is that array element. Array handling between actual and dummy argument lists is discussed in the section on the CALL statement in section 9. When a dummy argument is an array, it may have adjustable dimensions in the declaration in the subprogram, but such dimensions may only be variables appearing in the dummy argument list or as an element of a COMMON block. Thus, the following two statements reflect proper usage of these constructs:

```
SUBROUTINE SUBS(A,N)
REAL A(N,6)
```

Care should be taken that the dummy array is not dimensioned larger than the corresponding actual array, since this may result in an invalid subscript condition when the dummy array is referenced.

An asterisk in a dummy argument list reserves that position in the list for a statement label preceded by an ampersand (&) in the actual argument list; this argument is used in connection with nonstandard RETURN statements. (See section 9.)

Any slashes appearing in the dummy argument list will be ignored. A simple variable, array or array element will be referenced by name, rather than value, unless specified otherwise in the CALL statement by the use of parentheses or a unary + or –. (See section 9.)

Use of Subroutines

The executable statements in a subroutine body are executed when the subroutine is invoked by a CALL statement. The subroutine may cause program termination by executing a STOP statement or may cause control to subsequently return to the calling program unit by means of the RETURN statement.

The following are valid examples of SUBROUTINE statements:

```
SUBROUTINE SUBA
SUBROUTINE MULTMT(ARRAY1, ARRAY2, *)
SUBROUTINE MAX(A,B,C,D,E,*,F)
```

Direct recursion is prohibited. Indirect recursion is allowed, but only parameters passed by value and partially complete expressions will be unique upon re-entering the subroutine.

FUNCTION SUBPROGRAMS AND STATEMENT FUNCTIONS

FORTRAN FUNCTIONS are procedures which return a single value to a calling program unit at the point at which they are referenced. A function may consist of a program unit similar to a SUBROUTINE subprogram, called a FUNCTION subprogram, or may be declared as a single expression using a statement function declaration appearing in the program unit in which it is referenced.

Function Subprogram

A function subprogram is a sequence of statements initiated by a FUNCTION statement and terminated by an END statement.

FUNCTION Statement

The nonexecutable FUNCTION statement indicates the beginning of a function subprogram and specifies the dummy arguments employed in that subprogram to obtain the function value.

The proper forms of the FUNCTION statement are as follows:

FUNCTION STATEMENT
FUNCTION f(d) or t FUNCTION f(d)
where f is a FUNCTION name, d is a list of dummy arguments separated by single commas, and t is a type as defined in section 5.

FUNCTION Name

A FUNCTION name is constructed in the same manner as a variable name. A type is associated with this symbolic name which indicates the type of the value returned when the function is referenced. Default types are assigned to FUNCTION names in the same manner as variable and array names. FUNCTION names beginning with the letters A through H, and O through Z and $ are typed REAL, by default and FUNCTION names beginning with the letters I through N are typed INTEGER, by default.

The default type associated with a FUNCTION name may be altered using the item t in the FUNCTION statement. For example, the function COMPO will be of type LOGICAL if it is initiated by a FUNCTION statement of the form:

```
LOGICAL FUNCTION COMPO (A)
```

When a function is of a non-default type, then the FUNCTION name must appear in an appropriate type statement in every program unit in which that function is referenced. For example, a program unit referencing the preceding function must contain a statement of the sort:

```
LOGICAL COMPO
```

Once a symbolic name has been used as a FUNCTION name, it may not be used for any other purpose in that program. Such a name may appear on the left side of an equal sign in an assignment statement only in the body of the function subprogram. The value to be returned must be assigned to the FUNCTION name in this manner before a RETURN is executed.

Dummy Argument Lists

The list of dummy arguments in a FUNCTION statement is defined in exactly the same manner as the same portion of a SUBROUTINE statement, except nonstandard returns are not allowed. Every FUNCTION statement must contain at least one dummy argument.

The following is an example of a valid FUNCTION subprogram:

```
      FUNCTION MINVAL(A,K)
      REAL MINVAL
      LOGICAL NOMIN
      DIMENSION A(K)
      COMMON NOMIN
      MINVAL = A(1)
      DO 20 I=2,K
      IF(A(I)-MINVAL) 10,20,20
10    MINVAL = A(I)
      NOMIN = .TRUE.
20    CONTINUE
      RETURN
      END
```

Statement Function

A statement function is a function which may 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.

Statement Function Declaration

A statement function declaration is a nonexecutable statement of the form:

STATEMENT FUNCTION DECLARATION
f(d)=e
where f is a FUNCTION name, d is a list of dummy arguments separated by single commas, and e is an expression.

A statement function is local to the program unit in which the statement function declaration appears. It is referenced in that program unit exactly like any other function and may not be referenced by any other program unit.

A statement function must precede the first executable statement in the program unit and follow any specification statements.

Function Type

The type of the value returned when a statement function is referenced depends upon the type associated with the FUNCTION name. A default type is associated with this name in the same manner as a variable name. This default type may be changed by the appearance of the FUNCTION name in a type statement in the same program unit.

For example, the following is a valid combination of statements:

```
REAL MAX
LOGICAL FNAME,FLAG,VALID
FNAME(SUM,MAX) = SUM.GT.MAX
.
FLAG=VALID.AND.FNAME(TOTAL,BUDGET)
.
.
.
END
```

Dummy Argument List

The list of dummy arguments in a statement function may contain only variable names.

Use of Functions

A function is executed when a function reference is encountered in an expression. The value obtained from the function execution is returned to the point of the function reference.

A function reference has the form: f(a), where f is the FUNCTION name and a is a list of actual arguments.

Execution of the function begins with the first executable statement following the FUNCTION statement.

The actual argument list of a FUNCTION subprogram may contain the same items as the actual argument list of a CALL statement, except for labels. When a statement function is referenced, the actual argument list may contain only expressions.

If a function reference occurs in the output list of a WRITE statement, the execution of that function may not cause any WRITE statement to be executed.

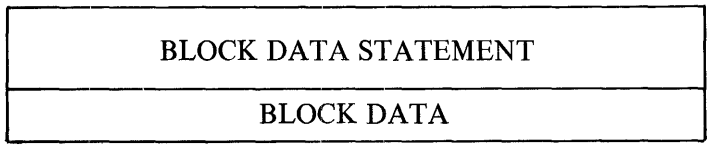
Recursion is not permitted in FUNCTION subprograms or statement functions.

Array handling in FUNCTION subprograms is performed in the same manner as for subroutines. Slashes in the dummy argument list will be ignored.

BLOCK DATA SUBPROGRAM

The BLOCK DATA subprogram is a nonexecutable program unit which has as its first statement a BLOCK DATA statement. The BLOCK DATA subprogram may not contain any executable statements and is used to initialize elements in labeled and unlabeled COMMON to predetermined values. Only one BLOCK DATA subprogram is allowed per program.

The proper form of the BLOCK DATA statement is as follows:



Note that the BLOCK DATA subprogram is only a means whereby elements in COMMON may be initialized at compile time. These elements, of course, may be reassigned a value at any time during the execution of the program.

The BLOCK DATA subprogram is a nonexecutable program unit whose initial statement is a BLOCK DATA statement. The statements following the BLOCK DATA statement define the elements in COMMON to be initialized and only the explicit type, DIMENSION, COMMON, EQUIVALENCE, and DATA declaration statements may be used. An END statement must physically be the last statement in the subprogram.

The construction of BLOCK DATA subprograms is subject to the following restrictions:

a. A BLOCK DATA subprogram must contain at least one COMMON statement.

b. All elements of a COMMON block up to and including the last element to be initialized must appear in the COMMON statement list even though some of those elements are not to be initialized.

c. All type, dimension, or equivalence information associated with the variables or arrays in a COMMON block must be declared in the BLOCK DATA subprogram.

d. There can be only one BLOCK DATA subprogram in a program.

e. More than one COMMON block may be initialized by a BLOCK DATA subprogram.

Example:

```
BLOCK DATA
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 the above example, elements in the common blocks labeled BLOC1 and BLOC2 are to be initialized; therefore, all the elements in these blocks up to and including the last to be initialized are listed in a COMMON statement. This is permissible, as more than one COMMON block may 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.

INTRINSIC FUNCTIONS

Intrinsic functions may be considered to be function subprograms which are known to the compiler and need not be supplied in the program.

Table 12-1 denotes the intrinsic function names recognized by the B 1800/B 1700 FORTRAN compiler. The type of the value returned by the intrinsic function reference is indicated as are the number and types of actual arguments passed to the intrinsic function. A brief description and definition of the function performed by each intrinsic function is also given. In such descriptions, A1 and A2 represent the first and second arguments passed to the intrinsic function, respectively. Multiple arguments must always be separated by single commas; such arguments must agree in number and type with the specification.

Intrinsic function names whose types do not agree with the default types which would be assigned to their names need not be named in type statements in the program units in which they are referenced. Intrinsic functions differ from other functions in this respect.

Most intrinsic functions may be redefined by the user in a function subprogram. The function subprogram may be called by any name, including the name of the function it is replacing. The user's function subprogram will not be entered in the system's intrinsic function file (FOR.INTRIN file) and must be compiled with the calling program unit each time its use is desired. Some intrinsic functions may not be redefined by the user with the same name as the intrinsic function.

Table 12-1 contains a list of intrinsic functions including their definitions, argument types, and function performed. Those that cannot be replaced are marked with a single asterisk (*).

Table 12-1. B 1800/B 1700 FORTRAN Intrinsic Functions

Intrinsic Function Name	Intrinsic Function Type	Argument Type	Number of Arguments	Function Performed	Definition
ABS*	Real	Real	1	Absolute value	|A1|
ACOS**	Real	Real	1	Arc cosine	ARCCOS (A1)
AIMAG*	Real	Complex	1	Obtain imaginary part	
AINT*	Real	Real	1	Truncation	Largest integer $\leq$ |A1|
ALGAMA	Real	Real	1	Log Gamma function	LN Gamma (A1)
ALOG	Real	Real	1	Natural logarithm	LN (A1)
ALOG10	Real	Real	1	Common logarithm	LOG (A1)
AMAXO*	Real	Integer	≥ 2	Choosing largest value	MAX(A1,A2,...)
AMAX1*	Real	Real	≥ 2	Choosing largest value	MAX(A1,A2,...)
AMINO*	Real	Integer	≥ 2	Choosing smallest value	MIN(A1,A2,...)
AMIN1*	Real	Real	≥ 2	Choosing smallest value	MIN(A1,A2,...)
AMOD*	Real	Real	2	Remaindering	A1(MOD A2)
AND*	Real	Real	2	36-Bit logical product	AND(A1,A2)
ARCOS	Real	Real	1	Arc cosine	ARCCOS(A1)
ARSIN	Real	Real	1	Arc sine	ARCSIN(A1)
ASIN**	Real	Real	1	Arc sine	ARCSIN(A1)
ATAN	Real	Real	1	Arctangent	ARCTAN(A1)
ATAN2	Real	Real	2	Arctangent	ARCTAN(A1/A2)
CABS*	Real	Complex	1	Absolute value	|A1|
CCOS	Complex	Complex	1	Trigonometric cosine	COS(A1)
CEXP	Complex	Complex	1	Exponential	e**A1
CLOG	Complex	Complex	1	Natural logarithm	LN(A1)
CMPLX*	Complex	Real	2	Create complex	(A1,A2)
CONJG*	Complex	Complex	1	Complex conjugate	
COS	Real	Real	1	Trigonometric cosine	COS(A1)
COSH	Real	Real	1	Hyperbolic cosine	COSH(A1)
COTAN	Real	Real	1	Trigonometric cotangent	COT(A1)

Table 12-1. B 1800/B 1700 FORTRAN Intrinsic Functions (Cont)

Intrinsic Function Name	Intrinsic Function Type	Argument Type	Number of Arguments	Function Performed	Definition
CSIN	Complex	Complex	1	Trigonometric sine	SIN (A1)
CSQRT	Complex	Complex	1	Square root	$\sqrt{A1}$
DABS*	Double	Double	1	Absolute value	|A1|
DARCOS	Double	Double	1	Arc cosine	ARCCOS (A1)
DARSIN	Double	Double	1	Arc sine	ARCSIN (A1)
DATAN	Double	Double	1	Arctangent	ARCTAN(A1)
DATAN2	Double	Double	2	Arctangent	ARCTAN(A1/A2)
DBLE*	Double	Real	1	Express single precision argument in double form	
DCOS	Double	Double	1	Trigonometric cosine	COS(A1)
DCOSH	Double	Double	1	Hyperbolic cosine	COSH(A1)
DCOTAN	Double	Double	1	Trigonometric cotangent	COT(A1)
DDIM*	Double	Double	2	Positive difference	A1-MIN(A1,A2)
DERF	Double	Double	1	Error function	Error Function(A1)
DERFC	Double	Double	1	Error function	1-Error Function(A1)
DEXP	Double	Double	1	Exponential	e**(A1)
DEFLOAT*	Double	Integer	1	Type of conversion	Conversion from integer to double precision
DGAMMA	Double	Double	1	Function	Gamma(A1)
DIM*	Real	Real	2	Positive difference	A1-MIN(A1,A2)
DINT*	Double	Double	1	Truncation	Largest integer $\leq$ |A1|
DLGAMA	Double	Double	1	Log Gamma function	LN Gamma(A1)
DLOG	Double	Double	1	Natural logarithm	LN(A1)
DLOG10	Double	Double	1	Common logarithm	LOG(A1)
DMAX1*	Double	Double	≥ 2	Choosing largest value	MAX(A1,A2,...)
DMIN1*	Double	Double	≥ 2	Choosing smallest value	MIN(A1,A2,...)
DMOD*	Double	Double	2	Remaindering	A1(MOD A2)
DSIGN*	Double	Double	2	Transfer of sign	(Sign of A2)* |A1|

Table 12-1. B 1800/B 1700 FORTRAN Intrinsic Functions (Cont)

Intrinsic Function Name	Intrinsic Function Type	Argument Type	Number of Arguments	Function Performed	Definition
DSIN	Double	Double	1	Trigonometric sine	SIN(A1)
DSINH	Double	Double	1	Hyperbolic sine	SINH(A1)
DSQRT	Double	Double	1	Square root	$\sqrt{A1}$
DTAN	Double	Double	1	Trigonometric tangent	TAN(A1)
DTANH	Double	Double	1	Hyperbolic tangent	TANH(A1)
ERF	Real	Real	1	Error function	Error Function(A1)
ERFC	Real	Real	1	Error function	1-Error Function(A1)
EXP	Real	Real	1	Exponential	e**A1
FLOAT*	Real	Integer	1	Type of conversion	Conversion from integer to real
GAMMA	Real	Real	1	Gamma function	Gamma(A1)
HFIX*	Integer	Real	1	Type conversion	Conversion from real to integer
IABS*	Integer	Integer	1	Absolute value	|A1|
IDIM*	Integer	Integer	2	Positive difference	A1-MIN(A1,A2)
IDINT*	Integer	Double	1	Truncation	Largest integer $\leq$ |A1|
IFIX*	Integer	Real	1	Type conversion	Conversion from real to integer
INT*	Integer	Real	1	Truncation	Largest integer $\leq$ |A1|
ISIGN*	Integer	Integer	2	Transfer of sign	(Sign of A2) * |A1|
LGAMMA**	Real	Real	1	Log Gamma function	LN Gamma(A1)
LOG**	Real	Real	1	Natural logarithm	LN(A1)
LOG10**	Real	Real	1	Common logarithm	LOG(A1)
MAX*,**	Integer	Integer	≥ 2	Choosing largest value	MAX(A1,A2,...)
MAX0*	Integer	Integer	≥ 2	Choosing largest value	MAX(A1,A2,...)
MAX1*	Integer	Real	≥ 2	Choosing largest value	MAX(A1,A2,...)
MIN*,**	Integer	Integer	≥ 2	Choosing smallest value	MIN(A1,A2,...)
MIN0*	Integer	Integer	≥ 2	Choosing smallest value	MIN(A1,A2,...)

Table 12-1. B 1800/B 1700 FORTRAN Intrinsic Functions (Cont)

Intrinsic Function Name	Intrinsic Function Type	Argument Type	Number of Arguments	Function Performed	Definition
MIN1*	Integer	Real	≥ 2	Choosing smallest value	MIN(A1,A2,...)
MOD*	Integer	Integer	2	Remaindering	A1(MOD A2)
OR*	Real	Real	2	36-bit logical sum	OR(A1,A2)
RANDOM	Real	Integer	1	Random number generator. Argument is updated by the intrinsic function.	
REAL*	Real	Complex	1	Obtain real part	
SIGN*	Real	Real	2	Transfer of sign	(Sign of A2) * |A1|
SIN	Real	Real	1	Trigonometric sine	SIN(A1)
SINH	Real	Real	1	Hyperbolic sine	SINH(A1)
SNGL*	Real	Double	1	Obtain most significant part of double precision argument	
SQRT	Real	Real	1	Square root	$\sqrt{A1}$
TAN	Real	Real	1	Trigonometric tangent	TAN(A1)
TANH	Real	Real	1	Hyperbolic tangent	TANH(A1)
TIME***	Integer	Integer	1	Time function	MCP

* These intrinsic functions cannot be redefined.

** The intrinsic functions MAX, MIN, LOG, LOG10, ASIN, ACOS, and LGAMMA are identical to the intrinsic functions MAX0, MIN0, ALOG, ALOG10, ARSIN, ARCOS, and ALGAMMA, respectively.

*** The argument of the TIME function must be the integer value 1, 2, 3, or 4 where:

1 = time in 1/10 second
2 = time as HHmm
3 = Julian date YYDDD
4 = date as YYMMDD

Table 12-2 contains a list of argument restrictions for the intrinsic functions.

Table 12-2. B 1800/B 1700 FORTRAN Intrinsic Function Restrictions

Intrinsic Function		Restrictions
ACOS	(A1)	-1. < = A1 < = 1.
ALGAMA	(A1)	0 < A1 < = 1000) A1 NOT = 0 AINT (A1) NOT) FOR A1 < 0 AMOD(AINT(A1),2) NOT = 0 FOR A1 < 0
ALOG	(A1)	A1 > 0
ALOG10	(A1)	A1 > 0
ARCOS	(A1)	-1, < = A1 < = 1.
ARSIN	(A1)	=1. < = A1 < = 1.
ASIN	(A1)	-1. < = A1 < + 1.
ATAN	(A1)	No restriction
ATAN2	(A1,A2)	NOT A1 = A2 = 0
CABS	(A1)	No restriction
CCOS	(A1)	-176 < IMAGINARY PART < 176
CEXP	(A1)	-177 <REAL PART <176
CLOG	(A1)	NOT REAL=IMAGINARY=0, NOT REAL = 0
COS	(A1)	ABS(A1+1.5707963) < = 2**24
COSH	(A1)	-177 < A1 < 176
COTAN	(A1)	AMOD(ABS(A1).3.141593) > 1.E-06 ABS(1:5707963-A1) < = 2**24
CSIN	(A1)	-176 < IMAGINARY PART < 176
CSQRT	(A1)	ABS(REAL PART) < = 0
DARCOS	(A1)	-1. < A1 < = 1.
DARSIN	(A1)	-1. < = A1 < = 1.
DATAN	(A1)	No restriction
DATAN2	(A1,A2)	NOT A1 = A2 = 0
DCOS	(A1)	ABS(A1+1.57079632679489662) < = 2**60
DCOSH	(A1)	-177 < A1 < 176

Table 12-2. B 1800/B 1700 FORTRAN Intrinsic Function Restrictions (Cont)

Intrinsic Function		Restrictions
DCOTAN	(A1)	DMOD(ABS(A1),3.14159265358979324D+00) = 1.D-06 ABS(1.5707963267948966 - A1) < = 2**60
DERF	(A1)	No restriction
DERFC	(A1)	No restriction
DEXP	(A1)	-177 < A1 < 176
DGAMMA	(A1)	-17 < A1 < = 58, A1 NOT = 0 DINT(A1) NOT = A1 FOR A1 < 0
DINT	(A1)	No restriction
DLGAMA	(A1)	-17 < A1 < 1000, A1 NOT = 0 DINT(A1) NOT = A1 FOR A1 < 0 DMOD(DINT(A1),2) NOT = 0 FOR A1 < 0
DLOG	(A1)	A1 > 0
DLOG10	(A1)	A1 > 0
DMOD	(A1,A2)	A1 NOT = 0
DSIN	(A1)	ABS(A1) < = 2**60
DSINH	(A1)	-177 < A1 < 176
DSORT	(A1)	A1 < = 0
DTAN	(A1)	ABS(A1) < = 2**60
DTANH	(A1)	No restriction
ERF	(A1)	No restriction
ERFC	(A1)	No restriction
EXP	(A1)	-177 < A1 < 176
GAMMA	(A1)	-17 < A1 < 58, A1 NOT = 0 AINT(A1) NOT = A1 FOR A1 < 0
LGAMMA	(A1)	-17 < A1 < 1000, A1 NOT = 0 AINT(A1) NOT = A1 FOR A1 < 0 AMOD(AINT(A1),2) NOT = 0 FOR A1 < 0
LOG	(A1)	A1 > 0
LOG10	(A1)	A1 < 0
SIN	(A1)	ABS(A1) < 2**24
SINH	(A1)	-177 < A1 < 176

Table 12-2. B 1800/B 1700 FORTRAN Intrinsic Function Restrictions (Cont)

Intrinsic Function		Restrictions
SORT	(A1)	A1 < = 0
TAN	(A1)	ABS(A1) < = 2**24
TANH	(A1)	No restriction

INTRINSICS

Table 12-3 contains a list of intrinsics, and includes a brief explanation of each intrinsic. Intrinsics are not directly accessible by the FORTRAN program, but are bound in to perform I/O, formatting, and other implicit functions of the FORTRAN program.

Table 12-3. List of Intrinsics

Intrinsic	Explanation of Responsibility
.ARFMT	ARRAY FORMAT
.AUXIO	AUXILIARY INPUT/OUTPUT STATEMENTS-POSITIONS AND CLOSES
.CPWRD	COMPLEX RAISED TO DOUBLE-PRECISION POWER
.CPWRI	COMPLEX RAISED TO AN INTEGER POWER
.CPWRR	COMPLEX RAISED TO A REAL POWER
.CXDIV	COMPLEX DIVIDE
.CXMUL	COMPLEX MULTIPLY
.DERR	DOUBLE-PRECISION DIVIDE BY ZERO, EXPONENT OVERFLOW AND UNDERFLOW
.DPWRD	DOUBLE-PRECISION BASE RAISED TO A DOUBLE-PRECISION EXPONENT
.DPWRI	DOUBLE-PRECISION BASE RAISED TO AN INTEGER EXPONENT
.DPWRR	DOUBLE-PRECISION BASE RAISED TO A REAL EXPONENT
.ERROR	DUMPS MEMORY, PRINTS AN ERROR MESSAGE AND STOPS
.FIXEC	SERVICE ROUTINE FOR I/O SYSTEM
.FSLH	PROCESSES A "/" FIELD SEPARATOR IN A FORMAT STATEMENT
.FT	PROCESSES A "T" FIELD DESCRIPTOR IN A FORMAT STATEMENT
.FTMAK	CALLED AT BOJ FOR EACH FILE ASSOCIATED WITH THE PROGRAM
.IERR	INTEGER DIVIDE BY ZERO
.IPWRD	INTEGER BASE RAISED TO A DOUBLE-PRECISION EXPONENT
.IPWRI	INTEGER BASE RAISED TO AN INTEGER EXPONENT
.IPWRR	INTEGER BASE RAISED TO A REAL EXPONENT
.IRFR	INITIALIZES RANDOM FORMATTED READS
.IRFW	INITIALIZES RANDOM FORMATTED WRITES
.IRUR	INITIALIZES RANDOM UNFORMATTED READS
.IRUW	INITIALIZES RANDOM UNFORMATTED WRITES
.ISFR	INITIALIZES SERIAL FORMATTED READS
.ISFW	INITIALIZES SERIAL FORMATTED WRITES
.ISNR	INITIALIZES SERIAL NAMELIST READ

Table 12-3. List of Intrinsics (Cont)

Intrinsic	Explanation of Responsibility
.ISNW	INITIALIZES SERIAL NAMELIST WRITE
.ISSR	INITIALIZES SERIAL FREE-FORMAT READ
.ISSW	INITIALIZES SERIAL FREE-FORMAT WRITE
.ISUR	INITIALIZES SERIAL UNFORMATTED READS
.ISUW	INITIALIZES SERIAL UNFORMATTED WRITES
.MTSPE	GETS THE NEXT FORMAT CONVERSION SPECIFIER
.NERR	NAMELIST error routine
.NLEWR	NAMELIST ERROR MESSAGE WRITE ROUTINE
.NLNWR	NAMELIST NAME PLACED IN THE BUFFER
.NNAME	GETS NAME FROM NAMELIST - ALSO "& < name > " AND "&END"
.NNEXT	NAMELIST - CHECKS END OF BUFFER AND READS NEXT RECORD
.NVAL	GETS A NAMELIST VALUE
.NWRIT	WRITES NAMELISTS - CAUSES MCP COMMUNICATE
.PAUSE	HANDLES "PAUSE" and "STOP" STATEMENTS
.RAAU	READS ANY UNFORMATTED ARRAY
.RASU	READS ANY UNFORMATTED SCALAR
.RCAA	READ COMPLEX ARRAY WITH ARRAY FORMAT
.RCAF	READS FORMATTED COMPLEX ARRAY
.RCAN	READS COMPLEX NAMELIST ARRAY
.RCAS	READS COMPLEX FREE-FORMAT ARRAY
.RCFMT	FORMAT CONVERSION FOR READ
.RCSA	READ COMPLEX SCALAR WITH ARRAY FORMAT
.RCSF	READS A COMPLEX FORMATTED SCALAR
.RCSS	READS COMPLEX FREE-FORMAT SCALAR
.PCSU	READS COMPLEX UNFORMATTED SCALAR
.RDAA	READ DOUBLE ARRAY WITH ARRAY FORMAT
.RDAF	READS DOUBLE-PRECISION FORMATTED ARRAY
.RDAN	READS DOUBLE-PRECISION WORD NAMELIST ARRAY
.RDAS	READS DOUBLE-PRECISION FREE-FORMAT ARRAY
.RDAU	READ DOUBLE ARRAY UNFORMATTED
.RDSA	READ DOUBLE SCALAR WITH ARRAY FORMAT
.RDSF	READS DOUBLE-PRECISION FORMATTED SCALAR
.RDSN	READS DOUBLE-PRECISION WORD NAMELIST SCALAR
.RDSS	READS DOUBLE-PRECISION FREE-FORMAT SCALAR
.RDSU	READ DOUBLE SCALAR UNFORMATTED
.READD	INPUT CONVERT DOUBLE
.READR	INPUT CONVERT SINGLE
.RERR	REAL DIVIDE BY ZERO, EXPONENT OVERFLOW AND UNDERFLOW
.RIAA	READ INTEGER ARRAY WITH ARRAY FORMAT
.RIAF	READS AN INTEGER FORMATTED ARRAY

Table 12-3., List of Intrinsics (Cont)

Intrinsic	Explanation of Responsibility
.RIAN	NAMELIST - READS INTEGER ARRAY
.RIAS	READS INTEGER FREE-FORMAT ARRAY
.RISA	READ INTEGER SCALAR WITH ARRAY FORMAT
.RISF	READS AN INTEGER FORMATTED SCALAR
.RISN	NAMELIST - READS INTEGER SCALAR
.RISS	READS INTEGER FREE-FORMAT SCALAR
.RLAA	READ LOGICAL ARRAY WITH ARRAY FORMAT
.RLAF	READS A LOGICAL FORMATTED ARRAY
.RLAS	READS LOGICAL FREE-FORMAT ARRAY
.RLSA	READ LOGICAL SCALAR WITH ARRAY FORMAT
.RLSF	READS A LOGICAL FORMATTED SCALAR
.RLSS	READS LOGICAL FREE-FORMAT SCALAR
.RPWRD	REAL BASE RAISED TO A DOUBLE-PRECISION EXPONENT
.RPWRI	REAL BASE RAISED TO AN INTEGER EXPONENT
.RPWRR	REAL BASE RAISED TO A REAL EXPONENT
.RPAA	READ REAL ARRAY WITH ARRAY FORMAT
.RRAF	READS A REAL FORMATTED ARRAY
.RRAN	READS REAL NAMELIST ARRAY
.RRAS	READS REAL FREE-FORMAT ARRAY
.RRSA	READ REAL SCALAR WITH ARRAY FORMAT
.RRSF	READS A REAL FORMATTED SCALAR
.RRSN	READS REAL NAMELIST SCALAR
.RRSS	READS REAL FREE-FORMAT SCALAR
.RSAU	READ SINGLE ARRAY UNFORMATTED
.RSSU	READ SINGLE SCALAR UNFORMATTED
.RZ	READS A HEXADECIMAL DATA ITEM
.SADDR	RETURNS THE ADDRESS OF A SCALAR
.STERR	RECORDS DATA, PARITY OR END OF FILE
.STKDP	PRINTS STACK RETURN CONTROL WORD (RCW) ADDRESSES
.TEN	SINGLE-PRECISION POWERS OF TEN
.TEND	DOUBLE POWERS OF TEN (.TEN NOW SINGLE)
.TFR	TERMINATES FORMATTED READS
.TFRL	TERMINATES FORMATTED READS WITH ACTION LABELS
.TEW	TERMINATE FORMATTED WRITE
.TFWL	TERMINATE FORMATTED WRITE WITH LABEL
.TRAC	GENERATE AND PRINT FORTRAN LEVEL TRACE LINE
.TRACE	SETS AND RESETS NORMAL STATE FLAGS
.TRAVC	SETS UP COMPLEX VALUE FOR TRACE
.TRN	TERMINATES READ NAMELIST
.TRAN	SAVE IDENTIFIER NAME FOR TRACE

Table 12-3. List of Intrinsics (Cont)

Intrinsic	Explanation of Responsibility
.TRNL	TERMINATES READ NAMELIST LABEL
.TRVAD	SET UP DOUBLE-PRECISION VALUE FOR TRACE
.TRVAJ	SET UP INTEGER VALUE FOR TRACE
.TRVAL	SAVE LOGICAL VALUE FOR TRACE
.TRVAR	SET UP REAL VALUE FOR TRACE
.TUR	TERMINATES UNFORMATTED READS
.TURL	TERMINATES UNFORMATTED READS WITH ACTION LABELS
.TUW	TERMINATE UNFORMATTED WRITE
.TUWL	TERMINATE UNFORMATTED WRITE WITH LABEL
.TW	TERMINATES A WRITE
.TWL	TERMINATES A WRITE ACTION LABELS
.TWN	TERMINATES WRITE NAMELIST
.TWNL	TERMINATES WRITE NAMELIST LABEL
.UNTOK	RETURNS FALSE IF UNIT IS NOT ASSOCIATED WITH AN FPB
.WAAU	WRITES ANY UNFORMATTED ARRAY
.WASU	WRITES ANY UNFORMATTED SCALAR
.WCAA	WRITE COMPLEX ARRAY WITH ARRAY FORMAT
.WCAF	WRITES COMPLEX FORMATTED ARRAY
.WCAN	WRITES COMPLEX NAMELIST ARRAY
.WCAS	WRITES COMPLEX FREE-FORMAT ARRAY
.WCFMT	FORMAT CONVERSION FOR WRITE
.WCSA	WRITE COMPLEX SCALAR WITH ARRAY FORMAT
.WCSF	WRITES COMPLEX FORMATTED SCALAR
.WCSN	WRITES COMPLEX NAMELIST SCALAR
.WCSS	WRITES COMPLEX FREE-FORMAT SCALAR
.WCSU	WRITES COMPLEX UNFORMATTED SCALARS
.WD	WRITES VALUE IN THE "D" FORMAT
.WDAA	WRITE DOUBLE ARRAY WITH ARRAY FORMAT
.WDAF	WRITES DOUBLE-PRECISION FORMATTED ARRAY
.WDAN	WRITES DOUBLE-PRECISION WORD NAMELIST ARRAY
.WDAS	WRITES DOUBLE-PRECISION FREE-FORMAT ARRAY
.WDAU	WRITE DOUBLE ARRAY UNFORMATTED
.WDSA	WRITE DOUBLE SCALAR WITH ARRAY FORMAT
.WDSF	WRITES DOUBLE-PRECISION FORMATTED SCALAR
.WDSN	WRITES DOUBLE-PRECISION NAMELIST SCALAR
.WDSS	WRITES DOUBLE-PRECISION FREE-FORMAT SCALAR
.WDSU	WRITE DOUBLE SCALAR UNFORMATTED
.WE	WRITES VALUE IN THE "E" FORMAT
.WED	WRITES DOUBLE-PRECISION VALUE IN "E" FORMAT
.WF	WRITES VALUE IN THE "F" FORMAT

Table 12-3. List of Intrinsics (Cont)

Intrinsic	Explanation of Responsibility
.WFD	WRITES DOUBLE-PRECISION VALUE IN "F" FORMAT
.WHAS	WRITES HOLLERITH FREE-FORMAT ARRAY
.WHSS	WRITES HOLLERITH FREE-FORMAT SCALAR
.WI	WRITES VALUE IN THE "I" FORMAT
.WIAA	WRITES INTEGER ARRAY WITH ARRAY FORMAT
.WIAF	WRITES INTEGER FORMATTED ARRAY
.WIAN	WRITES INTEGER NAMELIST ARRAY
.WIAS	WRITES INTEGER FREE-FORMAT ARRAY
.WISA	WRITE INTEGER SCALAR WITH ARRAY FORMAT
.WISF	WRITES INTEGER FORMATTED SCALAR
.WISN	WRITES INTEGER NAMELIST SCALAR
.WISS	WRITES INTEGER FREE-FORMAT SCALAR
.WLAA	WRITE LOGICAL ARRAY WITH ARRAY FORMAT
.WLAF	WRITES LOGICAL FORMATTED ARRAY
.WLAN	WRITES LOGICAL NAMELIST ARRAY
.WLAS	WRITES LOGICAL FREE-FORMAT ARRAY
.WLSA	WRITE LOGICAL SCALAR WITH ARRAY FORMAT
.WLSF	WRITES LOGICAL FORMATTED SCALAR
.WLSN	WRITES LOGICAL NAMELIST SCALAR
.WLSS	WRITES LOGICAL FREE-FORMAT SCALAR
.WRAA	WRITE REAL ARRAY WITH ARRAY FORMAT
.WRAF	WRITES REAL FORMATTED ARRAY
.WRAN	WRITES REAL NAMELIST ARRAY
.WRAS	WRITES REAL FREE-FORMAT ARRAY
.WRITE	WRITES FROM WORK AREA TO FILE
.WRSA	WRITE REAL SCALAR WITH ARRAY FORMAT
.WRSF	WRITES REAL FORMATTED SCALAR
.WRSN	WRITES REAL NAMELIST SCALAR
.WRSS	WRITES REAL FREE-FORMAT SCALAR
.WSAU	WRITE SINGLE ARRAY UNFORMATTED
.WSSU	WRITE SINGLE SCALAR UNFORMATTED
.WZ	WRITE A VALUE IN "Z" FORMAT
.XCHAR	INSERT CHARACTERS OF NUMBER INTO TRACE LINE
.XERRM	MESSAGE TO SHOW LOCATION OF RUN-TIME ERROR
.XEXIT	EXIT FROM A SUBPROGRAM
.XLIST	LIST COMPILER STATISTICS
.XNAME	INSERT SIX CHARACTER NAME INTO TRACE LINE
.XPROF	GENERATE RUN-TIME PROFILES
.XSTAT	SAVE LINE NUMBER OF STATEMENT, UPDATE LAST N STATEMENTS

Table 12-3. List of Intrinsics (Cont)

Intrinsic	Explanation of Responsibility
.XSUBP	SAVE SUBPROGRAM ENTRY INFORMATION - UPDATE SUBPROGRAM PROFILE
.XVALI	INSERT CHARACTERS OF INTEGER INTO TRACE LINE
.XVALR	OUTPUT REAL VALUE
.XVALX	OUTPUT VALUE ACCORDING TO TYPE
BLOCK.	BLOCK DATA FOR TRACEF
CHANGE	PROCESSES THE "CHANGE" STATEMENT - ISSUES MCP COMMUNICATE
DUMP	DUMPS BASE TO LIMIT MEMORY FOR ANALYSIS
EXIT	ROUND IN WHEN "CALL EXIT", SAME FUNCTION AS "STOP"
ZIP	PROCESSES THE "ZIP" STATEMENT - ISSUES MCP COMMUNICATE

13. COMPILER OPTION CONTROL CARDS

Compiler option control cards ($ cards) may be optionally included in the compilation deck and, if used, contain specifications to the compiler governing symbolic input and output.

A dollar sign ($) must appear in columns 1 or 2, and options with blanks as delimiters may immediately follow. If the dollar sign is placed in card column 2, the compiler option control card image is placed in the updated symbolic file (NEWSOURCE) if such a file is generated. Option control cards cannot be continued, and may be placed at any point in the compilation deck with the exception of ERRORTRACE and PROFILES.

COMPILER CONTROL CARD FORMAT

The format of the compiler control card is as follows:

Card Columns	Contents
1 or 2 through 72	$ or options is free-field format with blanks as delimiters.
73 through 80	Sequence number or blank.

The FORTRAN option control card has the following characteristics:

a. A $ sign may appear in column 1 or 2. When placed in column 2, the option control card is included in the new output source file if such a file is generated.

b. There must be at least one space between each item.

c. Options may be in any order.

d. Columns 73-80 are reserved for sequence numbering.

e. Any number of option control cards may appear within the source deck.

OPTIONS

The options that are available for the FORTRAN compiler are as follows:

BCD	Translates BCD source and card input into EBCDIC to allow the FORTRAN compiler to read it.
BIND	Causes the intermediate code files to be bound into an executable code file: This is a default option; if binding is not desired then NO BIND should be used. See appendix A for more information about intermediate code files.
CODE	Lists the object code for each source statement from the point of its insertion into the source deck.

CONTROL

NO CONTROL prevents option control cards from appearing on the compilation listing. The default is CONTROL; option control cards are listed by default.

DOUBLE

Causes output source listing to be double spaced.

DYNAMIC integer

This option specifies the size in words to be assigned for an object program's dynamic memory. If the number of words specified is greater than the total size of all data pages, the total data page size is used instead. By default, the compiler assigns the dynamic memory in one of two ways: (1) if the number of data pages is less than 10, it assigns a size equal to the sum of the data pages, or (2) if the data pages exceed 10, then the size of the 10 largest data pages is used.

ERRORTRACE

Provides a FORTRAN level trace of subprogram and statement usage prior to the detection of a run-time error. The ERRORTRACE option must be placed before the first executable statement of the main program or any subprogram. Once set, it may reset at any point by the NO ERRORTRACE option.

INITIAL

Provides a default initialization for variables which are not explicitly initialized. If a variable is never used, the compile-time warning variable NOT INITIALIZED occurs, but program execution is not aborted. If an array element is assigned a value in a DATA statement, the rest of the array is blanked (if strings were assigned) or zeroed (if numbers were assigned).

INITIAL is the default option; if initialization is not desired, NO INITIAL must be specified. A data item is considered uninitialized if it does not occur in one of the following cases:

a. As the left side of a replacement statement.

b. As the object of an ASSIGN statement.

c. As an element in an input list.

d. As a parameter in a CALL statement.

e. As a dummy parameter in a subprogram declaration.

INITIAL (Cont)

f. As an element in a COMMON statement.

g. As an element in an EQUIVALENCE statement.

h. As an element in a DATA statement. If an array element occurs in any of the above cases, the entire array is considered to be initialized.

INTERPPACK pack-id

Modifies the pack-id of the FORTRAN interpreter for the object program. Default is system disk.

INTRINPACK pack-id

Modifies the pack-id of the FORTRAN intrinsic file (FOR.INTRIN). Default is system disk.

LIBRARY multifile-id/file-id

Inserts source from another file into the compiled program but not into NEWSOURCE.

LIBRARYPACK pack-id

Changes the name of the library pack.

LIST

Creates a single spaced output listing of the source statements with error and/or warning messages. This is a default option.

MAP

Prints symbolic reference information about variables, subprograms, and labeled statements. If MAP is set, uninitialized data items are noted in the symbol table dump at the end of each program unit. In all other cases, warning messages are printed for each uninitialized data item at the end of the program unit.

The following information is generated by MAP:

a. A class of UNSPEC (unspecified) occurs when a variable is declared but not referenced.

b. Addresses of parameters are Return Control Word (RCW) relative, and are printed as "RCW+nn" or "RCW-nn", where nn is a word displacement.

c. Variables in COMMON are called "GLOBAL" in the notes portion of the listing.

MAP (Cont)

d. System routines do not have their number of arguments printed, and all intrinsics are given a class of "SUBRTN" regardless of their actual class. The correct number of arguments is printed on the listing.

e. Dummy parameters in statement function declarations do not appear in the listing.

f. In a function subprogram unit, the function name appears as a scalar.

g. Labels with a type of CONTROL are those to which transfers of control might be made, as opposed to FORMAT and DO-END (a specialized control label).

MERGE

The MERGE option allows source input from disk or tape (disk by default; file-identifier is SOURCE) to be merged with source statements from a card reader. The NEW option must be used with the MERGE option to create a new output source file. When the NEW option is not used, both the output listing and the object code file reflect the merged statements but a new source file is not created.

NEW

Creates a new source file having the file identifier NEWSOURCE. The new output file includes any changes made by the use of the MERGE option and any compiler option statements that have the dollar sign in column two.

NEWINTRINSIC

This option allows testing of potential intrinsics without placing them in the FORTRAN Intrinsic File (FOR.INTRIN). If this option is specified, the compiler first attempts to locate the intrinsic on disk by name before searching the FOR.INTRIN file. If a pack-id was specified on the option control card, the intrinsic must be located on the specified user pack rather than system disk.

NO

When used in conjunction with the following options, the NO option negates or puts them in a reset condition. There must be a space between NO and the option.

BIND
CODE
DOUBLE
ERRORTRACE
LIST
PROFILES
SEQERR
SAVEICM
SUPPRESS
TRACEC
TRACEF

PAGE	Causes the output listing to eject at that point and start a new page.
PROFILES	An optimization aid indicating those areas of a program that can be optimized to improve program performance. At run time the following information is provided by the PROFILES option: a. Frequency of subprogram usage. b. Time spent in each subprogram. c. Use of individual statements within a subprogram. d. Use of each statement during program execution. The PROFILES option must be placed before the first executable statement of the main or subprogram. To reset the option use NO PROFILES at any point within the program.
RANGE nnnnnnnn [nnnnnnnn]	Resets the sequence parameters if SEQ is set. The default increment is 100.
SAVEICM	Causes the intermediate code files for each syntax-free and error-free program part to be made a permanent disk file at the end of the compilation.
SEQ	Causes resequencing of the output listing and the new source file, if applicable, starting with the default number 00001000 and incrementing sequence numbers by 1000.
SEQ nnnnnnnn [nnnnnnnn]	Causes resequencing of the output listing and the new source file if applicable. SEQ is followed by either an eight-digit number which is the starting sequence number, or two eight-digit numbers with the first number being the starting sequence number and the second the resequencing increment value. The default resequence increment is 1000.
SEQERR	Causes a warning message to be printed in single spaced format.
SINGLE	Causes the output listing to be printed in single spaced format. This is a default option.
STACKSIZE	Specifies the size in words to be allocated for the object program evaluation stack. Default size is 100; maximum size is 4096.

SUPPRESS

Suppresses warning messages in the output listing. The default is NO SUPPRESS.

TRACEF

Causes a FORTRAN level trace to be printed for each FORTRAN statement executed in the program. This option may be inserted anywhere within the program. Once set, it remains set until reset by using NO TRACEF. Each TRACEF line contains the name of the current subprogram, the compiler-generated line number of the current statement in the subprogram, and an identification of the statement.

TRUNCATE.NAMES

Allows variable names to be extended to more than six characters without causing a syntax error. The FORTRAN compiler truncates the variable names to six characters. These truncated variable names must remain unique in the program.

VOID

Causes the source input image corresponding to the sequence number of the VOID card to be deleted from the input disk file.

VOID nnnnnnnn

Causes a series of source images to be deleted starting from the sequence number in the sequence number field through and including the sequence number of the VOID option.

XREF

Causes the compiler to produce a cross-reference listing after the source program listing and before the "CODE AND DATA MAPPING" table.

This listing consists of two parts: the first is a list of label numbers followed by a subprogram name (in parentheses) and the sequence numbers where the label appears in the subprogram. The second part of the XREF listing contains a list of variable names followed by a subprogram name (in parentheses) and the sequence numbers where the variable appears in the subprogram. Each sequence number in both parts is immediately followed by one of the following keys:

Key	Description
#	Specification statement or explicit label.
space	Identifier or label reference.
=	Left side of replacement or assignment statement.
*	Variable passed by address.

14. PROGRAM STRUCTURE

Every executable FORTRAN program consists of a sequence of statements, with each statement physically contained on one or more lines or card images.

SOURCE INPUT FORMAT

FORTRAN statements must be input to the compiler upon cards or card images which have the following format. (All blanks are ignored in source input except when they occur in Hollerith or proper strings.)

Source input cards are in general free form, with these exceptions:

a. Columns 1 through 5 of a card may contain a statement label. (See section 4.) This field is recognized as a label on the first card only of an executable or FORMAT statement. On all nonexecutable statements other than the FORMAT statement and on all continuation cards, this field is syntaxed. A label without an associated statement is ignored. Blanks and preceding zeros are ignored.

b. Column 6 of the first card of a statement must be blank or contain a zero. A statement may be continued on several cards by placing any non-blank and non-zero character in column 6 of the continuation cards.

c. Columns 7 through 72 of a card contain the FORTRAN statement.

d. Columns 73 through 80 may contain sequence numbers. This field is checked for ascending sequence numbering when $ MERGE or $ SEQERR is set; otherwise, the field is ignored.

If a card contains a "C" in column 1, the card is considered a comment card and is not interpreted. No comment or blank cards may be inserted immediately preceding a continuation card.

A card containing a $ in column 1 or 2 is a compiler option control card as discussed in section 13.

Blank characters are significant only in column 6 of a statement card and in string literals. With these exceptions, blanks may be used freely or omitted altogether without affecting the meaning of the FORTRAN program.

PROGRAM UNITS

Every executable FORTRAN program consists of exactly one main program unit which may be preceded and/or followed by as many subprograms as necessary.

Each program unit consists of a sequence of statements terminated by an END statement.

END Statement

The nonexecutable END statement is provided for use as the terminal statement of a program unit.

The proper form of the END statement is as follows:

END STATEMENT
END

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.

Main Program

The main program is the first (and perhaps only) program unit of an executable program to be executed.

The main program may invoke SUBROUTINE and FUNCTION subprograms using the CALL statement and function references, respectively. All FILE declaration statements must appear in the main program.

Statement Ordering

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

a. FILE declaration statements must precede all other statements of the main program.

b. IMPLICIT statement occurs next.

c. Specification statements must appear before any executable statement and statement function declaration statements.

d. FORMAT statements and comments may appear at any point in the program unit.

e. Statement function declarations must follow specification statements and precede any executable statements.

Thus, the recommended order of appearance of FORTRAN statements in a program unit is as follows:

a. IMPLICIT statement.

b. DIMENSION, COMMON, INTRINSIC, EXTERNAL, or explicit type statements in any order.

c. EQUIVALENCE statements.

d. DATA statements.

e. Statement function declarations.

f. Remainder of program unit.

g. END statement.

A. B 1800/B 1700 FORTRAN LANGUAGE SYSTEM

The purpose of this appendix is to provide an outline of the features of the B 1800/B 1700 FORTRAN language system. This includes the following:

a. A summary of system requirements.

b. A digest of user-oriented compiler information.

c. A complete discussion on control cards and the structure of the FORTRAN compilation deck.

The FORTRAN compiler described here and the object programs generated by it are designed to operate under control of the B 1800/B 1700 Master Control Program (MCP).

SYSTEM REQUIREMENTS

Required Hardware

Before the FORTRAN system may operate, the following minimum hardware devices must be provided:

a. B 1800/B 1700 processor.

b. Disk.

Required System Software

The FORTRAN system file requirements are as follows:

a. The compiler (which includes a binding phase).

b. The intrinsic file (which contains various subprograms supplied with the compiler).

c. The FORTRAN interpreter (which executes the object code).

The compiler, interpreter, and intrinsic files must all reside on the same disk, unless otherwise specified in a compiler option control card. If on a user cartridge, they are referenced by prefacing their names with the cartridge name.

USER/COMPILER INTERFACE

The purpose of the B 1800/B 1700 FORTRAN compiler is to accept application programs written in the FORTRAN language and to produce from these programs object code which may be executed on the B 1800/B 1700 system.

Concurrent to the production of this program object code, the user is provided with compile-time debugging and diagnostic facilities and the ability to a limited extent to control the function performed by the compiler (such as in the area of compiler file handling). The latter ability is available using the FORTRAN compiler option control card ($ card).

The debugging and diagnostic facilities provided by this 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.

a. Syntax-error numbers and messages, which are placed on the printer listing generally following the line of text bearing the incorrect statement.

b. Messages and numbers denoting warnings are optionally placed on the printer listing following the line bearing the incorrect statement.

c. Various compiler information messages.

d. A complete listing of compiler-generated code, optionally placed on the printer listing.

e. The status of the executing FORTRAN compiler may be queried by entering ⟨*mix number*⟩ AX ST on the SPO. One of the following responses will be displayed:

COMPILING ⟨*SUBPROGRAM*⟩ SEQ=N, ERRS=NN
BINDING ⟨*SUBPROGRAM*⟩ N OF M KNOWN SUBPROGRAMS

Error messages and error numbers that are generally sufficient to determine the cause of errors are provided. A list of FORTRAN run-time error messages is provided in appendix C.

All user communication with the compiler and all compiler output is handled using compiler files. A discussion of the interface between the user and the FORTRAN compiler is therefore an examination of the features of these compiler files. The system of compiler files is illustrated in figure A-1.

Intermediate Code Files

Depending on the option control cards 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 non-executable form. An executable file is produced by the binding part of the compiler using the ?COMPILE card specification or through the appropriate compiler control card ($ card) specification. For information on the use of option control cards, see section 13.

Compiler Files

Compiler communication is handled through various input and output files. The compiler files described in the following paragraphs are diagrammed in table A-1.

The compiler has the capability of merging, on the basis of sequence numbers, input from two files. 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 compiler can also generate an updated symbolic output file. These files may be created in addition to the compiler-generated output code file. Compiler input and output files are discussed in detail here.

Input Files

The primary compiler input file is a card file with the internal name CARDS; the secondary input file is a serial disk file with the internal name SOURCE. The presence of the primary file CARDS is required for each compilation; the presence of the secondary file SOURCE is optional for each compilation. File CARDS is coded with 80-character records and is unblocked. FILE SOURCE is coded with 80-character records and uses input blocking. Both the CARDS file and the SOURCE file may be label-equated (using the ?FILE control card) to change the file's external file name and hardware device. (See the discussion on the ?FILE card in the B 1800/B 1700 System Software Operational Guide, Form No. 1068731.)

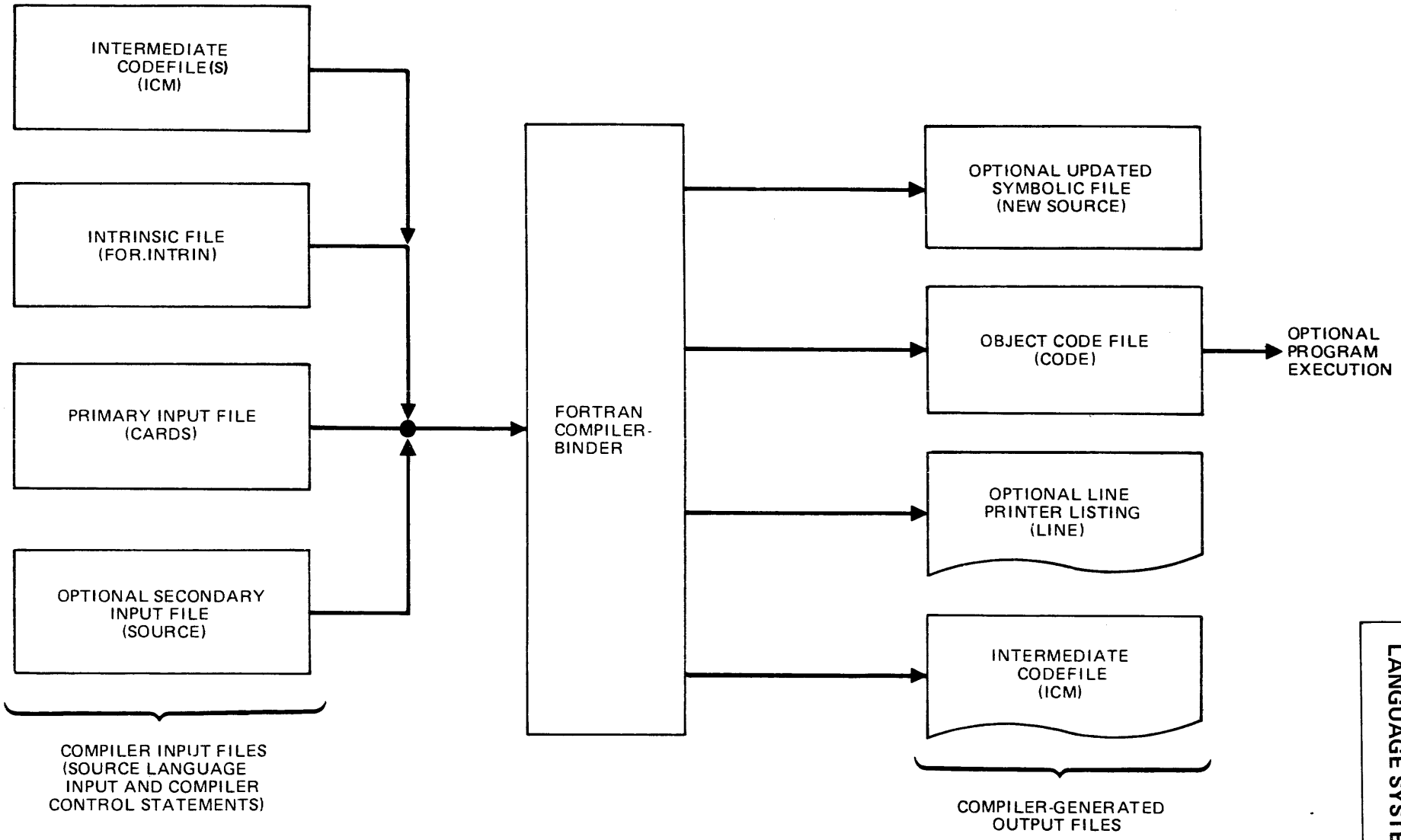


Figure A-1. FORTRAN Compilation System

Table A-1. FORTRAN Compiler File Names and Defaults

Internal Name	Purpose	Default Hardware Device	Record Size	Block Size	Comments
CARDS	Input Card File	CARD READER	80 characters	Unblocked	Required for each compilation. Primary compiler input file. Default external file name is CARDS. BUFFERS = 1.
SOURCE	Input Disk File	DISK	80 characters	Uses input blocking (default is 2 records per block)	Optional file; need not be present for each compilation. Secondary compiler input file, selected as input by setting MERGE compiler option. The default external name is SOURCE. BUFFERS = 1.
CODE	Executable Object Code File	DISK	1440 bits	1440 bits	Generated object code file. Saved or discarded and assigned the program-name. BUFFERS = 1.
NEWSOURCE	Updated Symbolic Output File	DISK	80 characters	Two records per block	Optional output file produced when $NEW compiler option is set. The default external name is NEWSOURCE. BUFFERS = 1.
LINE	Line Printer Listing	LINE PRINTER	120 characters	Unblocked	The external file name is LINE. BUFFERS = 1.
FOR.INTRIN	Intrinsics and Intrinsic Functions	DISK	1440 bits	1440 bits	The intrinsics and intrinsic functions file.
ICM	Intermediate Code File	DISK	1440 bits	1440 bits	Intermediate code file produced by the compiler.

Output Files

Output files produced by this compiler include intermediate and object code files, an updated symbolic 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 card specifies otherwise. The external file name of the saved code file is identical to the program-name appearing on the COMPILE card. (See the section on 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 (NEWSOURCE) generated only if the compiler option $NEW is set. This file contains the compilation source input or a selected portion of this input as specified by the compiler option NEW and may be used as the SOURCE file for a succeeding compilation.

The printer listing is an optional print file that is created unless the compiler option LIST is reset. (The LIST option is set by default.) The file has the internal name LINE, and contains the following information:

a. Source and compiler control statements input to the compiler.

b. Code segmentation information.

c. Error messages and error count.

d. Number of input statements scanned.

e. Elapsed compilation time per subprogram and total bind time.

f. Estimated size of the program's data space when the program is executed.

g. Estimated space needed for the program files.

h. Total number of bytes of object code generated for each subprogram.

i. Number of disk segments required for the program code file.

j. Estimated memory required to run the object program.

k. The date the compiler was compiled.

Depending upon the specified setting of the LIST, CODE, and MAP compiler options, the printer listing may contain more or less information than the basic items listed here.

Compiler File Names and Defaults

The FORTRAN input and output files are listed in table A-1, which provides information concerning the configuration of each file. Table A-1 lists the internal name of the file (i.e., the name used when the file is declared within the FORTRAN compiler), the purpose served by the file, the default hardware device of the file, the default record size (RECORD.SIZE) and block size (RECORDS.BLOCK) of the file, and a brief commentary on the file.

The attributes of any of these files may be changed through use of ?FILE control cards directed to the compiler. (See the discussion on the ?FILE card in the B 1800/B 1700 System Software Operational Guide, Form No. 1068731.)

MCP CONTROL CARDS

When a FORTRAN source program is compiled the actions to be performed are specified through the use of control cards. Control cards included in a compilation deck are of two general types: MCP control cards (?cards) and compiler option control cards ($cards). The structure of the FORTRAN compilation deck is explained in the text that follows.

Compilation of a FORTRAN source program is achieved by presenting the compilation card deck to the MCP. The entities comprising the structure of the FORTRAN compilation deck and the order of their occurrence are as follows:

Compilation Card Deck

a. ?COMPILE Card.

b. ?FILE Control Card(s).

c. ?DATA CARDS.

d. Compiler Option Control ($) Card(s) (optional).

e. File Declaration Statement(s) (optional).

f. Source input cards.

g. ?END (end-of-file) Card.

MCP control cards are made distinguishable from other cards by an invalid character in column 1 for 80-column cards, or a question mark (?) for 96-column cards. An invalid character is represented by a ? for clarity in this manual. MCP control information is punched in a free-form format in columns 2 through 72.

?COMPILE Card

The ?COMPILE card instructs the MCP to compile the indicated program-name with FORTRAN using one of the following options:

a. ?COMPILE program-name FORTRAN

This option causes the source program to be compiled, bound and executed (i.e., 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 card $SAVEICM is specified.

b. ?COMPILE program-name FORTRAN 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 option control card $SAVEICM is specified. Execution is specified by the execution statement, ?EXECUTE ⟨*program-name*⟩, placed after the ?END card. Further information on the ?EXECUTE card can be found in the B 1800/B 1700 System Software Operational Guide, Form No. 1068731.

c. ?COMPILE program-name FORTRAN 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 card $SAVEICM is specified.

d. ?COMPILE program-name FORTRAN SYNTAX

This option causes the source program to be compiled only for a syntax check.

In the absence of the ?COMPILE card, the system operator may manually execute one of the compile options through the console printer (SPO) by keying in the appropriate message.

For compile card options a, b, and c, the intermediate code files are created and left in the disk directory after compilation until binding occurs. If the program is terminated before it is bound or the compiler option NO BIND is specified, the intermediate code files will remain in the directory. (See Compiler Option Control Cards in section 13.) If any errors result during compilation, the error-free intermediate code files will remain in the directory and no binding will occur. Therefore, the error-free intermediate code files remaining in the directory will 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 "FILE NOT PRESENT" will be given during binding. Subprograms may be compiled independently or with the main program which references them.

PROGRAM-NAME

The program-name may consist of one, two, or three identifiers of up to 10 characters each, separated by slashes. The four forms a program-name can take are as follows:

a. family-name (an identifier which is a single file name).

b. family-name/file-identifier (an identifier which may be a single file name or a file with subprogram entries).

c. dp-id/family-name/file-identifier (the disk-pack-identifier is specified when a removable disk pack is used).

d. dp-id/family-name/ (a single file name residing on a removable disk pack).

An executable code file will have the program-name specified on the compile statements. Intermediate code files will have the program-name on the compile statement except the file-identifier will be replaced by the following:

a. The subprogram name in a SUBROUTINE or FUNCTION statement or

b. Zero (0) followed by the file-identifier (if any) for an unbound main program or

c. 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.

The following table 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	FORTRAN/MAIN/SUB	FORTRAN/MAIN/
Unbound Main Program	MAIN/O	MAIN/OSUB	FORTRAN/MAIN/OSUB	FORTRAN/MAIN/O
SUBROU-TINE X	MAIN/X	MAIN/X	FORTRAN/MAIN/X	FORTRAN/MAIN/X
BLOCK DATA Subprogram	MAIN/BLOCK.	MAIN/BLOCK.	FORTRAN/MAIN/ BLOCK.	FORTRAN/MAIN/ BLOCK.

MAIN is the family name.
SUB is the file-identifier.
FORTRAN is the disk pack identifier.

?FILE Card

The ?FILE control card may be optionally included in the compilation deck. The ?FILE control card may be used to modify the original attributes of the FORTRAN system files.

If used, the ?FILE control card(s) must immediately follow the ?COMPILE card and precede ?DATA CARDS. The general form of the ?FILE control card is:

```
?FILE  internal file name  file-attribute list;
```

A list of file-attributes and their uses can be found in the B 1800/B 1700 System Software Operational Guide, Form No. 1068731.

The FORTRAN compiler's internal file names for use with the ?FILE control card are as follows:

a. CARDS (Input file from the card reader (default)).

b. LINE (Compilation output to the line printer).

c. SOURCE (Symbolic input file on disk (default)).

d. NEWSOURCE (Updated symbolic output file on disk (default)).

The file SOURCE is used on input only when the compiler option MERGE is set.

?DATA CARDS Card

A card of the form:

```
?DATA CARDS
```

is required to label the source deck.

Source Input Cards

These cards are the FORTRAN statements comprising the source program.

?END Card

The ?END Card designates the end-of-file for the compilation deck to the MCP.

The ?END Card is coded as follows:

```
?END
```

The ?END Card must be the last card in the compilation deck.

The examples that follow show seven ways a program called JOB containing two subprograms (SUB1 and SUB2) can be compiled and executed, using the MCP control cards ?COMPILE, ?DATA, and ?END.

Example 1 - Compile-and-go only:

```
?COMPILE JOB FORTRAN
?DATA CARDS
C       MAIN PROGRAM JOB
        .
        .
        .
        END
        SUBROUTINE SUB1
        .
        .
        END
        FUNCTION SUB2
        .
        .
        END
?END
```

Example 2 - Compile, execute and save ICMs:

```
?COMPILE JOB FORTRAN                    or      ?COMPILE JOB FORTRAN SAVE
?DATA CARDS                                     ?DATA CARDS
$SAVEICM                                        $SAVEICM
C     [MAIN PROGRAM JOB                               [MAIN PROGRAM JOB
      |.                                              |.
      |.                                              |.
      |.                                              |.
      [END                                            [END
      [SUBROUTINE SUB1                                [SUBROUTINE SUB1
      |.                                              |.
      |.                                              |.
      [END                                            [END
      [FUNCTION SUB2                                  [FUNCTION SUB2
      |.                                              |.
      |.                                              |.
      [END                                            [END
?END                                            ?END
```

Notice that when the SAVE option is used, the object program is also entered in the disk directory.

Example 3 - Compile and execute in three separate steps:

```
?COMPILE JOB FORTRAN LIBRARY
?DATA CARDS
$SAVEICM
$NO BIND
      [SUBROUTINE SUB1              (An ICM named JOB/SUB1 is on disk.)
      |.
      |.
      [END
?END
?COMPILE JOB FORTRAN LIBRARY
?DATA CARDS
$SAVEICM
$NO BIND
C     [MAIN PROGRAM JOB             (An ICM named JOB/0 is on disk.)
      |.
      |.
      [END
?END
?COMPILE JOB FORTRAN LIBRARY    or      ?COMPILE JOB FORTRAN SAVE
?DATA CARDS                             ?DATA CARDS
$SAVEICM                                $SAVEICM
      [FUNCTION SUB2                          [FUNCTION SUB2
      |.                                      |.
      |.                                      |.
      [END                                    [END
?END                                    ?END
?EXECUTE JOB
```

B. LANGUAGE COMPATIBILITY

Source language programs written in ANSI FORTRAN are acceptable as input to the B 1800/B 1700 FORTRAN compiler, with the following exceptions:

a. The computed GO TO variable must be an integer expression.

b. Subprogram units may only be written in FORTRAN. This is because there is no machine facility to change the interpreter during program execution.

c. BN and BZ format specifiers are not acceptable.

d. Character type statement characters may be stored in integer (4 characters) or double precision (8 characters) variables.

e. SAVE statement is not acceptable. All local variables in a subprogram retain their previous values from one invocation to the next invocation.

f. OPEN statement is not acceptable. All opens are implicit.

g. Internal file is not acceptable. Internal files provide a means of transferring and converting data from internal storage to internal storage.

h. ENTRY statement is not acceptable.

The following FORTRAN IV extensions to the proposed 1976 ANSI FORTRAN subset (See Sigplan Notices, Vol. 11, No. 3, 1976 March), are included in B 1800/B 1700 FORTRAN.

a. Variables and constants.

1. A dollar sign may be used in variable names.

2. The variable length specifier REAL*8 translates into double precision. All other length specifiers are syntaxed as errors.

3. Variables are initialized unless NO INITIAL is specified.

4. Character strings may be delineated using both quote marks and apostrophes.

5. Hexadecimal constants are allowed in DATA statements.

b. Arrays

1. Up through 15 dimensions may be specified for arrays.

2. Array subscripts may be any integer expressions, but more efficient code may be emitted if the subscript can be reduced to a constant * variable ± constant.

3. Array names may be used in DATA and EQUIVALENCE statements.

c. Additional Statements

1. CALL CHANGE statement.

2. NAMELIST statement.

3. PAUSE n where n may be an integer or a string of length < characters.

4. PUNCH statement.

d. Arithmetic expressions.

1. Mixing of integer with real or double precision variables in relations and arithmetic expressions is allowed.

2. Exponentiation may include integer**real, real**integer, and double precision**integer.

e. DO loops.

1. Branching into or out of DO loops is permitted.

2. DO loop parameters may be changed during the execution of a DO loop.

3. The control variable at the end of a DO loop is defined as having the value of the final test.

4. Arithmetic expressions are allowed in the parameters of a DO loop.

f. END, EXIT, and STOP.

1. The intrinsic EXIT is provided and may be called to cause error-free termination of a program. In addition, the STOP statement will invoke the EXIT intrinsic.

g. Subprograms, arguments, and statement functions.

1. Indirect recursion is allowed.

2. Array elements may be referenced in the expression of a statement function.

3. An adjustable dimension array in a subprogram may have its dimensioned variables in COMMON.

4. Slashes delimiting formal parameters are ignored.

5. Alternate return from a subprogram:

	B 1800/B 1700 FORTRAN	ANSI FORTRAN
Dummy Argument	*	*
Actual Argument	& label	* label

6. SAVEICM causes the intermediate code file of a subprogram to be retained after compilation for future binding.

h. COMMON and BLOCK DATA.

1. A labeled COMMON block need not be the same size in all subprograms referencing it.

2. Variables in unlabeled COMMON may be initialized in BLOCK DATA.

3. The COMMON blocks in the BLOCK DATA subprogram are allowed to be abbreviated by specifying items up through the last item in a block to be initialized.

i. Files and I/O.

1. The NEW attribute must be specified in a FILE declaration statement to create the file.

2. FILE declaration statements must appear in the main program unit.

3. Random disk file accessing is allowed in I/O statements.

4. Record numbers may be specified in READ and WRITE statements using either an equal sign or an apostrophe.

5. An expression is allowed in the list of a WRITE statement.

6. Error (ERR), end-of-file (END) and data error (DATA) action specifiers are allowed in I/O statements.

j. FORMAT specifications.

1. The characters 1 through 9 will be interpreted as valid printer channel numbers when they appear as the first character in a line to be printed. All other non-standard carriage control characters will result in single spacing.

2. FORMAT specifiers Z (hexadecimal) and T (tabulate) may be used.

3. The type of an item in an I/O variable list does not need to match the type of the FORMAT specifier. The type in the FORMAT specifier must match the external data type.

C. WARNING AND ERROR MESSAGES

Error and warning diagnostics may occur during compilation, binding, or run-time. Warning messages only supply information about situations which might be a source of problems for the user. They do not cause premature termination of the compile, bind, and run-time procedures.

Error messages supply information about user and system problems which keep a program from completely going through the compile, bind, and run-time procedures. For a single run, errors during compilation prevent binding, errors during binding prevent execution, and errors during run-time prevent normal termination.

Error messages occurring during compilation or binding, and a further explanation of run-time errors is presented here.

There are two types of run-time error messages: those which appear on the line printer and those which appear on the console printer (SPO).

LINE PRINTER RUN-TIME ERROR MESSAGES

When there is an error in an intrinsic or intrinsic function, the output to the line printer consists of an error message and a stack dump of segments and displacements. For example, when the following program is executed:

```
R=SQRT(-7.0)        000075
STOP                000415
END                 000475
```

The line printer output is as follows:

```
ARGUMENT OUT OF RANGE
STACK DUMP    SEGMENT     DISPLACEMENT
              0006        00016845
              0003        00000306
              0001        00000385
```

This stack dump corresponds to the RETURN CONTROL WORD (RCW) addresses in the FORTRAN dump, except the first address which is always the .ERROR intrinsic.

In compiling the above program, the ADDRESS FINAL of the CODE AND DATA MAPPING section of the compile listing will show that segment 6 corresponds to the .ERROR intrinsic, segment 3 corresponds to the SQRT intrinsic and segment 1 corresponds to the main program. Code address segment 1, displacement 385, is between 75 and 415. Therefore, the statement "R=SQRT(-7.0)" at displacement 75 caused the error.

The following is a list of the run-time error messages generated by the intrinsics:

```
"&END" EXPECTED
ARGUMENT OUT OF RANGE
ARRAY BOUND EXCEEDED
BLANK MUST FOLLOW NAMELIST NAME
COMMA EXPECTED
COMPLEX DIVISOR OF ZERO
```

DATA ERROR
DECLARED RECORD LENGTH TOO LONG
DEVICE NOT DISK
END OF FILE
ERROR NUMBER - (AN INTRINSIC CALLED ERROR WITH INVALID PARAMS)
ERROR OCCURRED IN LINE n OF program ("$ERRORTRACE" PRINTS IF ERROR)
ERROR ON UNIT = n
EXPECT BOOLEAN
EXPONENT UNDERFLOW OR OVERFLOW
FUNCTION OUT OF RANGE
INCORRECT FORMAT
INCORRECT NUMBER
INCORRECT FUNCTION
INCORRECT TYPE
INVALID ASSIGNED GO TO
INVALID COMPLEX CONSTANT
INVALID CONSTANT
INVALID EXPONENT
INVALID HOLLERITH CONSTANT
INVALID LOGICAL CONSTANT
INVALID REPEAT
INVALID TERMINATOR
INVALID VARIABLE NAME
MISSING EQUAL
NAME NOT ELEMENT IN NAME LIST
PARAMETER MISMATCHED
PARITY ERROR
RANDOM ACCESS TO SERIAL FILE
RECORD NOT UNFORMATTED
RECORD SIZE EXCEEDED
RECORD SIZE IN FPB MUST BE LESS THAN OR EQUAL TO MAX DECLARED IN FILE STMT
REPEAT NOT ALLOWED ON SCALAR
RIGHT PAREN EXPECTED
SCALAR MAY NOT BE SUBSCRIPTED
SERIAL ACCESS TO RANDOM FILE
STRING GREATER THAN 4 CHARACTERS
SUBSCRIPT MUST BE POSITIVE INTEGER
TYPE MISMATCHED
UNDER/OVER FLOW
UNEXPECTED EOF
UNIT UNDEFINED
VALUE EXPECTED
ZIP ARRAY SIZE GREATER THAN 128 ELEMENTS

CONSOLE PRINTER RUN-TIME ERROR MESSAGES

When a run-time error message appears on the console printer, the user has the option of either discontinuing the program (mix-index DS) or initiating a memory dump and then aborting the program (mix-index DP). If the system option, TERM, is set, the program will automatically terminate and DS or DP need not be specified. For more information, see the B 1800/B 1700 System Software Operational Guide, Form No. 1068731.

The format of the run-time errors which appear on the console printer is as follows:

```
program-name = mix-index -- error message
: NXT INSTR = SEG @nnn@ DISP @nnnnnn@ (nnnn) (nnnn) DS OR DP
```

Numbers enclosed in @ are in hexadecimal format. Numbers enclosed in parentheses are in decimal format. Refer to the code segment number and displacement number appearing in the CODE AND DATA MAPPING section of the compile listing.

The following is a list of the run-time error messages which may appear on the console printer. The contents of these error messages is explanatory.

Error Message

STACK OVERFLOW - Refer to the $STACKSIZE option in section 13.
INVALID PARAMETER
INVALID SUBSCRIPT
DIVIDE BY ZERO
INVALID OPERATOR - This is a system error and if occurs, should be reported to the Burroughs Systems Representative.
UNINITIALIZED DATA ITEM
EXPONENT OVERFLOW
EXPONENT UNDERFLOW
INTEGER OVERFLOW

COMPILATION OR BINDING WARNING AND ERROR MESSAGES

The following is a list of warnings and errors that can occur during compilation or binding.

Warning Messages

000 MAXIMUM OF 8 CHARACTERS ALLOWED
004 SYSTEM INTRINSIC HAS BEEN REDEFINED
008 STATEMENT IGNORED
009 DOUBLE PRECISION VALUE GENERATED
011 SEQUENCE ERROR
012 TRUNCATED TO 10 CHARACTERS
013 SAVE FACTOR CHANGED TO 999
016 PARAMETER TYPE MISMATCH
017 DO VARIABLE USED IN ENCLOSING DO
018 ILLEGAL STATEMENT AT END OF DO LOOP
019 LABEL REQUIRED TO EXECUTE THIS STATEMENT
021 ZERO SUPPLIED FOR IMAGINARY PART
022 OVERLAPPING IMPLICIT DECLARATIONS
023 IN LINE SYSTEM ROUTINE CHANGED TO EXTERNAL PROCEDURE
024 HEXADECIMAL LITERAL TRUNCATED
025 STRING LITERAL TRUNCATED
026 LITERAL STRING TRUNCATED TO 4 CHARACTERS
027 LITERAL STRING TRUNCATED TO 8 CHARACTERS
028 LABEL FIELD ON CONTINUATION CARD NOT BLANK
029 MISSING DELIMITER
OPTION NAME CHANGED TO MAP
STARTING SEQ NUMBER EXCEEDS 8 DIGITS
RESEQUENCE INCREMENT EXCEEDS 8 DIGITS

Warning Messages

VOID LIMIT EXCEEDS 8 DIGITS
RANGE CARD REQUIRES STARTING SEQ NUMBER
CREATING A NEW SYMBOLIC INCOMPATIBLE WITH "$END" FUNCTION
NEW FILE TERMINATED HERE
LIBRARY CARD REQUIRES AT LEAST ONE NAME
UNEXPECTED SLASH
SLASH EXPECTED
SECOND IDENTIFIER EXPECTED
TOO MANY HMON REQUESTS
PREVIOUS TRACE REQUEST LOST
NUMBER EXPECTED
ERRORTRACE IGNORED
PROFILES IGNORED
UNRECOGNIZED PARAMETER
REDUNDANT COMMAS WILL BE IGNORED
IMPLIED DO INDEX IS NOT INTEGER
VARIABLE MAY HAVE BEEN INITIALIZED PREVIOUSLY
variable name EXHAUSTED WITH DANGLING CHARACTERS
SLASH EXPECTED
variable name BLANK FILLED
variable name ZERO FILLED
variable name IS NOT INITIALIZED
COMMON BLOCK common block name SIZE INCREASED FROM size TO size
OPTION NAME CHANGED TO MAP

Error Messages

001 COMPILER ERROR ONE
002 LABEL MUST BE NUMERIC GREATER THAN 0
003 RECURSIVE CALL NOT ALLOWED
004 DUPLICATE NAME
005 SUBSCRIPT EXPECTED
007 THIS TYPE OF STATEMENT NOT IMPLEMENTED
008 LEFT PARENTHESIS EXPECTED
009 INTEGER EXPECTED
011 TOO MANY CONTINUATION CARDS ..... MAX=19
012 INVALID STRING
013 INVALID PUNCTUATION
015 END STATEMENT SHOULD NOT BE LABELED
019 INVALID CONTINUATION
021 MISSING COMMA
025 MISSING EQUAL SIGN
026 MISSING SLASH
028 VARIABLE EXPECTED
031 ILLEGAL RETURN IN MAINLINE
032 ONLY SUBROUTINE ALLOWS NONSTANDARD RETURN
033 MAX NUMBER OF PARAMETERS EXCEEDED (MAX IS 63)
035 INCORRECT EXPRESSION
036 PARENTHESIS DO NOT BALANCE
037 END OF STATEMENT EXPECTED
038 OPERATOR EXPECTED
039 CANNOT DETERMINE STATEMENT TYPE
044 SIMPLE VARIABLE CANNOT BE FOLLOWED BY ...

Error Messages

046 SUBSCRIPT MUST BE INTEGER
047 INCORRECT FORMAT SPECIFIER
049 SYMBOL TABLE OVERFLOW
050 MAXIMUM LENGTH OF AN IDENTIFIER IS SIX CHARACTERS
053 STATEMENT HAS A DUPLICATE LABEL
055 PARAMETER OR SUBPROGRAM NAME MUST BE USED
056 LOGICAL IF CANNOT INCLUDE THIS STATEMENT TYPE
061 UNEXPECTED CHARACTER
062 INVALID HOLLERITH LENGTH
065 MISSING KEY WORD
066 EXPONENT UNDERFLOW OR OVERFLOW
067 FORMAT NESTING LEVEL EXCEEDED . . . 10 MAX
070 MISSING FORTRAN END CARD
072 IDENTIFIER EXPECTED
075 ADJUSTABLE ARRAY MUST BE A PARAMETER
076 RIGHT PARENTHESIS EXPECTED
078 ITEM MAY NOT BE DIMENSIONED
079 ITEM PREVIOUSLY DIMENSIONED
080 STRING SIZE EXCEEDED
082 LABEL EXPECTED
083 ILLEGAL LENGTH SPECIFIER
085 DATA INITIAL TABLE OVERFLOW-BREAK STMT INTO MULTIPLE STMTS
087 ITEM WAS PREVIOUSLY TYPED
089 NUMBER OF SUBSCRIPTS SPECIFIED IS GREATER THAN 15
090 EQUIVALENCE TABLE OVERFLOW-BREAK EQUIV. STMT INTO MULTIPLE STMTS
091 STATEMENT NOT ALLOWED IN EXECUTABLE STATEMENTS
093 TYPE OF ACTUAL PARAMETER DOES NOT AGREE WITH PREVIOUS USE
095 INVALID PARAMETER
097 LABEL PREVIOUSLY USED AND WAS NOT OF FORMAT TYPE
098 LABEL PREVIOUSLY USED AS FORMAT TYPE
099 LABEL ON DO APPEARS ON A PRECEDING STATEMENT
100 COMMA,LEFT PARENTHESIS OR LABEL EXPECTED FOLLOWING GOTO
101 INTEGER VARIABLE EXPECTED
102 ILLEGAL NEST OF DO
103 DO IS NESTED TOO DEEPLY FOR THE COMPILER
104 DATA/FORMAT TABLE OVERFLOW
105 ABSOLUTE VALUE OF INTEGER IS GREATER THAN 8589934591
106 REPEAT/FIELD WIDTH/DECIMAL WIDTH SIZE EXCEEDED-MAX IS 255
110 UNRECOGNIZED FILE OPTION
111 UNRECOGNIZED HARDWARE TYPE
112 MAXIMUM FILE SPECIFIER=99
113 FILE NAME EXPECTED
114 DUPLICATE ACTION LABELS
115 INVALID LIST ITEM
116 I/O LIST EXPECTED
117 ACTION LABEL EXPECTED
118 INVALID FORMAT SPECIFIER
119 LABEL TOO LONG
121 SIGN NOT ALLOWED
123 FIELD WIDTH/TABULATION MUST BE GREATER THAN 0
124 ILLEGAL FORTRAN CONSTANT
125 ILLEGAL PARAMETER IN CONTROL CARD
126 INVALID IMPLIED DO

Error Messages

127 COMPLEX NOT ALLOWED
129 LABEL ON STATEMENT IS A DUPLICATE
130 ILLEGAL MIXED TYPE
131 ILLEGAL SUBROUTINE REFERENCE
134 STATEMENT IS NOT ALLOWED IN BLOCK DATA SUBPROGRAM
135 WRONG NUMBER OF PARAMETERS
136 INTEGER,STRING OR END OF STMT EXPECTED
138 LABEL ON CONTINUATION LINE
140 INTEGER EXPRESSION EXPECTED
141 INVALID COMMON ELEMENT
142 MULTIPLE APPEARANCE IN COMMON
143 INVALID LENGTH FOR THIS TYPE
144 COMMA OR END OF STATEMENT EXPECTED
145 INCORRECT NUMBER OF SUBSCRIPTS
146 EQUIVALENCED TO MORE THAN ONE LOCATION
148 SLASH EXPECTED
150 NOT ALLOWED HERE
151 ILLEGAL FCN REF OR MISSING DIMENSION
153 INVALID SUBSCRIPT
154 ARITHMETIC IF REQUIRES 3 LABELS
155 EXCEEDED MAX RECORD SIZE (1000 CHARACTERS)

label IS A LABEL THAT HAS BEEN REFERENCED BUT NOT DECLARED
MISSING INTRINSIC intrinsic name
DATA DICTIONARY OVERFLOW
MISSING ICM FOR SUBPROGRAM subprogram name
UNEXPECTED END OF FILE
DEFINITION OF SUBPROGRAM subprogram name DISAGREES WITH PREVIOUS REFERENCE
WRONG NUMBER OF PARAMETERS FOR subprogram name PREV = number CURR = number
SYMBOL TABLE OVERFLOW
COMPILER ERROR CASE OP IN PROCESS.BIG.OPCODE.
NAME FORTRAN NOT ALLOWED ON COMPILE CARD
common block name COMMON BLOCK IS LARGER THAN PRIOR DECLARED SIZE
TOO MANY FILES - MAXIMUM ALLOWED 19
operator OPERAND operator IS AN ILLEGAL COMBINATION
OPERAND REQUIRED BETWEEN operator AND operator
OPERATOR REQUIRED BEFORE operand
OPERAND NOT ALLOWED BETWEEN operator AND operator
LOGICAL OPERAND REQUIRED BETWEEN operator AND operator

CANNOT CONVERT	UNKNOWN	TO	UNKNOWN
	UNKNOWN		LOGICAL
	UNKNOWN		INTEGER
	UNKNOWN		REAL
	UNKNOWN		DOUBLE
	UNKNOWN		COMPLEX
	LOGICAL		DOUBLE
	LOGICAL		COMPLEX
	INTEGER		UNKNOWN
	REAL		UNKNOWN
	REAL		LOGICAL
	DOUBLE		UNKNOWN
	DOUBLE		LOGICAL
	COMPLEX		UNKNOWN
	COMPLEX		LOGICAL

Error Messages

PARAMETER NO. number MISMATCHED IN subprogram name PREV = type CURR = type
subprogram name IS NOT ICM FILE -- BINDING ABORTED
subprogram name HAS SPECIAL I/O OPERATOR AND MUST BE IN THE INTRINSIC FILE
INTRINSIC VERSION = version COMPILER VERSION = version ARE MISMATCHED
subprogram name CONTAINS ARRAY FORMAT I/O USING OUTDATED .SETAF INTRINSIC.
RECOMPILE subprogram name
CODE FILE SIZE (BLOCKS.AREA) MUST BE ENLARGED FOR THIS PROGRAM
VARIABLE DIMENSION MUST BE INTEGER
VARIABLE DIMENSION MUST BE DUMMY OR IN COMMON
ELEMENTS OF DIFFERENT COMMON BLOCKS EQUIVALENCED
ELEMENTS OF SAME COMMON BLOCK EQUIVALENCED
PARAMETER MAY NOT BE INITIALIZED
IMPROPERLY FORMED IMPLIED DO LOOP
DO LOOP INDEX USED IN TWO NESTED IMPLIED DO LOOPS
THE REFERENCE TO THE ARRAY array name DOES NOT USE THE DO LOOP INDEX
variable name
BAD DO
THE SUBSCRIPT POLYNOMIAL FOR THE ARRAY array name IS OUT OF RANGE
ONLY COMMON MAY BE INITIALIZED IN BLOCK DATA
COMMON MAY ONLY BE INITIALIZED IN BLOCK DATA
ARRAY NOT DIMENSIONED
INTEGER CONSTANT EXPECTED
INTEGER SCALAR EXPECTED
TOO MANY SUBSCRIPTS
TOO FEW SUBSCRIPTS
LEFT PARENTHESIS EXPECTED
THE REFERENCE TO array name DOES NOT USE ALL PENDING DO LOOP INDICES
UNSUBSCRIPTED ARRAY REFERENCE IN IMPLIED DO LOOP
SCALAR OR ARRAY REFERENCE REQUIRED
UNEXPECTED IMPLIED DO LOOP END
UNEXPECTED DATA LIST ELEMENT
REPEAT FACTOR OUT OF RANGE - MAX IS number
ILLEGAL COMPLEX CONSTANT
COMMA EXPECTED
RIGHT PARENTHESIS EXPECTED
TOO MANY CONSTANTS
TOO FEW CONSTANTS
COMMA OR RIGHT PARENTHESIS EXPECTED
SIMPLE VARIABLE ENCOUNTERED IN IMPLIED DO
CONSTANT EXPECTED
IMPLIED DO'S NESTED DEEPER THAN 15
ONLY DELIMITERS ALLOWED HERE
EQUIVALENCE MAY NOT EXTEND COMMON LEFT
variable name IS IN COMMON BLOCK AND MAY NOT BE INITIALIZED IN BLOCK DATA
variable name IS NOT IN COMMON AND MAY NOT BE INITIALIZED IN BLOCK DATA
ARITHMETIC OPERAND REQUIRED BETWEEN previous operator and current operator
ILLEGAL COMBINATION OF type and type OPERANDS
COMPLEX ILLEGAL FOR operator
OPERATOR EXPECTED
MISSING RIGHT PARENTHESIS
EXPRESSION TOO NESTED
SUBSCRIPT EXPECTED
ILLEGAL SUBROUTINE REFERENCE
NAMELIST NAME ILLEGAL IN EXPRESSION

D. SAMPLE COMPILATION AND BIND LISTINGS

The following is an explanation of items appearing on compilation and bind listings. The letters correspond to the circled letters on the sample listings which appear on the following pages.

A. Compiler generated line number.

B. Compiler release version, date, and time.

C. Name of compiled program - ⟨*program name*⟩ appearing on ?COMPILE card.

D. Day, calendar date, and time of compilation.

E. User's source code.

F. User's sequence numbers.

G. Code displacement numbers.

H. Subprogram name.

I. The number of bytes of code compiled for the subprogram.

J. The elapsed compile time for the subprogram.

The following additional source listing information is provided when $MAP is included in the control cards:

K. MAP header.

L. Variable names declared or referenced.

M. Types of variable, subprogram, or label.

N. Class of variable or subroutine.

O. Dimension information for variable.

P. Common block name containing variable (blank if local).

Q. Relative address assigned to variable, block.

R. Pertinent information concerning variable.

S. Names of subprograms referenced.

T. Number of arguments passed to subroutine.

U. Labels found in subprogram.

V. Code/data displacement of labels found.

W. Common blocks declared.

The following letters and meanings apply to the CODE AND DATA MAPPING (bind phase) of the listing:

AA. Name of subprograms bound.

BB. Number of dummy arguments required.

CC. Relative entry address.

DD. Actual entry address.

EE. Size of code or data segment in bits, bytes, and words.

FF. Class of subprogram of data segment.

GG. Date and time of intrinsic compilation.

HH. Name of source file containing intrinsics.

II. Name of data segment bound.

JJ. Actual segment number of data segments.

KK. Memory requirement statistics.

LL. Interpreter requested.

MM. Total disk space required for the code file. If more than 700 disk sectors are required, a ?FILE card with the following form must be used to increase the amount of disk space:

```
?C0 program.name FORTRAN
?F1 CODE BLOCKS.AREA= integer
```

NN. Elapsed bind time.

OO. Elapsed time for complete compilation.

```
BURROUGHS B1800/B1700 FORTRAN COMPILER , MARK 7.0 RA 10/24/77 17:11 ,      MONDAY    12/19/77    08:40 AM
        /(NEAL)  /EXAMPLE0,
        :$MAP                                                                : 00000100              0EXAMPLE0
        :FILE  5=MORTGAGE/DATA,UNIT=READER                                   : 00001000              0EXAMPLE0
        :FILE  6=PAYMENT/SCHEDULE,UNIT=PRINTER                               : 00001100              0EXAMPLE0
A)0003  :      COMMON/BLOCK/ IYR,AMOUNT,AE,JSM,JSY                           : 00001200    000075    0EXAMPLE0
   0004 :      INTEGER PAGE,OUT,IN                                           : 00001300    000075    0EXAMPLE0
   0005 :      REAL              AS(480), AE,     RT, WORK, WSAVE, FINT      : 00001400    000075    0EXAMPLE0
   0006 :      INTEGER A  MONTH                                              : 00001500    000075    0EXAMPLE0
   0007 :      DIMENSION AMON(12)                                            : 00001600    000075    0EXAMPLE0
   0008 :      DATA MAXLIN,OUT,IN/50,6,5/                                    : 00001650    000075    0EXAMPLE0
        :C                                                                   : 00001700              0EXAMPLE0
        :C                                                                   : 00001800              0EXAMPLE0
        :C        IYR       NUMBER OF YEARS                                  : 00001900              0EXAMPLE0
        :C        AMOUNT    STARTING AMOUNT                                  : 00002000              0EXAMPLE0
        :C        AE        INTEREST                                         : 00002100              0EXAMPLE0
        :C        JSM       MONTH TO START COMPUTING                         : 00002200              0EXAMPLE0
        :C        JSY       YEAR TO START COMPUTING                          : 00002300              0EXAMPLE0
        :C                                                                   : 00002400              0EXAMPLE0
        :C                                                                   : 00002500              0EXAMPLE0
   0009 :1     PAGE = I                                                      : 00002600    000075    0EXAMPLE0
   0010 :      CALL INPUT(IN,&999999)                                        : 00002700    000329    0EXAMPLE0
   0011 :      WRITE ( 6, 70) AMOUNT, AE, IYR, PAGE                          : 00002800    000496    0EXAMPLE0
   0012 :      JSM = JSM - 1                                                 : 00002900    001137    0EXAMPLE0
   0013 :      MONTH = 12*IYR                                                : 00003000    001214    0EXAMPLE0
   0014 :      RT = AE    /1200.0                                            : 00003100    001291    0EXAMPLE0
   0015 :      WORK = 1.0/(1.0 + RT)                                         : 00003200    001394    0EXAMPLE0
   0016 :      WSAVE = WORK                                                  : 00003300    001545    0EXAMPLE0
   0017 :      TEQU = 0.0                                                    : 00003400    001583    0EXAMPLE0
   0018 :      DO 10 I = 1, MONTH                                            : 00003500    001652    0EXAMPLE0
   0019 :      AS(I) = (1.0 - WSAVE )/RT                                     : 00003600    001684    0EXAMPLE0
   0020 : 10   WSAVE = WSAVE*WORK                                            : 00003700    001882    0EXAMPLE0
   0021 :      WORK = AMOUNT/AS (MONTH)                                      : 00003800    002115    0EXAMPLE0
   0022 :      LINE = 0                                                      : 00003900    002267    0EXAMPLE0
   0023 :      WSAVE = AMOUNT                                                : 00004000    002299    0EXAMPLE0
   0024 :      DO 50 I = 1, MONTH                                            : 00004100    002348    0EXAMPLE0
   0025 :      IF (LINE .LE. MAXLIN)  GOTO  20                               : 00004200    002380    0EXAMPLE0
   0027 :      LINE = 0                                                      : 00004300    002466    0EXAMPLE0
   0028 :      PAGE = PAGE + 1                                               : 00004400    002498    0EXAMPLE0
   0029 :      WRITE(OUT, 70 ) AMOUNT, AE, IYR, PAGE                         : 00004500    002575    0EXAMPLE0
   0030 : 20   FINT = WSAVE*RT                                               : 00004600    003228    0EXAMPLE0
   0031 :      JMO = JSM + 1                                                 : 00004700    003317    0EXAMPLE0
   0032 :      JYR = JSY + ( JMO / 12 )                                      : 00004800    003394    0EXAMPLE0
   0033 :      JMO = 1 + JMO - (JMO/12 ) * 12                                : 00004900    003486    0EXAMPLE0
   0034 :      II = MONTH - I                                                : 00005000    003641    0EXAMPLE0
   0035 :      IF (II .NE. 0 ) GOTO 30                                       : 00005100    003730    0EXAMPLE0
   0037 :      WSAVE = 0.0                                                   : 00005200    003785    0EXAMPLE0
   0038 :      GO TO 40                                                      : 00005300    003854    0EXAMPLE0
   0039 : 30   WSAVE = WORK*AS(II)                                           : 00005400    003881    0EXAMPLE0
   0040 : 40   EQU = WORK - FINT                                             : 00005500    004033    0EXAMPLE0
   0041 :      TEQU = TEQU + EQU                                             : 00005600    004122    0EXAMPLE0
   0042 :      WRITE (6,80)AMONTH(JMO),JYR, WORK, SAVE, FINT, EQU, TEQU      : 00005700    004211    0EXAMPLE0
   0043 : 50   LINE = LINE + 1                                               : 00005800    005311    0EXAMPLE0
   0044 :100   GO  TO  1                                                     : 00005900    005532    0EXAMPLE0
   0045 : 70  0FORMAT (42H1AMORTIZATION SCHEDULE FOR A MORTGAGE OF $,F8.2," AT ",: 00006000    005559    0EXAMPLE0
   0045 :     1F7.4,"% PER YEAR FOR ",I2," YEARS  PAGE = ",I2,/18X,"MONTHLY",4X, : 00006100    005559    0EXAMPLE0
   0045 :     2"UNPAID",5X,"INTEREST",4X,"EQUITY",5X,"TOTAL"/" MONTH ** YEAR ** P: 00006200    005559    0EXAMPLE0
   0045 :     3AYMENT ** BALANCE ** PAYMENT ** PAYMANT ** EQUITY"/)              : 00006300    005559    0EXAMPLE0
   0046 :80    FORMAT (1X,A4,I9,5F11.2)                                      : 00006400    005559    0EXAMPLE0
   0047 :99999  STOP                                                         : 00006500    005559    0EXAMPLE0
   0048 :      END                                                           : 00006700    005618    0EXAMPLE0

SYMBOLIC REFERENCE INFORMATION
-------- --------- -----------
VARIABLES (LOCAL BASE SEG.=  2)   K
NOTE. DATA ADDRESSES IN THE REFERENCE TABLE FOR COMMON AND LOCAL
      ITEMS WILL APPEAR DIFFERENT THAN THOSE SHOWN IN A LISTING OF
      CODE. TO CORRELATE THE CODE LIST ADDRESS FOR COMMON BLOCK
      ITEMS THE COMPILE ASSIGNED SEGMENT ASSIGNED FOR THE BLOCK
      MUST BE SUBTRACTED FROM THE CODE LIST SEGMENT NUMBER.
      FOR LOCAL ITEMS THE COMPILE ASSIGNED LOCAL BASE SEGMENT
      MUST BE SUBTRACTED.
L)IDENT  M)TYPE    N)CLASS   O)DIM/ ELEM  P)COMMON    ADDR (SEG, DISP)   NOTES
IYR      INTEGER   SCALAR                 /BLOCK /        0, 000        GLOBAL
AMOUNT   REAL      SCALAR                 /BLOCK /        0, 001        GLOBAL
AE       REAL      SCALAR                 /BLOCK /        0, 002        GLOBAL
JSM      INTEGER   SCALAR                 /BLOCK /        0, 003        GLOBAL
JSY      INTEGER   SCALAR                 /BLOCK /        0, 004        GLOBAL
PAGE     INTEGER   SCALAR                                 1, 237        LOCAL
OUT      INTEGER   SCALAR                                 1, 246        LOCAL
IN       INTEGER   SCALAR                                 1, 238        LOCAL
AS       REAL      ARRAY     1/480                        0, 000        LOCAL
RT       REAL      SCALAR                                 1, 240        LOCAL
WORK     REAL      SCALAR                                 1, 241        LOCAL
WSAVE    REAL      SCALAR                                 1, 242        LOCAL
FINT     REAL      SCALAR                                 1, 247        LOCAL
AMON     REAL      ARRAY     1/12                         1, 224        LOCAL, UNINITIALIZED
MAXLIN   INTEGER   SCALAR                                 1, 245        LOCAL
I        INTEGER   SCALAR                                 1, 236        LOCAL
MONTH    INTEGER   SCALAR                                 1, 239        LOCAL
TEQU     REAL      SCALAR                                 1, 243        LOCAL
LINE     INTEGER   SCALAR                                 1, 244        LOCAL
```

```
JMO       INTEGER   SCALAR                              1, 248      LOCAL
JYR       INTEGER   SCALAR                              1, 249      LOCAL
II        INTEGER   SCALAR                              1, 250      LOCAL
EQU       REAL      SCALAR                              1, 251      LOCAL
EQU       REAL      SCALAR                              1, 252      LOCAL, UNINITIALIZED
SAVE      REAL      SCALAR                              1, 253      LOCAL, UNINITIALIZED

SUBPROGRAMS                 (N)       (T)       (R)
(S)IDENT  (M)TYPE     CLASS     ARG   NOTES
AMONTH    INTEGER   FCN             SYSTEM OR USER ROUTINE
.FTMAK              SUBRTN          SYSTEM ROUTINE
INPUT               SUBRTN      2   SYSTEM OR USER ROUTINE
.ISFW               SUBRTN          SYSTEM ROUTINE
.WRSF               SUBRTN          SYSTEM ROUTINE
.WISF               SUBRTN          SYSTEM ROUTINE
.TFW                SUBRTN          SYSTEM ROUTINE
.PAUSE              SUBRTN          SYSTEM ROUTINE

(U)LABELS     (M)         (V)
  LABEL     TYPE      ADDRESS
      1   CONTROL       291
  99999   CONTROL      5559
     70   FORMAT       1022
     10   DO END       1882
     50   DO END       5311
     20   CONTROL      3228
     30   CONTROL      3881
     40   CONTROL      4033
     80   FORMAT       1093
    100   CONTROL      5532


(W)COMMON BLOCKS
  IDENT    ADDR(SEG,DISP)
  BLOCK          1,0

                                                                                    (I)
NO ERRORS AND NO WARNINGS IN 48 STATEMENTS,CODE EMITTED = 5618 BITS (703 BYTES)0EXAMPLE0
COMPILE TIME IS 20.7 SECONDS FOR 59 CARDS AT 171 CARDS/MINUTE.
                (J)

BURROUGHS B1800/B1700 FORTRAN COMPILER , MARK 7.0 RA 10/24/77 17:11 ,        MONDAY    12/19/77   08:40 AM
        /(NEAL)    /EXAMPLE0
    0001   :                                                                : 00006800    000075  0EXAMPLE0
    0002   :       SUBROUTINE INPUT(UNIT,*)                                 : 00006900    000075  0EXAMPLE0
    0003   :       INTEGER YEAR,UNIT                                        : 00007000    000075  INPUT
    0004   :       COMMON/BLOCK/ YEAR,AMOUNT,AE,JSM,JSY                     : 00007100    000075  INPUT
    0005   : 90    FORMAT(I2                                                : 00007200    000075  INPUT
    0005   :      A      ,2F8.2                                            : 00007300    000075  INPUT
    0005   :      B      ,I2                                               : 00007400    000075  INPUT
    0005   :      C      ,I4                                               : 00007500    000075  INPUT
    0005   :      D      ,56X)                                             : 00007600    000075  INPUT
    0006   :       READ(UNIT,90,END=100)YEAR,AMOUNT,AE,JSM,JSY              : 00007700    000075  INPUT
    0007   :       RETURN 0                                                 : 00007800    000957  INPUT
    0008   :100    RETURN 1                                                 : 00007900    000979  INPUT
    0009   :       END                                                      : 00008000    001001  INPUT

SYMBOLIC REFERENCE INFORMATION
-------- --------- -----------
VARIABLES (LOCAL BASE SEG.=  2)
NOTE. DATA ADDRESSES IN THE REFERENCE TABLE FOR COMMON AND LOCAL
      ITEMS WILL APPEAR DIFFERENT THAN THOSE SHOWN IN A LISTING OF
      CODE. TO CORRELATE THE CODE LIST ADDRESS FOR COMMON BLOCK
      ITEMS THE COMPILE ASSIGNED SEGMENT ASSIGNED FOR THE BLOCK
      MUST BE SUBTRACTED FROM THE CODE LIST SEGMENT NUMBER.
      FOR LOCAL ITEMS THE COMPILE ASSIGNED LOCAL BASE SEGMENT
      MUST BE SUBTRACTED.
IDENT     TYPE     CLASS    DIM/ ELEM    COMMON    ADDR (SEG, DISP)  NOTES
UNIT      INTEGER  SCALAR                          RCW, -03          DUMMY
YEAR      INTEGER  SCALAR               /BLOCK /      0, 000        GLOBAL
AMOUNT    REAL     SCALAR               /BLOCK /      0, 001        GLOBAL
AE        REAL     SCALAR               /BLOCK /      0, 002        GLOBAL
JSM       INTEGER  SCALAR               /BLOCK /      0, 003        GLOBAL
JSY       INTEGER  SCALAR               /BLOCK /      0, 004        GLOBAL

SUBPROGRAMS
IDENT     TYPE     CLASS    ARG  NOTES
.ISFR              SUBRTN       SYSTEM ROUTINE
.RISF              SUBRTN       SYSTEM ROUTINE
.RRSF              SUBRTN       SYSTEM ROUTINE
.TFRL              SUBRTN       SYSTEM ROUTINE

LABELS
LABEL     TYPE     ADDRESS
   90   FORMAT       512
  100   CONTROL      979


COMMON BLOCKS
IDENT    ADDR(SEG,DISP)
BLOCK          1,0

NO ERRORS AND NO WARNINGS IN 9 STATEMENTS,CODE EMITTED = 1001 BITS (126 BYTES)INPUT
COMPILE TIME IS 6.1 SECONDS FOR 13 CARDS AT 128 CARDS/MINUTE.
```

```
BURROUGHS B1800/B1700 FORTRAN COMPILER , MARK 7.0 RA 10/24/77 17:11 ,        MONDAY    12/19/77    08:40 AM
        /(NEAL)    /EXAMPLEO
    0001   :                                                                 : 00008100    000075  OEXAMPLEO
    0002   :       INTEGER FUNCTION A MONTH (  INDEX  )                      : 00008200    000075  OEXAMPLEO
    0003   :       DIMENSION MONTH(12)                                       : 00008300    000075  AMONTH
    0004   :       DATA    MONTH/                                            : 00008400    000075  AMONTH
    0004   :      1              "JAN."                                      : 00008500    000075  AMONTH
    0004   :      2             ,"FEB."                                      : 00008600    000075  AMONTH
    0004   :      3             ,"MAR."                                      : 00008700    000075  AMONTH
    0004   :      4             ,"APR."                                      : 00008800    000075  AMONTH
    0004   :      5             ,"MAY "                                      : 00008900    000075  AMONTH
    0004   :      6             ,"JUNE"                                      : 00009000    000075  AMONTH
    0004   :      7             ,"JULY"                                      : 00009100    000075  AMONTH
    0004   :      8             ,"AUG."                                      : 00009200    000075  AMONTH
    0004   :      9             ,"SEPT"                                      : 00009300    000075  AMONTH
    0004   :      A             ,"OCT."                                      : 00009400    000075  AMONTH
    0004   :      B             ,"NOV."                                      : 00009500    000075  AMONTH
    0004   :      C             ,"DEC."  /                                   : 00009600    000075  AMONTH
    0005   :       A MONTH = MONTH(INDEX)                                    : 00009700    000075  AMONTH
    0006   :       RETURN                                                    : 00009800    000247  AMONTH
    0007   :       END                                                       : 00009900    000226  AMONTH

SYMBOLIC REFERENCE INFORMATION
-------- --------- -----------
VARIABLES (LOCAL BASE SEG.=  1)
NOTE. DATA ADDRESSES IN THE REFERENCE TABLE FOR COMMON AND LOCAL
      ITEMS WILL APPEAR DIFFERENT THAN THOSE SHOWN IN A LISTING OF
      CODE. TO CORRELATE THE CODE LIST ADDRESS FOR COMMON BLOCK
      ITEMS THE COMPILE ASSIGNED SEGMENT ASSIGNED FOR THE BLOCK
      MUST BE SUBTRACTED FROM THE CODE LIST SEGMENT NUMBER.
      FOR LOCAL ITEMS THE COMPILE ASSIGNED LOCAL BASE SEGMENT
      MUST BE SUBTRACTED.
IDENT     TYPE     CLASS    DIM/ ELEM    COMMON    ADDR (SEG, DISP)  NOTES
AMONTH    INTEGER  SCALAR                          RCW, +01          LOCAL
INDEX     INTEGER  UNSPEC                          RCW, -01          DUMMY
MONTH     INTEGER  ARRAY     1/12                    0, 000          LOCAL

NO ERRORS AND NO WARNINGS IN 7 STATEMENTS,CODE EMITTED = 226 BITS (29 BYTES)AMONTH
COMPILE TIME IS 5.1 SECONDS FOR 19 CARDS AT 224 CARDS/MINUTE.


NO ERRORS AND NO WARNINGS IN 64 STATEMENTS,CODE EMITTED = 6845 BITS (856 BYTES)ALL PROGRAM UNITS
COMPILE TIME IS 32.1 SECONDS FOR 91 CARDS AT 170 CARDS/MINUTE.

                                          ***CODE AND DATA MAPPING***
PROGRAM COMPILED ON SYSTEM DISK, INTRINSICS FILE ON SYSTEM DISK
SUBPROGRAM  DATA    NUM.    STARTING  ADDRESSES              LENGTH                  CLASS     DATE-TIME COMPILED     FAMILY
    ID        ID    ARG.    COMPILE       FINAL       BITS  BYTES   WORDS                                            ID

OEXAMPLEO             0       0,75        1,75       5618    703           MAIN LINE  12/19/77 08 40 AM   (NEAL)
          /BLOCK /            0,0       SEE BELOW     180     23       5  COMMON
          LOCAL               1,0         2,0       21132   2642     587  LOCAL DATA

AMONTH                1       0,75        2,75        226     29           INTEGER FN 12/19/77 08 40 AM   (NEAL)
          LOCAL               0,0         5,0         432     54      12  LOCAL DATA

.FTMAK                2       0,75        3,75       3283    411           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

INPUT                 2       0,75        4,75       1001    126           SUBROUTINE 12/19/77 08 40 AM   (NEAL)
          /BLOCK /            0,0       SEE BELOW     180     23       5  COMMON
          LOCAL               1,0         6,0         252     32       7  LOCAL DATA

.ISFW                 1       0,75        5,75       1166    146           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.WRSF                 2       0,75        6,75       4080    510           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.WISF                 2       0,75        7,75       2006    251           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.TFW                  0       0,75        8,75        563     71           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.PAUSE                2       0,75        9,75       1660    208           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.SADDR                1       0,75       10,75        141     18           INTEGER FN 10/25/77 02 42 PM   FOR.INTRIN

.ISFR                 1       0,75       11,75       1210    152           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.RISF                 2       0,75       12,75       3319    415           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.RRSF                 2       0,75       13,75       1868    234           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.TFRL                 4       0,75       14,75        937    118           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.UNTOK                1       0,75       15,75       1324    166           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.ERROR                1       0,75       16,75      15497   1938           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.WCFMT                1       0,75       17,75       3306    414           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.STERR                1       0,75       18,75        171     22           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.TEN                  1       0,75       19,75       2006    251           REAL FN    10/25/77 02 42 PM   FOR.INTRIN

.WF                   1       0,75       20,75       7612    952           SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN
```

```
.WD                 1       0,75        21,75          7585      949          SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.WE                 1       0,75        22,75          6434      805          SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.WI                 1       0,75        23,75          1470      184          SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.WZ                 1       0,75        24,75          1055      132          SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.WRITE              0       0,75        25,75          3862      483          SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.FIXEC              0       0,75        26,75           590       74          SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.READR              1       0,75        27,75          6738      843          REAL FN    10/25/77 02 42 PM   FOR.INTRIN

.RCFMT              1       0,75        28,75          3562      446          SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.RZ                 2       0,75        29,75          1191      149          SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.XERRM              0       0,75        30,75          4323      541          SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN
        LOCAL               2,0          7,0           1728      216     48   LOCAL DATA

.STKDP              1       0,75        31,75          3192      399          SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.FT                 1       0,75        32,75           883      111          SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.FSLH               1       0,75        33,75          2324      291          SUBROUTINE 10/25/77 02 42 PM   FOR.INTRIN

.TEND               1       0,75        34,75          5801      726          DOUBLE FN  10/25/77 02 42 PM   FOR.INTRIN

DINT                1       0,75        35,75          1445      181          DOUBLE FN  10/25/77 02 42 PM   FOR.INTRIN

FINAL FORM OF COMMON BLOCKS
        /.INTR /                    1,0          9216     1152     256
        /BLOCK /                    8,0           180       23       5
        /.XDATA/                    9,0         16128     2016     448
 MEMORY REQUIREMENTS ARE                        WORDS     BYTES      BITS
    STACK SIZE (STATIC MEMORY) .......           102       459      3672
    DYNAMIC AREA REQUIRED FOR ALL
      DATA PAGES TO BE IN MEMORY .....(     1363)(     6134)(    49068)
    DYNAMIC AREA ACTUALLY ASSIGNED ...      1363       6134     49068
    DYNAMIC MEMORY LINKS (MCP ADDED) .        60        268      2143
                                          --------  --------  --------
    TOTAL BASE TO LIMIT ..............      1525       6861     54883
    RUN STRUCTURE NUCLEUS ............        62        275      2198
    DATA DICTIONARY ..................        25        110       880
    F.I.B. DICTIONARY ................         9         40       320
                                          --------  --------  --------
    TOTAL RUN STRUCTURE ..............      1619       7286     58281
    DICTIONARY CONTAINER .............         4         17       131
    MASTER CODE SEGMENT DICTIONARY ...        80        360      2880
    LARGEST CODE SEGMENT .............       431       1938     15497
    MEMORY LINKS .....................        19         82       652
    FILE SPACE (IF ALL FILES OPEN) ...       179        802      6409
                                          --------  --------  --------
    TOTAL MEMORY REQUIREMENTS ........      2330      10482     83850
 INTERPRETER REQUESTED IS           /FORTRAN   /INTERP
 CODE  REQUIRES 0122 DISK SECTORS AT 180 BYTES/SECTOR
 ELAPSED BIND TIME = 51 SECONDS.
 ELAPSED TOTAL TIME =  1 MINUTE AND 23 SECONDS.


BURROUGHS B1800/B1700 FORTRAN COMPILER , MARK 7.0 RA 10/24/77 17:11 ,        MONDAY    12/19/77    08:40 AM
        /(NEAL)    /EXAMPLEO
         :$MAP                                                                    : 00000100            0EXAMPLEO
         :FILE  5=MORTGAGE/DATA,UNIT=READER                                       : 00001000            0EXAMPLEO
         :FILE  6=PAYMENT/SCHEDULE,UNIT=PRINTER                                   : 00001100            0EXAMPLEO
   0003  :      COMMON/BLOCK/ IYR,AMOUNT,AE,JSM,JSY                               : 00001200    000075  0EXAMPLEO
   0004  :      INTEGER PAGE,OUT,IN                                               : 00001300    000075  0EXAMPLEO
   0005  :      REAL              AS(480), AE,     RT, WORK, WSAVE, FINT          : 00001400    000075  0EXAMPLEO
   0006  :      INTEGER A  MONTH                                                  : 00001500    000075  0EXAMPLEO
   0007  :      DIMENSION AMON(12)                                                : 00001600    000075  0EXAMPLEO
   0008  :      DATA MAXLIN,OUT,IN/50,6,5/                                        : 00001650    000075  0EXAMPLEO
         :C                                                                       : 00001700            0EXAMPLEO
         :C                                                                       : 00001800            0EXAMPLEO
         :C       IYR       NUMBER OF YEARS                                       : 00001900            0EXAMPLEO
         :C       AMOUNT    STARTING AMOUNT                                       : 00002000            0EXAMPLEO
         :C       AE        INTEREST                                              : 00002100            0EXAMPLEO
         :C       JSM       MONTH TO START COMPUTING                              : 00002200            0EXAMPLEO
         :C       JSY       YEAR TO START COMPUTING                               : 00002300            0EXAMPLEO
         :C                                                                       : 00002400            0EXAMPLEO
         :C                                                                       : 00002500            0EXAMPLEO
   0009  :1     PAGE = I                                                          : 00002600    000075  0EXAMPLEO
   0010  :      CALL INPUT(IN,&99999)                                             : 00002700    000329  0EXAMPLEO
   0011  :      WRITE ( 6, 70) AMOUNT, AE, IYR, PAGE                              : 00002800    000496  0EXAMPLEO
   0012  :      JSM = JSM - 1                                                     : 00002900    001137  0EXAMPLEO
   0013  :      MONTH = 12*IYR                                                    : 00003000    001214  0EXAMPLEO
   0014  :      RT = AE    /1200.0                                                : 00003100    001291  0EXAMPLEO
   0015  :      WORK = 1.0/(1.0 + RT)                                             : 00003200    001394  0EXAMPLEO
   0016  :      WSAVE = WORK                                                      : 00003300    001545  0EXAMPLEO
   0017  :      TEQU = 0.0                                                        : 00003400    001583  0EXAMPLEO
   0018  :      DO 10 I = 1, MONTH                                                : 00003500    001652  0EXAMPLEO
   0019  :      AS(I) = (1.0 - WSAVE )/RT                                         : 00003600    001684  0EXAMPLEO
   0020  : 10   WSAVE = WSAVE*WORK                                                : 00003700    001882  0EXAMPLEO
   0021  :      WORK = AMOUNT/AS (MONTH)                                          : 00003800    002115  0EXAMPLEO
   0022  :      LINE = 0                                                          : 00003900    002267  0EXAMPLEO
   0023  :      WSAVE = AMOUNT                                                    : 00004000    002299  0EXAMPLEO
```

```
    0024   :       DO 50 I = 1, MONTH                                                    : 00004100    002348  0EXAMPLE0
    0025   :       IF (LINE .LE. MAXLIN)  GOTO  20                                       : 00004200    002380  0EXAMPLE0
    0027   :       LINE = 0                                                              : 00004300    002466  0EXAMPLE0
    0028   :       PAGE = PAGE + 1                                                       : 00004400    002498  0EXAMPLE0
    0029   :       WRITE(OUT, 70 ) AMOUNT, AE, IYR, PAGE                                 : 00004500    002575  0EXAMPLE0
    0030   : 20    FINT = WSAVE*RT                                                       : 00004600    003228  0EXAMPLE0
    0031   :       JMO = JSM + 1                                                         : 00004700    003317  0EXAMPLE0
    0032   :       JYR = JSY + ( JMO / 12 )                                              : 00004800    003394  0EXAMPLE0
    0033   :       JMO = 1 + JMO - (JMO/12 ) * 12                                        : 00004900    003486  0EXAMPLE0
    0034   :       II = MONTH - I                                                        : 00005000    003641  0EXAMPLE0
    0035   :       IF (II .NE. 0 ) GOTO 30                                               : 00005100    003730  0EXAMPLE0
    0037   :       WSAVE = 0.0                                                           : 00005200    003785  0EXAMPLE0
    0038   :       GO TO 40                                                              : 00005300    003854  0EXAMPLE0
    0039   : 30    WSAVE = WORK*AS(II)                                                   : 00005400    003881  0EXAMPLE0
    0040   : 40    EQU = WORK - FINT                                                     : 00005500    004033  0EXAMPLE0
    0041   :       TEQU = TEQU + EQU                                                     : 00005600    004122  0EXAMPLE0
    0042   :       WRITE (6,80)AMONTH(JMO),JYR, WORK, SAVE, FINT, EQU, TEQU              : 00005700    004211  0EXAMPLE0
    0043   : 50    LINE = LINE + 1                                                       : 00005800    005311  0EXAMPLE0
    0044   :100    GO  TO  1                                                             : 00005900    005532  0EXAMPLE0
    0045   : 70   0FORMAT (42H1AMORTIZATION SCHEDULE FOR A MORTGAGE OF $,F8.2," AT ",: 00006000    005559  0EXAMPLE0
    0045   :      1F7.4,"% PER YEAR FOR ",I2," YEARS  PAGE = ",I2,/18X,"MONTHLY",4X, : 00006100    005559  0EXAMPLE0
    0045   :      2"UNPAID",5X,"INTEREST",4X,"EQUITY",5X,"TOTAL"/" MONTH ** YEAR ** P: 00006200    005559  0EXAMPLE0
    0045   :      3AYMENT ** BALANCE ** PAYMENT ** PAYMANT ** EQUITY"/)                 : 00006300    005559  0EXAMPLE0
    0046   :80     FORMAT (1X,A4,I9,5F11.2)                                             : 00006400    005559  0EXAMPLE0
    0047   :99999  STOP                                                                : 00006500    005559  0EXAMPLE0
    0048   :       END                                                                   : 00006700    005618  0EXAMPLE0

SYMBOLIC REFERENCE INFORMATION
-------- --------- -----------
VARIABLES (LOCAL BASE SEG.=  2)
NOTE. DATA ADDRESSES IN THE REFERENCE TABLE FOR COMMON AND LOCAL
      ITEMS WILL APPEAR DIFFERENT THAN THOSE SHOWN IN A LISTING OF
      CODE. TO CORRELATE THE CODE LIST ADDRESS FOR COMMON BLOCK
      ITEMS THE COMPILE ASSIGNED SEGMENT ASSIGNED FOR THE BLOCK
      MUST BE SUBTRACTED FROM THE CODE LIST SEGMENT NUMBER.
      FOR LOCAL ITEMS THE COMPILE ASSIGNED LOCAL BASE SEGMENT
      MUST BE SUBTRACTED.
IDENT     TYPE     CLASS    DIM/ ELEM    COMMON    ADDR (SEG, DISP)  NOTES
IYR       INTEGER  SCALAR               /BLOCK /        0, 000       GLOBAL
AMOUNT    REAL     SCALAR               /BLOCK /        0, 001       GLOBAL
AE        REAL     SCALAR               /BLOCK /        0, 002       GLOBAL
JSM       INTEGER  SCALAR               /BLOCK /        0, 003       GLOBAL
JSY       INTEGER  SCALAR               /BLOCK /        0, 004       GLOBAL
PAGE      INTEGER  SCALAR                               1, 237       LOCAL
OUT       INTEGER  SCALAR                               1, 246       LOCAL
IN        INTEGER  SCALAR                               1, 238       LOCAL
AS        REAL     ARRAY     1/480                      0, 000       LOCAL
RT        REAL     SCALAR                               1, 240       LOCAL
WORK      REAL     SCALAR                               1, 241       LOCAL
WSAVE     REAL     SCALAR                               1, 242       LOCAL
FINT      REAL     SCALAR                               1, 247       LOCAL
AMON      REAL     ARRAY     1/12                       1, 224       LOCAL, UNINITIALIZED
MAXLIN    INTEGER  SCALAR                               1, 245       LOCAL
I         INTEGER  SCALAR                               1, 236       LOCAL
MONTH     INTEGER  SCALAR                               1, 239       LOCAL
TEQU      REAL     SCALAR                               1, 243       LOCAL
LINE      INTEGER  SCALAR                               1, 244       LOCAL
JMO       INTEGER  SCALAR                               1, 248       LOCAL
JYR       INTEGER  SCALAR                               1, 249       LOCAL
II        INTEGER  SCALAR                               1, 250       LOCAL
EQU       REAL     SCALAR                               1, 251       LOCAL
EOU       REAL     SCALAR                               1, 252       LOCAL, UNINITIALIZED
SAVE      REAL     SCALAR                               1, 253       LOCAL, UNINITIALIZED

SUBPROGRAMS
IDENT     TYPE     CLASS    ARG  NOTES
AMONTH    INTEGER  FCN           SYSTEM OR USER ROUTINE
.FIMAK             SUBRTN        SYSTEM ROUTINE
INPUT              SUBRTN   2    SYSTEM OR USER ROUTINE
.ISFW              SUBRTN        SYSTEM ROUTINE
.WRSF              SUBRTN        SYSTEM ROUTINE
.WISF              SUBRTN        SYSTEM ROUTINE
.TFW               SUBRTN        SYSTEM ROUTINE
.PAUSE             SUBRTN        SYSTEM ROUTINE

LABELS
LABEL    TYPE     ADDRESS
    1    CONTROL     291
99999    CONTROL    5559
   70    FORMAT     1022
   10    DO END     1882
   50    DO END     5311
   20    CONTROL    3228
   30    CONTROL    3881
   40    CONTROL    4033
   80    FORMAT     1093
  100    CONTROL    5532


COMMON BLOCKS
IDENT    ADDR(SEG,DISP)
BLOCK          1,0

NO ERRORS AND NO WARNINGS IN 48 STATEMENTS,CODE EMITTED = 5618 BITS (703 BYTES)0EXAMPLE0
COMPILE TIME IS 20.7 SECONDS FOR 59 CARDS AT 171 CARDS/MINUTE.
```

SAMPLE COMPILATION AND BIND LISTINGS

```
BURROUGHS B1800/B1700 FORTRAN COMPILER , MARK 7.0 RA 10/24/77 17:11 ,        MONDAY    12/19/77    08:40 AM
        /(NEAL)    /EXAMPLEO
    0001   :
    0002   :       SUBROUTINE INPUT(UNIT,*)                                  : 00006800    000075  0EXAMPLEO
    0003   :       INTEGER YEAR,UNIT                                         : 00006900    000075  0EXAMPLEO
    0004   :       COMMON/BLOCK/ YEAR,AMOUNT,AE,JSM,JSY                      : 00007000    000075  INPUT
    0005   : 90    FORMAT(I2                                                 : 00007100    000075  INPUT
    0005   :      A      ,2F8.2                                             : 00007200    000075  INPUT
    0005   :      B      ,I2                                                : 00007300    000075  INPUT
    0005   :      C      ,I4                                                : 00007400    000075  INPUT
    0005   :      D      ,56X)                                              : 00007500    000075  INPUT
    0006   :       READ(UNIT,90,END=100)YEAR,AMOUNT,AE,JSM,JSY               : 00007600    000075  INPUT
    0007   :       RETURN 0                                                  : 00007700    000075  INPUT
    0008   :100    RETURN 1                                                  : 00007800    000957  INPUT
    0009   :       END                                                       : 00007900    000979  INPUT
                                                                             : 00008000    001001  INPUT

SYMBOLIC REFERENCE INFORMATION
-------- --------- -----------
VARIABLES (LOCAL BASE SEG.=  2)
NOTE. DATA ADDRESSES IN THE REFERENCE TABLE FOR COMMON AND LOCAL
      ITEMS WILL APPEAR DIFFERENT THAN THOSE SHOWN IN A LISTING OF
      CODE. TO CORRELATE THE CODE LIST ADDRESS FOR COMMON BLOCK
      ITEMS THE COMPILE ASSIGNED SEGMENT ASSIGNED FOR THE BLOCK
      MUST BE SUBTRACTED FROM THE CODE LIST SEGMENT NUMBER.
      FOR LOCAL ITEMS THE COMPILE ASSIGNED LOCAL BASE SEGMENT
      MUST BE SUBTRACTED.
IDENT     TYPE     CLASS    DIM/ ELEM    COMMON     ADDR (SEG, DISP)  NOTES
UNIT      INTEGER  SCALAR                              RCW, -03       DUMMY
YEAR      INTEGER  SCALAR                /BLOCK /        0, 000       GLOBAL
AMOUNT    REAL     SCALAR                /BLOCK /        0, 001       GLOBAL
AE        REAL     SCALAR                /BLOCK /        0, 002       GLOBAL
JSM       INTEGER  SCALAR                /BLOCK /        0, 003       GLOBAL
JSY       INTEGER  SCALAR                /BLOCK /        0, 004       GLOBAL

SUBPROGRAMS
IDENT     TYPE     CLASS    ARG  NOTES
.ISFR              SUBRTN        SYSTEM ROUTINE
.RISF              SUBRTN        SYSTEM ROUTINE
.RRSF              SUBRTN        SYSTEM ROUTINE
.TFRL              SUBRTN        SYSTEM ROUTINE

LABELS
LABEL    TYPE     ADDRESS
   90  FORMAT       512
  100  CONTROL      979


COMMON BLOCKS
IDENT    ADDR(SEG,DISP)
BLOCK         1,0

NO ERRORS AND NO WARNINGS IN 9 STATEMENTS,CODE EMITTED = 1001 BITS (126 BYTES)INPUT
COMPILE TIME IS 6.1 SECONDS FOR 13 CARDS AT 128 CARDS/MINUTE.


BURROUGHS B1800/B1700 FORTRAN COMPILER , MARK 7.0 RA 10/24/77 17:11 ,        MONDAY    12/19/77    08:40 AM
        /(NEAL)    /EXAMPLEO
    0001   :
    0002   :       INTEGER FUNCTION A MONTH (  INDEX  )                      : 00008100    000075  0EXAMPLEO
    0003   :       DIMENSION MONTH(12)                                       : 00008200    000075  0EXAMPLEO
    0004   :       DATA   MONTH/                                             : 00008300    000075  AMONTH
    0004   :      1              "JAN."                                      : 00008400    000075  AMONTH
    0004   :      2             ,"FEB."                                      : 00008500    000075  AMONTH
    0004   :      3             ,"MAR."                                      : 00008600    000075  AMONTH
    0004   :      4             ,"APR."                                      : 00008700    000075  AMONTH
    0004   :      5             ,"MAY "                                      : 00008800    000075  AMONTH
    0004   :      6             ,"JUNE"                                      : 00008900    000075  AMONTH
    0004   :      7             ,"JULY"                                      : 00009000    000075  AMONTH
    0004   :      8             ,"AUG."                                      : 00009100    000075  AMONTH
    0004   :      9             ,"SEPT"                                      : 00009200    000075  AMONTH
    0004   :      A             ,"OCT."                                      : 00009300    000075  AMONTH
    0004   :      B             ,"NOV."                                      : 00009400    000075  AMONTH
    0004   :      C             ,"DEC."  /                                   : 00009500    000075  AMONTH
    0005   :       A MONTH = MONTH(INDEX)                                    : 00009600    000075  AMONTH
    0006   :       RETURN                                                    : 00009700    000075  AMONTH
    0007   :       END                                                       : 00009800    000247  AMONTH
                                                                             : 00009900    000226  AMONTH

SYMBOLIC REFERENCE INFORMATION
-------- --------- -----------
VARIABLES (LOCAL BASE SEG.=  1)
NOTE. DATA ADDRESSES IN THE REFERENCE TABLE FOR COMMON AND LOCAL
      ITEMS WILL APPEAR DIFFERENT THAN THOSE SHOWN IN A LISTING OF
      CODE. TO CORRELATE THE CODE LIST ADDRESS FOR COMMON BLOCK
      ITEMS THE COMPILE ASSIGNED SEGMENT ASSIGNED FOR THE BLOCK
      MUST BE SUBTRACTED FROM THE CODE LIST SEGMENT NUMBER.
      FOR LOCAL ITEMS THE COMPILE ASSIGNED LOCAL BASE SEGMENT
      MUST BE SUBTRACTED.
IDENT     TYPE     CLASS    DIM/ ELEM    COMMON     ADDR (SEG, DISP)  NOTES
AMONTH    INTEGER  SCALAR                              RCW, +01       LOCAL
INDEX     INTEGER  UNSPEC                              RCW, -01       DUMMY
MONTH     INTEGER  ARRAY     1/12                        0, 000       LOCAL
```

```
NO ERRORS AND NO WARNINGS IN 7 STATEMENTS,CODE EMITTED = 226 BITS (29 BYTES)AMONTH
COMPILE TIME IS 5.1 SECONDS FOR 19 CARDS AT 224 CARDS/MINUTE.

NO ERRORS AND NO WARNINGS IN 64 STATEMENTS,CODE EMITTED = 6845 BITS (856 BYTES)ALL PROGRAM UNITS
COMPILE TIME IS 32.1 SECONDS FOR 91 CARDS AT 170 CARDS/MINUTE.

                                       ***CODE AND DATA MAPPING***
PROGRAM COMPILED ON SYSTEM DISK, INTRINSICS FILE ON SYSTEM DISK
SUBPROGRAM  DATA   NUM.    STARTING  ADDRESSES             LENGTH              CLASS      DATE-TIME COMPILED      FAMILY
    ID       ID    ARG.    COMPILE     FINAL       BITS  BYTES  WORDS                                              ID
   (AA)     (II)   (BB)     (CC)       (DD)              (EE)               (FF)              (GG)
OEXAMPLEO           0       0,75       1,75        5618    703          MAIN LINE  12/19/77 08 40 AM      (NEAL)
          /BLOCK /          0,0      SEE BELOW      180     23      5   COMMON
          LOCAL             1,0        2,0        21132   2642    587   LOCAL DATA

AMONTH              1       0,75       2,75         226     29          INTEGER FN 12/19/77 08 40 AM      (NEAL)
          LOCAL             0,0        5,0          432     54     12   LOCAL DATA

.FTMAK              2       0,75       3,75        3283    411          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

INPUT               2       0,75 (JJ)  4,75        1001    126          SUBROUTINE 12/19/77 08 40 AM      (NEAL)
          /BLOCK /          0,0      SEE BELOW      180     23      5   COMMON
          LOCAL             1,0        6,0          252     32      7   LOCAL DATA                         (HH)

.ISFW               1       0,75       5,75        1166    146          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.WRSF               2       0,75       6,75        4080    510          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.WISF               2       0,75       7,75        2006    251          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.TFW                0       0,75       8,75         563     71          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.PAUSE              2       0,75       9,75        1660    208          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.SADDR              1       0,75      10,75         141     18          INTEGER FN 10/25/77 02 42 PM      FOR.INTRIN

.ISFR               1       0,75      11,75        1210    152          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.RISF               2       0,75      12,75        3319    415          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.RRSF               2       0,75      13,75        1868    234          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.TFRL               4       0,75      14,75         937    118          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.UNTOK              1       0,75      15,75        1324    166          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.ERROR              1       0,75      16,75       15497   1938          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.WCFMT              1       0,75      17,75        3306    414          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.STERR              1       0,75      18,75         171     22          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.TEN                1       0,75      19,75        2006    251          REAL FN    10/25/77 02 42 PM      FOR.INTRIN

.WF                 1       0,75      20,75        7612    952          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.WD                 1       0,75      21,75        7585    949          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.WE                 1       0,75      22,75        6434    805          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.WI                 1       0,75      23,75        1470    184          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.WZ                 1       0,75      24,75        1055    132          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.WRITE              0       0,75      25,75        3862    483          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.FIXEC              0       0,75      26,75         590     74          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.READR              1       0,75      27,75        6738    843          REAL FN    10/25/77 02 42 PM      FOR.INTRIN

.RCFMT              1       0,75      28,75        3562    446          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.RZ                 2       0,75      29,75        1191    149          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.XERRM              0       0,75      30,75        4323    541          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN
          LOCAL             2,0        7,0         1728    216     48   LOCAL DATA

.STKDP              1       0,75      31,75        3192    399          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.FT                 1       0,75      32,75         883    111          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.FSLH               1       0,75      33,75        2324    291          SUBROUTINE 10/25/77 02 42 PM      FOR.INTRIN

.TEND               1       0,75      34,75        5801    726          DOUBLE FN  10/25/77 02 42 PM      FOR.INTRIN

DINT                1       0,75      35,75        1445    181          DOUBLE FN  10/25/77 02 42 PM      FOR.INTRIN
```

```
                                                          (EE)
FINAL FORM OF COMMON BLOCKS                    (JJ)
        /.INTR /                               1,0          9216      1152      256
        /BLOCK /                               8,0           180        23        5
        /.XDATA/                               9,0         16128      2016      448
 MEMORY REQUIREMENTS ARE                        WORDS       BYTES      BITS
    STACK SIZE (STATIC MEMORY) .......           102         459      3672
    DYNAMIC AREA REQUIRED FOR ALL
      DATA PAGES TO BE IN MEMORY .....(         1363)(       6134)(    49068)
    DYNAMIC AREA ACTUALLY ASSIGNED ...          1363        6134     49068
    DYNAMIC MEMORY LINKS (MCP ADDED) .            60         268      2143
                                              --------    --------  --------
    TOTAL BASE TO LIMIT ..............          1525        6861     54883.
    RUN STRUCTURE NUCLEUS ............            62         275      2198
    DATA DICTIONARY ..................            25         110       880
    F.I.8. DICTIONARY ................             9          40       320
                                              --------    --------  --------
    TOTAL RUN STRUCTURE ..............          1619        7286     58281
    DICTIONARY CONTAINER .............             4          17       131
    MASTER CODE SEGMENT DICTIONARY ...            80         360      2880
    LARGEST CODE SEGMENT .............           431        1938     15497
    MEMORY LINKS .....................            19          82       652
    FILE SPACE (IF ALL FILES OPEN) ...           179         802      6409
                                              --------    --------  --------
    TOTAL MEMORY REQUIREMENTS ........          2330       10482     83850
(MM) INTERPRETER REQUESTED IS                 /FORTRAN    /INTERP  (LL)
    CODE  REQUIRES 0122 DISK SECTORS AT 180 BYTES/SECTOR
(NN) ELAPSED BIND TIME = 51 SECONDS.
(OO) ELAPSED TOTAL TIME =  1 MINUTE AND 23 SECONDS.
```

(II) /BLOCK /

(KK)

E. STORAGE ALLOCATION

Each B 1800/B 1700 FORTRAN 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 36-bit data word. The sum of all data memory requirements cannot exceed 262,144 data words.

The following FORTRAN features require special storage configurations:

a. Simple Variables.

b. Arrays.

c. EQUIVALENCEd Data Items.

d. Elements of COMMON Storage.

The method of storage allocation corresponding to each of these features is discussed in this appendix.

The notation, [m:n], is used in this appendix to describe data word fields. The 36 accessible bits of a data word are considered to be numbered, with the leftmost bit being bit 0 and the rightmost bit being bit 35. 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, the word field shown here (bits 19 through 23) would be described by [19:5].

0	4	8	12	16	20	24	28	32
1	5	9	13	17	21	25	29	33
2	6	10	14	18	22	26	30	34
3	7	11	15	19	23	27	31	35

NOTE

All variables are initialized unless $NO INITIAL is used (see Compiler Option Control Cards in section 13).

Hexadecimal constants are used extensively in the present appendix to indicate word contents. Such constants are particularly suited to describing the value of a data word as each digit in a hexadecimal constant indicates the contents of a 4-bit field. Such fields can be visualized as columns in the preceding picture of a data word.

STORAGE ALLOCATION

SIMPLE VARIABLES

Each simple variable requires one or two words of storage, depending upon the type of the variable. Each of these simple variable types corresponds to a special internal storage configuration:

a. Integer Variables.

b. Real Variables.

c. Double Precision Variables.

d. Logical Variables.

The internal handling of variables of these four types is discussed here.

Integer Variables

An integer variable requires one word of storage. The data word corresponding to an integer variable is partitioned as follows:

Field	Contents
[0:2]	Type bits (00 indicates integer)
[2:1]	Sign bit (1 = negative, 0 = positive)
[3:33]	Integer field

Integer values are represented internally in signed-magnitude notation. The sign of the value is denoted by bit 2 of the data word involved. This bit is 0 (zero) for zero and positive values and 1 for negative values. The magnitude of the value is stored right-adjusted in bits 3 through 33 and is preceded by zeros.

For example, the internal representation of the integer 10 described using hexadecimal constant notation is:

Z00000000A

The internal representation of -10 is:

Z20000000A

Integer values of up to 8589934591 in magnitude may be stored with accuracy. A larger value would require more than 33 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
[0:2]	Type bits (01 designates real type)
[2:1]	Mantissa sign bit (1 = negative, = 0 positive)
[3:9]	Exponent (in excess-256 notation)
[12:24]	Mantissa field

Real (floating-point) values are represented internally in signed-magnitude mantissa and excess-256-exponent notation. The sign of the value is denoted by the mantissa sign bit of the data word. This bit is 0 (zero) 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 binary point assumed to the left of the mantissa field. The exponent expressed in excess-256 notation represents the power of 2 (i.e., the "exponent value") by which the mantissa (in binary form) must be multiplied to determine its actual value. In excess 256 notation, an exponent equal to 256 represents an "exponent value" of 0 (zero).

The magnitude that can be stored in real form is 2**(-256) through (2**255)-2**231); (0.863616856E-77 through 0.57896041E+77).

Double Precision Variables

A double precision variable is allocated two adjacent words of storage. The first word is identical to the data word of a real variable, with the second word considered as a 36-bit extension to the right of the mantissa part of the first word. The two data words corresponding to a double precision variable are partitioned as follows:

Field	Contents
[0:2]	Type bits (11 designates double precision type)
[2:1]	Mantissa sign bit (1 = negative, 0 = positive)
[3:9]	Exponent (in excess-256 notation)
[12:60]	Mantissa field

Double precision values are represented internally in signed-magnitude mantissa and excess-256-exponent notation in the identical manner as described above for real variables. The magnitude of the mantissa is stored left-normalized within the 60-bit mantissa field.

The maximum magnitude that can be stored in double precision form is the same as for a real variable.

Logical Variables

A logical variable requires one word of storage. The data word corresponding to a logical variable is partitioned as follows:

Field	Contents
[0:2]	Type bits (00 designates integer)
[2:33]	Unused
[35:1]	Value bit

The leftmost 35 accessible bits of the data word containing the value of a logical variable are ignored, while the value represented by that variable is contained in bit 35 of the data word. This logical value is represented as follows:

When bit 35 is 1, the value of the variable is .TRUE; and
when bit 35 is 0, the value of the variable is .FALSE..

Usually, the internal representation of a logical value is identical to the representation of an integer 1 or an integer 0. However, since only the state of bit 35 is taken into consideration, the remaining 35 bits may assume any state.

For example, a logical variable may be EQUIVALENCEd to any odd-valued (i.e., 1, 3, 5, etc.) integer variable and the value of the logical variable will be .TRUE.. When EQUIVALENCEd to any even-valued (i.e., 0, 2, 4, etc.) integer variable, a logical variable will have the value .FALSE..

Complex Variables

A complex variable is allocated two adjacent words 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 words corresponding to a complex variable are partitioned as follows:

Field	Contents
First word (real part)	
[0:2]	Type bits (01 designates real type)
[2:1]	Mantissa sign bit (1 = negative, = 0 positive)
[3:9]	Exponent (in excess-256 notation)
[12:23]	Mantissa field
Second word (imaginary part)	
[0:2]	Type bits (01 designates real type)
[2:1]	Mantissa sign bit (1 = negative, = 0 positive)
[3:9]	Exponent (in excess-256 notation)
[12:24]	Mantissa field

The real and imaginary values are represented internally in signed-magnitude mantissa and excess-256-exponent notation in the identical manner as described above for real variables.

The maximum magnitude that can be stored in each word is the same as for a real variable. (See REAL VARIABLES, above).

ARRAYS

FORTRAN 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. The correspondence between the array elements and the group of internal storage words is discussed here.

A FORTRAN array of any number of declared dimensions is represented internally by a one-dimensional array (i.e., 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. Thus, 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. For one-dimensional arrays, the internal data word or word pair corresponding to each array element occurs in the same position in the internal array as the element occurs in the array.

For example, if the array A1 and A2 are declared using these statements:

```
DIMENSION A1(20)
DOUBLE PRECISION A2(20)
```

then the REAL array A1 will be allocated 20 words of storage (one word per element), and the DOUBLE PRECISION array A2 will be allocated 40 words of storage (two words per element). Therefore, the array element A1(2) will be assigned the second word of the internal array corresponding to A1, and A2(2) will be assigned the third and fourth words of the internal array corresponding to A2.

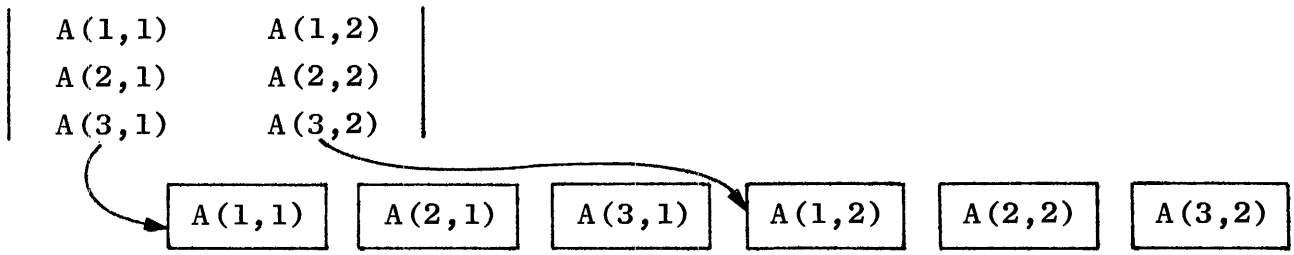
Each complex array element requires two words of storage. Thus, complex arrays are allocated twice as many internal data words as array elements.

For example, if arrays A1 and A2 are declared using these statements:

```
DIMENSION A1(20)
COMPLEX   A2(20)
```

then the real array A1 is allocated 20 words of storage (one word per element), and the complex array A2 is allocated 40 words of storage (two words per element). In addition, 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.

Arrays of more than one dimension are stored by columns into one continuous internal vector of storage words. This storage process may be visualized as occurring in the manner displayed in the example provided here.



The word(s) corresponding to each element of the two-dimensional array in this example are located in the internal array in the order shown. Beginning with element A(1,1), each successive element in the array may be thought of as being taken in the order in which they occur, proceeding down each column of the matrix, from left to right, and stored in the internal array. For three-dimensional arrays, this process is repeated for each successive layer of the array (i.e., each value of the third subscript, beginning with 1).

This, for any n-dimensional array, the elements of this array correspond in a set order to successive storage locations in the one-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 most rapidly, and so forth. As an example, the elements of the five-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.

An array A having n dimensions may be declared by means of an array declaration of the type:

A(D1,D2,D3, . . . , Dn)

The array A then contains elements of the form:

A(I1,I2,I3, . . . , In)

The position of the storage unit (i.e., the word or word pair) assigned to this array element within the corresponding internal array may be found by means of the following formula (called the array element successor function):

$$I = I1 + D1*(I2-1) + D1*D2*(I3-1) + \ldots + D1*D2*D3*\ldots*Dn-1*(In-1)$$

where I is the location index between 1 and (D1*D2*D3* . . . *Dn), inclusive. Thus, the linear index from the first address of the internal array to the array element is one less than this value (times 2 for DOUBLE PRECISION arrays).

An array having more than one dimension may be EQUIVALENCEd to a one-dimensional array and the elements of this latter array will correspond in order to the storage units assigned to the former 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 an I/O statement or as the item to be initialized in a DATA statement.

An array is stored internally in consecutive storage locations as long as the storage requirement of the array does not exceed 256 words. Arrays exceeding this size will be segmented for overlay purposes, but will appear logically consecutive. Each array segment will contain a maximum of 256 data words and will be overlayed and recalled as necessary during the execution of the program.

EQUIVALENCED DATA ITEMS

The FORTRAN EQUIVALENCE specification 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 (i.e., variable name or array element) may refer to one storage location.

The programming considerations involved with the use of the EQUIVALENCE statement are discussed here.

Single Storage Locations

The least complicated use of the EQUIVALENCE statement involves the assignment of data items requiring a single word of storage to a mutual storage location.

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

```
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 all 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 will contain equivalent changes and variables of different types will become undefined. In this example, if

INTEGER variable A is assigned a value, INTEGER array element AR(2) will be assigned the same value and REAL variable B2 will become 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 assignment statement:

```
L = .TRUE.
```

places the logical value .TRUE. into variable AL.

Multiple Storage Locations

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

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

The displayed EQUIVALENCE statement causes the REAL array A, the DOUBLE PRECISION variable D, and the REAL variable B to be assigned to identical data words. As A and D both require two data words, the first and second elements of A become equivalent to the first and second words, respectively, of the storage unit assigned to D. The variable B, requiring only one data word, will be assigned to the same location as A(1) and the first word of D.

Array Handling

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

a. An array name.

b. An array element with the same number of subscripts as contained in the declaration declaring the array.

c. An array element with one subscript from an array of more than one dimension.

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 $DE(2)
EQUIVALENCE (A,$DE), (B(1,1),C(1)),(D,F(26), E(2))
```

The first list in this EQUIVALENCE statement (i.e., (A,$DE)) causes the REAL array A and the DOUBLE PRECISION array $DE to share four storage words. The first two elements of A (i.e., A(1) and A(2)) become equivalent to the two words of the first element of $DE (i.e., $DE(1)), and the last two elements of A (i.e., A(3) and A(4)) become equivalent to the two words of the last element of $DE (i.e., $DE(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 (i.e., (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 two-dimensional array are stored internally in the manner described in the discussion of arrays in this appendix. The internal storage locations assigned to C occur in the same order as the elements of that array.

Hence, each C(I) is EQUIVALENCEd to the Ith internal element of B, and it follows that if the following two WRITE statements occur in the same program unit, identical output will be produced:

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

The final list in the sample EQUIVALENCE statement (i.e., (D,F(26), E(2))) indicates that the elements of the arrays D, F, and E are to be EQUIVALENCEd in such a manner that D(1), F(26), and the second internal element of the two-dimensional array E are to be assigned identical internal locations. Therefore, the last 25 elements of F (i.e., F(26) through F(50) become equivalent to the first 25 elements of D (i.e., D(1) through D(25)). Since E is stored internally in the manner described in the discussion of arrays in this section, equivalencing is handled in the manner illustrated below. Each of the following 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)		

Therefore, the following DATA statement would initialize the elements E(1,2), E(2,2), F(28), F(29), D(3), and D(4) with the value 6:

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

ELEMENTS OF COMMON STORAGE

The FORTRAN COMMON specification statement allows values to be communicated among program units without employing entries in SUBROUTINE and FUNCTION statement argument lists while permitting these data items to be referenced by the same or different symbolic names in each program unit. This statement also allows data initialization using the BLOCK DATA program unit. (See section 12.)

The storage handling of elements in COMMON blocks referenced in subprograms is performed in a different manner from dummy arguments in such subprograms. Each element of a COMMON block is allocated stor-

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

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. These two concepts are discussed in the following paragraphs.

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
```

The COMMON statement among the preceding statements will be assumed to provide the largest description of the size of the unlabeled COMMON block of any COMMON statement appearing in the executable program of which the program unit MSG is a part. The total size of this COMMON block is then 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 (which cannot cross a data segment boundary), 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 could be referenced, for example, by a COMMON statement within another program unit as indicated here:

```
      SUBROUTINE DUMP
      COMMON T(10)
      WRITE (6,1)T
1     FORMAT(1X,10E10.3)
      RETURN
      END
```

where T is a REAL array. The elements of this array therefore would be 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); and 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.

Additionally, the EQUIVALENCE statement may interact with the COMMON statement to expand the length of a COMMON storage block beyond the maximum size specified for the block by the COMMON statement. 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 discussion will illustrate this point.

As an example, 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
      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 two-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 K array 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.

The COMMON block could not have been extended backwards by the EQUIVALENCE statement. The following combination of statements is invalid:

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

F. DESCRIPTION OF UNFORMATTED I/O RECORDS

Each unformatted record is written as a 36-bit control word, followed by the values of the variables in the I/O variable list. Each item generates a 36-bit word (or 72-bit double-precision or complex word).

Thus, if three single-precision variables (A, B, C) were written unformatted, the record on disk would appear as one 36-bit control word, followed by a 36-bit word with the machine representation of A, followed by another 36-bit word with the machine representation of B, followed by a third 36-bit word with the machine representation of C. The 36-bit control word of an unformatted record consists of the following:

Bit	Meaning
1	When ON, indicates that the unformatted record begins in this logical record.
2	When ON, indicates that the unformatted record ends in this logical record.
3	Not used.
4-36	Length field. This is the total number of bits of all of the 36-bit words contained in the logical record, including the control word. To calculate the length field: a. Add 36 bits for each real variable, 36 bits for each integer variable, 72 bits for each double-precision variable, and 72 bits for each complex variable to be written. b. To the sum in item a above, add 36 bits for the control word.

In order to read an unformatted record, the I/O variable list of the read statement should agree in order and type with the list of the write statement originally used to write the unformatted record.

The default logical record size for disk is 180 bytes (1 segment), which is sufficient for the control word and 39 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 a control word followed by those items which did not fit into the first logical record. This process continues until the list is exhausted. If the list size for a random write exceeds the logical record size, a data error results.

G. OPTIMIZING PROGRAM EXECUTION

Increasing dynamic memory space ($DYNAMIC words) in the FORTRAN object program may result in faster program execution unless the program is I/O bound. Each FORTRAN subprogram has separate overlayable data segment(s) for local and COMMON blocks.

Each COMMON BLOCK starts a new segment. The user can determine data segmentation by placing items in separate labeled COMMON BLOCKS. If the COMMON BLOCK contains more than 256 single-precision words, another segment is created. All unlabeled COMMON BLOCKS are placed in one segment unless the sum of their elements is greater than 256 single-precision words. In general, the larger the dynamic memory size, the faster the program execution. This is not true if the dynamic memory competes with MCP overlays, the program's code segments, or other programs.

Local data for each subprogram is also placed in separate data segments. If local data in a subprogram exceeds 256 single-precision words, a new segment is created.

By default, the compiler sets DYNAMIC SPACES to 10. This allows memory links for 10 data segments. If more than 10 data segments are required in dynamic memory at the same time, modify the object program by entering the following:

```
? MODIFY program-id DYNAMIC.SPACES = n
```

To avoid overlays, work with arrays in the order in which they are stored. Accessing array elements in an order which requires a different segment for each element will increase overhead. (See Storage Allocation, appendix E.)

H. OPTIMIZING PROGRAM COMPILATION

Compilation with B 1800/B 1700 FORTRAN consists of a compile phase followed by a bind phase. The phase in which compiler files are opened is as follows (file names shown are internal file names):

Compile Phase	Bind Phase	Remarks
CARDS		
SOURCE		OPENED ONLY IF "$MERGE" USED
NEWSOURCE		OPENED ONLY IF "$NEW" USED
ICM	ICM	
LINE	LINE	
	CODE	
	FOR.INTRIN	

When compiling on B 1800/B 1700 systems with sufficiently large memory configurations, FORTRAN compilation times may be enhanced in the following ways:

a. Only one buffer is associated with each of the compiler's files. Since records are accessed in a serial manner from the CARDS, SOURCE, NEWSOURCE, and LINE files, the compiler may be modified so that additional buffers are associated with these files. This is accomplished by entering the following control statements:

```
?MODIFY FORTRAN
?FILE CARDS BUFFERS = integer
?FILE SOURCE BUFFERS = integer
?FILE NEWSOURCE BUFFERS = integer
?FILE LINE BUFFERS = integer
```

b. Compiler files may be assigned to different disk drives, thus enhancing compilation time by relieving disk arm contention. When a pack-id precedes the program-name on a compile card, each intermediate code file, as well as the object code file, is 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 compiler, the FOR.INTRIN built-in function and intrinsic file, and the FORTRAN interpreter are expected by the MCP to reside on the specified disk.

c. The size of the compiler's dynamic memory size is a factor that affects compilation time. Dynamic memory size may be increased from the default of 40000 bits by the following control statement:

```
? MODIFY FORTRAN 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.

I. FORTRAN/INTMAKER

The FORTRAN Intrinsic Maker is a program that allows user modifications to the FORTRAN intrinsic function and intrinsic file (FOR.INTRIN).

The program reads the old FOR.INTRIN file and/or user-selected ICM files and writes a new FOR.INTRIN file with the requested modifications. The input and output file names, the ICM pack and/or family names, the requested modifications, and the names of the intrinsic functions or intrinsics to be modified are included in a parameter deck with the external name of CARDS. Each option card must contain a dollar sign ($) in column 1, followed by the option name and any parameters in free-form format to column 72. Each option card may contain only one option, and each card naming an intrinsic function or intrinsic to be modified must contain only the name of that intrinsic function or intrinsic.

The allowable options are as follows:

ICM.PACK.FAMILY	This option should be followed by the family or pack and family names of the ICM files to be put into the file. If both pack and family names are specified, they must be separated by a slash ("/").
INTRIN.IN	This option should be followed by the file name of the intrinsic function and intrinsic file to be modified in standard B 1800/B 1700 naming format (default is "FOR.INTRIN").
ADD	This option should be followed (on separate cards) by the names of the intrinsic functions or intrinsics to be added to the existing file.
REMOVE	This option should be followed (on separate cards) by the names of the intrinsic functions or intrinsics to be removed from the existing file.
REPLACE	This option should be followed (on separate cards) by the names of the intrinsic functions or intrinsics to be replaced by ICM files in the existing file.

It is strongly advised that only intrinsic functions be removed or replaced.

The following example shows control cards that can be used to create ICM files.

Example:

```
? CO INTRINSICS FORTRAN LI
? DA CARDS
$ NO BIND
    REAL FUNCTION SIN (ARG)
    [ FORTRAN SOURCE ]
    END
    REAL FUNCTION F (X,Y)
    [FORTRAN SOURCE ]
    END
? END
```

There are two ICM files on disk named INTRINSICS/SIN and INTRINSICS/F. Assume that:

a. The intrinsic file to be modified is EXAMPLE/INTRIN and the output file will be named USER/EXAMPLE/INTRIN.

b. The INTRINSICS/SIN file is to replace the existing SIN intrinsic function.

c. The INTRINSICS/F file is to be added.

d. The CSQRT intrinsic function that is not used is to be removed.

The following example will accomplish these objectives:

Example:

```
? EX FORTRAN/INTMAKER
? DA CARDS
$ ICM.PACK.FAMILY INTRINSICS
$ INTRIN.IN EXAMPLE/INTRIN
$ INTRIN.OUT USER/EXAMPLE/INTRIN
$ REPLACE
SIN
$ ADD
F
$ REMOVE
CSQRT
? END
```

The program, after making all requested modifications, prints the directory of the new file.

J. COMPILER SIZE LIMITS FOR FORTRAN PROGRAMS

Item	Maximum
Executable code per subprogram	262,144 bits
Number of subprograms	1,023
Number of data segments	1,023
Number of parameters in a CALL statement	63
Number of common blocks	28-30
Record Size	2,047 characters
Variables and subprograms referenced per subprogram	16,384

Code files on the B 1000 system must be contained in one disk area. If the FORTRAN 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 > FORTRAN LI
FILE  CODE  BLOCKS.PER.AREA = < integer > ;
```

The value of < integer > must be greater than the default value of 700.

K. REMOTE FILE SCREEN FORMATTING AND FORMS MODE I/O

The following is a description of the screen formatting and forms mode I/O capabilities for remote files. The examples contained in this appendix assume the use of a TD830 series terminal. Screen formatting on other terminals can differ according to the firmware used with the terminal.

Burroughs FORTRAN permits the declaration of a remote terminal as a hardware input/output device. The following is an example of a FILE declaration statement used to declare a remote file.

```
FILE  8=RMT,  UNIT=REMOTE,  RECORD=1920
```

Only one remote file can be open at a time in a FORTRAN program. Each record of the remote file contains the number of characters declared in the RECORD attribute of the FILE declaration statement. In the preceding example, the entire screen is one record. Records are written with 80 characters per line, excluding control characters. When the end of a line is reached, the output continues on the next line.

Example:

```
      WRITE  (8,100)  (I(J),J=1,100)
100   FORMAT  (100A2)
```

In this example, the first two bytes (after the four bits of type information) of each of the first 100 elements of array I are written to the remote terminal.

NOTE

The first four bits contain type information and are stripped off before transmission to the screen.

Screen formatting characters can be sent to the terminal by using hexadecimal data initialization. Hexadecimal constants, which can only be assigned in a DATA statement, are assigned to a variable without conversion or regard to type.

Examples:

```
DATA  HOME,  BLINK  /Z00C000000,  Z018000000/
DATA  BRIGHT,  RS,  LF  /Z03F000000,  Z01E000000,  Z025000000/
```

In these examples, the first hexadecimal digit of each variable is zero since the first four bits are stripped off before transmission to the screen. The second and third hexadecimal digits contain a screen formatting character.

REMOTE FILE SCREEN FORMATTING AND FORMS MODE I/O

The variables initialized in these examples contain screen formatting characters which prescribe the following actions:

Variable	Action
HOME	Home the cursor and clear the screen.
BLINK	Initiate blink video data highlighting.
BRIGHT	Initiate bright video data highlighting.
RS	Terminate data highlighting.
LF	Move the cursor down one line.

The following is an example of a WRITE statement using the variables initialized in the preceding example:

```
    WRITE (8,100) HOME, (LF, I=1,11), BLINK, BRIGHT, RS
100 FORMAT (12A1, 38X, 2A1, 'HELP', A1)
```

These statements prescribe the following actions: 1) send the cursor to the home position and clear the screen, 2) move the cursor down eleven lines, 3) move the cursor to column 39, 4) print the word HELP with blink and bright video data highlighting, and 5) terminate the data highlighting.

A variable can be initialized with more than one formatting character.

Example:

```
DATA BLBR /Z0183F0000/
```

In this example, the screen formatting characters contained in the BLINK and BRIGHT variables are combined into the single variable BLBR. The following is an example of a WRITE statement using variables initialized in the previous examples:

```
    WRITE (8,100) HOME, (I(J), J=1,11), BLBR, RS
100 FORMAT (12A1, 38X, A2, 'HELP', A1)
```

The only change in the FORMAT statement from the previous example is that an A2 format specification is used to write one variable instead of a 2A1 format specification to write two variables.

The following example transmits a forms mode formatting character to the remote terminal:

```
    DATA FMODE /Z027E60300/
    WRITE (8,100) FMODE
100 FORMAT (A3)
```

The following DATA statement initializes two variables with the forms mode delimiting characters:

```
DATA US, RS /Z01F000000, Z01E000000/
```

This statement initializes the variable US with the left forms mode delimiter and initializes the variable RS with the right forms mode delimiter. The right forms mode delimiter in the RS variable can also be used to terminate data highlighting as shown in previous examples.

The following is a program using the variables initialized with forms mode delimiting characters:

```
      WRITE (8,100) US, RS, FMODE
100   FORMAT (37X, A1, 4X, A1, A3)
      READ (3,200) I
200   FORMAT (I4)
      STOP
      END
```

This program performs the following functions: 1) produces a 4-character forms field in the middle of the top line, 2) puts the screen in forms mode, and 3) reads a variable in integer format from the forms field. The forms mode character in the variable FMODE must be the last character written. READ operations in forms mode can only input information from the delimited forms fields. When more than one forms field is transmitted from the beginning of the first forms field, all information in all the forms fields is transmitted.

The following is a program that uses the capabilities of screen formatting and forms mode I/O for remote files. The program writes to the screen and receives input in forms mode. The input is written back to the terminal in the middle of the screen. Two forms fields are delimited and both accept input in A format.

```
FILE  9=REMO, UNIT=REMOTE, RECORD=1920
      IMPLICIT  INTEGER(A-Z)
      DATA  HOME, REV, FMODE, US, RS, CR/Z00C000000, Z00E000000,
     +            Z027E60300, Z01F000000, Z01E000000, Z00D000000/
      DATA  BLBR/Z0183F0000/
      WRITE  (9, 100) HOME, US, RS, US, RS, FMODE
  100 FORMAT (A1, 20X, A1, 4X, A1, 100X, A1, 4X, A1, A3 )
      READ  (9, 200)  I, J
  200 FORMAT (2A4)
      WRITE (9, 300)HOME, (CR, K=1, 11), BLBR, REV, I, RS, BLBR, J
  300 FORMAT(12A1, 36X, A2, A1, A4, A1, A2, A4)
      STOP
      END
```

For more information on the use of forms characters, refer to the TD730/TD830 System Reference Manual, form number 1093788.

INDEX