Burroughs

B 2500 and B 3500 SYSTEMS

FORTRAN REFERENCE MANUAL

Burroughs

B 2500 and B 3500 Systems

FORTRAN REFERENCE MANUAL

Burroughs Corporation
Detroit, Michigan 48232

$3.00

Printed in U. S. America 8-71 1030376

This revison incorporates the information
released under the following PCN:
1030376-001 (Jan 13, 1970)

Correspondence regarding this document should be forwarded using the Remarks Form at the back of the manual, or may be addressed directly to Systems Documentation, Sales Technical Services, Burroughs Corporation, 6071 Second Avenue, Detroit, Michigan 48232.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont)

TABLE OF CONTENTS (cont)

TABLE OF CONTENTS (cont)

LIST OF ILLUSTRATIONS

LIST OF TABLES

INTRODUCTION

This manual provides a complete description of the Burroughs B 2500/ B 3500 FORTRAN Compiler language.*

The FORTRAN language is designed for writing programs for scientific and engineering applications. Statements can be written in the general format of mathematical notation, thus increasing the ease of solving formula-oriented problems.

The B 2500/B 3500 FORTRAN Compiler operates under the control of the Master Control Program (MCP) and, similarly, the object code produced by the compiler is executed under the control of the MCP.

The B 2500/B 3500 FORTRAN Compiler language is based on ANSI (sometimes referred to as USASI) FORTRAN (refer to the publication: ASA X3.9-1966).

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

SECTION 1
GENERAL PROPERTIES

GENERAL.

Normally, a FORTRAN source program is prepared on punched cards. These cards are of three general types:

a. General program cards.

b. Comment cards.

c. Control and $ cards.

PROGRAM CARDS.

Program cards are used to contain FORTRAN source statements under the following limitations (see figure 1-1):

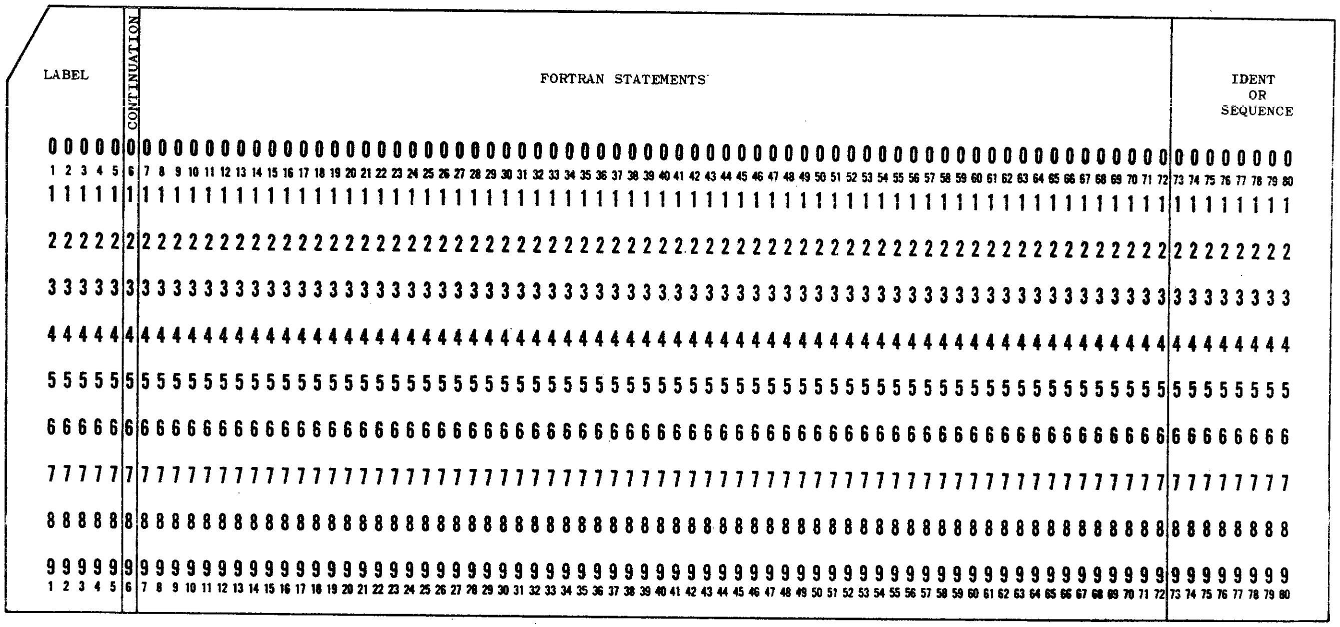


Figure 1-1. Program Card Layout

a. Columns 1-5. The label of a labeled statement consists of from one to five digits (unsigned integer) and may be placed anywhere within these columns with or without leading zeros, i.e., neither blank nor leading zeros are significant in differentiating statement labels. All labels within a program unit must be unique. The label field is ignored on all non-executable statements except for cards containing FORMAT statements.

b. Column 6. Column 6 of the initial card of a statement must be either blank or zero. Column 6 of a continuation card (any additional card after the initial card needed to contain the statement) must contain any character other than blank or zero. An unlimited number of continuation cards may follow an initial card.

c. Blank characters are significant only in column 6 of a non-comment card, in a Hollerith constant, or in a Hollerith field specification. With these exceptions, blanks may be used or omitted without affecting the interpretation of a FORTRAN statement.

d. Columns 7-72. Columns 7 through 72 contain the FORTRAN statement.

e. Columns 73-80. These columns are not interpreted by the compiler and may contain identification or sequencing information. However, this field is analyzed when changes are merged with a source tape (refer to appendix E).

f. Only one statement may be punched on a physical card.

g. A program unit must have an END statement as the final card. The sole purpose of the END statement is to inform the compiler that it has reached the end of a program unit. The END statement is a line with blanks in columns 1 through 6, the characters E, N, and D once each and in that order in columns 7 through 72, preceded by, interspersed with, or followed by blanks. It is not an executable statement; therefore, if a program attempts to execute an END statement, the program is aborted.

<u>COMMENT CARD</u>.

Comment cards are not interpreted by the compiler, but their information does appear on the compilation listing for documentation purposes. A comment card cannot be followed by a continuation card. Card punching limitations are as follows (see figure 1-2):

a. Column 1. A comment card must have the comment code, the letter C, in column 1.

b. Columns 2-72. Columns 2 through 72 may be used for comments.

c. Columns 73-80. These columns may contain identification or sequencing information or may be used with columns 2-72 to contain part of the comment.

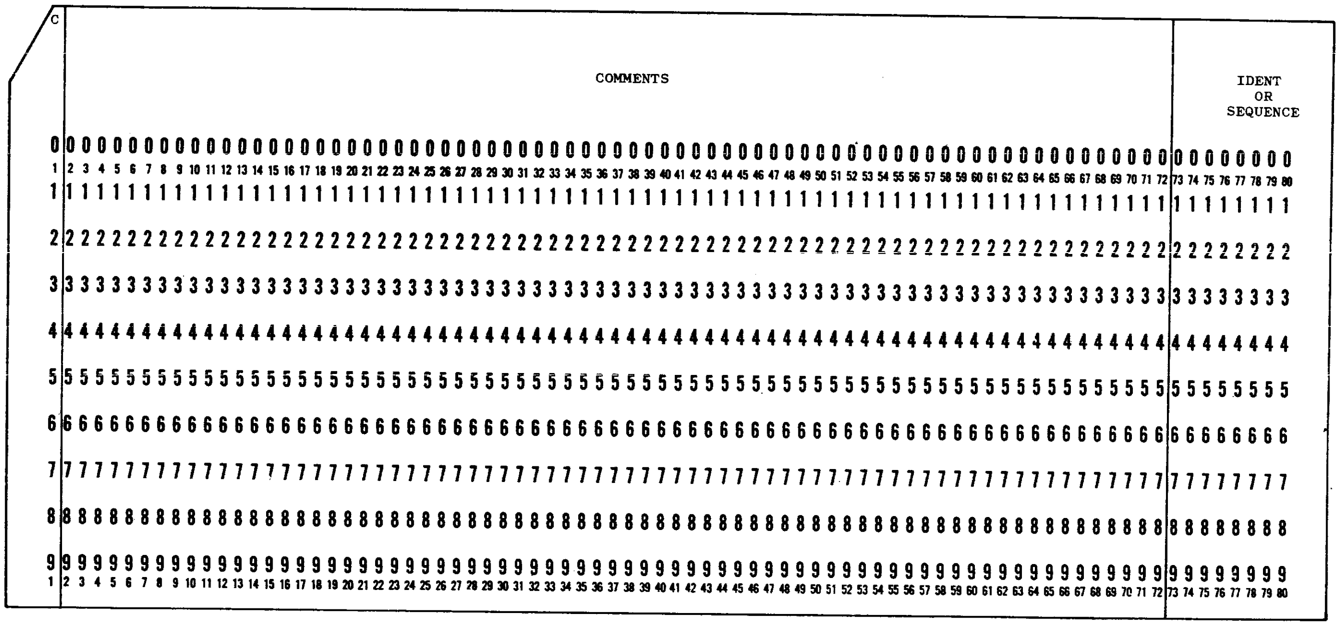


Figure 1-2. Comment Card

DECK STRUCTURE.

The basic deck structure is as follows:

? COMPILE ⟨JOBNAM⟩ FORTAN
or COMPILE ⟨JOBNAM⟩ FORTAN TO LIBRARY
or COMPILE ⟨JOBNAM⟩ FORTAN TO SYNTAX
? DATA CARDS
or DATAB CARDS

```
    or ┌? DATAB CARDS
       └HOLL
         ...┐
         ...│        Source Deck
         ...┘
    ?    END
```

Refer to appendix C for discussion of control cards and label equation cards.

SECTION 2
CHARACTER SET, CONSTANTS, VARIABLES

CHARACTER SET.

The FORTRAN character set consists of digits, letters, and special characters.

DIGITS.

A digit is any one of the following 10 characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

LETTERS.

A letter is any one of the following 26 characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.

SPECIAL CHARACTERS.

The special characters are as follows:

Character	Name
=	Equal sign
+	Plus sign
-	Minus sign
*	Asterisk
/	Slash
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Decimal point
	Blank
$	Currency symbol

The following BCL or BCD characters are recognized as alternatives to the standard FORTRAN set under the HOLL option.

FORTRAN Character	BCL Alternative
+	&
=	#
(	%
)	[

CONSTANTS.

Six basic types of constants are allowed in the FORTRAN programming language: integer, real, double precision, complex, logical, and Hollerith.

INTEGER CONSTANT.

An integer constant is formed by a string of decimal digits.

The general form is:

n
where $-99999 \leq n \leq 99999$

An integer constant is written without a decimal point or an exponent.

NOTE

Integer size may be increased to 47 digits through use of the SIZE Control Card.

Examples

```
12
-16729
36241
```

REAL CONSTANT.

A real constant is a string of decimal digits with a decimal point and, optionally, an exponent.

The general form is:

m.nEx
where m and n are strings of decimal digits, only one of which may be blank; x is a signed or unsigned 1- or 2-digit integer which is the exponent.

A real constant may be signed or unsigned.

An exponent is optional. If it is used, the letter E follows the mantissa and precedes the exponent. The exponent, if present, is interpreted such that 10^x is multiplied times the mantissa.

The range for a real datum is $-.99999999 \times 10^{99} \leq m.n \leq .99999999 \times 10^{99}$ by default. Refer to the SIZE Card in appendix C for extended ranges.

NOTE

Real size may be increased to 45 digits (mantissa size) through use of the SIZE Control Card.

<u>Examples</u>

```
56.9
.075
-253
71.32E+02
-71.32E-2
```

DOUBLE PRECISION CONSTANT.

A double precision constant is of the same form as a real constant, except that its mantissa, exponent, and sign can contain up to 20 decimal digits; and the format specifier D precedes the exponent part. If more than 20 digits are used, the mantissa is truncated to the 16 most-significant digits.

The range of a double precision constant is identical to that of a real constant.

A constant which does not have an exponent but which specifies more digits than a single precision value can maintain is not initialized as a double precision constant unless it occurs in a double precision expression.

Examples

```
12D-1
-5.36D+30
52D-07
.713D-17
```

COMPLEX CONSTANT.

A complex constant in the mathematical sense is composed of a real part and an imaginary part.

The general form is:

(m,n)
where m is the real part and n is the imaginary part.

Each of the two components may be a real constant.

Double precision components are not permitted.

Examples

Complex Constant	Mathematical Interpretation
(5,64.2)	5 + 64.2i
(0,-1)	-i
(3.5E-2,75.9)	.035 + 75.9i

NOTE

$i = \sqrt{-1}$

LOGICAL CONSTANT.

A logical constant may be either TRUE or FALSE.

The general form is:

.TRUE.
.FALSE.

<u>Examples</u>

```
.TRUE.
.FALSE.
```

HOLLERITH CONSTANT.

A Hollerith constant is a string of any valid FORTRAN characters.

The general form is:

wHs
where w is the width of the string and s is the string.

Blanks within the string must be included in the field width w.

<u>Examples</u>

```
2HbT
5HABCDE
11HJOHN SMITHE
8H*/(+)**/
```

NOTE

b represents blank

<u>VARIABLES</u>.

There are two forms of variables: simple and subscripted. Each of these is classified into six basic types: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and ALPHAnumeric. (Refer to page 6-8 for a discussion of the ALPHA type variable.)

SIMPLE VARIABLE.

A simple variable represents a single value.

The general form is:

From one to six alphanumeric characters, the first of which must be alphabetic.

A variable name with a first character of I, J, K, L, M, or N implicitly types that variable as an INTEGER variable. A variable name beginning with any other alphabetic character is implicitly typed as REAL unless otherwise defined in a Type statement.

A variable of type DOUBLE PRECISION, COMPLEX, LOGICAL, or ALPHA must be declared as such in a Type statement.

Examples

INTEGER Variables	REAL Variables
IB2	A123
J12	TSUB2
KALPHA	ZSQD
IJbK	ABbCD

NOTE

b represents blank

SUBSCRIPTED VARIABLE.

A subscripted variable refers to a particular element of an array.

The general form is:

$n(a_1,a_2,a_3)$
where n is the array name; a_1,a_2, and a_3 are arithmetic expressions which determine the values of the subscripts of the subscripted variable. An array may have one, two, or three dimensions.

A subscripted variable is named and typed according to the same rules as a simple variable.

All elements of an array must be of the same type; i.e., if N(2) is INTEGER, N(3) must also be INTEGER.

A subscript must be an integer constant or integer expression.

Subscripted variables must have their subscript bounds specified in a DIMENSION, Type, or COMMON statement prior to their first appearance in either an executable statement or in a DATA statement.

The maximum number of elements per array is 9999. The maximum number of dimensions is three.

Multi-dimensioned arrays are stored with the left-most subscript varying most rapidly and the right-most subscript varying least rapidly.

Examples

```
    B(I)
    GSUB(8*K,L,+3)
    DMIN(I,J,K)
```

SECTION 3
EXPRESSIONS

GENERAL.

An expression is any constant, variable, or function reference, or a combination of these separated by operators or parentheses. There are two types of expressions:

a. Arithmetic.

b. Logical.

ARITHMETIC EXPRESSION.

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

The general form is:

> Any constant, variable, or function reference, or a combination of these separated by operators. Parentheses may be used for grouping within an expression.

An arithmetic expression may contain the following arithmetic operators:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
( )	Grouping operator pair

Arithmetic expressions may be connected by arithmetic operators to form other arithmetic expressions, provided two operators do not appear in sequence and an arithmetic operator is not erroneously assumed present. Examples of invalid arithmetic expressions are:

```
A/-B
(A+2)(B-3)
A+-B
```

Any arithmetic expression can be enclosed in parentheses.

All actual arguments of a function reference are evaluated before the function is evaluated.

Parentheses may be used in an arithmetic expression to denote the order in which operations are to be performed. Parentheses have first precedence in determining the order of evaluation and, when nested parentheses occur, evaluation proceeds from the innermost to outermost set. There is no limit on the number of sets that may be nested together.

The precedence order (or hierarchy) used in evaluating an arithmetic expression is as follows:

a. Primary (unary + and -). Highest priority.
b. Exponentiation.
c. Multiplication and division.
d. Addition and subtraction. Lowest priority.

The precedence for successive operators of the same level is from left to right, e.g., A**B**C is evaluated as (A**B)**C.

For the operation A**B, the valid combinations and results are noted in table 3-1.

Any element may be combined with any other element through use of any of the arithmetic operators except exponentiation. The resultant type is listed in table 3-2 for A OP B, where A and B are operands and OP is either +, -, *, or /.

<u>Examples</u>

```
B
2.316
```

```
K + 1
(X + A(I,J,L) - SIN(Y(K)))
X - C + Y(I,L) * 16.397
```

Table 3-1

Resultant Type for Operation A**B

Base A	Exponent B			
	INTEGER	REAL	DOUBLE PRECISION	COMPLEX
INTEGER	INTEGER	REAL	DOUBLE PRECISION	Not permitted
REAL	REAL	REAL	DOUBLE PRECISION	Not permitted
DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	Not permitted
COMPLEX	COMPLEX	COMPLEX	COMPLEX*	Not permitted

* The DOUBLE PRECISION exponent is converted to REAL before exponentiation.

Table 3-2

Combination of Elements

A	B			
	INTEGER	REAL	DOUBLE PRECISION	COMPLEX
INTEGER	INTEGER*	REAL	DOUBLE PRECISION	COMPLEX
REAL	REAL	REAL	DOUBLE PRECISION	COMPLEX
DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	COMPLEX**

Table 3-2 (cont)

Combination of Elements

A	B			
	INTEGER	REAL	DOUBLE PRECISION	COMPLEX
COMPLEX	COMPLEX	COMPLEX	COMPLEX**	COMPLEX

* INTEGER division yields a truncated result.

** The DOUBLE PRECISION element is converted to REAL before the operation.

LOGICAL EXPRESSION.

A logical expression is a rule for computing a logical value.

The general form is:

> Any constant, variable, or function reference, or a combination of these separated by operators, logical operators, or parentheses.

Logical quantities may be combined by logical operators to form logical expressions in a manner analogous to the combination of arithmetic quantities by arithmetic operators.

A logical quantity, by itself, may also constitute a logical expression.

A logical quantity may be:

a. Any logical variable.
b. Either of the logical constants .TRUE. or .FALSE.
c. Any logical function reference.
d. Any relation.

The logical operators are defined in table 3-3.

Table 3-3

Definitions of Logical Operators

Operator	Definition
.NOT.	The expression .NOT. P is .TRUE. when P is .FALSE. The expression .NOT. P is .FALSE. when P is .TRUE.
.AND.	The expression P .AND. Q is .TRUE. when both P and Q are .TRUE. It is .FALSE. in any other case.
.OR.	The expression P .OR. Q is .TRUE. if either P or Q is .TRUE. It is .FALSE. if and only if both P and Q are .FALSE.

The precedence of operators in the evaluation of logical expressions is:

a. Function reference. Highest.

b. ** (exponentiation).

c. * and / (multiplication and division).

d. + and - (addition and subtraction).

e. .LT., .LE., .EQ., .NE., .GT., .GE.

f. .NOT.

g. .AND.

h. .OR. Lowest.

Parentheses may be used to alter the order of evaluation.

If A and B are logical expressions, each of the following examples is also a logical expression:

Examples

```
.NOT. B
A
A.OR.B
```

```
    (B)
    B .AND. A
```

RELATION.

A relation is a conditional logical expression.

The general form is:

a op b
where a and b are arithmetic expressions and op is a relational operator.

The relational operators and their meaning are noted in table 3-4.

Table 3-4

Relations and Meanings

Relation	Meaning
A_1 .GT. A_2	A_1 greater than A_2
A_1 .GE. A_2	A_1 greater than or equal to A_2
A_1 .LT. A_2	A_1 less than A_2
A_1 .LE. A_2	A_1 less than or equal to A_2
A_1 .NE. A_2	A_1 not equal to A_2
A_1 .EQ. A_2	A_1 equal to A_2

NOTE

A_1 and A_2 may be of type INTEGER, REAL, or DOUBLE PRECISION. Neither may be of type COMPLEX.

Relations, when evaluated, may have one of two values, TRUE or FALSE.

Chains of relations are not permitted, e.g.,

```
    A .LT. B .LT. C
```

A correct form is:

```
     A .LT. B .AND. B .LT. C
```

or

```
     A .LT. B .AND. A .LT. C
```

whichever is intended.

In the following examples A, B, Q, Z, E, F, X, G, H, and Y are arithmetic expressions.

Examples

```
     A .LT. B
     A .LT. B .AND. Q .GT. Z
     (E+F).NE.SIN(X).OR.(G-H).LT.ABS(Y)
```

SECTION 4
ASSIGNMENT STATEMENTS

GENERAL.

There are three types of assignment statements:

a. Arithmetic Assignment statement.

b. Logical Assignment statement.

c. ASSIGN statement.

ARITHMETIC ASSIGNMENT STATEMENT.

The Arithmetic Assignment statement causes the value represented by an arithmetic expression appearing to the right of the assignment operator (=) to be assigned to the simple or subscripted variable appearing to the left of the assignment operator.

The general form is:

v = a.e.
where v represents a variable name, simple or subscripted, and a.e. represents an arithmetic expression.

The variable v cannot be of type LOGICAL.

The rules provided in table 4-1 apply for type and value assignment in arithmetic expressions.

Examples

```
X = Y+Z
X(10) = A(5)+B(6)-(C/D)
X = 5.49
X(I,J) = A(I,J)+B(J,I)
X(4) = D-C**2
```

Table 4-1

Rules for Arithmetic Assignment
Statement (v = a)

v	a	Rule
INTEGER	INTEGER	Assign.
INTEGER	REAL	Truncate to INTEGER and assign.
INTEGER	DOUBLE PRECISION	Truncate to INTEGER and assign.
INTEGER	COMPLEX	Not permitted
REAL	INTEGER	Convert to REAL and assign.
REAL	REAL	Assign.
REAL	DOUBLE PRECISION	Assign most-significant part.
REAL	COMPLEX	Not permitted
DOUBLE PRECISION	INTEGER	Extend to DOUBLE PRECISION and assign.
DOUBLE PRECISION	REAL	Extend to DOUBLE PRECISION and assign.
DOUBLE PRECISION	DOUBLE PRECISION	Assign.
DOUBLE PRECISION	COMPLEX	Not permitted
COMPLEX	INTEGER	Not permitted
COMPLEX	REAL	Not permitted
COMPLEX	DOUBLE PRECISION	Not permitted
COMPLEX	COMPLEX	Assign.

LOGICAL ASSIGNMENT STATEMENT.

The Logical Assignment statement causes the value represented by the logical expression appearing to the right of the assignment operator

(=) to be assigned to the simple or subscripted variable of type LOGICAL appearing to the left of the replacement operator.

The general form is:

v = l.e.
where v is a simple or subscripted variable of type LOGICAL and l.e. represents a logical expression.

The variable v must be of type LOGICAL.

In the following examples K, L, M, and N are LOGICAL variables.

<u>Examples</u>

```
K = M .OR. N
L(J,5) = .TRUE.
M = A .LT. B
N = Q .GT. R .AND. Z .LT. P
```

<u>ASSIGN STATEMENT</u>.

The ASSIGN statement is used to initialize an Assigned GO TO statement.

The general form is:

ASSIGN n TO t
where n is a statement label referenced in an Assigned GO TO statement, and t is a simple INTEGER variable appearing in the same Assigned GO TO statement.

The statement label n must be referenced in the Assigned GO TO statement being initialized.

<u>Example</u>

```
ASSIGN 10 TO J
```

NOTE

Use of the ASSIGN statement requires an INTEGER size of at least 5, the default value.

SECTION 5
CONTROL STATEMENTS

GENERAL.

Control statements are used to alter the normal flow of a program. They may transfer control to another part of the program, terminate computation, or control iterative processes. Control may be transferred to labeled executable statements only. There are 10 different control statements:

a. Unconditional GO TO.
b. Computed GO TO.
c. Assigned GO TO.
d. Arithmetic IF.
e. Logical IF.
f. DO.
g. CONTINUE.
h. STOP.
i. RETURN.
j. CALL.

UNCONDITIONAL GO TO STATEMENT.

Execution of this statement causes control to be transferred to a statement other than that sequentially following the Unconditional GO TO statement.

The general form is:

GO TO n
where n is a statement label which exists within the same program unit.

A statement label n must be defined within the same program unit as the Unconditional GO TO statement which refers to it.

The statement labeled n may appear before or after the Unconditional GO TO statement referencing it.

Example

```
      GO TO 31
      ...
      ...
   31 ...
```

COMPUTED GO TO STATEMENT.

Execution of this statement causes control to be transferred to one of several statements other than that sequentially following the Computed GO TO statement.

The general form is:

GO TO $(n_1, n_2, \ldots, n_i)$, t
where $n_1, n_2, \ldots, n_i$ are statement labels and t is an integer expression.

Control is transferred to the statement label whose position in the list is equal to the value of the integer expression t, i.e., n_t.

The statement labels $n_1, n_2, \ldots, n_i$ must exist in the same program unit as the Computed GO TO statement.

The Computed GO TO statement is valid for values of t such that $1 \le t \le i$; otherwise, the program is terminated with an address error.

Example

```
      K=4
      GO TO (50,40,30,20,10),K
```

Execution of these two statements causes control to be transferred to statement 20.

ASSIGNED GO TO STATEMENT.

Execution of this statement causes control to be transferred to one of several alternative statements other than that sequentially following the Assigned GO TO statement.

The general form is:

GO TO t, $(n_1,n_2,...,n_i)$
where t is a simple INTEGER variable and $n_1,n_2,...,n_i$ are statement labels.

Control is transferred to the statement whose label has been ASSIGNed to t with an ASSIGN statement.

The values ASSIGNable to t are the actual statement labels appearing in the list $n_1,n_2,...,n_i$.

The variable t must be a simple INTEGER variable.

If t has not been assigned a label appearing in the list, an address error termination of the program results.

The statement labels $n_1,n_2,...,n_i$ must appear in the same program unit as the ASSIGN statement and the Assigned GO TO statement. The value of t must not be changed between execution of the ASSIGN statement and execution of the Assigned GO TO statement.

Example

```
    ...
    ...
    ASSIGN 10 TO J
    GO TO J,(50,40,30,20,10)
```

Execution of these two statements causes control to be transferred to statement 10.

ARITHMETIC IF STATEMENT.

Execution of the Arithmetic IF statement causes an arithmetic expression to be evaluated and a different branch to be made depending upon whether the expression is negative, zero, or positive.

The general form is:

$IF(a.e.)n_1,n_2,n_3$
where a.e. is an arithmetic expression and n_1,n_2, and n_3 are statement labels.

Execution of the Arithmetic IF statement causes control to be transferred to n_1,n_2, or n_3 if a.e. is less than, equal to, or greater than zero, respectively.

The arithmetic expression a.e. may not be COMPLEX.

Examples

```
     IF (A-B) 1,2,3
     IF(X(I,J)-C*E) 43,51,96
```

LOGICAL IF STATEMENT.

Execution of the Logical IF statement causes a logical expression to be evaluated and the sequence of execution of the program statements to be altered, depending upon whether the logical expression evaluated is TRUE or FALSE.

The general form is:

IF(l.e.) s
where l.e. is a logical expression and s is an executable FORTRAN statement.

The statement s may be any executable FORTRAN statement except:

a. A DO statement.

b. An IF statement.

Execution of the Logical IF statement results in the logical expression l.e. being evaluated. If l.e. is TRUE, statement s is executed. If l.e. is FALSE, statement s is not executed; and control is transferred to the next sequential executable statement following the Logical IF statement.

In the following examples X and Y are of type LOGICAL.

Examples

```
IF (X .AND. Y) A = 3.1
IF (A .LE. B .OR. I .EQ. 0) GO TO 5
```

DO STATEMENT.

The DO statement provides a means for controlling program loops.

The general form is:

DO m i=n_1,n_2,n_3
where m is a statement label, i is an INTEGER variable, and n_1,n_2, and n_3 are arithmetic expressions.

Execution of a DO statement results in the following actions:

a. The control variable i is set to the initial value n_1.

b. All executable statements up to and including the terminal statement are executed.

c. The control variable i is incremented by n_3.

d. The value of the control variable i is compared to the terminal value n_2. If the terminal value has been exceeded, control is transferred to the first executable statement following the terminal statement. Otherwise, steps b through d are repeated until the control variable comparison is satisfied.

The control variable i is a simple INTEGER; m is the label of an executable statement terminating the DO loop.

The initial, terminal, and incremental parameters n_1, n_2, and n_3, respectively, are each either an INTEGER variable or INTEGER constant.

If not specified, n_3 is assumed to be 1.

If present, n_3 must be greater than 0 (zero).

In the general form, n_2 must be greater than n_1. At the time of execution of the DO statement, n_1, n_2, and n_3 must be greater than 0 (zero).

The DO statement is always executed once with its initial value.

The control variable i is available for use by all statements within the DO loop, including the terminal statement, and may be modified as desired. The control variable i is available for computation when exiting from a DO loop by transferring outside the loop and not making a normal exit. When a normal exit is made from the DO loop, the control variable is undefined.

A DO statement may appear within a DO loop. This is defined as being a DO nest. However, all statements in the range of the latter DO loops must be within the range of the initial DO loop (see figure 5-1).

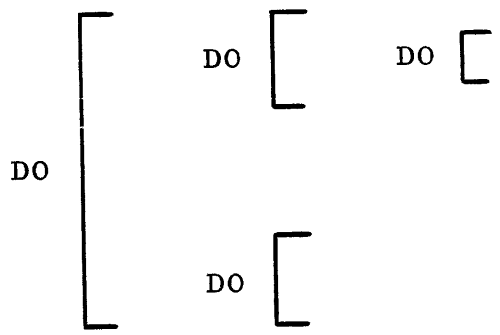


Figure 5-1. DO Nesting

Nested DO's may specify the same statement as their last statement m.

A maximum of nine DO statements may be nested within the range of another DO statement.

The terminal statement of a DO loop may not be:

a. A GO TO of any form.

b. IF statement.

c. RETURN.

d. STOP.

e. PAUSE.

f. DO statement.

g. REREAD statement.

A DO statement has an extended range if both of the following apply:

a. If there is a GO TO statement or an Arithmetic IF statement within the range of the <u>innermost</u> DO of a completely nested nest that can cause control to pass out of that nest (a completely nested nest is one in which both the nested DO statement and its terminal statement are <u>within</u> the outer loop).

b. If there is a GO TO statement or Arithmetic IF statement not within the nest that can cause control to be returned into the range of the <u>innermost</u> DO of the nest.

If a statement is the terminal statement of more than one DO statement, the statement label of that terminal statement may not be used in any GO TO or Arithmetic IF statement that occurs anywhere but in the range of the most deeply nested DO with that terminal statement.

<u>Examples</u>

```
      DO 10 I=2,200,4
      ...
      ...
   10 ...
      DO 5 INDEX=5,10
      DO 5 J=1,10
      ...
      ...
    5 ...
```

```
      DO 10 I=1,5,2
      DO 5 J=1,10
      ...
      ...
    3 ...
      GO TO 20
    5 CONTINUE
      ...
      ...
   10 ...
      ...
      ...
   20 X=Y+Z
      GO TO 3
```

CONTINUE STATEMENT.

The CONTINUE statement is considered a dummy statement because it causes no action in the execution of a program. It is frequently used as the terminal statement of a DO loop to provide a transfer point for an IF or GO TO statement.

The general form is:

$\boxed{\texttt{CONTINUE}}$

Example

```
      DO 30 J=2,N
      B(J)=NM(J-1) + INC
      INC=INC+1
      IF (NPARM(J).LT.MAX) GO TO 30
      K=J-1
   30 CONTINUE
```

STOP STATEMENT.

The STOP statement causes immediate termination of the program. At least one STOP statement or a CALL EXIT statement (refer to appendix B) must appear in a FORTRAN program.

The general form is:

$\boxed{\texttt{STOP}}$

Example

```
    STOP
```

RETURN STATEMENT.
Execution of the RETURN statement causes control to be transferred from a subprogram to the calling program.

The general form is:

$$\boxed{\text{RETURN}}$$

Every subprogram must contain at least one RETURN statement, but more than one may appear in a subprogram.

Control returns to the point of reference in the calling unit.

CALL STATEMENT.
A subroutine is referenced by a CALL statement.

The general form is:

1. CALL n
2. CALL n(a_1,a_2,...,a_n)
where n is the name of the subroutine and a_1,a_2,...,a_n are the actual parameters.

The actual parameters are:

a. A Hollerith constant.
b. A variable name.
c. An array element name.
d. An array name.
e. An expression.
f. The name of a subprogram.

Execution of a subroutine reference results in an association of actual parameters with all appearances of formal parameters in executable statements in the subroutine body, and in an association of actual parameters with variable dimensions in the subroutine, if any exist.

Following the above associations, control is transferred to the first executable statement in the subroutine body.

If an actual parameter is a subscripted variable with an arithmetic expression as a subscript, then, effectively, the arithmetic expression is evaluated, and the resulting subscripted variable is associated with the corresponding formal parameter in the subroutine.

If a formal parameter of a subroutine is an array name, the corresponding actual parameter must be an array name or an array element name.

Examples

```
CALL FALL(X,Y,Z)
CALL KOST(A(I+J,2),B,4HHEAD)
```

NOTE

An equivalenced array whose elements are of a size less than the SU (storage unit) number of digits must be passed to a subroutine in COMMON, not as a parameter.

SECTION 6
DECLARATIVE STATEMENTS

GENERAL.

The declarative statements are non-executable statements used to supply variables and array information and storage allocation information. The six different declarative statements are:

a. DIMENSION.
b. COMMON.
c. EQUIVALENCE.
d. Type.
e. EXTERNAL.
f. DATA.

Declarative statements must appear preceding all executable statements in the program part.

DIMENSION STATEMENT.

The DIMENSION statement provides a means for specifying a collection of values with a single name, and at the same time specifying to the compiler the structure which is imposed on the collection.

The general form is:

DIMENSION $a_1(i_1)$, $a_2(i_2)$, $a_3(i_3)$ . . . .
where each a is an array name and each i represents dimension information having the form of one or more subscript bounds separated by commas.

Each bound is an integer constant.

Variable names appearing with subscripts in the source program must have dimension information specified for them prior to their use.

Dimension information may be given in a DIMENSION, COMMON, or Type statement; however, the dimension information for a specific array name must appear only once in the program unit.

The magnitude of the values of the subscript bounds indicates the maximum values the subscripts may obtain in any reference to the array. The lower subscript bound is always one. The maximum number of elements in an array is 9999. The maximum number of dimensions in a multi-dimensional array is three.

VARIABLE DIMENSIONS.

An array may have variables for its subscript bounds in a FUNCTION or SUBROUTINE subprogram. In this case, the array name and all variables used as subscript bounds must appear as formal parameters in the subprogram.

The advantage to this is that a given subprogram can perform calculations on such a generally stated array with specific dimensions provided from any calling program.

The actual values assumed by these variables are not determined until the subprogram is entered at execution time.

The general form is:

DIMENSION $a_1(i_1)$, $a_2(i_2)$, $a_3(i_3)$ . . . .
where each a is an array name and each i is one or more subscript bounds separated by commas. Each bound is an integer variable.

Specific dimensions passed to the subprogram from the calling program must be identified in a DIMENSION statement of the calling program.

Specific variable size can be passed through more than one level of a subprogram to a given subprogram using the variable as a dimension.

<u>Example</u>

```
DIMENSION A(10,20)
...
...
...
I=5
J=7
```

```
CALL SUB(A,I,J))
...
...
...
END
SUBROUTINE SUB(B,K,L)
DIMENSION B(K,L)
...
...
...
END
```

COMMON STATEMENT.

The COMMON statement provides a means for sharing core storage among the main program and its subprograms, or among the subprograms. Information appearing in the storage area reserved by a COMMON statement is ordered in the sequence specified by the COMMON statement. The ordered information is relative to the beginning of a given COMMON block. There are two types of COMMON storage: labeled and unlabeled.

The general form is:

COMMON/x_1/a_1/x_2/a_2/. . ./x_n/a_n
where each a in the COMMON statement is a list containing any combination of variable names, array names, or dimensioned array names; and each x is a block name or is empty. If x_1 is empty, the first two slashes are optional.

Array names in a COMMON statement may have their dimensioning information appended to them. When arrays are dimensioned in a COMMON statement, they cannot be dimensioned in a Type or DIMENSION statement as well.

COMMON area storage is assigned in the order of appearance of the elements within the COMMON block list.

Block names may be duplicated within a program unit, causing the associated elements from each COMMON block list having the same name to be cumulatively assigned to one block with the same name. The effect is the same as declaring the block name once and listing all elements for that block in the COMMON block list. This is also true for multiple unlabeled COMMON block lists within a given program unit.

Variables and array names may not be duplicated in COMMON statements.

COMMON elements may be assigned initial values through use of the BLOCK DATA subprogram.

The number and type of variables appearing in the COMMON block list and related EQUIVALENCE statements specify the length of the COMMON block.

All subscript bounds for any array which appears in a COMMON statement must be integer constants.

All variables stored in COMMON are stored in SU digits; SU is defined as the <u>maximum</u> of:

a. 2 x ALPHA precision (2x6 or 12 by default).
b. REAL precision + 4 (8+4 or 12 by default).
c. INTEGER precision + 1 (5+1 or 6 by default).

Therefore, when default precisions are used, an integer stored in COMMON is left-justified in a 12-digit field instead of occupying six digits. In this case, half of the space in an all-integer COMMON is wasted.

An element in COMMON which is double precision or complex must be an odd-numbered element (i.e., first, third, fifth, etc.), counting from the left.

Labeled COMMON statements are specified by a COMMON block name, between slashes, preceding the list of elements assigned to that labeled COMMON block. Termination of the list of elements assigned to a block is by:

a. Termination of the COMMON statement.

b. Introduction of a new block name.

c. Introduction of an unlabeled COMMON block.

COMMON block names are unique identifiers. A maximum of 10 unique COMMON block names may be defined in a program part.

Blocks of labeled COMMON statements in different program units which have the same block name reference the same storage area. Therefore, there is a direct correspondence of variable names in the COMMON statements.

Unlabeled COMMON statements are specified by a blank block name, e.g., / /, followed by the unlabeled COMMON block list. The two slashes may be omitted if they appear at the beginning of a COMMON statement list. Termination of an unlabeled COMMON block is accomplished by the introduction of a block name or termination of the COMMON statement.

Examples

```
COMMON X,Y,Z
COMMON //X,Y,Z
COMMON /YY/ K(5,5),L
COMMON A,B,C/S/D(10,10),E
COMMON /Y/Q,R,S/ /K(5,5),L
```

EQUIVALENCE STATEMENT.

By using the EQUIVALENCE statement, a storage location can be given more than one name. Thus, variables or array elements not listed in an EQUIVALENCE statement have unique storage assignments.

The general form is:

EQUIVALENCE $(q_1),(q_2),(q_3),....,(q_n)$
where each q is a list of two or more simple or subscripted variables or array names separated by commas.

Subscripts must be positive integer constants and must correspond in number to the declared number of dimensions of the array, or be single subscripted by equating the element position in the array to a single subscript. For an explanation of the latter, refer to table 6-1.

Example

```
     DIMENSION C(120)
     DIMENSION B(4,5,6) element referenced B(3,2,1)
     EQUIVALENCE(B,C)
     B(3,2,1) = C([3]+[4x(2-1)]+[4x5x(1-1)]) = C(7)
```

An array name without subscripts is considered as that identifier with a subscript of one.

Elements may be entered into COMMON blocks by setting them equivalent to an element appearing in a COMMON statement list. If the element is an array element, the whole array is brought into COMMON. This may extend the size of the COMMON block involved at its end only.

When two elements share storage because of their appearance in one or more EQUIVALENCE statements, only one may appear in a COMMON statement.

All subscript bounds for an array which appears in an EQUIVALENCE statement must be integer constants.

An EQUIVALENCE statement must precede any reference to the elements equivalenced.

Example

```
     DIMENSION A(10), B(5,5) D(3,3,3)
     EQUIVALENCE (A(3), B(5,4), D(1,1,1)), (A(1),E)
```

The above statements assign specific variable values to the same storage locations, as shown below, where each horizontal line is one memory location.

Table 6-1

Equivalencing Multiple Subscripts To One Subscript

Number of Dimensions	Array Declarations	Array Element	Same Array Element With One Subscript	Maximum Single-Subscript Value
1	A(I)	A(i)	A(i)	I
2	A(I,J)	A(i,j)	A(i+Ix(j-1))	IxJ
3	A(I,J,K)	A(i,j,k)	A(i+Ix(j-1)+IxJx(k-1))	IxJxK

```
A(1)     B(3,4)     ...         E
A(2)     B(4,4)     ...         ...
A(3)     B(5,4)     D(1,1,1)    ...
A(4)     B(1,5)     D(2,1,1)    ...
A(5)     B(2,5)     D(3,1,1)    ...
A(6)     B(3,5)     D(1,2,1)    ...
A(7)     B(4,5)     D(2,2,1)    ...
A(8)     B(5,5)     D(3,2,1)    ...
A(9)     ...        D)1,3,1)    ...
A(10)    ...        D(2,3,1)    ...
```

Variables placed in an EQUIVALENCE statement are stored in storage units, SU digits. SU is defined as the maximum of:

a. 2 x ALPHA precision (2x6 or 12 digits by default).

b. REAL precision + 4 (8+4 or 12 digits by default).

c. INTEGER precision + 1 (5+1 or 6 digits by default).

Therefore, when default precisions are used, an integer placed in an EQUIVALENCE statement is left-justified in a 12-digit field.

TYPE STATEMENT.

Type statements are used to declare the type of variables, array names, and function names.

The general form is:

1. INTEGER type list
2. REAL type list
3. DOUBLE PRECISION type list
4. COMPLEX type list
5. LOGICAL type list
6. ALPHA type list
where a type list is composed of variable names, array names, or statement function names separated by commas. In addition, arrays may be dimensioned by appending the dimension information to the array name in one or more subscript positions.

Implicit type assignment is overridden by Type statements.

A variable name must be typed prior to its use in an executable statement or DATA statement. If the first letter of a variable name is I, J, K, L, M, or N, it is implicitly declared of type INTEGER and need not appear in a Type statement. If a variable name begins with any other letter, it is implicitly declared of type REAL and need not appear in a Type statement.

NOTE

Type statements must appear before all executable statements in the associated program part. An ALPHA type statement must be specified for all alphanumeric variables and arrays.

Examples

```
INTEGER X,Y,Z,A(10,10)
REAL H,I,J,K
LOGICAL ATEST
ALPHA WORD
```

EXTERNAL STATEMENT.

When an actual parameter list of a function or subroutine reference contains a function or subroutine name, that name must appear in an EXTERNAL statement.

The general form is:

EXTERNAL $n_1,n_2,\ldots\ldots,n_n$
where the n's are the names of the functions or subroutines appearing in the parameter list of a function or subroutine reference.

The EXTERNAL statement appears in the calling program unit.

NOTE

When a function is referenced in an EXTERNAL statement, its name must conform to default naming conventions, e.g., an INTEGER function must have a name beginning with one of the characters I through N.

Example

```
EXTERNAL SIN,COS
CALL SUBT (SIN,COS)
...
...
END
SUBROUTINE SUBT(AA,BB)
TANX=AA(X)/BB(X)
...
...
RETURN
END
```

DATA STATEMENT.

The DATA statement permits variables and arrays to be initialized to predetermined values. It must be the last non-executable statement in the program part in which it appears.

The general form is:

DATA list$_1$/d_1,d_2,d_3,...,d_n/,list$_2$/d_1,d_2,...,d_n/,.../
A list element may be an array name or a simple or subscripted variable name, where the subscripts must be integer constants. The d_i represents a constant, or has the form i*c, where i is a repeat count and c is a constant.

The constants may be any of the following:

a. Integer, real, or double precision constant.

b. Logical constants.

c. Hollerith constants.

d. Complex constants. (A complex constant must be enclosed in parentheses.)

A one-to-one correspondence must exist between the list elements and the constants.

DATA statement variables retain their values from one execution to the next, as they are initialized only once by the DATA statement at compilation time.

Example

```
COMPLEX VAR
ALPHA A(3)
DATA A/6HABCDEF,6HGHIJKL,6HMNOPQR/,VAR/(8.3,4.5)/
```

Elements in a COMMON block may appear in a DATA statement only in a BLOCK DATA subprogram (refer to section 8).

Variables assigned quantities by a DATA statement may be assigned other values during execution.

When an array name without subscripts appears in the list, the entire array is initialized.

Subscripted variables appearing in a program must have their subscript bounds specified in a DIMENSION, COMMON, or Type statement prior to the first appearance of the subscripted variable in a DATA statement.

Example

```
DIMENSION N(5,5),A(8)
LOGICAL ATEST,BTEST
DATA I,J,H/1,3,5.7/,X,Y,ATEST/6.2,99.99,.FALSE./
DATA K,N,Z,BTEST,A/0,25*0,-99.,.TRUE.,8*77.77/
```

SECTION 7

INPUT/OUTPUT

GENERAL.

The following areas of input/output (I/O) are covered in this section:

a. Input statements.
b. Output statements.
c. I/O lists.
d. Implied DO loop.
e. Action labels.
f. Auxiliary I/O statements.
g. FORMAT statement.

INPUT STATEMENTS.

In explanations presented in this section of the manual, the symbols u, r, f, k, and l have the following meanings unless otherwise specified.

Symbol	Meaning
u	File specifier or unit number. The file specifier is an integer variable or integer constant whose value identifies the file being used for input or output. Unless otherwise specified by a FILE Card, it is assumed at object time that the file specifier designates the default hardware type as defined on page C-5.
r	Random record number. It is an integer constant or variable whose value represents a particular record within a random disk file.
f	Format specifier. It may be the label of a FORMAT statement or an array identifier.
l	Action label. It specifies a statement label to which a branch is made if a parity error or an End-of-File condition is encountered during execution of an input statement.

Symbol	Meaning
k	Input/output list. It may be a blank or it may contain one or more variables and/or implied DO loops, in any combination.

Execution of any of the READ statements causes the next record to be read from the input file. The information is scanned and converted as specified by the format specifier f if the statement is a formatted READ. The values are assigned to the elements specified by the list k. If the list is not specified in an unformatted READ, a record is skipped; if the list is not specified in a formatted READ, data are read into the locations in storage occupied by the FORMAT statement.

FORMATTED INPUT STATEMENTS.

Formatted input statements are always associated with a FORMAT statement or any array containing FORMAT specifications.

The general form is:

1.	READ (u,f) k
2.	READ (u,f,l) k
3.	READ (u=r,f) k
4.	READ (u=r,f,l) k

In all four forms, the input list may be empty (i.e., blank).

When the first or second form is used, input is assumed to be from whatever peripheral device is associated by default with the specified unit number, unless otherwise specified by a FILE Card.

When the third or fourth form is used, input should be from random disk file. In this instance, a FILE Card must be used.

In using the third or fourth form, the random record number r, when evaluated, must have a non-negative integer value.

Examples

```
READ(8,IBID)((I,J,A(I,J),J=6,9),I=1,5)
READ(IUNIT,75)X,Z,A
READ(14,LISTA)
READ(6=5,25,END=101,ERR=77) ARAY
```

UNFORMATTED INPUT STATEMENTS.

Unformatted input statements do not have a format specifier associated with them. Input must be from a variable length tape file or disk file which has been created with an unformatted output statement.

The general form is:

1.	READ(u) k
2.	READ(u,l) k
3.	READ(u=r) k
4.	READ(u=r,l) k

In all four forms, the input list k may be empty (i.e., blank).

If the list k is not specified, a record is skipped.

The file used for input must have been previously created with a similar unformatted output statement.

When either of the first two forms is used, input must be from a tape or serial disk file.

When either of the last two forms is used, input must be from a random disk file.

Examples

```
READ(9) I,A,J,B,D
READ(2,ERR=37) SAM
READ(IUNIT=10,END=99) FEAT,HAMER
```

OUTPUT STATEMENTS.

In the following explanation, the symbols u, r, f, 1, and k have the same meanings as outlined under input statements.

Execution of any of the output statements causes the next record in the output file to be created. The information is converted and positioned on output as specified by the format specifier f if the statement is a formatted output statement. If the list is not specified, either a record is skipped or data contained in the locations in storage occupied by the FORMAT statement are outputted. When output is to a serial disk file and a format is not specified, a blank record is written. When output is to a random disk file, a record is not written.

FORMATTED OUTPUT STATEMENTS.

Formatted output statements are always associated with a FORMAT statement or an array containing FORMAT specifications.

The general form is:

1.	WRITE(u,f) k
2.	WRITE(u=r,f) k

In both forms, the output list k may be empty (i.e., blank).

When either form is used, output is to whatever peripheral device is associated by default with the unit number specified, unless otherwise specified by a FILE Card.

When the second form is used, output is to a random disk file. In this instance, a FILE Card must be used.

In using the second form, the random record number r, when evaluated, must have a non-negative integer value.

If output is to the line printer and the associated FORMAT statement specifies more information than can be printed on one line, data are not lost. Beginning with the first element that does not completely fit on the print line, the remainder of the record is written on the next print line.

Examples

```
WRITE(3=3,68)MATR
WRITE(NO,I) ROW
WRITE(3=J,68) MATRIX
```

UNFORMATTED OUTPUT STATEMENTS.
Unformatted output statements do not have a format specifier associated with them. Output must be to a tape or disk file.

The general form is:

1.	WRITE(u) k
2.	WRITE(u=r) k

In both forms, the output list k may be empty (i.e., blank).

When the first form is used, output must be to a tape or serial disk file.

When the second form is used, output must be to a random disk file.

Examples

```
WRITE(OUT) (X(K),K=I,J),XX
WRITE(11=IREC) BOOL
```

I/O LISTS.
An input list k in an input statement specifies the variables to which values are assigned on input. An output list k specifies the variables whose values are transmitted on output. The input and output lists are of the same form.

The general form is:

$k_1, k_2, \ldots, k_n$
where $k_1, k_2, \ldots, k_n$ are variables, array names, or implied DO loops, or any combination thereof.

An element k_i of an I/O list may be a simple variable, a subscripted variable, an array name without subscripts, an implied DO loop, or any combination of these elements.

An array name without subscripts in an I/O list is equivalent to inputting or outputting the entire array in the same order in which the elements are stored in memory, i.e., column-wise, with the left-most subscripts varying most rapidly.

Examples

```
I,J,A,KP,B(I)
(A(INDEX),LP,INDEX=1,20),ZIP,ZAP
```

IMPLIED DO LOOP.

An implied DO loop is used as an element in an I/O list to specify a repeated cycle of list elements.

The general form is:

1. $(L, i=n_1, n_2, n_3)$
2. $((L, i=n_1, n_2, n_3), k=m_1, m_2, m_3)$
where L is a list of I/O elements which may contain an implied DO loop, and i, n_1, n_2, n_3, and their counterparts k, m_1, m_2, m_3 are as defined for the DO statement.

Example

```
WRITE(6,35) ((I,B(I,J),I=1,3),J=6,7)
```

The output for the above statement takes the following form:

```
    1  B(1,6)
    2  B(2,6)
    3  B(3,6)
    1  B(1,7)
    2  B(2,7)
    3  B(3,7)
```

where the subscripted B's represent the values of those elements.

ACTION LABELS.

The formatted and unformatted input statements can be extended to programmatically recover from either End-of-File conditions or non-recoverable parity conditions, or both, through use of action labels.

The general form is:

1. ERR=n_1
2. END=n_2
3. ERR=n_1,END=n_2
4. END=n_2,ERR=n_1
where n_1 and n_2 are statement labels.

When an attempt is made to read a record which has a parity error from which the operating system cannot recover, control is transferred to the statement labeled n_1.

When an attempt is made to read an End-of-File, control is transferred to the statement labeled n_2.

The program is terminated immediately by the operating system if either of the above conditions occurs and the associated label is not specified in the input statement being executed.

An End-of-File condition can occur under the following circumstances:

a. Attempting to read a card with an invalid character in column one.

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

c. Attempting to read a record from an area of disk which has not been written.

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

Examples

```
READ(3,END=99)
READ(6=R,35,ERR=70) A
READ(11,85,END=77,ERR=78) J,S,V
```

AUXILIARY I/O STATEMENTS.

There are four types of auxiliary I/O statements:

a. REWIND.
b. BACKSPACE.
c. ENDFILE.
d. REREAD.

REWIND STATEMENT.

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

The general form is:

$$\boxed{\text{REWIND u}}$$

Execution of the REWIND statement causes the file u to be positioned to its initial point.

If the last reference to the file 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 tape or disk files.

Examples

```
     REWIND 5
     REWIND IUNIT
```

BACKSPACE STATEMENT.
If the pointer in file u is positioned at record n, execution of the BACKSPACE statement causes the file pointer to point at record (n-1).

The general form is:

$$\boxed{\texttt{BACKSPACE u}}$$

If file u is positioned at its initial point, execution of this statement has no effect.

Examples

```
     BACKSPACE 8
     BACKSPACE N
```

ENDFILE STATEMENT.
The ENDFILE statement causes a tape mark and ending label to be written on the specified file and the file to be closed.

The general form is:

$$\boxed{\texttt{ENDFILE u}}$$

The ENDFILE statement is undefined for anything other than a tape file.

When an ENDFILE statement follows a WRITE statement on the same file u, an End-of-File record is written and the tape is positioned such that the next record written follows the End-of-File record.

When an ENDFILE statement follows a READ statement on the same file u, the tape is positioned to the beginning of the next file on the tape.

When an ENDFILE statement follows a BACKSPACE statement on the same file u, the tape is positioned to the beginning of the file u.

When an ENDFILE statement follows REWIND or another ENDFILE statement on the same file u, the ENDFILE statement is ignored.

Examples

```
     ENDFILE IF1
     ENDFILE 7
```

REREAD STATEMENT.
The REREAD statement causes the last record read in any file to be reaccessed.

The general form is:

$$\boxed{\text{REREAD}}$$

A REREAD statement is associated with the next READ statement to be executed and yields the last record read from that file. REREAD should immediately precede the READ statement for which this function is desired.

REREAD may not be the terminal statement of a DO loop.

Example

```
     FILE  4=TAPFIL,UNIT=TAPE,FIXED
     .......
           READ(5,70) A
           ....
           READ(4,20) ARAY
           IF(A.EQ.CNTR) GO TO 10
           REREAD
           READ(5,91) B,C,D
           ....
        10 CONTINUE
```

FORMAT STATEMENT.

The FORMAT statement specifies what type of conversion is to be performed on data from external representation to internal machine representation or vice versa.

The general form is:

n FORMAT(f_1,f_2,...,f_n)
where n is a statement label and f_1, f_2,...,f_n are format specifications.

The FORMAT statement is non-executable.

The FORMAT statement is always associated with one or more formatted input and/or output statements.

Each FORMAT specification must agree in type with the corresponding variable in the list of the associated I/O statement.

When inputting data under a numeric format specification (I, F, E, D, G), leading blanks are not significant and other blanks are interpreted as zeros.

Plus signs are optional on input and may be omitted.

When inputting data under a real format specification (F, E, G, D), a decimal point appearing in the input field overrides the decimal point placement specified.

Any blanks read in under a numeric format specification (I, F, E, D, G) which are outputted without an action being performed on them between inputting and outputting appear in the output field as negative zeros.

In the following FORMAT discussions, the symbols w, d, b, and s have the following meanings.

Symbol	Meaning
w	Total input or output field width. A positive unsigned integer
d	Number of decimal places. A non-negative unsigned integer
b	Blank
s	A string of any valid FORTRAN characters

INTEGER CONVERSION ON INPUT USING Iw.

The integer format specification Iw on input causes the value of the integer datum in the input field to be assigned to the corresponding integer variable in the input list.

The general form is:

$$\boxed{\text{Iw}}$$

The integer datum must be in the form of an integer constant right-justified in the input field.

Examples

Input Field	Specification	Internal Value
567	I3	+567
bb-329	I6	-329
-bbbb27	I7	-27
27bbb	I5	+27000
-bb234	I6	-234

INTEGER CONVERSION ON OUTPUT USING Iw.

The integer format specification Iw on output causes the value of the corresponding integer variable in the output list to be written on the specified output file.

The general form is:

$\boxed{\texttt{Iw}}$

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

The plus sign is omitted for positive numbers.

The appearance of asterisks in the output field indicates that the value is larger than its format specification.

Examples

Internal Value	Specification	Output Field
+23	I4	bb23
-79	I4	b-79
+67486	I5	67486
0	I3	bb0
37216	I4	****

REAL CONVERSION ON INPUT USING Fw.d.
The real format specification Fw.d on input causes the value of the real datum in the input field to be assigned to the corresponding real variable in the input list.

The general form is:

$\boxed{\texttt{Fw.d}}$

If there is no decimal point in the input field, a decimal point is inserted d places from the right side of the input field.

The field width w must be greater than or equal to the specified number of decimal places d.

Examples

Input Field	Specification	Internal Value
36725931	F8.4	+3672.5931
3.672593	F8.4	+3.672593
-367259.	F8.4	-367259

REAL CONVERSION ON OUTPUT USING Fw.d.

The real format specification Fw.d on output causes the value of the corresponding real variable in the output list to be written on the specified output file.

The general form is:

$$\boxed{\texttt{Fw.d}}$$

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.

The appearance of asterisks in the output field indicates that the value is larger than its format specification.

Examples

Internal Value	Specification	Output Field
+36.7929	F7.3	b36.793
-0.0316	F6.3	-0.032
0.0	F6.4	0.0000
0.0	F6.2	bb0.00
+579.645	F6.2	579.65
27.15	F4.2	****

REAL CONVERSION ON INPUT USING Ew.d.

The real format specification Ew.d on input causes the value of the real datum in the input field to be assigned to the corresponding real variable in the input list.

The general form is:

$$\boxed{\text{Ew.d}}$$

If there is no decimal point in the input field, a decimal point is inserted d places from either the right side of the input field or from the E denoting the exponent, if there is one.

The field width w must be greater than or equal to the specified number of decimal places d.

An input datum may or may not have an exponent.

Examples

Input Field	Specification	Internal Value
bbbbbb25046	E11.4	+2.5046
bbbbb25.046	E11.4	+25.046
-bb25046E-3	E11.4	-0.0025046
bb250.46E-3	E11.4	+0.25046
b-b25.04678	E11.4	-25.04678

REAL CONVERSION ON OUTPUT USING Ew.d.

The real format specification Ew.d on output causes the value of the corresponding real variable in the output list to be written on the specified output file.

The general form is:

$$\boxed{\text{Ew.d}}$$

The real number is placed right-justified and rounded to a d-digit mantissa, together with a 4-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 significant digits are not written to the left of the decimal point in the output field. The plus sign is omitted for positive numbers. This rule should be followed:

$$(w - d) \geq 6$$

If a scale factor n is used, it controls the decimal normalization between the number part and the exponent part as follows:

a. If $n \leq 0$, then $|n|$ zeros are placed immediately to the right of the decimal point with $(d-|n|)$ significant digits following the zeros.

b. If $n \geq 0$, then n significant digits are placed to the left of the decimal point and $(d-n+1)$ significant digits are placed to the right of the decimal point.

<u>Examples</u>

Internal Value	Specification	Output Field
+36.7929	E12.5	bb.36793Eb02
-36.7929	E11.5	-.36793Eb02

DOUBLE PRECISION CONVERSION ON INPUT USING Dw.d.
The double precision format specification Dw.d on input causes the value of the real datum in the input field to be assigned to the corresponding variable of type DOUBLE PRECISION in the input list.

The general form is:

$$\boxed{\text{Dw.d}}$$

Aside from the fact that a double precision value may contain twice as many significant mantissa digits as a single precision real, and that the exponent in the input field is preceded by a D rather than an E, the double precision format specification Dw.d functions in the same manner as Ew.d.

DOUBLE PRECISION CONVERSION ON OUTPUT USING Dw.d.
The double precision format specification Dw.d on output causes the value of the corresponding double precision variable in the output list to be written on the specified output file.

The general form is:

$$\boxed{\text{Dw.d}}$$

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

a. The value associated with it is stored with twice the number of mantissa digits as a real field.

b. The variable name associated with the value must be of type DOUBLE PRECISION.

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

REAL CONVERSION ON INPUT USING Gw.d.
The real format specification Gw.d on input is identical to Fw.d.

The general form is:

$$\boxed{\text{Gw.d}}$$

REAL CONVERSION ON OUTPUT USING Gw.d.
The real format specification Gw.d on output causes the value of the corresponding real variable in the output list to be written on the specified output file.

The general form is:

$$\boxed{Gw.d}$$

The representation in the output field is a fraction of the magnitude of the real number being outputted.

If n is the magnitude of the number being outputted, table 7-1 shows how the number appears in the output field.

Table 7-1
Datum Conversion

Magnitude of Datum	Equivalent Conversion Effected
$0.1 \leq n < 1$	F(w-4). d, 4X
$1 \leq n < 10$	F(w-4). (d-1), 4X
.	.
.	.
.	.
$10^{d-2} \leq n < 10^{d-1}$	F(w-4).1, 4X
$10^{d-1} \leq n < 10^{d}$	F(w-4).0, 4X
Otherwise	Ew.d

If a scale factor is used, it has no effect on output conversion unless the magnitude of the number being written is outside the range which permits effective use of F conversion.

Examples

Internal Value	Specification	Output Field
+10.	G12.5	bb10.000
+1000.	G12.5	bb1000.0

Internal Value	Specification	Output Field
+100000.	G12.5	bb.10000Eb06
+1000000.	G12.5	bb.10000Eb07

LOGICAL CONVERSION ON INPUT USING Lw.

The logical format specification Lw on input causes the value of the logical datum in the input field to be assigned to the corresponding variable of type LOGICAL in the input list.

The general form is:

$$\boxed{\text{Lw}}$$

The input field width w must be greater than or equal to one. There may be leading blanks. The first character encountered in the field exclusive of leading blanks must be either T or F, for TRUE or FALSE, respectively. Any characters following the T or F are ignored.

If the input field for a logical variable contains neither a T nor an F, the internal value is FALSE.

Examples

Input Field	Specification	Internal Value
T	L1	TRUE
bbF	L3	FALSE
bbbTRU	L6	TRUE

LOGICAL CONVERSION ON OUTPUT USING Lw.

The logical format specification Lw on output causes the logical value of the corresponding variable of type LOGICAL in the output list to be written on the specified output file.

The general form is:

$$\boxed{\text{Lw}}$$

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

Examples

Internal Value	Specification	Output Field
FALSE	L1	F
FALSE	L3	bbF
TRUE	L2	bT

ALPHANUMERIC CONVERSION ON INPUT USING Aw.

The alphanumeric format specification Aw on input causes the character string of width w in the input field to be assigned to the corresponding variable in the input list.

The general form is:

$$\boxed{\text{Aw}}$$

The variable must be of type ALPHA. If the field width w is greater than alpha size (six by default), the input field is right-truncated. If w is less than six, the alpha string is stored left-justified in memory with (alpha size - w) trailing blanks. Alpha size is six in the following examples.

Examples

Input Field	Specification	Internal Value
ABCDEFGHIJK	A3	ABCbbb
ABCDEFGHIJK	A6	ABCDEF
ABCDEFGHIJK	A11	ABCDEF

ALPHANUMERIC CONVERSION ON OUTPUT USING Aw.

The alphanumeric format specification Aw on output causes the character string assigned to the corresponding variable in the output list to be written on the specified output file.

The general form is:

$$\boxed{\text{Aw}}$$

The string is placed left-justified in the output field over a field of blanks.

Examples

Internal Value	Specification	Output Field
ABCbbb	A3	ABC
ABCbbb	A5	ABCbb

ENTERING A CHARACTER STRING AS INPUT USING wHs.

The Hollerith field specification wHs on input causes the character string of width w in the input field to replace the character string s of the Hollerith field specification in a FORMAT statement.

The general form is:

$$\boxed{\text{wHs}}$$

The Hollerith field specification on input may be used to read in page headings which are to be printed on output, but which may vary in content from one run to another.

Example

```
   READ(5,15)
15 FORMAT(2X,9HDUMMYbbbb)
   WRITE(6,15)
```

Input

```
                  1 1
1 2 3 4 5 6 7 8 9 0 1 (card column)
X Y b A b S A M P L E
```

Output

```
b b A b S A M P L E
```

Note that in the printed output, although 2X has been specified, only one blank is printed since the first blank is a carriage control character (refer to carriage control, page 7-25).

PRODUCING A CHARACTER STRING AS OUTPUT USING wHs.
The Hollerith field specification wHs on output causes the character string s of width w of the Hollerith field in a FORMAT statement to be written on the specified output file.

The general form is:

$$\boxed{\text{wHs}}$$

The string s remains unchanged.

Example

```
   WRITE(7,95)
95 FORMAT(12HbBURROUGHSbb)
```

Output

```
                    1 1 1
1 2 3 4 5 6 7 8 9 0 1 2 (card column)
b B U R R O U G H S b b
```

SKIPPING CHARACTERS USING nX.
The format editing specification nX on input or on output causes n characters to be skipped in the respective input or output field.

The general form is:

$$\boxed{\text{nX}}$$

Tn FORMAT SPECIFICATION
The Tn format specification causes data transmission on input or on output to begin in the nth character position. When used with a line

printer file, data actually begin in the (n-1)st character position because the first character of the record is used for carriage control.

The general form is:

$$\boxed{\text{Tn}}$$

Example

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

Output

```
1 2 3 4 5 6 7 (printer position)
b b A B C D E
```

Example

```
  WRITE(7,8)
8 FORMAT(T4,7HEXAMPLE)
```

Output

```
                  1
1 2 3 4 5 6 7 8 9 0 (card column)
b b b E X A M P L E
```

SCALE FACTOR ON INPUT.
For F, E, G, and D format specifications on input, when the input datum does not have an exponent, the input datum is multiplied by 10^{-n}, where n is the scale factor. For example, the datum 573.19 read with a format of 2PF6.2 is stored internally as 5.7319. If the input datum contains an exponent, the scale factor has no effect.

SCALE FACTOR ON OUTPUT.
For F, E, and D format specification on output, when the output datum does not have an exponent, the output datum is multiplied by 10^{n}, where n is the scale factor. For example, the number stored internally as 5.7319 and written with a format of 2PF6.2 has the external

value of 573.19. If the output datum contains an exponent, the datum is multiplied by 10^n and the exponent is reduced by n. Therefore, the value is not changed. For example, the number stored internally as .57319E+02 and written with a format of 1PE11.3 has the external value of 5.732E+01.

For the G format specification on output, the effect of the scale factor is suspended unless the magnitude of the datum being outputted is outside the range that permits effective use of F conversion. If the use of E conversion is required, the scale factor has the same effect as when using the E format specification on output.

For further information refer to real conversion on page 7-15 using Ew.d.

FORMAT SPECIFICATION IN AN ARRAY.

Any of the formatted input/output statements may contain an array name in place of a FORMAT statement label. At the time the input/output statement containing the array reference is executed, the array must contain the equivalent of a FORMAT statement, with the first character being a left parenthesis. Any characters in the array following the final right parenthesis of the FORMAT statement in the array are ignored. There may not be embedded blanks between the left and right parentheses.

<u>Example</u>

<u>Program</u>

```
   ALPHA FORM(5)
   DIMENSION INFO(6)
   READ(5,75) FORM
75 FORMAT(5A6)
   READ(20,FORM) Q,R,(INFO(I),I=1,6)
   ...
   ...
```

<u>Input</u>

```
(F6.2,3X,E15.8,6I3)bbbbbbbbbbb
```

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:

Character	Action
Blank	One space before printing
Zero	Double space before printing
1	Skip to channel 1 of carriage control tape before printing.
Plus sign	No advance before printing

One of the above characters must be specified by a Hollerith constant, a skip (1X = blank), or a Tn; otherwise, a single space before printing carriage control is assumed, and the first character is not printed.

Examples

```
25 FORMAT(1HO,E12.6,A5)
```
Causes the carriage to double space before printing.

```
35 FORMAT(6H+TITLE)
```
Provides no carriage advance before printing.

```
45 FORMAT(3X,6I5)
```
Causes the carriage to single space before printing.

```
55 FORMAT(1H1,5HTITLE)
```
Causes the carriage to skip to channel 1 and print TITLE.

USE OF SLASH (/).

A slash in a FORMAT statement 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 FORMAT statement.

On output, the current record is terminated and any subsequent output is placed in the next record. Multiple slashes may be used to skip several records on input or create several blank records on output.

REPEAT SPECIFICATIONS.
Repetition of any format specification except nX or wHs is accomplished by preceding it with a positive integer constant called the repeat count. If the I/O list warrants it, the specified conversion is interpreted repetitively up to the specified number of times. If a scale factor is included, it must precede the repeat count.

Repetition of a group of format specifications is accomplished by enclosing them within parentheses and preceding the left parenthesis with a positive integer constant called the group repeat count, which indicates the number of times to interpret the enclosed groupings. If a group repeat count is not given, the group is repeated until the I/O list is exhausted. Grouping with parentheses may be continued to nine levels.

Example

```
     85 FORMAT(3E16.6,5(F10.5,I3,4A2))
```

FORMAT AND I/O LIST INTERACTION.
The execution of a formatted I/O statement initiates format control. If there is an I/O list, at least one format specification other than wHs or nX must exist in the FORMAT statement referenced.

When a formatted input statement is executed, one record is initially read. No other records are read unless specified by the FORMAT statement. The I/O list associated with a FORMAT statement may not require more data of a record than it contains.

When a formatted output statement is executed, writing of a new record occurs each time the FORMAT statement referenced so specifies. Terminating execution of a formatted output statement causes the current record to be written. A slash also causes the record to be written.

Except for the effects of repeat counts, the FORMAT statement is interpreted from left to right.

To each I, F, E, G, D, A, or L format specification there corresponds one element in the I/O list. A list element of type COMPLEX is considered, for purposes of I/O conversion, as two list elements of type REAL. Thus, there must be two format specifications (or a format specification preceded by a repeat count) for every list element of type COMPLEX.

There is no corresponding I/O list element for any wHs, Tn, or nX format specification. Instead, the information is inputted or outputted directly to or from the FORMAT statement.

If, under format control, the right-most right parenthesis of the FORMAT statement is encountered and the I/O list is still not exhausted, format control reverts to the last previously encountered left parenthesis. If a group repeat count precedes this left parenthesis, it also takes effect.

If, during execution of a formatted I/O statement, the I/O list is exhausted but the right-most right parenthesis of the specified FORMAT statement has not been encountered, execution of the I/O statement is complete.

SECTION 8
SUBPROGRAMS

GENERAL.

A subprogram is a program unit, a self-contained and independent routine, which may be referenced by the main program and by other subprograms. There are three types of subprograms:

a. FUNCTION.
b. SUBROUTINE.
c. BLOCK DATA.

FUNCTIONS.

In mathematics if the value of one quantity is dependent on the value or values of another quantity, it is said to be a function of the other quantity. The first quantity is called the function and the other quantities are called the arguments. For example, in

arctan(x)

arctan is the function and x is the argument.

Functions may be divided into three categories:

a. Statement.
b. Intrinsic.
c. External.

STATEMENT FUNCTIONS.

A statement function is declared within the program unit in which it is referenced. It is defined by a single statement similar in form to an Arithmetic or Logical Assignment statement.

The general form is:

$f(x_1,x_2,\ldots,x_n)=e$
where f is the statement function name, $x_1,x_2,\ldots,x_n$ are the dummy arguments, and e is an expression.

The rules for naming a function subprogram are the same as those for naming a variable (refer to section 2). The dummy arguments may be simple or subscripted variables. They represent values which are passed to the function subprogram and are used in the expression e in order to evaluate the function f. The dummy arguments are undefined outside of the statement function and may be redefined within the program unit. Together, f and e must conform to the rules for Arithmetic or Logical Assignment statements.

Aside from the dummy arguments, the expression e may contain:

a. Variables used in the program unit.
b. Intrinsic function references.
c. References to previously defined statement functions.
d. External function references.

A statement function must be defined before it is referenced.

A statement function is referenced in the same manner as a FUNCTION subprogram.

The name of a statement function must not appear in an EXTERNAL statement, nor as a variable name or an array name in the same program unit.

<u>Example</u>

```
       DIMENSION A(10)
       LOGICAL STAFUN,Y,Z
       STAFUN(N)=X .LT. SIN(A(N))
       READ(5,25)X,Y,(A(I),I=1,10)
    25 FORMAT(F8.2,L2,10F7.2)
       DO 50 J=1,10
       Z=Y .AND. STAFUN(J)
       ...
    50 ...
```

INTRINSIC FUNCTIONS.

The intrinsic functions are those functions made available to a FORTRAN object program by the operating system. The names, types, and definitions of the intrinsic functions are predefined, so they need only be referenced in order to be used.

An intrinsic function name may be redefined within a program unit. However, if it has been redefined, that intrinsic function is no longer recognized by the compiler, but its identifier is used as it has been redefined.

An intrinsic function is referenced by using it as a primary in an arithmetic or logical expression. The actual parameters which constitute the parameter list must agree in type, number, and order with the specifications in table 8-1, and may be any expression of the specified type.

Execution of an intrinsic function reference results in the passing of the actual parameter values to the corresponding formal parameters of the intrinsic function and an evaluation of the intrinsic. The resultant value is then assigned to the intrinsic function identifier and thereby passed back to the intrinsic function reference.

Examples

```
    IBIG = MAXO(I,J,K,LEST)
    TANGE = SIN(X+Y)/COS(A-B)
```

EXTERNAL FUNCTIONS.

An external function is a program unit which has as its first statement a FUNCTION statement.

The general form is:

t FUNCTION f(a_1,a_2,...,a_n)
where t is either INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or empty; f is the symbolic name of the function being defined; and a_1,...,a_n are formal parameters which may be either a variable name, an array name, a subroutine, or function name.

An external function must be referenced by another program unit, not by itself.

Table 8-1

Resulting Actions of an Intrinsic Function

Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Absolute value	$\|a\|$	1	ABS IABS DABS CABS	Real Integer Double Complex	Real Integer Double Real
Truncation	Sign of a times largest integer $\leq \|a\|$	1	AINT INT IDINT	Real Real Double	Real Integer Integer
Remaindering*	a_1 (mod a_2)	2	AMOD MOD DMOD	Real Integer Double	Real Integer Double
Choosing largest value	Max $(a_1, a_2 \ldots)$	≥ 2	AMAX0 AMAX1 MAX0 MAX1 DMAX1	Integer Real Integer Real Double	Real Real Integer Integer Double
Choosing smallest value	Min $(a_1, a_2 \ldots)$	≥ 2	AMIN0 AMIN1 MIN0 MIN1 DMIN1	Integer Real Integer Real Double	Real Real Integer Integer Double
FLOAT	Conversion from integer to real	1	FLOAT	Integer	Real

Table 8-1 (cont)

Resulting Actions of an Intrinsic Function

Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of sign	sign of a_2 times $\|a\|$	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double
Positive difference	a_1-Min (a_1, a_2)	2	DIM IDIM	Real Integer	Real Integer
Obtain most significant part of double precision argument		1	SNGL	Double	Real
Express single precision argument in double precision form		1	DBLE	Real	Double
Obtain real part		1	REAL	Complex	Real
Obtain imaginary part		1	AIMAG	Complex	Real
Create complex	$C = A_1 + ia_2$	2	CMPLX	Real	Complex
Complex conjugate	$C = X - iY$	1	CONJG	Complex	Complex
Exponential	e^a	1 1	EXP DEXP CEXP	Real Double Complex	Real Double Complex

Table 8-1 (cont)

Resulting Actions of an Intrinsic Function

Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Natural logarithm	$\log_e$ (a)	1	ALOG	Real	Real
		1	DLOG	Double	Double
		1	CLOG	Complex	Complex
Common logarithm	$\log_{10}$ (a)	1	ALOG10	Real	Real
		1	DLOG10	Double	Double
Trigonometric sine	sin (a)	1	SIN	Real	Real
		1	DSIN	Double	Double
		1	CSIN	Complex	Complex
Trigonometric cosine	cos (a)	1	COS	Real	Real
		1	DCOS	Double	Double
		1	CCOS	Complex	Complex
Arctangent	arctan (a)	1	ATAN	Real	Real
		1	DATAN	Double	Double
Arctangent	arctan (a_1/a_2)	2	ATAN2	Real	Real
		2	DATAN2	Double	Double
Square root	$(a)^{1/2}$	1	SQRT	Real	Real
		1	DSQRT	Double	Double
		1	CSQRT	Complex	Complex
Hyperbolic tangent	tanh (a)	1	TANH	Real	Real

Table 8-1 (cont)

Resulting Actions of an Intrinsic Function

Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Trigonometric tangent	tan (a)	1	TAN	Real	Real
	NOTE Where applicable, trigonometric functions must be in radians.				

* The functions MOD, AMOD, and DMOD (a_1,a_2) are defined as $a_1 - [a_1/a_2]*a_2$, where [a] denotes the integral part of a.

The construction of external functions is subject to the following conditions:

a. The function name must be used as a variable within the function subprogram to the left of the replacement operator (=) in an assignment statement at least once. Its value at the time of execution of any RETURN statement within the function subprogram is the value of the function.

b. The name of the function must not appear in any non-executable statement in the function subprogram, except for the FUNCTION statement.

c. The symbolic names of the formal parameters may not appear in an EQUIVALENCE, COMMON, or DATA statement in the function subprogram.

d. The function subprogram may define or redefine one or more of its parameters to effectively return results in addition to the value of the function.

e. The function subprogram may contain any statements except SUBROUTINE, another FUNCTION statement, or BLOCK DATA.

f. The function subprogram must contain at least one RETURN statement.

g. An END statement must be the last statement of the subprogram body.

<u>Example</u>

```
   FUNCTION EVAL(U,V)
   IF(U .LT. V) GO TO 1
   EVAL=V/U
   RETURN
1  EVAL=U/V
   RETURN
   END
```

REFERENCING EXTERNAL FUNCTIONS.

An external function is referenced by using it as a primary in an arithmetic or logical expression. The actual parameters, which constitute the parameter list, must agree in order, number, and type with the corresponding formal parameters in the defining program. An actual parameter in an external function reference must be one of the following:

a. A Hollerith constant.
b. A variable name.
c. An array element name.
d. An array name.
e. An expression.
f. The name of a function or a subroutine.

If an actual parameter is a function name (external or intrinsic) or a subroutine name, the corresponding formal parameter must be used as a function name or a subroutine name, respectively.

If an actual parameter corresponds to a formal parameter that is defined or redefined in the referenced subprogram, the actual parameter must be a variable name, an array element name, or an array name. Execution of an external function reference, as described in the foregoing, results in an association of actual parameters with all appearances of corresponding formal parameters in the executable statements of the subprogram, and in an association of actual parameters with variable dimensions, if present, in the subprogram. Following these associations, execution of the first executable statement of the subprogram body is undertaken.

An actual parameter which is an array element name containing variables in the subscript can, in every case, be replaced by the same parameter with a constant subscript containing the same values as can be derived by computing the variable subscript just before association of parameters takes place.

If a formal parameter of an external function is an array name, the corresponding actual parameter must be an array name or array element name.

Example

```
     TOTAL=EVAL(P,X) + CPS(Y)
```

SUBROUTINE.

A subroutine is defined externally to the program unit that references it. A subroutine defined by a FORTRAN statement headed by a SUBROUTINE statement is called a subroutine subprogram.

DEFINING SUBROUTINE SUBPROGRAMS.

The SUBROUTINE statement is one of the forms:

1. SUBROUTINE n
2. SUBROUTINE n $(a_1,a_2,\ldots,a_n)$
where n is the symbolic name of the subroutine to be defined; the a's are formal parameters which may be either a variable name, an array name, a function or subroutine name.

The construction of subroutine subprograms is subject to the following restrictions:

a. The symbolic names of the formal parameters may not appear in an EQUIVALENCE, COMMON, or DATA statement in the subprogram.

b. The subroutine subprogram may define or redefine one or more of its parameters in order to effectively return results.

c. The subroutine subprogram may contain any statements except FUNCTION, another SUBROUTINE statement, or BLOCK DATA.

d. The subroutine subprogram must contain at least one RETURN statement.

e. An END must be physically the last statement.

Example

```
SUBROUTINE FALL(T,V,S)
G=32.172
S=G*T**2/2
V=G*T
RETURN
END
```

BLOCK DATA.

Further use of the DATA statement is in the BLOCK DATA subprogram. It is used to enter data into COMMON blocks; however, the following must be observed:

a. There may be no executable statements in a BLOCK DATA subprogram. The first statement of the subprogram must be BLOCK DATA.

b. The subprogram may contain only Type, EQUIVALENCE, DATA, DIMENSION, and COMMON statements.

c. All elements of a COMMON BLOCK must appear in the COMMON statement list even though some do not appear in the DATA statement list.

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

e. There may be as many BLOCK DATA subprograms as desired in a program. Any common block identifier may occur in only one BLOCK DATA subprogram unless an INITIAL Card is used to specify the BLOCK DATA subprogram to be used for initialization.

f. Variables in a DATA statement must be listed in the order in which they appear in a COMMON statement.

Example

```
      BLOCK DATA
      COMMON/TEST/K,L,S/AATWO/B,C
      DIMENSION C(10)
      DATA L,S/ 1, 3.5/, C/ 10*16.2/
      END
```

APPENDIX A

B 2500/B 3500 FORTRAN VERSUS B 5500 FORTRAN

The constructs cited below are characteristics of B 2500/B 3500 FORTRAN and indicate differences between B 2500/B 3500 and B 5500 FORTRAN:

a. Only one statement is allowed per card as opposed to two or more statements separated by semicolons.

b. The character set does not include the (") quote sign. Therefore, all literal strings must be designated as Hollerith (H) fields.

c. The relational operators $<$, $\leq$, $\neq$, $>$, and $\geq$ are not allowed. The FORTRAN mnemonics .LT., .LE., .NE., .GT., and .GE. must be used.

d. The maximum number of dimensions which can be declared for an array is three. The maximum number of elements in an array is 9999.

e. A subscript may not be a REAL expression. It may, however, be an INTEGER constant, variable, or expression.

f. In the statement:

   GO TO i, $(k_1, k_2, \ldots, k_n)$

   i may be an INTEGER variable, never a REAL variable.

g. In the statement:

   GO TO $(k_1, k_2, \ldots, k_n)$, i

   i may be an INTEGER variable, never a REAL variable.

h. In the statement:

   IF (l.e.)s

s may be any executable statement except a DO statement or an IF statement. B 5500 FORTRAN permits usage of IF statements for s.

i. B 5500 FORTRAN allows the terminal statement of a DO loop to be any executable statement. B 2500/B 3500 FORTRAN forbids usage of a GO TO of any form, an IF statement, RETURN, STOP, PAUSE, or DO statement.

j. In the statement:

DO m i = n_1, n_2, n_3

i may not be a REAL variable, only an INTEGER variable. The terms n_1, n_2, n_3 may be either INTEGER constants or INTEGER variables. They may not be INTEGER or REAL expressions as is permitted in B 5500 FORTRAN.

k. CLOSE u, LOCK u, and PURGE u are not permitted.

l. In I/O and auxiliary I/O statements, the unit number (u) may not be an expression.

m. NAMELIST is not allowed.

n. PRINT and PUNCH I/O statements are not permitted.

o. The intrinsics: COTAN, ARSIN, ARCOS, ERF, GAMMA, ALGAMA, AND, OR, COMPL, EQUIV, CONCAT, and TIME are not available.

p. Non-standard returns from subroutines are not permitted.

q. Multiple entry points to subprograms are not permitted.

r. Each of the two components of a COMPLEX constant may be REAL only, not INTEGER.

B 2500/B 3500 FORTRAN VERSUS B 5500 FORTRAN

s. The format specified Ow is not permitted.

t. Recursive subroutines are not allowed.

u. STOP n and PAUSE are not available.

v. An EQUIVALENCE statement may not be used to extend the size of a COMMON block at its beginning, only at its end.

w. No more than nine DO statements may be nested within the range of another DO statement. In other words, DO statements may be nested no more than nine deep.

x. CALL EXIT is treated as a STOP statement.

y. The maximum number of parameters allowed in a subprogram argument list is 42.

z. There is no check for divide by zero or exponent overflow or underflow.

APPENDIX B
SOFTWARE LIBRARY

Three types of FORTRAN subprograms appear on the Systems tape. The first of these, READ., WRITE., RWIND., EXPON., ALOG, EXP, DLOG, DEXP, are called by the FORTRAN Compiler. READ and WRITE are implicitly called when a READ and WRITE statement are used in the program. The other five routines are called when the ** operator is used. Another group of subprograms are described in table 8-1.

In addition to these functions, some commonly used subroutines are provided. The purpose and parameters of these subroutines are described below.

SUBROUTINE ACCEPT.

The purpose of the ACCEPT subroutine is to receive data from another job in the mix. It is of the form:

```
    CALL ACCEPT (DATA, NCH, LV, HOLLER)
```

DATA may be a variable name or an array name of any data type. It specifies into which variable or array the data which are received are to go.

NCH is an integer constant or integer variable name. It specifies the number of characters of DATA to be received. NCH must be less than 10000. Note that one character is equivalent to two digits.

LV is a logical variable. If ACCEPT actually receives the data from a sending program, LV is set to .TRUE. If there is no program sending data when ACCEPT is ready to receive data, LV is set to .FALSE.

HOLLER is a Hollerith constant or variable containing Hollerith data. It specifies the program identifier from which the NCH characters of DATA are being sent. Note that HOLLER is not a subprogram name, but the name of a code file. The Hollerith data are exactly six characters, ending in blanks if necessary.

NOTE

The core-to-core option (CRCR) must be set in CP--S versions of the Master Control Program.

Example

A program expects to receive a real array with 10 elements from another program in the mix called TRANS.

```
      LOGICAL LV
      DIMENSION BRAY (10)
           .
           .
10    CALL ACCEPT (BRAY, 60, LV, 6HTRANSb)
      IF (.NOT. LV) GO TO 10
           .
           .
      STOP
      END
```

Assume a default size of 12 digits or six characters per real element. Then NCH is 60 since 10 elements at six characters per element is 60 characters.

The data being received from TRANS go into BRAY.

The Logical IF statement states that if LV is .FALSE. (i.e., data have not been received), loop back to statement 10 and try again. In other words, the program keeps looping and waiting for TRANS to send the data.

NOTE

Refer to the description of SUBROUTINE SEND for the method of sending data.

SUBROUTINE CHANGE.

This subroutine changes the file identification associated with a logical unit number and can be used to reduce the core requirement of a program when logically distinct files with identical attributes are to be processed. It is of the form:

```
    CALL CHANGE(⟨file number⟩,⟨new file identifier⟩)
```

The ⟨file number⟩ is an integer constant or variable which is the unit number of the file to be changed.

The ⟨new file identifier⟩ is a 6-character Hollerith constant or ALPHA variable.

All processing on the first file should be completed before it is changed. The first file is closed with no rewind by the CHANGE subroutine.

The attributes of the new file must be identical to the attributes of the original file.

Example

```
    FILE   8=TAPE1,BUFFERS=1,FIXED,RECORD=80,BLOCKING=10
           ...........
           READ(8,10) A,B,C
           READ(8,20) D,E,F
           ...........
           ...........
           CALL CHANGE(8,6HTAPE2 )
           ...........
           WRITE(8,50) RSULT,XMAX
```

In the example all references to unit 8 prior to execution of the CHANGE subroutine are to FILE1. Subsequent references are to FILE2.

SUBROUTINE CLOSE.

The purpose of the CLOSE subroutine is to close a file. When CLOSE is not specified by the programmer, files opened implicitly by a READ or WRITE remain open until the end of execution. Its form is:

```
     CALL CLOSE (IUNIT, HOLLER)
```

The term IUNIT is an integer constant or integer variable name which specifies the unit number of the file to be closed.

HOLLER signifies a Hollerith constant or a variable containing Hollerith data which specify the type of CLOSE to be performed.

Only the first two characters of HOLLER are used and have the following meanings:

First Character	Meaning
R	Reel close (for use with multireel tape files)
F	File close

Any other character (including a blank) is regarded as an implicit FILE CLOSE.

Second Character	Meaning
Blank	Normal CLOSE (rewind, retain)
N	CLOSE, NO REWIND, RETAIN
R	CLOSE WITH RELEASE and REWIND*
L	CLOSE WITH REWIND and LOCK
P	CLOSE WITH REWIND and PURGE*
C	CLOSE WITH CLOBBER (disk file with same name is removed)
Any other	CLOSE WITH REWIND and RETAIN

* Magnetic tape units are returned to the MCP as available for use with other programs in the mix.

<u>SUBROUTINE DATE</u>.

The purpose of the DATE subroutine is to obtain the contents of the current date word used by the MCP. It is of the form:

```
     CALL DATE (IM, ID, IY)
```

IM is an integer variable which, after the execution of DATE, contains the month that is in the MCP date word. The value is an integer between 1 and 12 inclusive.

ID is an integer variable which, after the execution of DATE, contains the day that is in the MCP date word. The value is an integer between 1 and 31 inclusive.

IY is an integer variable which, after the execution of DATE, contains the year that is in the MCP date word.

NOTE

Each of the variables IM, ID, IY may be printed out with the format specification of I2.

The MCP date word may be set by the operator with a DT SPO message. DT 12/20/68 sets the value of the MCP date word to 122068. Thus, subsequent use of the DATE subroutine sets the value of IM to 12, the value of ID to 20, and the value of IY to 68.

<u>SUBROUTINE EXIT</u>.

The purpose of the EXIT subroutine is to stop the execution of an object program. Its form is:

```
     CALL EXIT
```

The CALL EXIT statement does exactly the same things as a STOP statement. It is included to allow existing FORTRAN programs that already use CALL EXIT to run without making any changes.

SUBROUTINE FMDUMP.

The purpose of this subroutine is to produce an analysis of the memory of a program during its execution. It is of the form:

```
CALL FMDUMP(N,L)
```

N is the unit number of a PRINTER file; the output from FMDUMP appears on this printer.

L is a logical variable. If its value is TRUE when FMDUMP is called, a memory dump is produced with the analysis. If the value of L is FALSE, only the analysis is produced.

The analysis provides the following information:

a. Sizes of data types and storage unit size.

b. Contents of index registers.

c. Number of program segments and the segment last brought into memory.

d. Analysis of each program segment giving its status (IN or NOT IN), base relative beginning and ending addresses, and the address of the first executable instruction.

e. Analysis of each file giving the unit number, address of the file information block (FIB), device type, label, status (closed or open), blocking factor, address of the work area if one is used, number of buffers, and the disk key for a random disk file.

f. Analysis of stack in reverse order of calling (latest stack entry to first entry), giving the address in the stack where the entry begins, contents of the return control word, number of parameters passed with the associated NTR, and the stack address of the first five parameters.

SUBROUTINE SEND.

The purpose of the SEND subroutine is to send data to another job in the mix. It is of the form:

```
CALL SEND (DATA, NCH, HOLLER)
```

DATA may be a constant, a variable name, or an array name of any data type. DATA specifies the data to be transferred.

NCH is an integer constant or integer variable name. It specifies the number of characters of DATA to be transferred. Note that one character is equivalent to two digits.

HOLLER is a Hollerith constant or variable containing Hollerith data. It specifies the program identifier to which the NCH characters of DATA are being sent. Note that HOLLER is not a subprogram name, but the name of the code file. The Hollerith data are exactly six characters, ending in blanks if necessary.

Example

Suppose a programmer decides to send a real array of 10 elements to a program in the mix called ACCT. Let the name of this array be ARAY. The FORTRAN program includes the following:

```
DIMENSION ARAY (10)
      .
      .
      .
      CALL SEND (ARAY, 60, 6HACCTbb)
           .
           .
      STOP
      END
```

NCH has a value of 60 since we are assuming that a real element contains 12 digits or six characters.

NOTE

The core-to-core option (CRCR) must be set in CP--S versions of the Master Control Program.

If the program specified in HOLLER (ACCT in the example) is not yet ready to receive the data, the sending program waits until the receiving program is ready.

Refer to the description of SUBROUTINE ACCEPT for the method of receiving data.

<u>SUBROUTINE SPOACP</u>.

The purpose of this subroutine is to accept input from the system SPO. The data are placed in a specified variable and may be either INTEGER or ALPHA. It is of the form:

```
    CALL SPOACP(N,VAR)
```

N is an integer constant or variable whose value is 0 or 1. If N=0, the input is to be considered INTEGER; if N=1, the input is to be considered ALPHA.

VAR is an integer or alpha variable (simple or subscripted) which receives the SPO input. VAR must be of type ALPHA or INTEGER depending on the type of input designated by N.

If the input is integer, only the integer digit of each character is placed in VAR.

If the SPO input is longer than the length of the receiving variable, it is truncated. If it is shorter, it is left-justified in the field with trailing blanks or zeros.

Example

```
        ALPHA MEST(4)
        DO 20 I=1,4
        CALL SPOACP(1,MEST(I))
     20 CONTINUE
```

SUBROUTINE SPOMSG.

The purpose of this subroutine is to display a message at the system SPO. It is of the form:

```
CALL SPOMSG(⟨character count⟩,⟨message⟩)
```

The ⟨character count⟩ is an integer constant denoting the number of characters in the message to be displayed.

The ⟨message⟩ is a Hollerith constant of up to 60 characters which is the message to be displayed.

Example

To display "MOUNT A SCRATCH TAPE ON A 9-CHANNEL DRIVE" at the system SPO, code:

```
    CALL SPOMSG(41,41HMOUNT A SCRATCH TAPE ON A 9-CHANNEL DRIVE)
```

SUBROUTINE TIME.

The purpose of this subroutine is to obtain the contents of the current time word used by the MCP. It is of the form:

```
CALL TIME (T1, T2)
```

T1 may be either a real constant or a real variable name. It is used as input to the TIME routine.

T2 is a real variable. After execution of TIME, T2 contains the difference between the current time and T1. That is, if the current time is TC, then T2 has the value TC - T1. The unit of time is milliseconds.

To obtain the current time, the following CALL TIME statement can be used:

```
     CALL TIME (O., T2)
```

Example

To time out a certain DO loop, the following FORTRAN statements can be used:

```
          CALL TIME (O., TME)
          DO 20 I = 1, 1000
              .
              .
      20  CONTINUE
          CALL TIME (TME, TME)
```

TME contains the length of time (in milliseconds) required to execute the DO 20 loop.

SUBROUTINE TRACE.

The TRACE subroutine is used to turn a trace on and off during the execution of an object program. It may also be used to give a complete or partial dump of the object program. Its form is:

```
     CALL TRACE (I)
     CALL TRACE (21,I,J)
```

When using the first option, I is an integer constant or variable and is used as an input parameter to the TRACE routine. I can have the following values:

Value of I	Meaning
0	Turn off TRACE
1	Turn on normal state TRACE
2	Turn on control state TRACE*
3	Turn on normal and control state TRACE*
20	Dump entire program

When tracing, the program should be alone in the mix.

The second option permits the user to selectively dump part of his program. I is an integer constant or variable which is the base relative address at which the dump begins. J is an integer constant or variable which is the base relative address at which the dump ends.

Examples

To trace the object time execution of a set of FORTRAN statements:

```
CALL TRACE (1)
.
.
.  series of statements
.  to be traced
.
CALL TRACE (0)
```

To dump core between base relative addresses 3150 and 9000:

```
CALL TRACE(21,3150,9000)
```

SUBROUTINE ZIP.

The ZIP subroutine is used to execute a control card from a currently executing program. Its form is:

```
CALL ZIP (ARGM)
```

* Can be used only if MCP TRACE option is set (i.e., TRAC=1)

ARGM must be a Hollerith constant or variable or an ALPHA array name containing Hollerith data. The Hollerith data may be any valid control card ending with a period. Note that all control cards start with the two characters CC.

The execution caused by the execution of the control card contained in ARGM is carried on concurrently with the execution of the program that contains the CALL ZIP statement.

Example

```
? COMPILE ZIPRGM WITH FORTRAN
? DATA CARDS
                 .
                 .
        10 CALL ZIP (18 HCC EXECUTE MATMPY.)
                 .
                 .
                 .
           STOP
           END
? END
```

When the execution of ZIPRGM reaches statement 10, the execution of the ZIP routine causes the MCP to start the execution of the program MATMPY (if it exists on disk). The executions of ZIPRGM and MATMPY then proceed simultaneously.

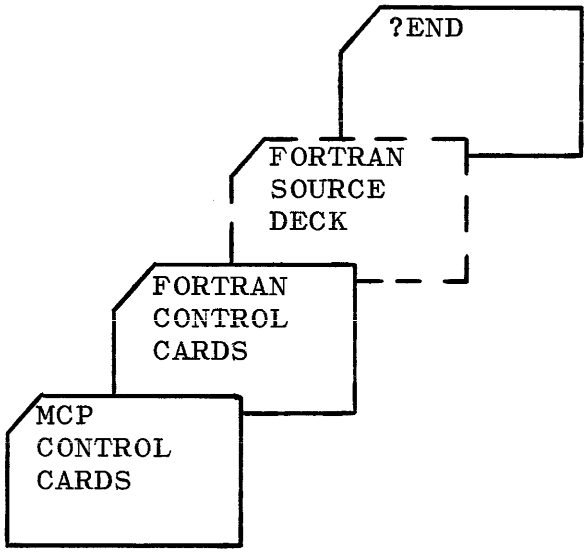
The same results can be obtained with the following FORTRAN program:

```
    ALPHA ARAY (3)
    DATA ARAY / 18HCC EXECUTE MATMPY. /
        .     / 6HCC EXE,6HCUTE M, 6HATMPY./
        .
        .
10  CALL ZIP (ARAY)
        .
        .
        .
    STOP
    END
```

APPENDIX C
CONTROL CARDS

GENERAL.

When a FORTRAN program is compiled the functions to be performed by the compiler are specified through the use of control cards. The format and effect of each of these cards are described below. The following diagram presents the fundamental components of a FORTRAN source deck.



MCP CONTROL CARDS.

The first control card instructs the MCP to compile with FORTAN (the FORTRAN Compiler) the indicated program name ⟨p-n⟩ using one of the following options:

a. ?COMPILE ⟨p-n⟩ WITH FORTAN. This option causes the symbolic program to be compiled and executed. The pseudo code file (also referred to as Independently Compiled Subroutine, ICS, file) of each program part is entered on disk, but the object program is not placed in the library. (Refer to appendix F for a description of an ICS file.)

b. ?COMPILE ⟨p-n⟩ WITH FORTAN LIBRARY. This option causes the symbolic program to be compiled and the object program to be entered in the disk directory with the ID ⟨p-n⟩. An ICS is also created and placed on disk for each program part. The ICS file of a main program is given the ID PROGAM unless an IDENT Card is used; the ICS file of a subroutine has the name of the subroutine as the file identifier. (If an IDENT Card is used, the name specified must be the subroutine name; refer to IDENT Card which is described below.)

c. ?COMPILE ⟨p-n⟩ WITH FORTAN SYNTAX. This option causes the symbolic program to be compiled and checked for syntax errors. The ICS file for each program part automatically replaces its namesake on disk if it is syntax- and flag-free. The compilation of a program part that contains flags, but no errors, causes a DUP LIB message to be displayed on the SPO, leaving its replacement on disk to the operator's discretion. The compilation of a program part which contains errors does not result in an ICS file being placed on disk. The object program is <u>not</u> entered in the disk directory.

NOTE

The logic governing automatic replacement of ICS files on disk described in c above is used for all compile options.

The second control card is the label card which provides file identification. Its format is:

a. ?DATA CARDS for EBCDIC source language input.
b. ?DATAB CARDS for BCL source language input.

NOTE

Refer to HOLL Card, which is described below, for control cards to be used when the input is BCD.

CONTROL CARDS

The last card in the deck is the END Card, which signifies the physical end of the card file to the MCP. Its format is:

```
?END
```

FORTRAN CONTROL CARDS.

The following control cards may be included optionally in the source deck to define particular user requirements to the compiler. Each control card consists of a key word which must begin in column 1 and additional information, coded in columns 7 through 72. Continuation cards are permitted as defined for the FORTRAN language. Blanks appearing in columns 7 through 72 are ignored, and commas are used as delimiters.

FILE CARD.

A FILE Card is used to define the attributes of a file when other than default attributes are desired. A FILE Card may contain the description of one file; multiple FILE Cards are permitted.

A FILE Card is coded as follows:

a. The key word FILE is coded in columns 1-4.

b. n=⟨file ID⟩,UNIT=⟨hardware type⟩, optionally followed by an ⟨attribute list⟩, is coded in columns 7-72. Specifications are free-field and delimited by commas. Continuation cards may be used.

The ⟨file ID⟩ may be specified as ⟨multi-file ID⟩/⟨file ID⟩ for tape files only.

The ⟨hardware type⟩ is specified as follows:

Reserved Word	Device Type
PRINTER	Line printer
PRINT	Line printer

CONTROL CARDS

Reserved Word	Device Type
READER	Card reader
TAPE	7- or 9-channel magnetic tape
TAPE9	9-channel magnetic tape
TAPE7	7-channel magnetic tape
DISK	Disk
PUNCH	Card punch
PTP	Paper tape punch
PTR	Paper tape reader

By default the FORTRAN Compiler associates a unit number referenced in an I/O statement with a particular hardware type, which is, in turn, associated with a file description. These default associations are given in table C-1.

The FILE Card may be used to specify associations different from those given above. Any number of attributes listed for each file description may be redefined; those attributes not specifically defined in a FILE Card retain the default condition for the hardware type. For example, FILE 10=DSKFIL,UNIT=DISK,RECORD=100 redefines unit number 10 and associates it with a disk file named DSKFIL. The record length is redefined at 100 characters. All other attributes are those assigned by default for disk, such as blocking factor of 2, sequential access, and so forth.

Table C-1

Unit Number/Hardware Type Default Associations

Unit Number	Hardware Type	Parity	Label	No of Buffers	Work Area	Blocking Factor	Record Length	Records Per Area
1-4 10-19	Magnetic tape (7- or 9-channel	Odd	FILEn (n = unit no.)	2	Yes	Variable	100UA maximum (variable)	
5	Card reader		FILE5	2	No	1	80UA	
6-8	Line printer	Even	FILE6	2	No	1	132UA	
7	Card punch		FILE7	2	No	1	80UA	
9	Disk	Even	FILE9	2	No	2	50UA	100

CONTROL CARDS

The reserved words which may be used in the ⟨attribute list⟩ and their meanings are given below.

Definition (Reserved Word)	Meaning
	All Hardware Devices
UNLABELED	Unlabeled file (not valid for disk)
OPTIONAL	Optional file
BLOCKING=⟨unsigned integer⟩	Number of logical records per block
BUFFERS=⟨unsigned integer⟩	Number of buffers
RECORD=⟨unsigned integer⟩	Record length in characters
WORKAREA	Assigns work area for file
	Magnetic Tape
ALPHA	Even parity
FIXED	Fixed length record (must be specified when I/O statements reference a FORMAT)
SAVE=⟨unsigned integer⟩	Assigns save factor of ⟨unsigned integer⟩ days to retain file
TRANSLATE	Causes translation of characters from 8-bit form to 6-bit form for 7-channel tape. (This attribute is for 7-channel tape only and must be used.)

CONTROL CARDS

Definition (Reserved Word)	Meaning
	Disk
LOCK	Enters file in disk directory at CLOSE time (when STOP or CALL EXIT is executed)
RANDOM	Random access technique. (A work area is assigned in addition to two buffers.)
SEQIODISK	Sequential I/O access technique
EU=⟨unsigned integer⟩	Specifies EU on which file is to reside
AREA=⟨unsigned integer⟩	Number of records per area (20 areas are assumed)
PACKED	Item in unformatted record is length of its data size (by default each item is SU size in length)
	Line Printer
BACK	Forces file to backup
NOBACK	Prohibits file from going to backup

A file must be defined with a FILE Card when an integer variable is used as its unit designator in an I/O source statement.

Examples

```
FILE  8=TAPER,UNIT=TAPE,FIXED,
     *RECORD=80,BLOCKING=10
      DIMENSION PRAY(8)
      ........
      WRITE(8,10) PRAY
```

```
    10 FORMAT (8F10.3)
       ........
       STOP
       END
```

In the above example unit 8 is designated a tape file with the ID TAPER. Records are a fixed length of 80 characters with 10 records per block. The FIXED specification is necessary because the records being written are formatted.

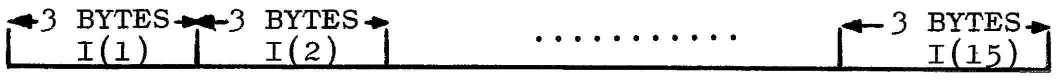
```
  FILE 5=FILE5,UNIT=READER
       DIMENSION P(10,15),R(10,15)
       N=5
       DO 10 I=1,15
       READ (N,20) P(1,I),R(2,I)
    20 FORMAT (2F10.3)
    10 CONTINUE
       ........
       STOP
       END
```

In the above example the FILE Card is needed because the integer variable N is used as a unit designator in a READ statement. All default attributes are associated with the file.

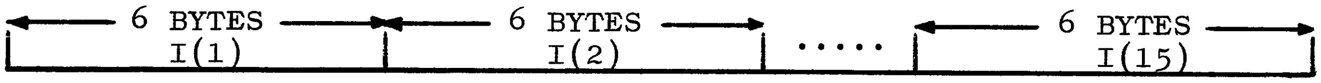
```
  FILE 9=RANFIL,UNIT=DISK,RANDOM,PACKED,AREA=500
       DIMENSION I(15)
       WRITE(9=5) I
       ........
       ........
       STOP
       END
```

In the above example RANFIL is a random disk file whose records consist of packed data written without a FORMAT. There are 500 records per disk area, with the file expandable to 10,000 records.

By default each element written unformatted to a disk file occupies SU number of digits on disk. The PACKED specification causes each element to occupy a number of bytes on disk equal to the size of its data type. In the above example, with the PACKED specification and default sizes, the following record is written.



Without PACKED, RECORD=90 must be declared, and the following record is written.



```
FILE  4=MASFIL/FIRST,UNIT=TAPE
```

In the above example the FILE Card defines a multi-file tape file with the multi-file ID MASFIL and the file ID FIRST. All other attributes are default, i.e., variable length records, and so forth.

HOLL CARD.
The HOLL Card must be used when BCD symbolic source input is to be compiled.

The HOLL Card is coded with the letters HOLL in columns 1-4.

?DATAB CARDS is required as the MCP label control card.

The HOLL Card causes the following characters to be translated for compilation:

a. % to (.
b. [to).
c. & to +.
d. # to =.

<u>Example</u>

```
?COMPILE ⟨p-n⟩ FORTAN
?DATAB CARDS
```

```
HOLL
.
.       BCD source
.       deck
.
?END
```

IDENT CARD.

The IDENT Card may be used to specify an identifier for a main program or a subprogram.

The IDENT Card immediately precedes the program part it names and is coded as follows:

a. The key word IDENT is coded in columns 1-5.

b. An identifier which consists of one to six alphanumeric characters, the first of which is alphabetic, is coded in columns 7-72.

The specified identifier is used as the ICS file ID for the associated program part. When an IDENT Card does not precede the main program, its ICS file has the default ID, PROGAM. The ICS file ID of a subprogram is the subprogram name by default.

An IDENT Card which names a subroutine must specify the name of the subroutine as the identifier.

If a REPLACE Card is used, the symbolic tape must contain IDENT Cards for every program part. (Refer to REPLACE Control Card which is described below.)

The program name, ⟨p-n⟩, used in the COMPILE Card should not be specified as an identifier in an IDENT Card. This causes a ** DUP LIBRARY ⟨p-n⟩ message to be displayed, and an RM message causes the ICS file with the ID, ⟨p-n⟩, to be removed and the object code file to be placed on disk.

Example

```
    IDENT MAIN
          READ (5,10) DATA
       10 FORMAT (F10.2)
          CALL MYSUB
          ..........
          ..........
          END
    IDENT MYSUB
          SUBROUTINE MYSUB
          ...........
          ...........
          RETURN
          END
```

INITIAL CARD.

The INITIAL Card specifies the names of BLOCK DATA subprograms which have been compiled independently and are to be used to initialize COMMON. It must be used in all cases when an IDENT Card is used to name a BLOCK DATA subprogram.

The INITIAL Card is coded as follows:

a. The key word INITIAL is coded in columns 1-7.

b. One or more BLOCK DATA subroutine names are coded in columns 9-72, delimited by commas.

By default BLOCK DATA subroutines are named BD.001, BD.002,..., BD.00n sequentially as they appear in the source file. The term n is equal to the number of BLOCK DATA subprograms in the source file. An IDENT Card may be used to assign a specific identifier to a BLOCK DATA subprogram, in which case an INITIAL Card must be included in the source deck.

An INITIAL Card is not necessary when the required BLOCK DATA subprogram is included in the compile deck of the program for which initialization is desired and is not preceded by an IDENT Card.

CONTROL CARDS

The INITIAL Card is _needed_ in the following example.

Example

```
?COMPILE XTRA FORTAN SYNTAX
?DATA CARDS
IDENT FSTBLK
      BLOCK DATA
      COMMON /FIRST/ K,M,N
      DATA K,M,N/1,2,3/
      END
      BLOCK DATA
      COMMON /SECND/ R,S,T
      DATA R,S,T/5.2,3.98,1.0/
      END
?END
?COMPILE BIGPRO FORTAN LIBRARY
?DATA CARDS
SEGMENT READ.,WRITE.
LOAD  PROGAM
INITIAL FSTBLK, BD.001
?END
```

An INITIAL Card is _not_ _needed_ in the following example.

Example

```
?COMPILE BIGPRO FORTAN LIBRARY
?DATA CARDS
SEGMENT READ., WRITE.
      COMMON /FIRST/ K,M,N /SECND/ R,S,T /THIRD/ A(20)
      ..........
      ..........
      ..........
      END
      BLOCK DATA
      COMMON /FIRST/ K,M,N
      DATA K,M,N/1,2,3/
      END
      BLOCK DATA
      COMMON /SECND/ R,S,T
      DATA R,S,T/5.2,3.98,1.0/
      END
?END
```

LOAD CARD.

The LOAD Card is used to create an object code file when the main program is not included in the source deck.

The LOAD Card is coded as follows:

a. The key word LOAD is coded in columns 1-4.
b. The identifier of the main program is coded in columns 7-72.

Unless it is preceded by an IDENT Card, the main program identifier is PROGAM.

The LOAD Card provides the name of the ICS file of the main program to the compiler. This is essential for proper linkage during the "load" phase of the compilation.

All desired FORTRAN control cards (except IDENT for the main program) must be included in the source deck, i.e., FILE Cards, SIZE Card, and so forth.

Only one LOAD Card may be included in a source deck.

A LOAD Card may be used in conjunction with a REPLACE Card.

Example

```
?COMPILE BIGPRO FORTAN LIBRARY
?DATA CARDS
LOAD  MAIN
?END
```

The above example creates an object program with the ID BIGPRO and enters it in the disk directory. It is assumed that the ICS files for the main program and any referenced subroutines reside on disk, and that the main program has been compiled with an IDENT Card containing the identifier MAIN.

Example

```
?COMPILE SMPRO FORTAN
?DATA CARDS
LOAD  PROGAM
FILE  9=RANFIL,UNIT=DISK,RANDOM
SIZE  REAL=20
      SUBROUTINE INERR(R,N,T)
      .........
      .........
      .........
      RETURN
      END
?END
```

When a sizable program is compiled and only part of the program requires recompilation, the LOAD Card can be used effectively to reduce recompilation time. As in the above example, the subroutine to be recompiled is included in the source deck, and a LOAD Card causes the object code file to be created. Thus, recompilation of the entire program is not performed.

REPLACE CARD.
The REPLACE Card is used to recompile a program part (main program or subroutine) which resides on a symbolic tape file.

The REPLACE Card is coded as follows:

a. The key word REPLACE is coded in columns 1-7.

b. The program part identifier for which recompilation is desired is coded in columns 9-72.

An IDENT Card must precede every program part on the symbolic tape. The identifier coded in the REPLACE Card is that which appears in the IDENT Card for the desired program part.

Patch cards for the program part follow the REPLACE Card.

The specified program part is recompiled and its ICS file is replaced on disk according to the rules outlined under MCP Control Cards (discussed previously in this appendix).

The symbolic tape must have been created as follows:

a. The main program is first on the tape.

b. The IDENT Card for the main program is the first card image on the tape.

c. No card images are between an END Card and a subsequent IDENT Card.

d. Sequence numbers are present to allow merging and/or replacement with patch cards.

The following card deck creates a symbolic tape file against which a REPLACE Card can be used.

Example

```
?COMPILE MATINV FORTAN DATA CARDS
$CARD LIST NEW TAPE
IDENT MAIN
      DIMENSION X(5),Y(20,3),Z(10,10)
      .........
      .........
      .........
      END
IDENT SUB1
      SUBROUTINE SUB1
      ..........
      ..........
      END
IDENT SUB2
      SUBROUTINE SUB2
      ..........
      ..........
      END
?END
```

The REPLACE Card may be used in conjunction with a LOAD Card to create an object code file if the replaced program part is syntax-error free.

<u>Example</u>

```
?COMPILE APPROX FORTAN SYNTAX
REPLACE SUB1
      A=SQRT(B*C)                                     00000200
      CALL SUB2                                       00000210
      IF (A-2) 3,4,6                                  00000211
?END
```

The above example causes the compiler to search a symbolic tape for the IDENT, SUB1, recompile the program part with the patch cards, and, if syntax-error free, enter the resulting ICS file on disk with the ID SUB1 (replacing its namesake if necessary).

<u>Example</u>

```
?COMPILE COMPUT FORTAN
LOAD  MAIN
REPLACE C
      .
      .
      .   patch cards for C
      .
?END
```

In the above example, after recompiling subroutine C and finding it syntax-error free, an object code file is created and executed.

SEGMENT CARD.

The SEGMENT Card may be used to make specified subroutines overlayable, and thus reduce the core requirement of a program.

The SEGMENT Card is coded as follows:

a. The key word SEGMENT is coded in columns 1-7.

b. The names of two or more subroutines which are to be segmented (made overlayable) are coded in columns 9-72, delimited by commas.

Subroutines which are referenced by a chain of CALLs may not overlay one another, i.e., if A CALLs B, B CALLs C, and C CALLs D, these subroutines may not overlay each other. (However, they may overlay subroutines not in the CALL chain.) If a subroutine name is coded in a SEGMENT Card and the compiler determines that it cannot be overlaid with any other subroutines, a segment dictionary entry (32 digits) is generated for it although the subroutine is always core resident.

No segmentation is performed by default.

The SEGMENT Card may be used and an object program may be reloaded without recompiling any of its program parts. (Refer to LOAD Card.)

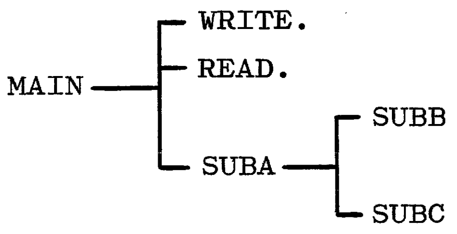
A subroutine name which appears in a SEGMENT Card must be referenced in the program. Otherwise, the syntax error message SEGMENT CARD: UNKNOWN PROGRAM IDENTIFIER is generated by the compiler.

Example

```
SEGMENT SUBA,SUBB,SUBC,READ,WRITE.
IDENT MAIN
      READ (5,10) A,B,C
      WRITE (6,20) A,B,C
   10 FORMAT (3F10.3)
   20 FORMAT (1H1,3F10.3)
      CALL SUBA(A)
      ...........
      ...........
      STOP
      END
      SUBROUTINE SUBA(P)
      ...........
      CALL SUBB
      ...........
      CALL SUBC
```

```
.........
END
SUBROUTINE SUBB
.........
.........
END
SUBROUTINE SUBC
............
............
END
```

The above sequence of CALLs may be represented by the following diagram.



In the above diagram, SUBC and SUBB overlay each other; WRITE., READ., and SUBA overlay one another.

SIZE CARD.
The SIZE Card is used to specify the amount of core to be allocated for REAL, INTEGER, and/or ALPHA variables when other than default sizes are desired.

By default a REAL variable occupies 12 digits (one digit for the sign of the exponent, a 2-digit exponent, one digit for the sign of the mantissa, and an 8-digit mantissa). The default size of an INTEGER variable is six digits (a 1-digit sign and five digits of precision). The default size of an ALPHA variable is six bytes (12 digits).

The SIZE Card is coded as follows:

a. The key word SIZE appears in columns 1-4.

b. One or more of the following specifications appear in columns 7-72, delimited by commas:

1) INTEGER = n.
2) REAL = m.
3) ALPHA = l.

The term n is the number of digits of precision; m is the mantissa size; and l is the number of characters. The maximum sizes which may be specified are: n=48, m=45, and l=24.

Example

```
SIZE  REAL=10
     *ALPHA=4
```

The cards in the above example specify that all real variables in the program are to have 10-digit mantissas, causing 14 digits to be allocated for each real variable. All alpha variables are allocated four bytes (eight digits) of memory.

Example

```
SIZE  REAL=12,  INTEGER=6,  ALPHA=4
```

The SIZE Card in the above example specifies that all real variables are to have 12-digit mantissas, all integer variables are to have six digits of precision (seven digits of core allocated for each), and all alpha variables are to be four bytes in length.

STACK CARD.
The STACK Card may be used at compile time to define the size of the stack* of a program. (By default the stack size is 1000 digits.)

The STACK Card is coded as follows:

a. The key word STACK is coded in columns 1-5.

* A stack is used by the NTR and EXT instructions. Refer to appendix F for a detailed explanation.

b. An integer constant which is the number of digits to be allocated for the stack is coded in columns 7-72.

There is no upper limit to stack size other than program size, which may not exceed 100,000 digits (50 KB).

The stack resides at the top of memory of a FORTRAN program. Because the MCP allocates core to a program in a MOD 1000 digit area, the stack actually occupies its specified size (1000 digits by default) plus any remaining core up to the limit register.

Example

A DC PROG SPO inquiry to the MCP reveals that PROG requires 100,000 digits. The MCP allocates 101,000 digits of core for its execution (MOD 1000). Since PROG is given a 1000-digit stack by default, the usable stack size is actually 1900 digits. By recompiling PROG with the control card, STACK 900, the core requirement of the program is reduced by 1000 digits. (DC PROG now yields a requirement of 100,000 digits.)

NOTE
Refer to appendix E for suggested usage of the STACK Card.

USE CARD.
The USE Card causes a specified subroutine to be referenced instead of another subroutine each time a CALL to the latter appears in the program. The card makes recompilation unnecessary.

The USE Card is coded as follows:

a. The key word USE is coded in columns 1-3.

b. One or more equations of the form ⟨identifier⟩ = ⟨identifier⟩ appear in columns 7-72, delimited by commas. The first ⟨identifier⟩ is the name of the subroutine to be used in place of the second ⟨identifier⟩.

In the following example a program contains CALLs to subroutine A, and the user wishes to use subroutine B in its place. He may compile B independently, then reload his object program (refer to LOAD Card). This assumes that the original program has already been compiled and its ICS files reside on disk.

<u>Example</u>

```
?COMPILE DUMMY FORTAN SYNTAX
?DATA CARDS
      SUBROUTINE B
      .
      .
      .
      END
?END
?COMPILE ORIGIL FORTAN LIBRARY
?DATA CARDS
LOAD MAIN
USE   B=A
?END
```

After execution of the above, the ORIGIL object program is on disk and, when executed, CALLs B where a CALL A has been coded in the original source statements.

NOTE

Intrinsic functions such as SIN, COS, and SQRT are considered to be subroutines.

<u>FORTRAN CONTROL CARDS FOR DEBUGGING AIDS</u>.

The following FORTRAN control cards produce program debugging aids. Their application is explained in detail in appendix E.

DEBUGN HEADINGS CARD.

The DEBUGN HEADINGS Card causes the following information for specified program parts to be printed after the symbolic listing:

a. Base relative beginning address of program part (LOW ADRS).

b. Base relative ending address of program part (HIGH ADRS).

c. Base relative address at which data begin (DATA BASE).

d. Base relative address at which executable code begins (CODE BASE).

e. Base relative addresses for the beginning of Junk (JUNK BASE), double precision temporaries (DBLT BASE), and single precision temporaries (SNGT BASE).

f. The length in digits of the following: JUNK, DBLT, SNGT, and DATA and their total length.

g. The amount of code in digits.

h. Subprogram names called by the program part with their base relative beginning addresses and segment numbers.

i. Names of referenced common blocks with their base relative beginning addresses.

The DEBUGN HEADINGS Card is coded as follows:

a. The key words DEBUGN HEADINGS are coded in columns 1-15.

b. The program part identifiers for which headings are desired are coded in columns 17-72, delimited by commas. Continuation cards may be used.

As many as 100 identifiers may be specified in the DEBUGN HEADINGS Card.

Unless it is preceded by an IDENT Card, the main program identifier is PROGAM.

Example

```
?COMPILE PREP FORTAN LIBRARY
?DATA CARDS
DEBUGN HEADINGS MAIN, READ., WRITE., SUBA
     *SUBB
IDENT MAIN
      .........
      CALL SUBA
      .........
      END
      SUBROUTINE SUBA
      .........
      CALL SUBB
      .........
      END
      SUBROUTINE SUBB
      .........
      END
?END
```

DEBUGN CARD.

The DEBUGN Card causes the compiler to produce a listing of generated object code and headings following the symbolic listing.

The DEBUGN Card is coded as follows:

a. The key word DEBUGN is coded in columns 1-6.

b. Optionally, program part identifiers for which code and headings are desired are coded in columns 8-72, delimited by commas.

Unless it is preceded by an IDENT Card, the main program identifier is PROGAM.

When identifiers are not specified in the DEBUGN Card, code and headings are produced for every program part, including referenced FORTRAN intrinsics.

The headings produced are identical to those generated by DEBUGN HEADINGS.

Object code for a program part is listed following a heading for that program part. Addresses are base relative.

<u>Examples</u>

```
DEBUGN SPOMSG
DEBUGN
```

MAP CARD.
The MAP Card produces a list of variable names and associated addresses following the symbolic listing of each program part.

The MAP Card is coded with the key word MAP in columns 1-3.

There are three groupings of variables which, if applicable, appear in MAP output:

a. Data relative identifiers. These variables are local to the program part. Their associated addresses are relative to the beginning of the data area for the program part. The base relative address of an identifier is the sum of the DATA BASE address generated by a DEBUGN or DEBUGN HEADINGS and the address generated in the MAP listing.

b. COMMON block identifiers. These variables are defined in COMMON statements in the program part. They are listed by COMMON block name, with /   / for blank COMMON, and their associated addresses are relative to the beginning of the

COMMON block in which they are defined. The base relative address of a variable is the sum of the address of the COMMON block which is generated by a DEBUGN or DEBUGN HEADINGS and the address generated by MAP for that variable.

c. Stack address of arguments. These variables are the formal parameters listed in the argument list of a SUBROUTINE statement. Their associated addresses are relative to the beginning of the stack entry which is created by the NTR executed for a CALL to the subroutine.

The MAP Card should be used in conjunction with a DEBUGN Card or a DEBUGN HEADINGS Card.

Example

```
?COMPILE TROUBL FORTAN LIBRARY
MAP
DEBUGN
        .
        .  FORTRAN source statements
        .
?END
```

FORTRAN DOLLAR SIGN CONTROL CARDS.

Dollar sign control cards are optional and contain specifications to the compiler governing symbolic input and output.

The dollar sign control card is coded as follows:

a. The $ symbol is coded in column 1.

b. Options from the table below are coded in columns 2-72 in free field format with blanks as delimiters.

The following dollar sign options can be used.

CONTROL CARDS

Option	Effect	Default
LIST	Lists source program	Lists only lines with syntax errors or flags
CARD	Source code from cards	Card
TAPE	Source code from tape	Card
NEW TAPE	Creates source language tape (ID is TAPES)	No tape output
DEBUGN	Lists pseudo code interspersed with symbolic code	No listing of pseudo code
CHECK	Checks order of sequence numbers	No sequence check
SEQ nnnnnnnn + nnnnnn	Resequences beginning with 8-digit number using 6-digit increment	No sequence. If starting sequence not specified, 1000 assumed
SPACE nn	Prints nn lines per page on symbolic listing	Standard number of lines per page
JAPN	Symbolic listing begins in column 37.	Symbolic listing begins in column 1.

By default the compiler assumes $CARD LIST.

Dollar sign control cards may be freely interspersed within a FORTRAN source deck. When the compiler encounters a dollar sign card, options specified on a previous one, and not repeated, are negated.

CONTROL CARDS

NOTE

A "9's Card" must be included in a symbolic deck which specifies TAPE in a dollar sign card. This card immediately precedes the ?END Card and has a sequence field of eight 9's. With the exception of the sequence field, the 9's Card may be blank.

Example

```
?COMPILE MYPRO FORTAN LIBRARY DATA CARDS
$TAPE NEW TAPE SEQ 00000100 + 000010 LIST
      .
      .   Patch cards
      .
                                                    99999999
?END
```

In the above example the compiler applies the patch cards included in the symbolic deck to a symbolic tape, creating a new symbolic tape which is resequenced. A symbolic listing of the entire program is printed.

APPENDIX D
READ/WRITE INTRINSICS ADDITIONAL INFORMATION

GENERAL.

This appendix contains information to be used in file planning and debugging. The different possibilities for layouts of data when written to various devices are described, followed by a brief summary of run-time error messages displayed by the READ. and WRITE. intrinsics, and programming suggestions for efficient I/O execution.

FORMATTED INPUT AND OUTPUT.

A format is used to read or write data which are in byte (EBCDIC/display) form, regardless of device type. The length of the item, i.e., the number of bytes, is determined by the field width in the format specifier. An exception occurs when a WRITE to a line printer references a FORMAT which does not contain an explicit carriage control specification, such as 1X or 1H1. In this case, the first format specifier is used for carriage control, and the length of the first item is decreased by one (effectively dropping the first character).

Example

```
      I=310
      R=25.264
      WRITE (9,10) I,R
   10 FORMAT (2HI=,I5,3X,2HR=,F7.3)
      STOP
      END
```

The above program produces the following 50-byte record on disk:

```
C97E4040F3F1F0404040D97E40F2F54BF2F6F440...40
```

UNFORMATTED INPUT AND OUTPUT.

Data may be read from or written to disk or tape by an I/O statement which does not reference a FORMAT statement. The format of this data as it appears in the file differs slightly depending on whether:

a. The file is tape or disk.

b. The file is defined with a PACKED attribute if disk.

READ/WRITE INTRINSICS ADDITIONAL INFORMATION

c. The file is 7-channel or 9-channel if tape. The "formats" for unformatted records are described below. References are to writing a record, with the understanding that an unformatted READ performs an identical operation in reverse.

DISK.

By default (no PACKED specification in a FILE Attributes Card), an unformatted WRITE to a disk file produces a record comprised of fixed-length data items. The length is the size of a storage unit (SU), which is, in turn, determined by the "sizes" of the various data types. SU is the largest of the following: N+1, M+4, and 2xL, where N is the number of digits of precision for an integer, M is the mantissa size for a real, and L is the number of characters for an alpha variable. (Refer to SIZE Card in appendix C.)

Unformatted data on disk is in digit form (UN/COMP), left-justified in a SU size field and blank filled to the right. A complex variable occupies two fields, with the real part in the first and the imaginary part in the second. A double precision variable also occupies two fields, left-justified with trailing blanks.

The following example assumes default sizes (giving SU=12 digits or six bytes) and default attributes for disk.

<u>Example</u>

```
      LOGICAL L
      ALPHA A
      COMPLEX CMP
      DOUBLE PRECISION D
      DATA A/4HABCD/
      I=56789
      R=1234.56
      L=.TRUE.
      D=-12345.67890D0
      CMP=(111.222,333.444)
      WRITE(9) A,I,R,L,D,CMP
      STOP
      END
```

READ/WRITE INTRINSICS ADDITIONAL INFORMATION

The variables A,I,R,L,D, and CMP are stored in core as follows:

```
A=C1C2C3C44040
I=C56789
R=C04C12345600
L=1
D=C05D1234567890000000
CMP=C03C111222000C03C33344400
```

The disk record produced by the above program is:

```
◄── A ───►◄─── I ────►◄─── R ────►◄─── L ───►
C1C2C3C44040C56789404040C04C123456001040404040
◄────────── D ──────────►◄──────── CMP ────────►Filler
C05D1234567890000000404040C03C111222000C03C333444004040
```

An unformatted WRITE to a disk file defined with a PACKED specification produces a record comprised of data items whose lengths are determined individually by the "size" of the data type each represents. (Refer to SIZE Card in appendix C.) This packed, unformatted data is on disk in digit form, as it appears in core.

<u>Example</u>

Refer to the program in the example above. With the FILE Card coded as FILE 9=FILE9, UNIT=DISK, PACKED, the following 50-byte record is written:

```
◄── A ────►◄─ I ─►◄─── R ───►L ◄────── D ──────►
C1C2C3C44040C56789C04C1234560010C05D1234567890000000
◄──────── CMP ────────►◄Filler►
C03C111222000C03C333444004040...40
```

MAGNETIC TAPE.

An unformatted WRITE to a magnetic tape produces a variable number of variable length records, the actual number being dependent upon the number of variables to be written, storage unit (SU) size, and the maximum physical record length. When using unformatted I/O statements, a magnetic tape <u>must</u> be variable length, the default attribute.

READ/WRITE INTRINSICS ADDITIONAL INFORMATION

An unformatted WRITE produces one or more records with the following format:

4UA	1UA	
R-L	R C F	DATA

Record length (R-L) is a 4-character field containing a count of the number of characters in the record, excluding itself. The Record Continuation Flag (RCF) is a 1-character field used by the compiler to group records associated with each I/O statement in the FORTRAN program.

RCF is one of the following:

Character	Meaning
0	One and only one record
1	First record
2	Intermediate record
3	Last record

The compiler writes as many records in the above format as needed to exhaust an I/O list. The number of records varies depending on whether the tape is 7-channel or 9-channel because of a difference in data representation.

Data items on 9-channel tapes are in digit (UN/COMP) format and are fixed in length. The length is the size of a storage unit (SU), which is, in turn, determined by the "sizes" of the various data types. SU is the largest of the following: N+1, M+4, 2xL, where N is the number of digits of precision for an integer, M is the mantissa size for a REAL variable, and L is the number of characters for an ALPHA variable. (Refer to SIZE Card in appendix C.) Data are left-justified in the field with trailing blanks. A complex variable occupies two fields, with the real part in the first and the imaginary part in the second. A double precision variable is left-justified in two SU size fields with trailing blanks.

Data items on 7-channel tape are in byte (UA/DISPLAY) format and are fixed in length. The length is twice the size of a storage unit (SU), the determination of which is described in the preceding paragraph. Data are left-justified in the field with trailing blanks. A sign and the numeric digit immediately following it are represented in one byte, with the sign as the zone digit. All other numeric digits are represented with the numeric subset zone digit, F. Double precision and complex variables each occupy two, fixed-length fields, or a field four times SU size. The real part of a complex number occupies the first field; and the imaginary part, the second field. An ALPHA variable is not expanded; it is written as it appears in core followed by trailing blanks if necessary.

<u>Examples</u>

```
FILE  3=MAGTAP. UNIT=TAPE,RECORD=26
      DIMENSION I(3),R(4)
      ALPHA ALP(3)
      COMPLEX CMP
      DOUBLE PRECISION BIG
      DATA I,R,ALP,CMP,BIG/2*5,76543,6.5,10.194,
     1-7.0,0.,6HABCDEF,3HMY ,4HNAME,(9.9,.2),
     *-.00099654387091/
      WRITE(3) I,R,ALP,CMP,BIG
      STOP
      END
```

Assuming default data sizes, the above variables are stored in core as follows:

```
I(1)=C00005
I(2)=C00005
I(3)=C76543
```

APPENDIX D (cont)

READ/WRITE INTRINSICS ADDITIONAL INFORMATION

```
R(1)=C01C65000000
R(2)=C02C10194000
R(3)=C01D70000000
R(4)=D99C00000000
ALP(1)=C1C2C3C4C5C6
ALP(2)=D4E840404040
ALP(3)=D5C1D4C54040
CMP=C01C99000000C00C20000000
BIG=D03D99654387091000000
```

The records produced on a 9-channel tape by the above program are:

```
                R
                C
      ◄-R-L-►F ◄--I(1)----►◄---I(2)----►◄---I(3)----►◄---R(1)--►
#1    FOFOF2F6F1C00005404040C00005404040C76543404040C01C6500000040
                R
                C
      ◄-R-L-►F ◄-- R(2)----►◄--- R(3)----►◄--- R(4)----►◄-- ALP(1)--►
#2    FOFOF2F6F2C02C10194000C01D70000000D99C00000000C1C2C3C4C5C640
                R
                C
      ◄-R-L -►F ◄--ALP(2)--►◄---ALP(3)--►◄---------- CMP ----------►
#3    FOFOF2F6F2D4E840404040D5C1D4C54040C01C99000000C00C2000000040
                R
                C
      ◄-R-L-►F ◄--------BIG ------------►
#4    FOFOF1F4F3D03D996543870910000040404 0
```

NOTE

An extra blank is added to each
record to make the physical
record an even number of bytes.

The same WRITE statement produces the following records on 7-channel tape:

```
                R
                C
      ◄-R-L-►F ◄-------------I(1)-------------►◄-----------I(2)---------►
#1    FOFOF2F6F1COFOFOFOF540404040404040COFOFOFOF540404040404040
```

APPENDIX D (cont)

READ/WRITE INTRINSICS ADDITIONAL INFORMATION

```
                 R
                 C
        ←R-L→ F ←────── I(3) ──────→←────── R(1) ──────→
#2      F0F0F2F6F2C7F6F5F4F340404040404040C0F1C6F5F0F0F0F0F0F0404040
                 R
                 C
        ←R-L→ F ←────── R(2) ──────→←────── R(3) ──────→
#3      F0F0F2F6F2C0F201F0F1F9F4F0F0F04040C0F1D7F0F0F0F0F0F0F0404040
                 R
                 C
        ←R-L→ F ←────── R(4) ──────→←────── ALP(1) ─────→
#4      F0F0F2F6F2D9F9C0F0F0F0F0F0F04040C1C2C3C4C5C640404040404040
                 R
                 C
        ←R-L→ F ←───── ALP(2) ─────→←────── ALP(3) ─────→
#5      F0F0F2F6F2D4E84040404040404040404040D5C1D4C5404040404040404040
                 R
                 C
        ←R-L→ F ←─────────────── CMP ───────────────→
#6      F0F0F2F6F2C0F1C9F9F0F0F0F0F0F04040C0F0C2F0F0F0F0F0F0F0404040
                 R
                 C
        ←R-L→ F ←─────────────── BIG ───────────────→
#7      F0F0F2F6F3D0F3D9F9F6F5F4F3F8F7F0F9F1F0F0F0F0F040404040404040
```

RUN TIME ERROR MESSAGES.

Each time an I/O statement is encountered during program execution, the READ and WRITE intrinsics verify information which is passed to them. File number, format, if referenced, and record length are checked. If invalid information is found, one of the error messages below is displayed on the SPO followed by an ADDR ERROR, DS, or DP message. The error message is preceded by R-- for a READ error or W-- for a WRITE error.

Message	Meaning
IFN	Invalid file number
GIC	Invalid format character
IFC	Invalid format character
RTL	Record too long

READ/WRITE INTRINSICS ADDITIONAL INFORMATION

<u>Example</u>

```
**⟨program-name⟩=⟨mix-index⟩ W--RTL
--ADDR ERROR ⟨program-name⟩=⟨mix-index⟩ nnnnn nnnn
** ⟨program-name⟩=⟨mix-index⟩ DS OR DP
```

Conditions which are determined during program execution may cause invalid information, impossible to syntax-check at compilation time, to be passed to the READ/WRITE intrinsics. Frequent causes of run-time errors are described below.

INVALID FILE NUMBER.
A unit designator (file number) is an integer constant or variable with permissible values of 0-19. An integer variable may have a value which is negative or greater than 19 at the time it is referenced as a unit designator in an I/O statement. When this occurs, an IFN message is displayed.

INVALID FORMAT CHARACTER.
A format reference in an I/O statement may be an array name, with the assumption that when the statement is executed the array contains a valid format. If the format is not valid, an IFN message is displayed.

The FORTRAN Compiler "deblanks" a FORMAT statement before making it part of a FORTRAN program. For example, 20 FORMAT (3A6, I2) appears in core as (3A6,I2). Thus, an array used as a format may not contain blanks before the final right parenthesis, and only valid format characters may appear between the initial and final parentheses.

The following example illustrates a common error in using an array for a format.

Example

Data Card

```
              1111111111
1234567890123456789 (card columns)
(I5,F10.4,2E20.4)
```

Program

```
   ALPHA FMT(6)
   READ(5,10)FMT
10 FORMAT(6A3)
   ...
   ...
   READ(9,FMT) J,R,RMAX,RMIN
   ...
   STOP
   END
```

The above program causes an R--IFN message to be displayed on the SPO because blanks appear within the format stored in the array FMT. Assuming default sizes, each element of FMT is six characters in length; and when filled using an A3 format specifier, the array in core is (lower case b indicates blank):

```
I5,bbbF10bbb.4,bbb2E2bbb0.4bbb)bbbbb
```

The program can be corrected by including a SIZE ALPHA=3 Control Card or by changing statement 10 to FORMAT(3A6).

Another error which is often reflected by an IFC message is that of subscripting an array beyond its dimensioned size. This error could cause a format in core to be overwritten with data. Although a format in an array is more susceptible to being overwritten, compiler-generated formats immediately follow array storage in core and are also subject to overwriting.

READ/WRITE INTRINSICS ADDITIONAL INFORMATION

In addition to programmatically subscripting beyond the dimensioned size of an array, overwriting of data can occur when an A format specifier is used to read data into a variable which is not typed ALPHA. If the overwritten data is a format, an IFC message is displayed.

RECORD TOO LONG.
A logical record size is defined for a file either through a FILE Card or the default associations of the compiler. If a READ or a WRITE statement specifies a logical record of more data items than can be contained in the defined record length, an RTL message is displayed on the SPO. This applies to disk files, fixed-length tape files (i.e., formatted I/O to tape), and card files. The READ/WRITE intrinsics handle record overflow for the line printer and variable length tapes (i.e., unformatted I/O to tape).

The series of paragraphs relating to data representation in this appendix should be used to determine the cause of an RTL message and as a guide to specifying optimum record lengths for efficient core and disk utilization.

PROGRAMMING FOR EFFICIENT I/O EXECUTION.

For each I/O statement the FORTRAN Compiler generates a series of NTR instructions to the READ. or WRITE. intrinsic. Information is passed to the intrinsic through the STACK entry created by the execution of each NTR. Included in these instructions is an NTR for each variable name in an I/O list. An unsubscripted array name in an I/O list causes the array to be read or written in the order in which it is stored in core (i.e., column order). When an implied DO loop is coded, one NTR is generated; but this NTR is executed once for every value the dummy subscripts assume.

Examples

```
    DIMENSION ARAY(10.10)
    WRITE (6,20) ((ARAY(I,J),I=1,10),J=1,10),BRAY
```

READ/WRITE INTRINSICS ADDITIONAL INFORMATION

Two NTR's are generated to pass the addresses of ARAY and BRAY. The NTR generated to pass ARAY is executed 100 times.

```
WRITE (6,30) ALP,ARG,ZMAX,PSQ,RIN,ZAP,TOP
```

Seven NTR's are generated to pass the addresses of the variables in the I/O list.

```
DIMENSION ARAY(10,10)
WRITE(6,40) ARAY
```

One NTR is generated to pass the address of ARAY; it is executed once, although 100 values are printed.

In view of the above considerations, the following suggestions are made:

a. Do not use implied DO loops in I/O statements where they are not necessary.

b. When there are many variables to be read or written, EQUIVALENCE a dummy array to the variables and code the array name in the I/O statement.

<u>Example</u>

```
 DIMENSION DUM(10)
 EQUIVALENCE (DUM(1),A),(DUM(2),B),(DUM(3),C),
*(DUM(4),D),(DUM(5),E),(DUM(6),F),(DUM(7),G),
*(DUM(8),H),(DUM(9),I),(DUM(10),J)
 WRITE(9,10) DUM
 ..........
 END
```

Generated code for the above statements is more efficient than the following:

```
WRITE(9,10) A,B,C,D,E,F,G,H,I,J
```

APPENDIX E
DEBUGGING AIDS

This appendix is a guide to the use of FORTRAN-supplied debugging aids and contains information about the more common causes of address errors.

A FORTRAN PROGRAM IN MEMORY.

The diagram on the next page illustrates the core layout of a FORTRAN object program.

In the explanation given below for the FORTRAN program shown in the illustration, some lengths are determined using the following notation:

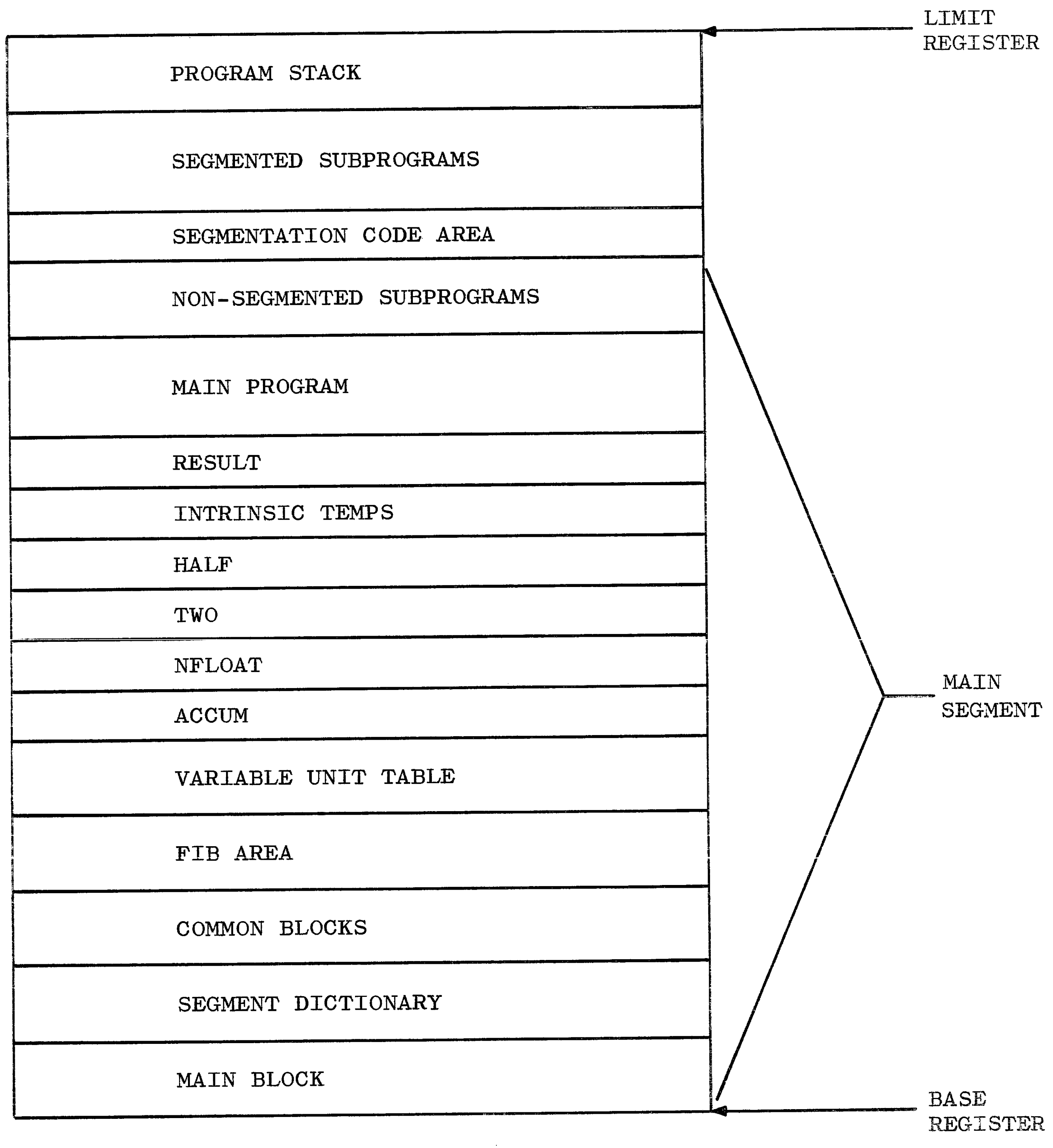
Abbreviation	Meaning
M	Mantissa length for REALs
N	Precise digits for INTEGERs
L	ALPHA length in characters
SU	Storage unit = MAX(M+4,N+1,2xL)

Starting at the base register (base relative address 00000), a FORTRAN program consists of the following:

a. MAIN BLOCK - data size information as follows:

Base Relative Address	Length in Digits	Meaning
00000	2	2xM+3
00002	2	N
00004	2	M
00006	2	M+3
00046	2	2xM+4
00048	2	SU

DEBUGGING AIDS



Base Relative Address	Length in Digits	Meaning
00050	2	N+1
00052	2	M+4
00054	2	2xM
00056	2	2xM+3
00058	2	2xL
00060	2	2xSU-1
00062	2	2xSU

b. SEGMENT DICTIONARY - data used by the overlay code of the MCP (refer to MCP Reference Manual for details). The length of the dictionary is 32(NOSEGS+1), where NOSEGS is the number of segments requested in a SEGMENT Control Card plus one.

c. COMMON BLOCKS - core area allocated for variables declared in COMMON statements.

d. FIB AREA - core allocated for File Information Blocks and, as defined, buffers and work areas.

e. VARIABLE UNIT TABLE - FIB addresses for units 0-19. The table consists of 20, 6-digit entries.

f. ACCUM - temporary storage used to avoid C-field overlap in floating-point add and subtract instructions (length is 2xM+4).

g. NFLOAT - used for integer to real conversion. The length is MAX(2xM+4,N+4).

h. TWO - used for real to integer conversion (length is N+5).

i. HALF - used for rounding REALS (length is M+5).

j. INTRINSIC TEMPS - temporary storage used by in-line intrinsic functions (length is 4xSU).

k. RESULT - used to pass the result of function subprograms (length is 2xSU).

l. MAIN PROGRAM - the main program, as well as subroutines and function subprograms, have the following format:

CODE
DATA
SINGLE PRECISION TEMPS
DOUBLE PRECISION TEMPS
JUNK CELLS

JUNK CELLS are optional and contain indirect addressing code used for subscripted variables. DOUBLE and SINGLE PRECISION TEMPS (temporaries) are optional, their presence being dictated by the generated code. Local DATA and CODE are self-explanatory, being generated from the symbolic input for the program part.

m. NON-SEGMENTED SUBPROGRAMS - program parts residing sequentially in core and having the same format as the main program (described in paragraph l).

n. SEGMENTATION CODE AREA - 26-digit entries, one created for each declared segment. An entry consists of two instructions used to initiate an overlay branch communicate.

o. SEGMENTED SUBPROGRAMS - overlay area, the length of which is determined by the longest group of segmented subprograms referenced in a chain of calls.

p. PROGRAM STACK - area of core utilized by the NTR instruction. By default, this area is 1000 digits; optionally, a STACK Control Card may be used to specify its length (refer to appendix C).

<u>FORTRAN SUPPLIED DEBUGGING AIDS</u>.

Control cards may be introduced at compilation to provide information for debugging purposes. These are described in appendix C and are:

a. DEBUGN HEADINGS.
b. DEBUGN.
c. MAP.
d. $DEBUGN.

In addition to the above control cards, the intrinsic FMDUMP, when CALLed, provides a memory analysis of the object program and, optionally, the memory dump (refer to appendix B).

A DEBUGN Control Card produces a listing of machine instructions, grouped by program part following the symbolic listing. Addresses in the DEBUGN listing are base relative. Each group of instructions is preceded by a DEBUGN HEADING for the program part, and a page of general information for the program is provided. The generated machine instructions in a DEBUGN listing may be used as an aid in debugging by correlating them to the symbolic code. With this, a dump can be used effectively, and a normal-state trace becomes an invaluable tool for debugging.

In addition to machine instructions, the locations of identifiers and temporary storage areas must be known when debugging from a dump or trace. The output generated by a MAP Control Card in conjunction with a DEBUGN HEADING locates specific identifiers in core by their base relative addresses. An address associated with a local variable is "data relative," i.e., relative to the data base of the program part in which it is defined. An address associated with a variable in COMMON is "COMMON relative," i.e., relative to the beginning base relative address of the COMMON block in which it is defined.

An argument passed as a parameter to a program part is referenced through the stack (refer to appendix F). Thus, in a DEBUGN HEADING, the "stack address of an argument" is relative to the beginning of the stack entry created by an NTR to the program part.

A DEBUGN HEADING also contains the base relative address of the single precision temporaries, double precision temporaries, and junk cells for the program part. In some cases, knowledge of these addresses is necessary to understand the generated machine code.

In general, relative addresses are associated with identifiers in MAP output, and the addresses to which these identifiers are relative are provided in a DEBUGN HEADING. An example is shown on the next page to clarify the correlation of the two debugging aids.

The "pseudo code" generated by a $DEBUGN Control Card is interspersed with the symbolic code and may be used to facilitate the correlation of machine instructions in a DEBUGN listing to symbolic instructions.

The dump analysis produced by the subroutine FMDUMP provides a snapshot of core during program execution, a valuable tool when the status of a file, subprogram, stack entry, etc., at a particular instance is questioned.

SYMBOLIC LISTING

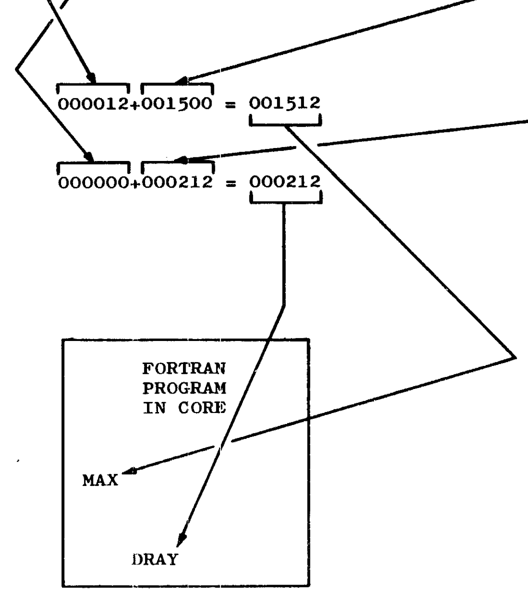
```
      SUBROUTINE NON(X,J)
      COMMON RR,IJ,SM(5)/FIRST/DRAY,GRAY(6)
      T=X**2
      MAX=J-1
      RETURN
      END

04/16/71     6:52 AM   ASR#4.3    71011    COMPILER
  0 MIN  3 SEC FOR COMPILATION PASS
   10 CARDS AT 196 CARDS PER MINUTE
    36 DIGITS DATA.    98 DIGITS CODE.   168 DIGITS COMMON.
```

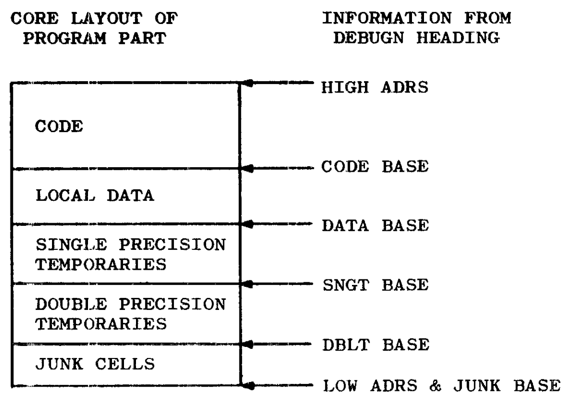
MAP OUTPUT

```
STACK ADDRESS OF ARGUMENTS
    X          000030
    J          000038
DATA RELATIVE IDENTIFIERS
    NON        000000
    T          000000
    MAX        000012
/      / COMMON BLOCK 01 ----- 00084 DIGITS LONG
    RR         000000
    IJ         000012
    SM         000024
/FIRST / COMMON BLOCK 02 ----- 00084 DIGITS LONG
    DRAY       000000
    GRAY       000012
```

DEBUGN HEADING

```
SYMBOLIC LISTING FOR NON    :

LOW ADRS  HIGH ADRS     DATA BASE CODE BASE
 001486    001622        001500    001524

JUNK BASE DBLT BASE SNGT BASE
 001486    001486    001486

JUNK LGTH DBLT LGTH SNGT LGTH DATA LGTH  TOTAL   CODE LGTH
 000000    000000    000014    000024    000038  000098

CALLED SUBPROGRAMS :
NAME     ADDRESS   SEG. NO.
EXPON.   001622     001

COMMON BLOCKS REFERENCED :
NAME    ADDRESS
         000128
FIRST    000212
```




CORE LAYOUT OF PROGRAM PART
INFORMATION FROM DEBUGN HEADING
CODE
LOCAL DATA
SINGLE PRECISION TEMPORARIES
DOUBLE PRECISION TEMPORARIES
JUNK CELLS
HIGH ADRS
CODE BASE
DATA BASE
SNGT BASE
DBLT BASE
LOW ADRS & JUNK BASE

DEBUGGING AIDS

ADDRESS ERRORS.

COMPILE-TIME.
An address error in the FORTRAN Compiler generally indicates a software malfunction, one over which the programmer has no control. However, there is one possible cause of an address error which should be investigated: stack overflow. An involved expression requires a series of NTR instructions to be executed in the compiler; and if the stack size is exceeded before the compiler completes the expression, an address error occurs. A dump of the compiler after the address error occurs reveals this problem. As in a FORTRAN object program, the stack of the compiler is located at the top of the program. If the stack is "full" and base relative address 00040 points past or close to the limit register, a stack overflow is the probable cause of the failure. Successful compilation can then be achieved by breaking the involved expression(s) into subexpressions or by giving the compiler more core in which to execute, for example:

```
?COMPILE MYPRO FORTAN CORE 60000
?DATA CARDS
.
.
.
```

EXECUTION-TIME.
An address error may occur during the execution of a FORTRAN program as a result of logic errors that are impossible to syntax-check during compilation. If the cause of an address error is not apparent, the following steps should be folllowed to isolate the problem:

a. Recompile with DEBUGN and MAP Control Cards. Strategic CALLs to TRACE and/or FMDUMP may be included in the source deck.

b. Execute and perform a DP in response to the DS or DP SPO message. The instruction at which an address error occurs is the instruction at or immediately preceding the PAR address in the dump.

c. Use of additional debugging aids may become necessary at this point.

The more common causes of execution-time address errors are explained below.

PROGRAM SIZE. A FORTRAN program may have a maximum size of 100,000 digits (50 KB), excluding disk file headers which reside beyond the limit register. This core restriction is a result of the 6-digit address inherent in the hardware design of the systems. The most-significant digit of an address is used for index register and address controller specifications, with the five remaining digits specifying the actual base relative address. Thus, the largest address which can be directly specified is 99999. With local data physically residing in a program part and instructions which are self-modifying (i.e., addressed as data), the 50 KB limit, when exceeded, usually results in execution errors. Usually the result is an address error.

Frequently, the compiler can determine that the size of a program is greater than 50 KB and generates a TREEANALYSIS syntax error. Because the compiler is unable to detect this error in all cases, when the size of a program is questionable, a DC SPO inquiry should be made before attempting its execution. However, a dump also indicates the core allocation of a program.

The following techniques can be used to effectively reduce core requirements:

DEBUGGING AIDS

a. SEGMENT (make overlayable) existing subroutines, including FORTRAN intrinsics when feasible. (Refer to SEGMENT Control Card in appendix C.)

b. Fracture the main program and/or subroutines to create more subroutines which can be segmented.

c. Decrease the sizes of data types, remembering that a variable in a COMMON or EQUIVALENCE statement occupies a storage unit in core. (Refer to SIZE Control Card in appendix C.)

d. EQUIVALENCE large arrays. This is effective when the elements of the arrays are a storage unit in length, i.e., with default sizes, REAL or ALPHA.

e. Reduce the number of FIB's and the amount of core used for buffers and work areas by using the CHANGE intrinsic. (Refer to appendix B.)

f. Decrease the stack size of the program with a STACK Control Card. If a program is executable, the amount of unused stack can be determined from a dump taken after a reasonable amount of processing; a continuous field of numeric zeros is the unused portion.

g. Use FILE Control Cards to specify attributes which reduce the amount of core assigned to a file, i.e., shorten record lengths, reduce blocking factors, specify fewer buffers, and so forth. (Refer to the table of default file attributes in appendix C.)

h. Check array structures and specify minimum DIMENSIONs according to usage.

i. Place in COMMON, variables that are repeatedly passed as arguments to subroutines.

j. Divide a very large FORTRAN program into two programs, and utilize the SEND and ACCEPT intrinsics for interprogram communication.

CONTROL STATEMENTS. Address errors may be caused by the improper use of control statements. Common errors are described in the following paragraphs.

Use of an ASSIGN statement requires an integer size of at least 5. If integer size is less than 5, the result of executing an Assigned GO TO statement is unpredictable with an invalid instruction or address error likely.

At the time of execution of an Assigned GO TO statement, the control variable must have been ASSIGNed an integer value equal to one of the statement numbers in the GO TO list. If the variable does not correspond to a label, an address error may occur. The control variable should not be referenced between its appearance in an ASSIGN statement and an Assigned GO TO statement, nor should it be EQUIVALENCEd to a variable which is referenced in the interim.

Similarly, the control variable in a Computed GO TO statement must be an integer value greater than zero but no greater than the number of statements in the GO TO list when the statement is executed. A violation of this rule causes an address error.

READ/WRITE INTRINSICS. In appendix D, run-time error messages displayed by the READ/WRITE intrinsics and their meanings are described. These error messages are followed by a programed address error to terminate execution. The following procedure may be employed to isolate the failing READ or WRITE statement:

a. Recompile with a DEBUGN Control Card and execute.

b. Perform a DP in response to the DS or DP message.

c. In the dump, go to the address contained in base relative location 00040.

d. Carefully examine the stack entries preceding this memory location, working backward through the stack until the most recent NTR to the intrinsic is found. (AN R-- message prefix indicates the READ intrinsic; a W-- message prefix indicates the WRITE intrinsic.) This stack entry contains a return address which is not within the intrinsic, a fact which is ascertained from the DEBUGN output.

e. The return address in this stack entry is the address of the first executable instruction following an NTR to the intrinsic for the failing I/O statement.

f. Find the instruction in the DEBUGN listing using its address and associate the preceding NTR instruction with an I/O statement in the symbolic listing.

In addition to the programming errors reflected by the meanings of the run-time error messages (e.g., W--RTL indicates an attempt to write a record which is longer than the defined record length of the file), an "IFC" message may be the result of an error which is not related to the format of the I/O statement in question. Because formats immediately follow arrays in core, a violation of array bounds may cause a format to be overwritten with data. This can occur when a subscript exceeds the dimension of an array or when data are read with an A format specifier into an array or variable which is not typed ALPHA. Thus, a format which is syntactically perfect during compilation may contain invalid format characters when referenced by an I/O intrinsic during execution.

MISCELLANEOUS. Address errors may be caused by a stack overflow or an instruction time-out. A stack overflow is detected from a dump of the program taken after the address error. If the stack is "full" and base relative address 00040 points beyond or near the limit register, a stack overflow is suspect. The program should execute properly after recompilation with a STACK Control Card that sufficiently increases the stack size.

An instruction time-out can occur when an operand contains undigits that are meaningless to the instruction, i.e., integer arithmetic performed on a floating point or alpha value and floating point arithmetic performed on an alpha value. When variables of different data types are EQUIVALENCEd in FORTRAN, it is possible to reference a variable in a statement, forgetting that it previously has been referenced by an EQUIVALENCEd identifier of a different data type and assigned a value.

APPENDIX F
FORBLR

GENERAL.

FORBLR is a program which assembles subroutines written in assembler language to interface with a FORTRAN program. Symbolic input to FORBLR is identical to assembler symbolic code except where additions, differences, or limitations are noted in this appendix. The Assemblers Reference Manual should be used in conjunction with this appendix, as the repertoire of available instructions is not repeated.

The FORBLR Assembler is used as follows:

```
?EXECUTE FORBLR
?DATA (B) CARDS
.
.    Assembler
.    symbolic
.    code
.
?END
```

An IDNT pseudo must be included in the assembler symbolic code.

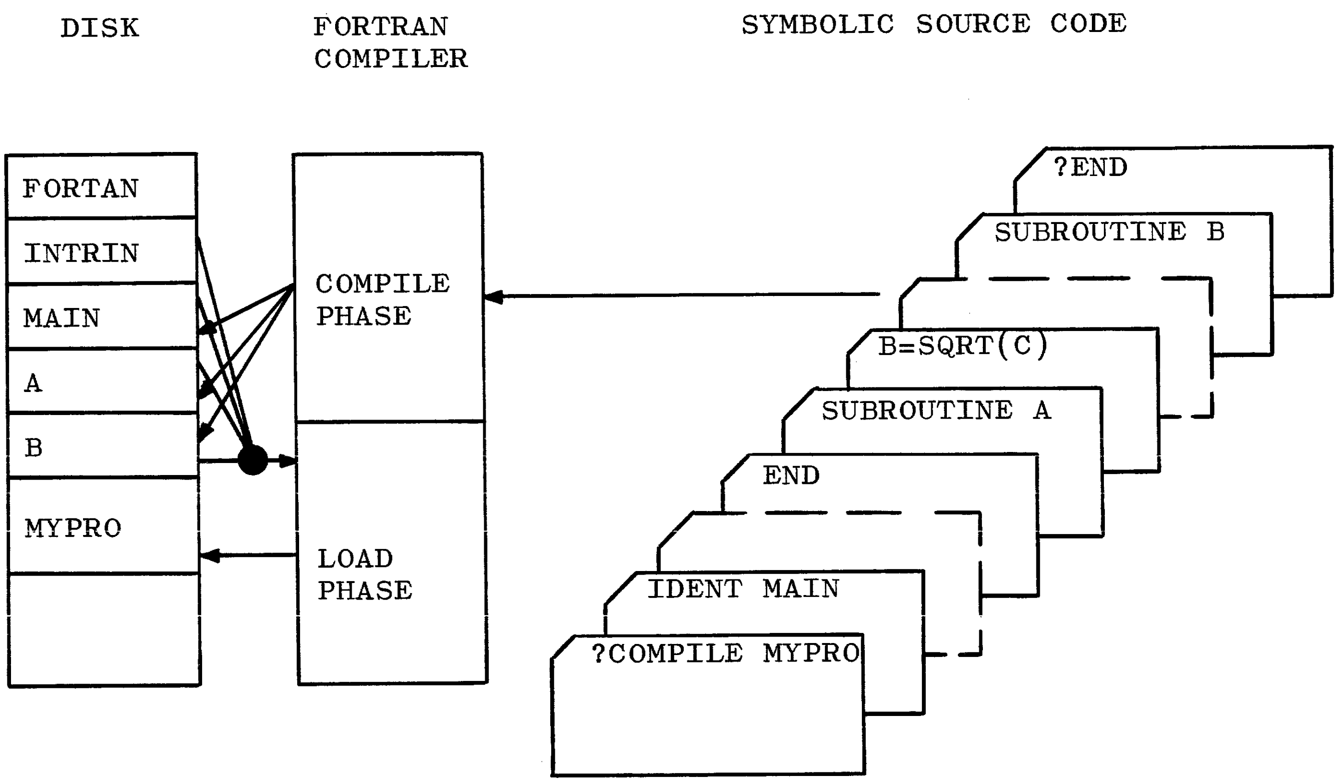
NOTE

Do not COMPILE with FORBLR. If this is done, the MCP displays a message on the SPO indicating that the program contains syntax errors, regardless of whether it does or not.

The remainder of this appendix contains information which should be known to correctly interface a FORBLR routine to a calling program part.

THE FORTRAN COMPILER AND ICS FILES.

The FORTRAN Compiler performs two distinct functions, which, through the use of control cards, may be mutually exclusive. It compiles symbolic code (compile phase) and creates object code files (load phase). The two phases of compiler execution are illustrated by the following diagram and are described in detail below.



COMPILE PHASE.

The FORTRAN Compiler generates a pseudo code file for each program part introduced for compilation in a symbolic code file. A pseudo code file is a permanent disk file and is referred to as an Independently Compiled Subroutine file (ICS file). The file ID of an ICS file is one of the following:

a. For a main program - the identifier coded in an IDENT Card or, by default, PROGAM.

b. For a SUBROUTINE, FUNCTION, or FORBLR subprogram - the subprogram name. (The identifier in an IDENT Card for a subprogram must be the subprogram name.)

The disk format of an ICS file is shown on the following page.

An ICS file has a fixed format. The first five disk segments each contain specific information and are continued, if necessary, on subsequent segments in the file. The last eight digits in a segment are a link to the segment on which its information is continued. An ICS file contains the following data:

a. IDENT segment.

1) Identifiers of referenced program parts (6UA each).

2) Names of defined COMMON blocks (6UA each) with the number of storage units of data associated with each (4UN).

3) If files are defined, their internal file names, multi-file ID's, and external file ID's (18UA per file).

b. INITIALIZED DATA segment.

1) Data local to the program part and initialized by a DATA statement in FORTRAN or a CNST declaration in FORBLR. Data are in the following format:

```
| CODE | ADDRESS | LENGTH | DATA ————————→
|______|_________|________|_______________________
◄2UN►◄—— 6UN ——►◄——3UN—►◄ #digits specified by LENGTH ——►
```

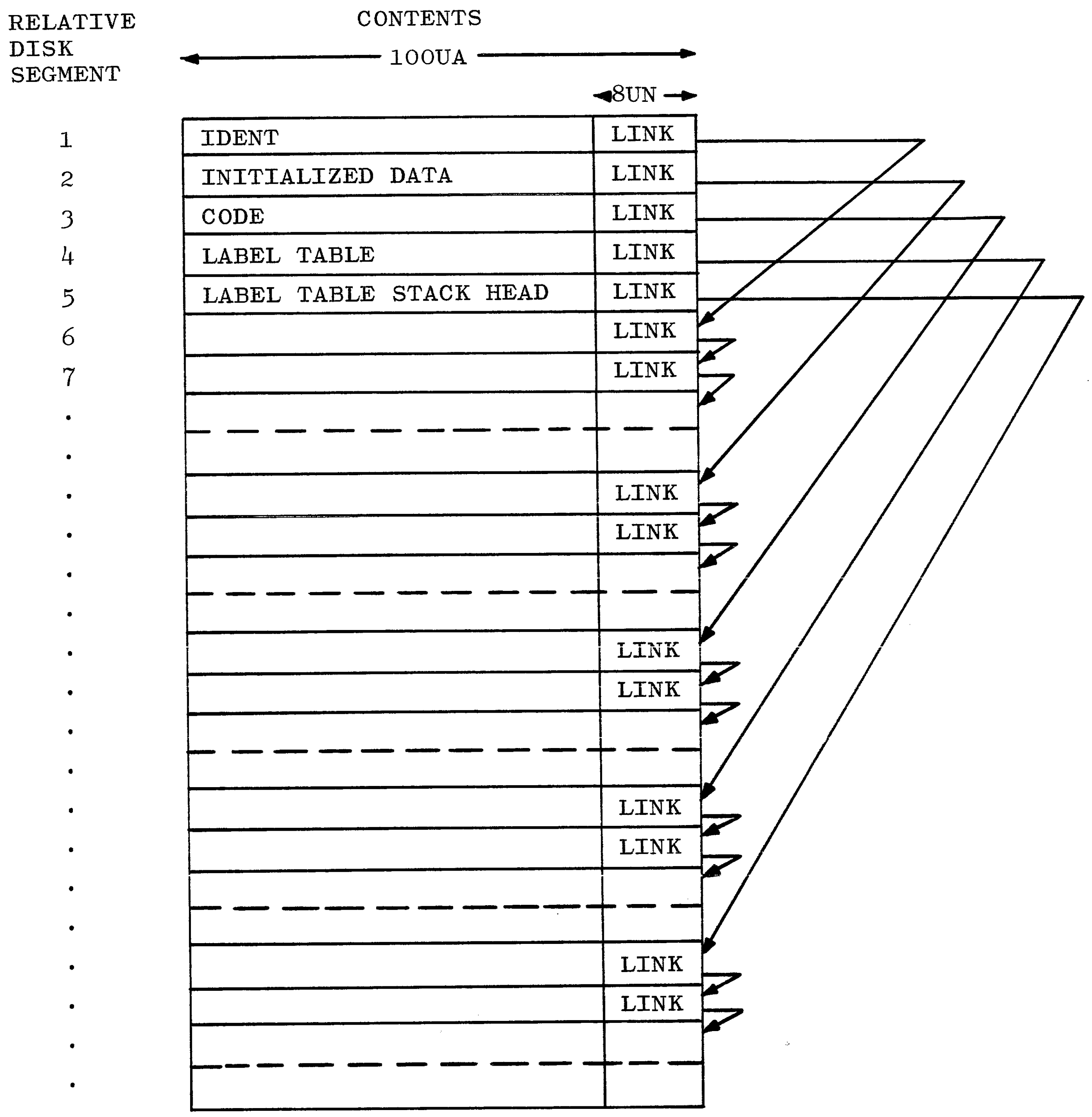
The above address is relative to an address which is determined during the load phase of the compiler and specified by the 2-digit CODE prefix. Refer to the table of storage allocation codes given below.

2) If files are defined, information used by the FORTRAN Compiler to build file information blocks (FIB's).

c. CODE segment - machine instructions generated for the program part in a pseudo format. Each address in an instruction is

APPENDIX F (cont)

FORBLR



preceded by a 2-digit storage allocation code, specifying the address to which it is relative.

d. LABEL TABLE segment - labels defined within the program part.

e. LABEL TABLE STACK HEAD segment - reserved.

As stated above, all addresses in an ICS file are relative to a specified address which is determined during the load phase (described below) of the FORTRAN Compiler. A 2-digit code which precedes each address indicates the particular base relative address in the object program to which the given address is relative. These "storage allocation codes" are defined below.

Code	Meaning
50	Base register relative
51	Data relative
52	Code relative
53	Function table relative
54	Single temp relative
55	Junk relative
56	Parameter relative
57	File information blocks
58	Double temp relative
59	Label table relative
60	TWO relative
61	Function results
62	Common area relative
63	HALF relative

FORBLR

Code	Meaning
64	ACCUM relative
65	NFLOAT relative
66	Intrinsic temps relative
67	Variable Unit Table relative
68	First COMMON generated by FORBLR
69	Second COMMON generated by FORBLR
70	Third COMMON generated by FORBLR
71	Fourth COMMON generated by FORBLR
72	Fifth COMMON generated by FORBLR
73	Sixth COMMON generated by FORBLR
74	Seventh COMMON generated by FORBLR
75	Eighth COMMON generated by FORBLR
76	Ninth COMMON generated by FORBLR
77	Tenth COMMON generated by FORBLR
78	FORBLR generated file

Pertinent areas referenced by the storage allocation codes are described in appendix E (Debugging Aids).

LOAD PHASE.

In its load phase, the FORTRAN Compiler creates an object code file (executable program) by accessing ICS files of the main program and the INTRN. file, and those of referenced program parts. The program part identifiers in the IDENT segments of each ICS file provide the linkage necessary to create a complete object program. As the code file is built, the base relative addresses are associated with storage allocation codes, and addresses in the pseudo code files are then adjusted to be base relative. The compiler builds the object code file using data from INITIALIZED DATA segments and instructions from CODE segments; it provides for specified segmentation where possible.

COMMUNICATIONS BETWEEN PROGRAM PARTS.

At the time of program execution, a FORBLR routine is part of the object code of a FORTRAN program and is referenced as though it were a FORTRAN subroutine. From symbolic code, the FORBLR Assembler creates a pseudo code file having the same format as that produced for each program part by the FORTRAN Compiler. (Refer to compile phase above.) The file ID of the ICS file of a FORBLR routine is taken from the IDNT Card. During the load phase of the compiler, all referenced FORBLR routines are made part of the object code file of the FORTRAN program.

A FORBLR subroutine is executed through an NTR instruction from the calling program part. In FORTRAN this is accomplished with a CALL statement to the name coded in the IDNT Card in the routine. The FORTRAN Compiler generates an NTR instruction for a CALL statement with constants generated for parameters passed in an argument list following the NTR. A FORBLR routine may call another FORBLR routine or a FORTRAN subroutine by executing an appropriate NTR instruction.

Example

```
?EXECUTE FORBLR
?DATA CARDS
             SPEC    CARD
             IDNT    ROUT
              ........
              ........
             FINI
?END
?COMPILE FORPRO FORTAN
?DATA CARDS
       ..........
       ..........
```

```
        CALL ROUT
        .........
        END
?END
```

The FORTRAN Compiler generates the following code for a CALL statement:

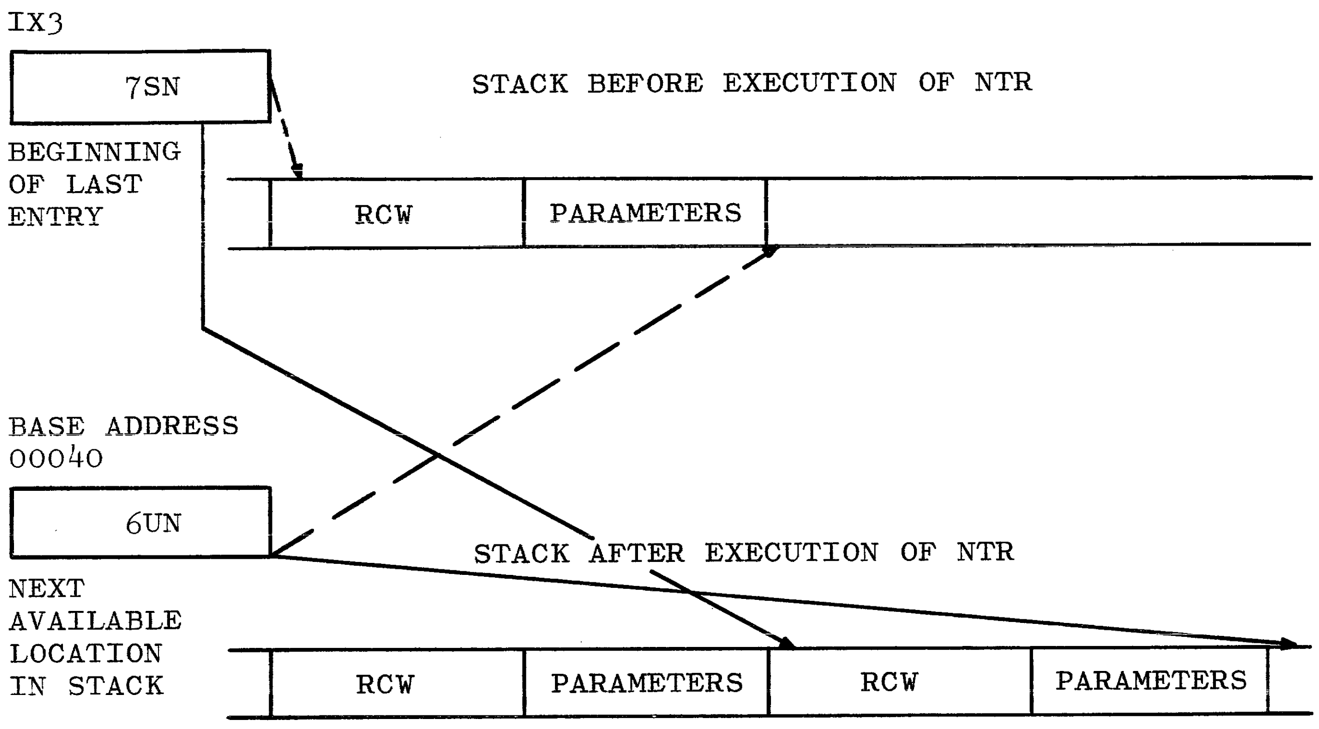
```
NTR xxxx nnnnnn     xxxx = number of bytes passed; nnnnnn = first
                    executable instruction in subroutine

CNST  12 UN         Reserved

CNST   2 UN         Parameter length      ─┐
                                           ├── One pair per
CNST   6 UN         Address of parameter  ─┘   parameter passed

............

............
```

The address in the NTR instruction is the address of the instruction that is executed next (nnnnnn above).

Execution of an NTR instruction creates an entry in the stack of the program; in a FORTRAN program the stack is a reserved area of core at the top of the program.

A stack entry is made beginning at the address contained at base relative address 00040. The contents of index register 3 (IX3) and the address of the first executable instruction following the NTR are stored in the stack entry as part of the return control word, and the address in location 40 is placed in IX3. Thus, IX3 points to the most recently created entry in the stack. After execution of an NTR, the address of the first digit following the last stack entry is in location 40. The following diagram illustrates the execution of an NTR.

FORBLR


IX3
7SN
STACK BEFORE EXECUTION OF NTR
BEGINNING OF LAST ENTRY
RCW
PARAMETERS
BASE ADDRESS 00040
6UN
STACK AFTER EXECUTION OF NTR
NEXT AVAILABLE LOCATION IN STACK
RCW
PARAMETERS
RCW
PARAMETERS

The stack entry consists of a return control word and constants and/or address constants following the NTR instruction. Its format when an NTR for a CALL statement is executed is:

Number of Digits	Use	
6	Return address	Return Control Word (RCW)
8	Contents in IX3 at NTR	
1	Zero (0)	
1	Toggles	
12	Reserved	
2	Parameter length	One pair for each parameter passed
6	Address of parameter	
.		
.		
.		

FORBLR

It is necessary to understand the execution of an NTR instruction, for it is through the stack that a calling program part passes and receives variables to and from a FORBLR subroutine.

ADDITIONAL PSEUDOS AVAILABLE WITH FORBLR.

To provide a facility for communications between a FORBLR routine and the FORTRAN program of which it is a part, several special-purpose pseudos are available with FORBLR. Descriptions of these pseudos and their functions follow.

SUBR PSEUDO.

A SUBR pseudo must be used when a FORBLR subroutine calls another subroutine (this includes a supplied routine from the INTRN. file). One SUBR pseudo is coded for each subroutine called. This pseudo creates an entry in the IDENT segment of the ICS file of the FORBLR routine, ensuring proper linkage with the referenced subprogram.

The format of the SUBR pseudo is:

	SEQ NO.	LABEL	OP CODE	VAR		A ADDRESS				B ADDRESS				C ADDRESS			
				AF	BF	LABEL	±INC	AI	AC	LABEL	±INC	BI	BC	LABEL	±INC	CI	CC
01	02	08	14	18	20	22	28	31	32	34	40	43	44	46	52	55	56
		SIN	SUBR														

The name of a subprogram which is called by the FORBLR subroutine is coded in the label field of a SUBR pseudo.

<u>Example</u>

	SEQ NO.	LABEL	OP CODE	VAR		A ADDRESS				B ADDRESS				C ADDRESS			
				AF	BF	LABEL	±INC	AI	AC	LABEL	±INC	BI	BC	LABEL	±INC	CI	CC
0 1	0 2	0 8	1 4	1 8	2 0	2 2	2 8	3 1	3 2	3 4	4 0	4 3	4 4	4 6	5 2	5 5	5 6
		• • • • • •	• • • •	• •	• •	•											
		VALUE	CNST		5	SN				C00654							
		SQRT	SUBR														
		• • • • • •	• • • •	• •	• •	•											
		• • • • • •	• • • •	• •	• •	•											
			NTR	00	10	SQRT											
			CNST		12	UN											
			CNST		2	UN				06							
			ACØN			VALUE											
		• • • • • •	• • • •	• •	• •	•											

COMN AND ENDC PSEUDOS.

The COMN and ENDC pseudos are used to declare COMMON blocks. A COMMON block containing a variable which is referenced in a FORBLR routine must be defined in that routine. The COMN pseudo has the following format:

	SEQ NO.	LABEL	OP CODE	VAR		A ADDRESS				B ADDRESS				C ADDRESS			
				AF	BF	LABEL	±INC	AI	AC	LABEL	±INC	BI	BC	LABEL	±INC	CI	CC
0 1	0 2	0 8	1 4	1 8	2 0	2 2	2 8	3 1	3 2	3 4	4 0	4 3	4 4	4 6	5 2	5 5	5 6
		CØMN1	CØMN														

The name of the referenced COMMON block is coded in the label field of the COMN pseudo. A blank label field defines blank COMMON.

FORBLR

DATA declarations follow the COMN pseudo to define COMMON elements, and an ENDC pseudo terminates the declaration. The ENDC pseudo has the following format:

	SEQ NO.	LABEL	OP CODE	VAR		A ADDRESS				B ADDRESS				C ADDRESS			
				AF	BF	LABEL	±INC	AI	AC	LABEL	±INC	BI	BC	LABEL	±INC	CI	CC
01	02	08	14	18	20	22	28	31	32	34	40	43	44	46	52	55	56
			ENDC														

Only DATA declarations may be coded between the COMN and ENDC pseudos.

DATA declarations following a COMN pseudo are considered individually to determine the total number of storage units associated with the COMMON block. The FORBLR Assembler assigns an even number of storage units to each DATA declaration and writes the total number of storage units following the block name in the ICS file of the routine.

Storage unit size is determined as it is by the FORTRAN Compiler, and the same default data sizes are assumed. Thus, by default, a storage unit is 12 digits.

When a FORBLR subroutine contains COMMON declarations, storage unit size must correspond to that used in the FORTRAN program in which the routine is referenced. Data sizes can be specified, as in the FORTRAN SIZE Control Card, to alter storage unit size with the REAL and INTG pseudos described below.

The following example illustrates communications between a FORTRAN program with COMMON declared and a FORBLR subroutine. (Assume default data sizes.)

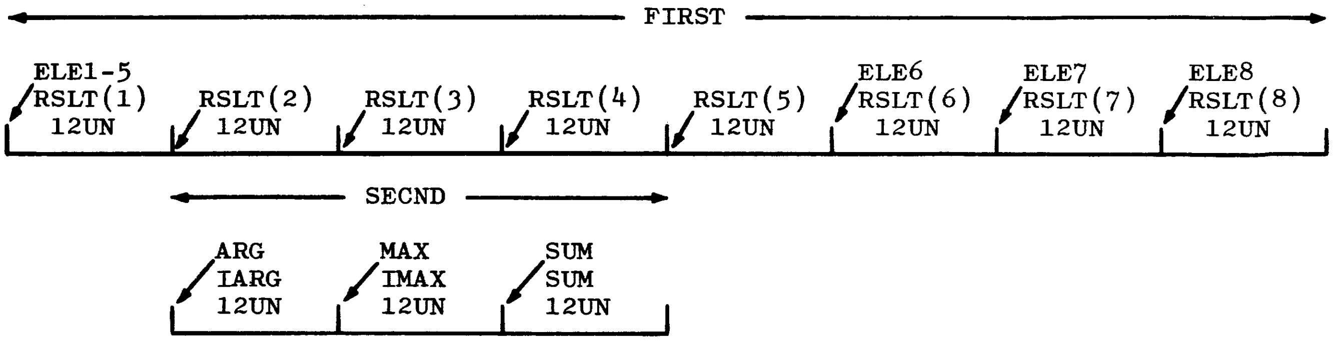
Example

```
COMMON /FIRST/RSLT(8)/SECND/IARG,IMAX,SUM
..........
..........
CALL FORB(IANS)
.........
STOP
END
```

				VAR		A ADDRESS				B ADDRESS				C ADDRESS			
	SEQ NO.	LABEL	OP CODE	AF	BF	LABEL	±INC	AI	AC	LABEL	±INC	BI	BC	LABEL	±INC	CI	CC
0 1	0 2	0 8	1 4	1 8	2 0	2 2	2 8	3 1	3 2	3 4	4 0	4 3	4 4	4 6	5 2	5 5	5 6
		•	• • • •	• •													
			IDNT			FØRB											
		FIRST	CØMN														
		ELE1-5	DATA		60	UN											
		ELE6	DATA		11	SN											
		ELE7	DATA		12	UN											
		ELE8	DATA		12	UN											
			ENDC														
		SECND	CØMN														
		ARG	DATA		5	SN											
		MAX	DATA		6	UN											
		SUM	DATA		11	SN											
			ENDC														
		•	• • • •	• •													
			ADD	5	5	ARG				MAX		5	N	BASE	30	3	IA
		•	• • • •	• •													
			EXT														
		•	• • • •	• •													

The COMMON blocks defined in the above example appear in core and are referenced as follows:

FORBLR



REAL AND INTG PSEUDOS.

The REAL and INTG pseudos "define" data sizes for real integer variables, respectively. The pseudos <u>only</u> effect the determination of storage unit size, which, in turn, defines the lengths of COMMON blocks declared in a FORBLR subroutine. The REAL and INTG pseudos have the following formats:

	SEQ NO.	LABEL	OP CODE	VAR		A ADDRESS				B ADDRESS				C ADDRESS			
				AF	BF	LABEL	±INC	AI	AC	LABEL	±INC	BI	BC	LABEL	±INC	CI	CC
01	02	08	14	18	20	22	28	31	32	34	40	43	44	46	52	55	56
			REAL			10											
			INTG			08											

For both pseudos, a 2-digit size is specified left-justified in the A ADDRESS field. The size coded in a REAL pseudo defines a mantissa length for real variables. The size coded in an INTG pseudo defines the number of digits of precision for integer variables.

Storage unit size is determined as with the FORTRAN Compiler, i.e., the largest of M+4, N+1, and 2xL; where M is the mantissa size of a real, N is the number of digits of precision for an integer, and L is the number of characters for an ALPHA variable.

EQIV PSEUDO.

The EQIV pseudo is used as it is in B 3500 Assembler. In addition, it may be used to associate a label to an address which is determined by the FORTRAN Compiler during its load phase. An EQIV pseudo with the following format performs this function.

	SEQ NO.	LABEL	OP CODE	VAR		A ADDRESS				B ADDRESS				C ADDRESS			
				AF	BF	LABEL	±INC	AI	AC	LABEL	±INC	BI	BC	LABEL	±INC	CI	CC
01	02	08	14	18	20	22	28	31	32	34	40	43	44	46	52	55	56
		FIBRAY	EQIV	00	06	00000											

A 2-digit storage allocation code (from the table on page F-5) is coded, left-justified, in the B ADDRESS field. This code specifies the address which is to be associated with the label. A length of 0006 is coded in the VAR (variant) field for a 6 UN address. The A ADDRESS field contains a dummy address of five zeros; at load time this address is "replaced" by the address specified by the storage allocation code.

In the following example the label FIBRAY points to the beginning of the variable unit table, which is specified by storage allocation code 67. The address of the FIB for unit 9 is moved to FIBADR; it is the tenth 6-digit value in the variable unit table (i.e., an increment of 6x9, or 54, to FIBRAY).

APPENDIX F (cont)

FORBLR

	SEQ NO.	LABEL	OP CODE	VAR		A ADDRESS				B ADDRESS				C ADDRESS			
				AF	BF	LABEL	±INC	AI	AC	LABEL	±INC	BI	BC	LABEL	±INC	CI	CC
01	02	08	14	18	20	22	28	31	32	34	40	43	44	46	52	55	56
		•	••••	••	••												
		FIBRAY	EQIV	00	06	00000				67							
		•	••••	••	••												
			MUN	02	07	54			NL	IX1			SN				
			MUN	06	06	FIBRAY		1	UN	FIBADR			UN				
		•	••••	••	••												
			BCT	01	14												
			BUN			*	020		UN								
		FIBADR	CNST		6	UN				000000							
		EØFLAB	CNST		6	UN				000000							
		•	••••	••	••												

PROGRAMMING CONSIDERATIONS.

Considerations to be observed when writing a FORBLR subroutine follow:

a. Assembler pseudos not available with FORBLR are: LOCN, SEGM, ENSG, ALOC, SORT, KEYA, KEYD, SKEY, SETT, RSET, BUMP, DECR.

b. An IDNT Card must be used. The identifier coded in this card is used for the file ID of the ICS file of the routine and is the name by which it is "called."

c. Declarations must precede executable code, i.e., file declarations and SUBR, COMN, EQIV, and DATA pseudos.

d. Declarations may be mixed freely.

e. In-line constant declarations, such as those coded after a BCT, must each be modulo-2 digits. However, one modulo-2 constant may be coded, with non-modulo-2 parts of it referenced through EQIV pseudos.

f. A STOP Card is not used unless program termination during execution of the routine is desired.

g. An EXT instruction is used to "terminate" execution of a FORBLR routine. Execution of an EXT returns control to the instruction following the NTR to the routine.

h. Macro and library routines (MACR,LIBR) may not be defined or referenced in FORBLR.

PROGRAMMING EXAMPLE.

Following is an example program using FORTRAN and FORBLR.

```
C THIS PROGRAM CALLS A FORBLR ROUTINE WHICH TAKES THE SQUARE ROOT OF
C THE PASSED PARAMETER.  THE RESULT IS PLACED IN A VARIABLE IN COMMON.
      REAL IRSLT,IARG
      COMMON /SCAN/IRSLT,ARAY(5)
      IARG=20.
C CALL FORBLR ROUTINE, PASSING IARG
      CALL DUMMY(IARG)
C THE SQUARE ROOT OF IARG + 5. IS IN IRSLT
      WRITE(6,10) IRSLT
   10 FORMAT(1X,F10.5)
      STOP
      END
```

			VAR		A ADDRESS				B ADDRESS				C ADDRESS				
SEQ NO.	LABEL	OP CODE	AF	BF	LABEL	±INC	AI	AC	LABEL	±INC	BI	BC	LABEL	±INC	CI	CC	REMARKS
02	08	14	18	20	22	28	31	32	34	40	43	44	46	52	55	56	58 80
		SPEC			CARD												LIST
		IDNT			DUMMY												ROUTINE NAME
	RSLT	CNST	00	11	SN												
	DUM	CNST	00	11	SN												
	SQRT	SUBR															EXTERNAL RØUTINE
	SCAN	CØMN															DEFINE CØMMØN BLØCK
	BACK	DATA		12	UN												
		DATA		60	UN												FILLER FØR ARRAY
		ENDC															END CØMMØN DEFINITIØN
	ANS	EQIV	00	06	00000				61								ADRS ØF FUNCTIØN RSLTS
	ARG	EQIV	00	06	BASE	30											ARGUMENT ADRS IN STACK
	I	MUN	11	11	ARG		3	IA	DUM								
		FAD	01	08	+08CS			SL	DUM				RSLT				
		NTR	00	10	SQRT												USE SQRT INTRINSIC TØ
		CNST		12	UN												TAKE SQRT ØF RSLT
		CNST		2	UN				12								
		ACØN			RSLT												
		MUN	12	12	ANS			UN	BACK								PLACE ANS IN CØMMØN
		EXT															RETURN
		FINI															

APPENDIX G
FORTRAN ERROR MESSAGES AND FLAGS

<u>Error Message</u> — <u>Error Number</u>

IMPROPER STATEMENT TYPE — 1

Self-explanatory.

ILLEGAL STATEMENT ORDER — 2

a. The non-executable statements immediately preceding this error message are not in acceptable order.

b. Statement order may be misarranged if one or more preceding statements are illegally begun in column 6 or if one or more executable statements have been interspersed with the non-executable statements.

c. If one or more of the non-executable statements is mispunched or misrepresented, an error message for that error is issued and that statement is ignored. This, therefore, can cause a change in sequence.

ILLEGAL COMBINATION OF OPERATOR AND OPERANDS — 24

Variables and/or constants must be separated by an operator. Two operators must not appear together. Parentheses may not be used as operators.

MISSING EQUAL SIGN IN ASSIGNMENT STATEMENT — 25

Self-explanatory.

IMPROPER NESTING OF DO STATEMENTS OR MISSING DO TERMINATION — 26

Self-explanatory.

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
INVALID OP CODE - COMPILER ERROR	27
SUBSCRIPTED VARIABLE IN EQUIVALENCE STATEMENT IS NOT DIMENSIONED	28
A subscripted variable is being used in an EQUIVALENCE statement without having been dimensioned prior to that statement.	
THIS STATEMENT CANNOT BE EXECUTED	29
Self-explanatory.	
REPEAT COUNT ON A FORMAT DESCRIPTOR IS ZERO	30
Repeat count must be an integer constant greater than zero.	
REPEAT COUNT ON A FORMAT GROUP IS ZERO	31
Repeat count must be an integer constant greater than zero.	
D, E, F OR G FORMAT DESCRIPTOR FIELD WIDTH IS ZERO	32
An I/O statement is attempting to describe a field width of zero.	
I, A OR L FORMAT DESCRIPTOR FIELD WIDTH IS ZERO	33
An I/O statement is attempting to describe a field width of zero.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
D, E, F OR G FORMAT DESCRIPTOR FIELD WIDTH IS LESS THAN FRACTIONAL WIDTH	34
A format field descriptor is specifying a field which is smaller than its decimal part. Example: F5.8.	
ARRAY ELEMENT IS NOT SUBSCRIPTED	35
A variable name declared as an array name is being used without a subscript.	
MISSING RIGHT PARENTHESIS IN SUBSCRIPT	36
Self-explanatory.	
IMPROPER SUBSCRIPT DELIMITER	37
Self-explanatory.	
SUBSCRIPT EXPRESSION IS NOT TYPE INTEGER	38
An expression used as a subscript must consist of integer constants and/or integer variables only.	
MISSING LEFT PARENTHESIS BEFORE UNIT NUMBER IN READ/WRITE	39
Self-explanatory.	
MISSING RIGHT PARENTHESIS BEFORE IO LIST IN READ/WRITE	40
Self-explanatory.	
HOLLERITH CONSTANT LENGTH IS ZERO	41
A Hollerith constant must be defined with a length greater than zero.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
IMPROPER FORMAT SPECIFIER IN READ/WRITE	42

Character read as format specifier is not recognized as valid. Check Hollerith constant lengths and punctuation.

LAST STATEMENT NOT COMPLETELY PROCESSED — 43

This message is self-explanatory and usually follows another error message.

IMPROPER DELIMITER IN IO LIST — 44

Elements in an I/O list must be separated by commas. Check the lengths associated with Hollerith constants.

IMPROPER ELEMENT IN IO LIST — 45

Self-explanatory.

IMPROPER IMPLIED DO INITIALIZATION PARAMETER — 46

At the time of execution of the DO statement, the initial parameter must be greater than zero, must be an integer constant or integer variable, and must not be greater than the final parameter.

MISSING LEFT PARENTHESIS IN IMPLIED DO — 47

Self-explanatory.

MISSING RIGHT PARENTHESIS AFTER IMPLIED DO — 48

Self-explanatory. Carefully examine syntax of implied DO usage.

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
COMPLEX EXPRESSIONS NOT ALLOWED WITH RELATIONALS	49
Complex constants and variables are not permitted in a relational expression.	
INVALID UNIT NUMBER	50
Unit number may be integers between 1 and 19, inclusive.	
MISSING RIGHT PARENTHESIS IN WHAT APPEARS TO BE A COMPLEX CONSTANT	51
Complex constants are coded as two real constants delimited by a comma and enclosed by parentheses. Example: (3.2,4.5)	
IMPROPER SUBROUTINE NAME IN SUBROUTINE STATEMENT	52
Subroutine name is one to six alphanumeric characters, the first of which must be alphabetic.	
MISSING LEFT PARENTHESIS IN SUBROUTINE STATEMENT	53
The parameter list in a SUBROUTINE statement must be enclosed by parentheses.	
COMPLEX EXPRESSION NOT ALLOWED IN IF STATEMENT	54
Self-explanatory.	
IMPROPER DUMMY VARIABLE	55
Dummy variable in argument list must follow naming	

Error Message	Error Number
conventions for variables; it must be of the same type as its corresponding actual parameter.	
IMPROPER FUNCTION NAME IN FUNCTION STATEMENT	56
Function name is from one to six alphanumeric characters, the first of which must be alphabetic.	
MISSING LEFT PARENTHESIS IN FUNCTION STATEMENT	57
An argument list associated with a FUNCTION statement must be enclosed by parentheses.	
INCOMPLETE LIST IN DIMENSION STATEMENT	58
DIMENSION statement is incorrect.	
IMPROPER IDENTIFIER IN DIMENSION STATEMENT	59
Identifiers must adhere to naming conventions.	
MISSING LEFT PARENTHESIS IN DIMENSION STATEMENT	60
Dimensions must be enclosed by parentheses.	
ARRAY BOUND IS TOO LARGE	61
The maximum array size which may be specified in a DIMENSION statement is 9999 elements.	
ADJUSTABLE DIMENSIONS CAN ONLY BE USED WITH DUMMY VARIABLES	62
Variables in the main program may not have adjustable dimensions.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
IMPROPER ARRAY BOUND	64
Array bound must be an integer constant, or an integer variable when adjustable dimensioning is used.	
IMPROPER ARRAY DELIMITER	65
Array declarations must be separated by commas. Array dimensions must be separated by commas.	
IMPROPER ARRAY DELIMITER	66
Array declarations must be separated by commas. Array dimensions must be separated by commas.	
IMPROPER COMPUTED GO TO VARIABLE	67
Control variable must be an integer expression whose value is no greater than the number of statement labels in the list.	
INDETERMINATE STATEMENT TYPE. ASSIGNMENT STATEMENT ASSUMED	68
Statements immediately preceding this message may not be in legal order.	
IMPROPER ARRAY DECLARATOR DELIMITER	69
DIMENSION statement is incorrect. List-elements should be separated by commas.	
IMPROPER SUBROUTINE IDENTIFIER IN CALL STATEMENT	70
CALL statement is coded incorrectly.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
MISSING LEFT PARENTHESIS IN CALL STATEMENT	71
Actual parameters in CALL statement must be enclosed by parentheses.	
TOO MANY IDENTIFIERS IN DATA STATEMENT	72
Compiler table is 450 digits. Nine digits are used per named variable; 13 digits are used per array name. Break the DATA statement into two or more accordingly.	
MISSING LEFT PARENTHESIS IN EQUIVALENCE STATEMENT	73
Equivalenced variables must be enclosed by parentheses.	
MISSING RIGHT PARENTHESIS IN EQUIVALENCE STATEMENT	74
Equivalenced variables must be enclosed by parentheses.	
UNIT NUMBER MUST BE LESS THAN 20	75
Unit number must be between 1 and 19, inclusive.	
DO TERMINATION LABEL SHOULD NOT APPEAR ON THIS STATEMENT	76
DO may not terminate on a GO TO, IF, RETURN, STOP, DO, or REREAD statement.	
LABEL MAY NOT APPEAR ON END STATEMENT	77
Self-explanatory.	
MISSING RIGHT PARENTHESIS IN ASSIGNED GO TO STATEMENT	78
Statement numbers must be enclosed by parentheses.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
MISSING RIGHT PARENTHESIS IN I/O STATEMENT	79
Parentheses must surround unit specifier, and FORMAT statement label if used.	
MISSING COMMA IN I/O STATEMENT PARAMETER LIST OR IMPROPER ELEMENT IN VARIABLE LIST	80
Elements must be separated by commas. Check the lengths of Hollerith constants.	
TWO MAIN PROGRAMS--CHECK FOR "SUBROUTINE", "BLOCKDATA" OR "FUNCTION" CARD STARTING IN COLS. 1-6	81
Self-explanatory.	
SUBSCRIPTED VARIABLE NOT SEEN IN DIMENSION STATEMENT	82
Array bounds must be specified in a DIMENSION, TYPE, or COMMON statement before a subscripted variable is referenced.	
IMPROPER DELIMITER IN COMMON LIST	83
List elements in a COMMON statement must be separated by commas.	
MISSING SLASH IN COMMON STATEMENT	84
COMMON block names must be enclosed by slashes.	
CONTINUATION CARD MUST CONTINUE SOMETHING	85
Self-explanatory. Check keypunching.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
IMPROPER COMMON LIST ELEMENT	86
A COMMON list consists of COMMON block names, identifiers, and, optionally, dimensions.	
EMPTY COMMON LIST	87
There are no identifiers associated with COMMON block.	
IMPROPER ELEMENT IN TYPE STATEMENT	88
Elements in TYPE statement must be identifiers, simple or subscripted.	
TWO END STATEMENTS	89
A program part may contain only one END statement as the last card in the symbolic code for that program part.	
INVALID DATA IN SIZE CARD	90
The size of a data type is expressed as an integer.	
IMPROPER DELIMITER IN TYPE STATEMENT LIST	91
Elements in TYPE statement are delimited by commas.	
IMPROPER LIST ELEMENT IN DATA STATEMENT	92
Elements must be identifiers, subscripted or simple.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
ARRAY IDENTIFIER MUST BE FOLLOWED BY A LEFT PARENTHESIS IN A DATA STATEMENT	93

If array identifier is subscripted, subscript must be enclosed by parentheses.

```
Example:  DATA ARAY(2)/3.4/
          but, DATA ARAY/3.4,4.4,5.6/
```

INCORRECT NUMBER OF SUBSCRIPTS OF AN ARRAY ELEMENT IN A DATA STATEMENT LIST	94

Number of subscripts must correspond to the number defined in DIMENSION statement.

IMPROPER DELIMITER IN DATA STATEMENT LIST	95

Check commas and slashes for correct positioning in DATA statement.

MISSING STOP CARD--LAST OUTPUT LINE MAY NOT BE PRINTED	96

Check that program has at least one STOP Card or a CALL EXIT statement.

MISSING END CARD	97

Check that each program part is terminated with an END statement.

IMPROPER DELIMITER IN VALUE LIST OF DATA STATEMENT	98

Elements in value list must be delimited by commas.

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
THE LENGTHS OF THE DATA AND VALUE LISTS IN A DATA STATEMENT ARE UNEQUAL The number of values must correspond to the number of variables to be initialized. Check punctuation.	99
ILLEGAL LOAD CARD--ONLY ONE ALLOWED PER COMPILE Self-explanatory.	100
IMPROPER CONSTANT IN VALUE LIST OF DATA STATEMENT Constant must correspond in type to variable to be initialized.	101
EXTERNAL STATEMENT MAY NOT APPEAR IN BLOCKDATA SUBPROGRAM Self-explanatory.	102
DECIMAL POINT ASSUMED IMMEDIATELY AFTER THE MANTISSA When decimal point is not coded in a real constant, it is assumed to follow the mantissa.	103
IMPROPER STATEMENT NUMBER IN ASSIGN STATEMENT A statement number must be an integer constant or variable greater than zero and less than six digits when used in an ASSIGN statement.	107 and 108
ASSIGN STATEMENT VARIABLE MUST BE INTEGER VARIABLE A variable used in an ASSIGN statement must be of type INTEGER.	109

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
INTEGER PRECISION MUST BE GREATER THAN 4 TO USE ASSIGN	110
Self-explanatory.	
INTEGER SIZE TOO LARGE COULD CAUSE RUN TIME ERRORS	111
Self-explanatory.	
ALPHA SIZE TOO LARGE COULD CAUSE RUN TIME ERRORS	112
REAL SIZE TOO LARGE COULD CAUSE RUN TIME ERRORS	113
IMPROPER EQUIVALENCE STATEMENT-POSSIBLE MISSPELLING	118
Self-explanatory.	
ILLEGAL CHARACTER IN FORTRAN STATEMENT	119
Check character set for appearance of characters in statement containing error. Frequent error is use of quote marks instead of legal H format specifier to specify Hollerith data.	
STACK OVERFLOW CAN OCCUR. USE SUB-EXPRESSIONS	120
Involved expression may cause compiler stack overflow. Use sub-expressions or give compiler more core in which to execute with MCP CORE Card.	
IMAGINARY PART OF WHAT APPEARS TO BE A COMPLEX CONSTANT IS NOT REAL	121
Complex values must be expressed as floating point values.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
MISSING PERIOD FOLLOWING RELATIONAL OR LOGICAL OPERATOR, OR LOGICAL VALUE	122
Self-explanatory.	
INVALID RELATIONAL OR LOGICAL OPERATOR, OR LOGICAL VALUE	123
Self-explanatory.	
INVALID SPECIAL CHARACTER	124
It is illegal to mix character codes within one program.	
IDENTIFIER CONTAINS MORE THAN SIX CHARACTERS	125
Self-explanatory.	
A REAL CONSTANT WILL HAVE TO BE TRUNCATED	126
The constant cited has exceeded the prescribed precision.	
INVALID EXPONENT	127
Self-explanatory.	
AN ALPHA CONSTANT WILL HAVE TO BE TRUNCATED	128
The constant cited has exceeded the prescribed precision.	
A DOUBLE PRECISION CONSTANT WILL HAVE TO BE TRUNCATED	129
The constant cited has exceeded the prescribed precision.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
INVALID DOLLAR SIGN OPTION	130
Self-explanatory.	
STATEMENT NUMBER MUST BE INTEGER	131
Self-explanatory.	
STACK OVERFLOW CAN OCCUR, USE SUB-EXPRESSIONS	132
Involved expression may cause compiler stack overflow. Use sub-expressions or give compiler more core in which to execute with MCP CORE Card.	
NORMALIZED EXPONENT IS TOO LARGE	133
Self-explanatory.	
EXPONENT HAS TOO MANY DIGITS	134
Exponent may have only two digits.	
INTEGER CONSTANT TOO LARGE	135
An integer constant appearing immediately before this message exceeds the precision for integer values.	
STACK OVERFLOW OCCURRED, USE SUB-EXPRESSIONS	136
An involved expression has caused a compiler stack overflow. Use sub-expressions or give compiler more core in which to execute with MCP CORE Card.	
STACK OVERFLOW OCCURRED, USE FEWER PARAMETERS OR SUB-EXPRESSIONS	137
Check function call preceding error message.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
UNEXPECTED COMMA TREATED AS TERMINATOR	138
Self-explanatory.	
SEQUENCE ERROR	139
Self-explanatory.	
RETURN STATEMENT NOT ALLOWED IN MAIN PROGRAM OR BLOCKDATA SUBPROGRAM	140
Self-explanatory.	
NO RETURN STATEMENT IN THIS SUBROUTINE	141
A subroutine must contain at least one RETURN statement.	
INCORRECT USE OF COMMA IN AN EXPRESSION	142
Self-explanatory.	
HOLLERITH CONSTANT NOT ALLOWED IN AN EXPRESSION	143
Self-explanatory. A Hollerith constant may be used only in an assignment statement such as ALF=2HAB.	
COMPILER ERROR (INVALID ID IN EXPRESSION)	144
A subroutine ID, file ID, or COMMON block ID may not appear in an expression.	
ILLEGAL QUANTITY IN EXPRESSION	145
Self-explanatory.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
MISSING OPERATOR IN EXPRESSION	146
Self-explanatory.	
MISSING OPERATOR IN EXPRESSION	147
Self-explanatory.	
MISSING OPERAND IN AN EXPRESSION	148
Self-explanatory.	
STATEMENT NUMBER IS GREATER THAN FIVE DIGITS	149
Self-explanatory.	
ASSIGNMENT STATEMENT HAS AN OPERATOR PRECEDING THE EQUALS	150
Self-explanatory.	
EQUAL SIGN IN NON-ASSIGNMENT STATEMENT EXPRESSION	151
Self-explanatory.	
ILLEGAL OR MISSING OPERATOR IN AN EXPRESSION	152
Self-explanatory.	
ILLEGAL OR MISSING OPERATOR IN AN EXPRESSION	153
Self-explanatory.	
ILLEGAL COMBINATION OF TYPES IN AN EXPRESSION	154
Mixed modes are not permitted in this expression.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
CANNOT DETERMINE STATEMENT TYPE Self-explanatory.	156
MIXED INTEGER AND REAL TYPES IN EXPRESSION Self-explanatory.	157
A SEQ DOLLAR SIGN OPTION WITH A PLUS REQUIRES AN INCREMENT Self-explanatory.	158
THE WORD NEW SHOULD BE FOLLOWED BY THE WORD TAPE IN $ CARD Dollar sign option for creating a new tape is incorrectly coded.	159
ILLEGAL COMBINATION OF TYPES ACROSS AN EQUAL SIGN Self-explanatory.	160
NUMBER LONGER THAN 100 DIGITS Self-explanatory.	161
FRACTIONAL PART LONGER THAN 100 DIGITS Self-explanatory.	162
INCORRECT STOP STATEMENT STOP statement is coded incorrectly.	163
COMPILER ERROR (MISTAKENLY IN EXPRESSION BLOCK) Expression block expects an assignment, IF, or CALL statement which has not been found.	164

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
IMPLIED DO CONTROL VARIABLE MUST BE INTEGER	169
Self-explanatory.	
IMPLIED DO INITIALIZATION PARAMETER MUST BE INTEGER TYPE	170
Self-explanatory.	
IMPLIED DO TERMINATION PARAMETER MUST BE INTEGER TYPE	171
Self-explanatory.	
IMPLIED DO INCREMENTATION PARAMETER MUST BE INTEGER TYPE	172
Self-explanatory.	
STACK OVERFLOW CAN OCCUR. USE SUB-EXPRESSIONS	174
This error message is generated when a statement contains an excessive number of operations. Use sub-expressions or give the compiler more core in which to execute with the MCP CORE Card.	
COMPILER ERROR (PUTIN CANNOT FIND INFO ENTRY)	175
Check statement immediately preceding error message. If correctly coded, replace deck in card reader for recompilation (possible error in reading card).	
INCORRECT QUANTITY FOLLOWING AN IF STATEMENT	176
Self-explanatory.	
A FUNCTION IDENTIFIER APPEARS WITH NO ARGUMENTS	177
Self-explanatory.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
EXPRESSION CAUSED STACK OVERFLOW. USE SUB-EXPRESSIONS	178
An excessive number of operations are contained within one statement. Use sub-expressions or give compiler more core in which to execute with MCP CORE Card.	
MISSING RIGHT PARENTHESIS IN FUNCTION CALL	179
Self-explanatory.	
A FUNCTION IDENTIFIER PRECEDES THE EQUAL SIGN IN AN ASSIGNMENT STATEMENT	180
Check for array identifier that has not been dimensioned.	
IMPROPER DELIMITER FOLLOWING A DUMMY ARGUMENT OF AN ARITHMETIC STATEMENT FUNCTION	182
Dummy arguments must be separated by commas.	
COMPILER ERROR (CHECKROW: INFO ROW DOES NOT EXIST)	183
IMPROPER COMMON BLOCK NAME	184
Self-explanatory.	
A VARIABLE APPEARS TWICE IN COMMON	186
Self-explanatory.	
A VARIABLE HAS BEEN DIMENSIONED TWICE	188
Check for variable dimensioned in COMMON that is also dimensioned in a TYPE or DIMENSION statement.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
IMPROPER ELEMENT IN EQUIVALENCE STATEMENT Self-explanatory.	189
A DUMMY ARGUMENT CANNOT BE EQUIVALENCED Self-explanatory.	190
AN EQUIVALENCE GROUP MUST CONTAIN MORE THAN ONE ELEMENT An element must be equivalenced to something.	191
CONTRADICTORY OR REDUNDANT EQUIVALENCE STATEMENT Self-explanatory.	192
IMPROPER QUANTITY FOLLOWING AN EQUIVALENCE STATEMENT Self-explanatory.	193
SUBSCRIPT IN EQUIVALENCE IS NOT INTEGER Self-explanatory.	194
MAXIMUM SUBSCRIPT SIZE IS FIVE DIGITS Maximum subscript is 9999.	195
TOO MANY SUBSCRIPTS IN AN EQUIVALENCED VARIABLE Number of subscripts must correspond to dimension of variable.	196
MISSING RIGHT PARENTHESIS FOLLOWING THE SUBSCRIPT OF AN EQUIVALENCED VARIABLE	197

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
Equivalenced variables must be enclosed by parentheses. Subscripts must be enclosed by parentheses.	
TOTAL ARRAY SIZE IS TOO LARGE	198
An array may have no more than 9999 elements.	
TOTAL ARRAY SIZE IS TOO LARGE	199
Array may contain no more than 9999 elements.	
ARRAY REQUIRES MORE THAN 100,000 DIGITS	201
The core requirement of a program may not exceed 100,000 digits. This error message indicates that an array alone exceeds 100,000 digits.	
SUBSCRIPTED VARIABLE IN AN EQUIVALENCE STATEMENT IS NOT DIMENSIONED	202
A variable which appears in an EQUIVALENCE statement must have been dimensioned previously.	
SUBSCRIPT VALUE IN AN EQUIVALENCE STATEMENT IS TOO LARGE	203
Value exceeds declared dimension of identifier.	
CONTRADICTION IN EQUIVALENCING VARIABLES	204
Self-explanatory.	
DIMENSIONED VARIABLE IN TYPE STATEMENT PREVIOUSLY DIMENSIONED	206
A variable may be dimensioned only once.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
INTEGER PRECEDING X FORMAT DESCRIPTOR MAY NOT EXCEED 132	208
Self-explanatory.	
COMPILER ERROR (FRSTX: COMMON NUMBER CANNOT BE FOUND)	209
IMPROPER QUANTITY FOLLOWING A RETURN STATEMENT	210
RETURN statement is coded incorrectly.	
IMPROPER QUANTITY FOLLOWING AN EXTERNAL STATEMENT	211
EXTERNAL statement is coded incorrectly.	
FUNCTION NAME HAS NOT APPEARED LEFT OF THE EQUAL SIGN IN AN ASSIGNMENT STATEMENT	212
In FUNCTION subprogram, the function name must appear at least once on the left side of an equal sign in an assignment statement.	
END OF FILE AND PARITY ACTION LABELS MUST BE PRECEDED BY AN EQUALS	213
In I/O statement, action labels are coded as END=n or ERR=n.	
END OF FILE AND PARITY ACTION LABELS MUST BE STATEMENT NUMBERS	214
Self-explanatory.	
COMMON BLOCK NAME MUST BE AN IDENTIFIER	215
Self-explanatory.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
CANNOT FIND AN IDENT CARD ON TAPE	216
IDENT Card must be used in creating a new tape when a REPLACE Card is to be used.	
UNIT NUMBER NOT TYPE INTEGER IN READ/WRITE STATEMENT	217
Variable used as unit specifier must be of type INTEGER.	
MISSING LEFT PARENTHESIS PRECEDING THE UNIT NUMBER IN READ/WRITE STATEMENT	218
Self-explanatory.	
FIRST ARGUMENT IN AN INTRINSIC IS THE WRONG TYPE	219
Self-explanatory. Check function usage.	
SECOND ARGUMENT IN AN INTRINSIC IS THE WRONG TYPE	220
Self-explanatory. Check function usage.	
ARGUMENT IN MAX/MIN INTRINSIC IS THE WRONG TYPE	221
Self-explanatory. Check function usage.	
FORMAT ARRAYS MUST HAVE AN EVEN ELEMENT LGTH UNLESS EQIV/COMMON REQUIRING EVEN STORAGE UNIT LGTH	222
Array used as FORMAT must be of type ALPHA.	
AN IDENTIFIER IN A BLOCK DATA SUBPROGRAM DATA STATEMENT MUST HAVE BEEN PREVIOUSLY SEEN	223

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
Identifier must be defined in COMMON statement within BLOCK DATA subprogram.	
A DUMMY ARGUMENT MAY NOT APPEAR IN A DATA STATEMENT	224
Self-explanatory.	
COMMON VARIABLES MAY ONLY APPEAR IN A DATA STATEMENT IN A BLOCKDATA SUBPROGRAM	225
Self-explanatory.	
IN A BLOCK DATA SUBPROGRAM ALL DATA STATEMENT VARIABLES MUST BE IN COMMON	226
Self-explanatory. Check for declaration of variables in COMMON statement in BLOCK DATA subprogram as well as main program.	
IN A DATA STATEMENT THE QUANTITY PRECEDING AN ASTERISK MUST BE AN INTEGER CONSTANT	227
Repeat-specification must be an integer.	
A SUBSCRIPT IN A DATA STATEMENT HAS MORE THAN FOUR DIGITS	228
An array may have no more than 9999 elements.	
THE PRODUCT OF TWO SUBSCRIPTS IN A DATA STATEMENT HAS MORE THAN FOUR DIGITS	229
An array may contain no more than 9999 elements.	
A SUBSCRIPT IN A DATA STATEMENT IS NOT AN INTEGER CONSTANT	230
Self-explanatory.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
THE PRODUCT OF THE SUBSCRIPTS IN A DATA STATEMENT HAS MORE THAN FOUR DIGITS	231
An array may contain no more than 9999 elements.	
INVALID CHARACTER IN ALPHA STATEMENT	232
ALPHA statement is not coded correctly.	
ALPHA TYPE ON FUNCTION IS NOT ALLOWED	233
Self-explanatory.	
SECOND ARGUMENT IN CMPLX INTRINSIC IS WRONG TYPE	234
Self-explanatory. Check intrinsic usage.	
INVALID FORMAT DESCRIPTOR	235
FORMAT statement is coded incorrectly.	
INVALID FORMAT LIST ELEMENT	236
Self-explanatory.	
UNASSOCIATED INTEGER IN FORMAT LIST	237
Self-explanatory.	
INVALID CHARACTER IN FORMAT LIST	238
Self-explanatory.	
INVALID SCALE FACTOR FOLLOWING MINUS	239
Compiler assumes an intended scale factor when it encounters a minus sign in a FORMAT statement.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
MISSING P-DELIMITER FOLLOWING NEGATIVE SCALE FACTOR	240
P must follow scale factor and precede format specifier.	
MISSING RIGHT PARENTHESIS IN FORMAT	241
FORMAT statement must terminate with a parenthesis.	
P-DELIMITER NOT PRECEDED BY AN INTEGER IN FORMAT	242
Scale factor must be specified explicitly by an integer preceding P.	
X FORMAT DESCRIPTOR NOT PRECEDED BY AN INTEGER	243
Self-explanatory.	
INVALID FIELD WIDTH FOLLOWING D, E, F OR G DESCRIPTOR	244
FORMAT statement is coded incorrectly.	
MISSING DOT FOLLOWING FIELD WIDTH	245
Period must separate field width and factional part of format specifier.	
INVALID FRACTION FIELD WIDTH FOLLOWING DOT	246
Fraction part is missing or larger than field width.	
INVALID FIELD WIDTH FOLLOWING A, I OR L DESCRIPTOR	247
FORMAT statement is coded incorrectly.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
MISSING DELIMITER BEFORE HOLLERITH CONSTANT IN FORMAT	248
FORMAT statement is coded incorrectly.	
TOO MANY RIGHT PARENTHESES IN FORMAT	249
Too many right parentheses or garbage is found following final right parenthesis.	
PROGRAM UNIT CONTAINS MORE THAN 100,000 DIGITS. SUBROUTINIZE AND SEGMENT OR OPTIMIZE DATA	250
Program part itself contains more than 100,000 digits, the maximum total program size. Try to make part of program unit a subroutine(s) and make subroutine overlayable through SEGMENT Card.	
EXCESSIVE USE OF STATEMENT NUMBERS IN COLUMNS 1-5 CAUSES TABLE OVERFLOW. IF NECESSARY SUBROUTINIZE	251
Self-explanatory.	
STATEMENT NUMBER 00000 IS DUPLICATED IN COLUMNS 1-5. ONLY THE LAST APPEARANCE WILL BE REFERENCED	252
Self-explanatory.	
STATEMENT NUMBER 00000 IS REFERENCED BUT DOES NOT APPEAR IN COLUMNS 1-5	253
Self-explanatory.	
STATEMENT NUMBER 00000 IS DUPLICATED BY A FORMAT NUMBER IN COLUMNS 1-5	254
Self-explanatory.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
SEGMENT CARD: UNKNOWN PROGRAM IDENTIFIER	501
A program part identifier which appears in a SEGMENT Card must be referenced in the program.	
USE CARD: UNKNOWN PROGRAM IDENTIFIER	502
Self-explanatory.	
INITIALIZE CARD: UNKNOWN PROGRAM IDENTIFIER	504
Self-explanatory.	
FILE CARD: INVALID FILE NUMBER MUST BE BETWEEN 0 AND 19	510
Self-explanatory.	
FILE CARD: INVALID UNIT DESIGNATOR	511
Self-explanatory.	
FILE CARD: INVALID FILE ATTRIBUTE	512
Self-explanatory.	
DUPLICATE FILE DECLARATIONS: ONLY FIRST WILL BE USED	513
More than one FILE Card has been coded specifying same unit number.	
STACK CARD: INVALID STACK SIZE	520
STACK Card is coded incorrectly.	
CONTROL CARD: INDETERMINATE CARD TYPE	530
Self-explanatory.	

FORTRAN ERROR MESSAGES AND FLAGS

Error Message	Error Number
CONTROL CARD: IDENTIFIER TOO LONG, 11 CHARACTER MAXIMUM	531

Self-explanatory.

HANDLECODE: OP CODE nn FOUND IN _____ BUT NOT IMPLEMENTED	540

Floating point hardware or SEA (Search) is not in machine.

TREEANALYSIS: DATA FOR ______ EXCEEDS 100,000 DIGIT LIMIT	541

Data for the listed program part(s) exceeds the 100,000 digit program size limit. Program should be segmented.

HANDLECODE: LABEL NOT FOUND	542

Self-explanatory.

TREEANALYSIS: CODE FOR ______ EXCEEDS 300,000 DIGIT LIMIT	543

Self-explanatory. Theoretically, <u>code</u> may reside above the 100,000 limit. This is true only if that code is not self modifying.

Flag Message	Flag Number
AN EXECUTABLE STATEMENT MAY NOT APPEAR IN A BLOCK DATA SUBPROGRAM	29

Self-explanatory.

HOLLERITH CONSTANT TOO LONG	35

Hollerith constant used in FORMAT statement may not exceed 132 characters. Hollerith constant used in

Flag Message	Flag Number
other statements should not exceed ALPHA size for program.	
INTEGER PRECISION MUST BE AT LEAST 5 TO ACCOMMODATE AN ASSIGNED GO TO	68
Self-explanatory. If integer precision is not at least 5, address error may occur when Assigned GO TO is executed.	
A GO TO MUST BE FOLLOWED BY A STATEMENT NUMBER, INTEGER ID, OR LIST	96
Self-explanatory.	
IMPROPER STATEMENT NUMBER IN A COMPUTED GO TO LIST	97
Statement number must be integer of one to five digits.	
STATEMENT NUMBERS IN A COMPUTED GO TO MUST BE DELIMITED BY A COMMA OR RIGHT PARENTHESIS	129
Self-explanatory.	
A COMPUTED GO TO LIST MUST BE FOLLOWED BY A COMMA	130
A comma must follow the right parenthesis of the statement list and must precede the control variable.	
A LOGICAL IF MAY NOT CONTAIN A LOGICAL IF	131
A Logical IF may not contain an IF of any type.	
ANYTHING AFTER THIRD LABEL IN AN IF STATEMENT IS INVALID	157
Self-explanatory.	

FORTRAN ERROR MESSAGES AND FLAGS

<u>Flag Message</u>	Flag <u>Number</u>
NOT ENOUGH STATEMENT NUMBERS IN AN IF STATEMENT	158
IF statement must contain three statement numbers.	
COMPILER ERROR	212
LOGICAL QUANTITY . (DOT) NOT ALLOWED IN AN EXPRESSION	219
a. An illegal mixing of modes causes this message.	
b. If a period has been incorrectly called in an expression, this message may occur.	
MISSING TERMINATION STATEMENT NUMBER IN DO STATEMENT	220
Self-explanatory.	
EXCEEDED MAXIMUM NESTING OF DO STATEMENTS	502
DO statement may be nested a maximum of nine deep.	
IMPROPER DO CONTROL VARIABLE	504
Control variable must be an integer variable.	
MISSING EQUAL SIGN IN DO STATEMENT	510
Self-explanatory.	
COMMA MUST FOLLOW INITIALIZATION PARAMETER IN DO STATEMENT	512
Self-explanatory.	
ILLEGAL INITIAL PARAMETER IN DO STATEMENT	513
Self-explanatory.	

FORTRAN ERROR MESSAGES AND FLAGS

Flag Message	Flag Number
ILLEGAL TERMINATION PARAMETER IN DO STATEMENT	530
Self-explanatory.	
ILLEGAL INCREMENT PARAMETER IN DO STATEMENT	531
Self-explanatory.	

INDEX

INDEX (cont)

INDEX (cont)

BURROUGHS CORPORATION
DATA PROCESSING PUBLICATIONS
REMARKS FORM

TITLE: B 2500 and B 3500 SYSTEMS
FORTRAN Reference Manual

FORM: 1030376
DATE: 8-71

CHECK TYPE OF SUGGESTION:

☐ ADDITION ☐ DELETION ☐ REVISION ☐ ERROR

GENERAL COMMENTS AND/OR SUGGESTIONS FOR IMPROVEMENT OF PUBLICATION:

FROM: NAME ______________ DATE ______________
TITLE ______________
COMPANY ______________
ADDRESS ______________
____

cut alo dotted line

STAPLE

FOLD DOWN SECOND FOLD DOWN

Postage Will Be Paid by Addressee

No Postage Stamp Necessary If Mailed in the United States

BUSINESS REPLY MAIL
First Class Permit No. 817, Detroit, Mich. 48232

Burroughs Corporation
6071 Second Avenue
Detroit, Michigan 48232

attn: Sales Technical Services
Systems Documentation

FOLD UP FIRST FOLD UP

Wherever There's Business There's **Burroughs**

1030376 8-71 Printed in U. S. America