Cromemco FORTRAN IV

Instruction Manual

Twenty Dollars

CROMEMCO

FORTRAN IV

Instruction Manual

CROMEMCO, INC.
28Ø Bernardo Avenue
Mountain View, CA 94Ø43

Part No. Ø23-ØØ38 March 1979

This manual was produced on a Cromemco model 3355 printer using the Cromemco Word Processing System.

CROMEMCO FORTRAN
Reference Manual

Contents

PART 2
FORTRAN USER'S MANUAL

SECTION 1

INTRODUCTION

FORTRAN is a universal, problem oriented programming language designed to simplify the preparation and check-out of computer programs. The name of the language - FORTRAN - is an acronym for FORmula TRANslator.

The syntactical rules for using the language are rigorous and require the programmer to define fully the characteristics of a problem in a series of precise statements. These statements, called the source program, are translated by a system program called the FORTRAN processor into an object program in the machine language of the computer on which the program is to be executed.

This manual defines the FORTRAN source language for the Cromemco Z-80 microcomputers. This language includes the American National Standard FORTRAN language as described in ANSI document X3.9-1966, approved on March 7, 1966, plus a number of language extensions and some restrictions. These language extensions and restrictions are described in the text of this document and are listed in Appendix A.

Examples are included throughout the manual to illustrate the construction and use of the language elements. The programmer should be familiar with all aspects of the language to take full advantage of its capabilities.

Section 2 describes the form and components of a Cromemco FORTRAN source program. Sections 3 and 4 define data types and their expressional relationships. Sections 5 through 9 describe the proper construction and usage of the various statement classes.

SECTION 2

FORTRAN PROGRAM FORM

Cromemco FORTRAN source programs consist of one program unit called the Main Program followed by any number of program units called Subprograms. A discussion of subprogram types and methods of writing and using them can be found in Section 9 of this manual.

Programs and program units are constructed of an ordered set of statements which precisely describe procedures for solving problems and which also define information to be used by the FORTRAN processor during compilation of the object program. Each statement is written using the FORTRAN character set and following a prescribed line format.

2.1 FORTRAN CHARACTER SET

To simplify reference and explanation, the FORTRAN character set is divided into four subsets and a name is given to each.

2.1.1 LETTERS

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,$

2.1.2 UPPER CASE AND LOWER CASE

No distinction is made between upper and lower case letters. However, for clarity and legibility, exclusive use of upper case letters is recommended.

2.1.3 DIGITS

Ø,1,2,3,4,5,6,7,8,9

2.1.4 HEXADECIMAL DIGITS

Strings of digits representing numeric quantities are normally interpreted as decimal numbers. However, in certain statements, the interpretation is in the hexadecimal number system in which case the letters A, B, C, D, E, F may also be used as hexadecimal digits. Hexadecimal usage is defined in the descriptions of statements in which such notation is allowed.

2.1.5 ALPHANUMERICS

A sub-set of characters made up of all letters and all digits.

2.1.6 SPECIAL CHARACTERS

	Blank
=	Equality Sign
+	Plus Sign
-	Minus Sign
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point

NOTES:

1. The following special characters are classified as Arithmetic Operators and are significant in the unambiguous statement of arithmetic expressions.

   + Addition or Positive Value
   - Subtraction or Negative Value
   * Multiplication
   / Division
   ** Exponentiation

2. The other special characters have specific application in the syntactical expression of the FORTRAN language and in the construction of FORTRAN statements.

3. Any printable character may appear in a Hollerith or Literal field.

2.2 FORTRAN LINE FORMAT

A FORTRAN program line consists of 80 character positions or columns, numbered 1 through 80. A line is divided into four fields.

1. Statement Label (or Number) field-
   Columns 1 through 5 (See definition of statement labels).

2. Continuation character field-
   Column 6

3. Statement field-
   Columns 7 through 72

4. Identification field-
   Columns 73 through 80

The identification field is available for any purpose the FORTRAN programmer may desire and is ignored by the FORTRAN processor.

The lines of a FORTRAN statement are placed in Columns 1 through 72 formatted according to line types. The four line types, their definitions, and column formats are:

1. Comment line -- used for source program annotation at the convenience of the programmer.

   1. Column 1 contains the letter C.

   2. Columns 2 - 72 are used in any desired format to express the comment or they may be left blank.

   3. A comment line may be followed only by an initial line, an END line, or another comment line.

4. Comment lines have no effect on the object program and are ignored by the FORTRAN processor except for display purposes in the listing of the program.

   Example:

```
C     COMMENT LINES ARE INDICATED BY THE
C        CHARACTER C IN COLUMN 1.
C  THESE ARE COMMENT LINES
```

2. END line -- the last line of a program unit.

   1. Columns 1-5 may contain a statement label.

   2. Column 6 must contain a zero or blank.

   3. Columns 7-72 contain one of the characters E, N or D, in that order, preceded by, separated by, or followed by blank characters.

   4. Each FORTRAN program unit must have an END line as its last line to inform the Processor that it is at the physical end of the program unit.

   5. An END line may follow any other type line.

      Example:

```
      END
```

3. Initial Line -- the first or only line of each statement.

   1. Columns 1-5 may contain a statement label to identify the statement.

   2. Column 6 must contain zero or blank.

   3. Columns 7-72 contain all or part of the statement.

   4. An initial line may begin anywhere within the statement field.

Example:

```
C THE STATEMENT BELOW CONSISTS
C    OF AN INITIAL LINE
C
        A= .5*SQRT(3-2.*C)
```

4. Continuation Line -- used when additional lines of coding are required to complete a statement originating with an initial line.

   1. Columns 1-5 are ignored, unless Column 1 contains a C.

   2. If Column 1 contains a C, it is a comment line.

   3. Column 6 must contain a character other than zero or blank.

   4. Columns 7-72 contain the continuation of the statement.

   5. There may be as many continuation lines as needed to complete the statement.

   Example:

```
C THE STATEMENTS BELOW ARE AN INITIAL LINE
C      AND 2 CONTINUATION LINES
C
 63  BETA(1,2) =
    1     A6BAR**7-(BETA(2,2)-A5BAR*5Ø
    2     +SQRT (BETA(2,1)))
```

5. Statement Labels -- A statement label may be placed in columns 1-5 of a FORTRAN statement initial line and is used for reference purposes in other statements.

   The following considerations govern the use of statement labels:

   1. The label is an integer from 1 to 99999.

   2. Leading zeros and blanks are not significant. Trailing blanks are not significant.

   3. A label must be unique within a program unit.

4. A label on a continuation line is ignored by the FORTRAN Processor.

Example:

```
C  EXAMPLES OF STATEMENT LABELS
C
  1
  101
99999
 763
```

2.3 STATEMENTS

Individual statements deal with specific aspects of a procedure described in a program unit and are classified as either executable or non-executable.

Executable statements specify actions and cause the FORTRAN Processor to generate object program instructions. There are three types of executable statements:

1. Replacement statements.

2. Control statements.

3. Input/Output statements.

Non-executable statements describe to the processor the nature and arrangement of data and provide information about input/output formats and data initialization to the object during program loading and execution. There are five types of non-executable statements:

1. Specification statements.

2. DATA Initialization statements.

3. FORMAT statements.

4. FUNCTION defining statements.

5. Subprogram statements.

The proper usage and construction of the various types of statements are described in Sections 5 through 9.

2.4 Order of Statements in a Program Unit

The statements in a program unit must appear in the following order. Some statements are required, some are optional. Those that are required are specified.

1. PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA

   One of these is required to begin each program unit; only one may be used.

2. LOGICAL, LOGICAL*1, LOGICAL*2, BYTE, INTEGER, INTEGER*1, INTEGER*2, REAL, REAL*4, REAL*8, DOUBLE PRECISION, EXTERNAL, DIMENSION

   These may appear in any order relative to each other.

3. COMMON

4. EQUIVALENCE

5. DATA

6. Statement Functions

7. ASSIGN, BACKSPACE, CALL, CONTINUE, DO, ENDFILE, GO TO, IF, PAUSE, READ, Replacement Statements, RETURN, REWIND, STOP, WRITE

   These statements may appear in any order relative to each other, subject to the rules specified in this manual.

8. END

   This statement is required to end each program unit.

The FORMAT statement may appear anywhere between the PROGRAM, SUBROUTINE, or FUNCTION statement and the END statement.

SECTION 3

DATA REPRESENTATION / STORAGE FORMAT

All data used in FORTRAN programs must be identified by name and data type. This section defines the form, representation, and usage of the permissable kinds of data.

3.1 DATA NAMES AND TYPES

3.1.1 DATA NAMES

Data are represented in FORTRAN expressions by the use of symbolic names. A symbolic name is a unique string of 1 to 6 alphanumeric characters. The first character must be a letter.

NOTE

System variable names and runtime subprogram names are distinguished from other variable names in that they begin with the dollar sign character ($). It is strongly recommended that in order to avoid the possibility of conflicts, symbolic names used by the programmer begin with any letter other than "$".

1. Constant - An explicitly stated datum.
2. Variable - A symbolically identified datum.
3. Array - An ordered set of data in 1, 2 or 3 dimensions.
4. Array Element - One member of the set of data of an array.

3.1.2 DATA TYPES

Data are classified into four data types:

LOGICAL
INTEGER
REAL
DOUBLE PRECISION

The data type is specified by one of the following methods.

1. Implicit Typing.

   The first letter of the symbolic name specifies INTEGER or REAL type. Unless explicitly typed (see paragraph 2), symbolic names beginning with I, J, K, L, M, or N represent INTEGER data, and symbolic names beginning with letters other than I, J, K, L, M, or N represent REAL data.

   Examples:

Integer Data Names	Real Data Names
ITEM	ALPHA
J(1,1)	BETA(2,2)

2. Explicit Typing.

   Data may be specified to be of a particular type, overriding the type implied by the first character of the symbolic name.

   The LOGICAL statement types data as LOGICAL.

   The INTEGER statement types data as INTEGER.

   The REAL statement types data as REAL.

   The DOUBLE PRECISION statement types data as DOUBLE PRECISION.

   (See Section 6.3 for a detailed discussion of type statements.)

The description, range, and precision of each type of data follows.

1. LOGICAL - One byte representation of the logical values TRUE and FALSE. TRUE is represented by a non-zero value. FALSE is represented by a zero value.

   A symbolic name with type LOGICAL may also be used to store one byte integer values in two's - complement notation.

   RANGE:
   -128 to +127
   PRECISION:
   3 digits

2. INTEGER - Two byte representation of integer numbers in two's - complement notation.

   RANGE:
   -32768 to +32767
   (-2**15 to 2**15-1)
   PRECISION:
   5 digits

3. REAL - Four byte floating point approximations of real numbers.

   APPROXIMATE MAGNITUDE RANGE:
   1Ø**-38 to 1Ø**38
   (2**-127 to 2**127)
   PRECISION:
   7+ significant digits

4. DOUBLE PRECISION - Eight byte floating point approximations of real numbers.

   APPROXIMATE MAGNITUDE RANGE:
   same as REAL
   PRECISION:
   16+ significant digits

The following types are allowed for readability and compatibility with other FORTRAN implementations.

BYTE INTEGER*1 LOGICAL*1	Equivalent to LOGICAL
INTEGER*2 LOGICAL*2	Equivalent to INTEGER

REAL*4	Equivalent to REAL
REAL*8	Equivalent to DOUBLE PRECISION

Hollerith, or Alphanumeric, data is a string of any number of characters from the computer's character set. All characters, including blanks, are significant. Hollerith data may be stored in any type data item and require one byte of storage for each character in the data string.

3.2 CONSTANTS

FORTRAN constants are specified by stating their actual value.

A Signed Constant is a constant preceded by a plus sign (+) or a minus sign (-). The plus sign is optional in front of a constant with a positive value. The minus sign must be used to specify a constant with a negative value.

An Unsigned Constant is a constant with no preceding plus sign or minus sign.

The data type of a constant is implicit in its form.

The possible forms of constants are defined in Table 3-1, on the following page.

Table 3-1.

FORMS OF CONSTANTS

TYPE	FORM AND RULES OF USE	EXAMPLES
INTEGER	1. 1 to 5 decimal digits interpreted as a decimal number.	-763 1 +ØØ672
	2. A preceding plus (+) or minus (-) sign is optional.	-32768 +32767
	3. No decimal point (.) or comma (,) is allowed.	
	4. Value range: -32768 through +32767 (i.e., -2**15 through 2**15-1).	
	5. INTEGER constants may be assigned to LOGICAL variables, and the permissible range is -128 to +127.	
REAL	1. A decimal number with precision to 7 digits and represented in one of the following forms: + or - .f + or - i.f + or - i. E + or - e + or - .f E + or - e + or - i.f E + or - e where i, f, and e are each strings representing integer, fraction, and exponent respectively.	345. -.345678 +345.678 +.3E3 -73.E4

2. Plus (+) and minus (-) characters are optional.

3. In the form shown in 1 above, if r represents any of the forms preceding E+ or -e (i.e., rE+ or -e), the value of the constant is interpreted as r times 1Ø**e, where -38<=e<=38.

4. If the constant preceding E+ or -e contains more significant digits than the precision for real data allows, truncation occurs, and only the most significant digits in the range will be represented.

DOUBLE PRECISION	A decimal number with precision to 16 digits. All forms and rules are identical to those for REAL constants, except D is used in place of E. Note that a REAL constant is assumed single precision unless it contains a "D" exponent.	+345.678 +.3D3
LOGICAL	.TRUE. generates a non-zero byte (hexadecimal FF) .FALSE. generates a byte in which all bits are Ø. When LOGICAL constants are used in arithmetic expressions, .TRUE. has a value of -1, and .FALSE. has a value of Ø.	.TRUE. .FALSE.

LITERAL

In the literal form, any number of characters may be enclosed by single quotation marks. This form is as follows:

```
'X1X2X3...Xn'
```

where each Xi is any character other than '. Two quotation marks in succession may be used to represent the quotation mark character within the string, i.e., if X2 is to be the quotation mark character, the string appears as the following:

```
'X1''X3...Xn'
```

A literal constant is left justified in its storage unit.

HEXADECIMAL

1. The letter Z or X followed by a single quote, up to 16 hexadecimal digits (Ø-9 and A-F) and a single quote is recognized as a hexadecimal value.

```
Z'12'
X'AB1F'
Z'FFFF'
X'1F'
```

2. A hexadecimal constant is right justified in its storage unit.

3.3 VARIABLES

A variable datum is identified and referenced by the use of a symbolic name that is 1-6 characters long (see 3.1.1).

Each variable name must be unique within a program unit, and is assigned to the appropriate amount of storage when FORTRAN first encounters the variable name in a type statement or some other FORTRAN statement.

A variable has a data type, either LOGICAL, INTEGER, REAL or DOUBLE PRECISION, which is specified implicitly by the first letter of its symbolic name or explicitly by a type statement (see 3.1.2).

Examples

I, A, RATE, RHO, A1$C

3.3.1 Variable Data Values

The intial values of data variables are undefined unless the programmer has specified initial values using the DATA statement (see Section 6.8).

Variables receive numeric values through initial DATA statements, the execution of replacement statements (Section 5), or the execution of input/output statements (Section 8).

Variables receive Hollerith or literal data through initial DATA statements or the execution of input/output statements. Any type variable may be used to store Hollerith or literal data (see Section 3.6).

3.4 ARRAYS AND ARRAY ELEMENTS

An array is an ordered set of data characterized by the property of dimension. An array may have 1, 2 or 3 dimensions and is identified and typed by a symbolic name in the same manner as a variable except that an array name must be so declared by an array declarator. Complete discussions of the array declarators appear in Section 6 of this manual. An array declarator also specifies the number of dimensions and size of each dimension of the array. An array element is one member of the data set that makes up an array. Reference to an array element in a FORTRAN statement is made by appending a subscript to the array name. Each array element has the same data type as the array. The term array element is synonymous with the term subscripted variable used in some FORTRAN texts and reference manuals.

An initial value may be assigned to any array element by a DATA statement or its value may be derived and defined during program execution.

The value of an array element is initially undefined if no value is assigned in a DATA statement.

3.5 SUBSCRIPTS

A subscript follows an array name to identify uniquely an array element. In use, a subscript in a FORTRAN statement takes on the same representational meaning as a subscript in familiar algebraic notation.

Rules that govern the use of subscripts are as follows:

1. A subscript contains 1, 2 or 3 subscript expressions (see Rule 4) enclosed in parentheses.

2. If there are two or three subscript expressions within the parentheses, they must be separated by commas.

3. The number of subscript expressions must be the same as the specified dimensionality of

the array declarator except in EQUIVALENCE statements (Section 6).

4. A subscript expression is written in one of the following forms:

K	C*V
V	C*V+K
V+K	C*V-K
V-K	

where C and K are INTEGER constants and V is an INTEGER variable name (see Section 4.1.3 for a discussion of expression evaluation).

5. Subscripts themselves may not be subscripted.

6. The value of a subscript must always be greater than or equal to 1.

Examples of valid subscript expressions:

X(2*J-3,7) A(I,J,K) I(2Ø) C(L-2) Y(I)

3.6 DATA STORAGE ALLOCATION

Allocation of storage for FORTRAN data is made in numbers of storage units. A storage unit is the memory space required to store one REAL data value (4 bytes).

Table 3-2 defines the word formats of the four data types.

Hexadecimal data may be associated (via a DATA statement) with any type data. Its storage allocation is the same as the associated datum.

Hollerith or literal data may be associated with any data type by use of DATA initialization statements (Section 6).

Up to eight Hollerith characters may be associated with DOUBLE PRECISION type storage, up to four with REAL, up to two with INTEGER and one with LOGICAL type storage.

TABLE 3-2. STORAGE ALLOCATION BY DATA TYPES

TYPE	ALLOCATION
LOGICAL	1 byte (1/4 storage unit) Zero (false) or non-zero (true) A non-zero valued byte indicates true (the logical constant .TRUE. is represented by the hexadecimal value FF). A zero valued byte indicates false. When used as an arithmetic value, a LOGICAL datum is treated as an integer in the range -128 to +127.
INTEGER	2 bytes (1/2 storage unit) Sign and Binary Value Negative numbers are the 2's complement of positive representations.
REAL	4 bytes (1 storage unit) Characteristic Sign and Mantissa Mantissa Mantissa The first byte is the characteristic expressed in excess-8Ø hex (= 2ØØ octal = 128 decimal) notation; i.e., a value of 8Ø hex corresponds to a binary exponent of Ø. Values less than 8Ø hex correspond to negative exponents, and values greater than 8Ø hex correspond to positive exponents. By definition, if the characteristic is zero, the entire number is zero.

The next three bytes constitute the mantissa. The mantissa is always normalized such that the high order bit is one, eliminating the need to actually save that bit. The high bit is used instead to indicate the sign of the number. A one indicates a negative number, and zero a positive number. The mantissa is assumed to be a binary fraction whose binary point is to the left of the mantissa.

DOUBLE PRECISION

8 bytes (2 storage units)

The internal form of DOUBLE PRECISION data is identical with that of REAL data except DOUBLE PRECISION uses 4 extra bytes for the mantissa.

NOTE

The data bytes for INTEGER, REAL and DOUBLE PRECISION data types are placed into memory such that the least significant byte of data occupies the byte of memory with the lowest address, the next most significant byte occupies the byte of memory with the next greater address, and so on. Thus, when memory is inspected, the value of the data is read from right to left, byte by byte.

SECTION 4

EXPRESSIONS

A FORTRAN expression is composed of a single operand or a string of operands connected by operators. Three expression types - arithmetic, relational, and logical - are provided by FORTRAN. The operands, operators, forms, and rules of use for all types are described in the following paragraphs.

4.1 ARITHMETIC EXPRESSIONS

Arithmetic expressions specify numeric computations and are used to produce a single numeric value for use in a FORTRAN statement.

An arithmetic expression consists of a single arithmetic operand, or two or more arithmetic operands connected by arithmetic operators.

4.1.1 Arithmetic Operands

An arithmetic operand must be one of the following:

Unsigned constant - LOGICAL, INTEGER, REAL, or DOUBLE PRECISION.

Variable - LOGICAL, INTEGER, REAL, or DOUBLE PRECISION.

Array element - LOGICAL, INTEGER, REAL, or DOUBLE PRECISION.

Function reference - LOGICAL, INTEGER, REAL, or DOUBLE PRECISION.

Unsigned arithmetic expression.

Signed arithmetic expression enclosed in parentheses.

An unsigned arithmetic expression is an arithmetic expression whose first character is not a plus sign (+) or a minus sign (-). A signed arithmetic

expression is an arithmetic expression whose first character is a plus sign (+) or a minus sign (-).

4.1.2 Arithmetic Operators

The arithmetic operators are:

**	Exponentiation
/	Division
*	Multiplication
+	Addition
-	Subtraction

The arithmetic operators have a precedence that determines the order in which the operations will be performed when an expression includes more than one operator and there are no parentheses that specify the order. From highest to lowest, this precedence is:

1. Exponentiation
2. Division and Multiplication
3. Addition and Subtraction

4.1.3 Form of Arithmetic Expressions

An arithmetic expression must have one of the following forms:

1. A single arithmetic operand appearing with no operators.

   Examples:

   ```
   375
   375.1
   RATE
   SCALE(I)
   SQRT(A+B)
   ```

2. An arithmetic operand preceded by a plus (+) or a minus (-) sign.
   This is called a signed arithmetic expression. The plus sign (+) has no effect on the value of the expression. The minus sign (-) negates the value of the expression.

Examples:

```
-375
+375.1
-RATE
+SCALE(I)
-SQRT(A+B)
```

3. An arithmetic expression enclosed in parentheses.
Enclosing an arithmetic expression in parentheses represents the numeric value of the expression.

Examples:

```
(-RATE)
-(SCALE (I))
(SQRT (A+B))
```

4. Two arithmetic operands connected with one of the arithmetic operators.
The operand to the right of the operator must be an unsigned operand, i.e., its first character must not be a plus (+) or a minus (-) sign.

Examples:

```
B**2
B**2 - (4*A*C)
SQRT (B**2 - 4*A*C)
1.7E-2 ** (X+5.Ø)
-(B(I,J) + ALOG (A+B(I,J)))
```

5. An arithmetic expression containing nested parenthesized arithmetic expressions.
The number of left paretheses must be the same as the number of right parentheses.

Example:

```
A * (Z-((Y+X)/T))**J
```

where

Y+X is the innermost element, (Y+X)/T is the next innermost, and Z-((Y+X)/T) is the next.

4.1.4 Evaluation of Arithmetic Expressions

Arithmetic expressions are evaluated according to the following rules:

1. An evaluated expression will be of type LOGICAL, INTEGER, REAL, or DOUBLE PRECISION. The type is determined by the data types of the operands of the expression. If the operands are not all of the same type, the type of the expression is determined by the operand having the highest type. The data type hierarchy, highest to lowest, is:

   DOUBLE PRECISION
   REAL
   INTEGER
   LOGICAL

2. Parenthesized operands are evaluated first. If parenthesized operands are nested, the innermost operand is evaluated, followed by the next innermost, until the expression has been evaluated.

3. Within parentheses and wherever parentheses are not used to specify the order of evaluation, the hierarchy of operations, from highest to lowest, is:

   1. FUNCTION evaluation
   2. Exponentiation
   3. Multiplication and Division
   4. Addition and Subtraction

Example:

The expression

```
A*(Z-((Y+R)/T))**J+VAL
```

is evaluated as follows:

```
v1 = Y+R
v2 = v1/T
v3 = Z-v2
v4 = v3**J
v5 = A*v4
v6 = v5+VAL
```

where the vi are intermediate values.

4. Wherever operations of equal hierarchy are involved, evaluation proceeds from left to right.

Example:

expression	evaluated as
W*X/Y*Z	((W*X)/Y)*Z
B**Z-4.*A*C	(B**Z)-((4.*A)*C)
X-Y-Z	(X-Y)-Z
X/Y/Z	(X/Y)/Z
-X**3	-(X**3)

5. The expression

```
X**Y**Z
```

is not allowed. It must be written as:

```
(X**Y)**Z
```

or

```
X**(Y**Z)
```

6. Use of an array element reference requires the evaluation of its subscript. Subscript expressions are evaluated as arithmetic expressions.

4.2 RELATIONAL EXPRESSIONS

A relational expression consists of two arithmetic expressions connected by a relational operator. The value of a relational expression is a logical .TRUE. or a logical .FALSE.

4.2.1 Relational Operators

The relational operators are:

.LT.	Less than
.LE.	Less than or Equal to
.EQ.	Equal to
.NE.	Not Equal to
.GT.	Greater than
.GE.	Greater than or Equal to

4.2.2 Form of Relational Expressions

The form of a relational expression is:

```
el r e2
```

where el and e2 are arithmetic expressions and r is a relational operator.

Examples:

```
A.EQ.B
(A**J).GT.(Z*(RHO*TAU-A))
```

4.2.3 Evaluation of Relational Expressions

The value of a relational expression is logical .TRUE. if the relation defined by the operator is true. Otherwise the value is .FALSE.

4.3 LOGICAL EXPRESSIONS

A logical expression consists of a single logical operand, or two or more logical operands connected by logical operators.

4.3.1 Logical Operands

A logical operand must be one of the following:

Logical Constant
Logical Variable
Logical Array Element
Logical Function Reference (see 9.4)
Relational Expression
Logical Expression

Additionally, logical operators may be used to perform bitwise logical operations on the following:

Integer Constant
Integer Variable
One or Two Byte Hollerith Constant
One or Two Byte Literal Constant
One to Four Digit Hexadecimal Constant

4.3.2 Logical Operators

The logical operators are:

.NOT.	Logical Negation
.AND.	Logical Product, or Conjunction
.OR.	Logical Sum, or Inclusive Disjunction
.XOR.	Logical Exclusive Disjunction

The logical operators have a precedence that determines the order in which the operations will be performed when an expression includes more than one operator and there are no parentheses that specify the order. From highest to lowest, this precedence is:

1. .NOT.
2. .AND.
3. .OR., .XOR.

The definitions of the operators follow. Ø and 1 represent bit values.

B	.NOT.B
Ø	1
1	Ø

B1	B2	B1.AND.B2	B1.OR.B2	B1.XOR.B2
Ø	Ø	Ø	Ø	Ø
Ø	1	Ø	1	1
1	Ø	Ø	1	1
1	1	1	1	Ø

Since logical operators operate on logical operands bit by bit, and since a logical expression is defined to be false if its bit value is zero and true if its bit value is non-zero, the following tables are equivalent to the preceding tables.

E	.NOT.E
false	true
true	false

E1	E2	E1.AND.E2
false	false	false
false	true	false
true	false	false
true	true	true

E1	E2	E1.OR.E2
false	false	false
false	true	true
true	false	true
true	true	true

E1	E2	E1.XOR.E2
false	false	false
false	true	true
true	false	true
true	true	false

Examples:

```
If      L1 = Ø1101100
        L2 = 11001001

Then
   .NOT.L1 = 10010011
L1.AND.L2 = Ø1001000
 L1.OR.L2 = 11101101
L1.XOR.L2 = 10100101
```

4.3.3 Form of Logical Expressions

A logical expression must have one of the following forms:

1. A single logical operand appearing with no logical operators.

2. A logical expression enclosed in parentheses. Enclosing a logical expression in parentheses represents the logical value of the expression: true or false.

3. A logical expression preceded by the .NOT. operator.
   Expressions containing more than one operand must be enclosed in parentheses.

4. Two logical expressions connected by one logical operator.

5. Two logical expressions connected by two logical operators.
   The second operator must be the .NOT. operator.

   Examples:

```
VALID:      A.AND..NOT.B
INVALID:    A.AND..XOR.B
```

4.3.4 Evaluation of Logical Expressions

The following rules control the evaluation of a logical expression:

1. Parentheses surrounding an expression specify that the enclosed expression is to be evaluated first. When parenthesized expressions are nested within other parenthesized expressions, the innermost expression is evaluated first, then the next innermost, until the entire expression has been evaluated.

2. Non-parenthesized expressions containing operators of non-equal types and precedence are evaluated in the order specified by the hierarchy of type and precedence that follows. The hierarchy is listed from highest to lowest.

Type of Operator	Precedence Within Type
Arithmetic	Function Reference Exponentiation Multiplication and Division Addition and Subtraction
Relational	.LT., .LE., .EQ., .NE., .GT., .GE.
Logical	.NOT. .AND. .OR.,.XOR.

3. Non-parenthesized expressions containing operators of equal precedence are evaluated left to right.

Examples:

The lvi represent intermediate logical values. The avl represents an intermediate numeric value.

The expression
```
     A.OR.B.XOR.C
```
is evaluated
```
     lv1 = A.OR.B
     lv2 = lv1.XOR.C
```

The expression
```
     A.XOR.B.AND.C
```
is evaluated
```
     lv1 = B.AND.C
     lv2 = A.XOR.lv1
```

The expression
```
     (A.XOR.B).AND.C
```
is evaluated
```
     lv1 = A.XOR.B
     lv2 = lv1.AND.C
```

The expression

```
X.AND.Y.OR.B(3,2)**2.GT.Z
```

is evaluated

```
avl = B(3,2)**2
lvl = avl.GT.Z
lv2 = X.AND.Y
lv3 = lv2.OR.lvl
```

4.4 HOLLERITH, LITERAL, AND HEXADECIMAL CONSTANTS IN EXPRESSIONS

Hollerith, literal and hexadecimal constants may be used in expressions wherever integer constants may appear, subject to the following considerations:

1. With two exceptions, these constants always evaluate to an integer value and thus must be limited to a maximum length of two bytes.

2. Hollerith or literal constants longer than two bytes may be used as subroutine parameters.

3. Hollerith, literal, and hexadecimal constants in DATA statements have the following maximum lengths:

DATA TYPE	MAXIMUM LENGTH
LOGICAL	1 BYTE
INTEGER	2 BYTES
REAL	4 BYTES
DOUBLE PRECISION	8 BYTES

SECTION 5
REPLACEMENT STATEMENTS

Replacement statements define computations and are used similarly to equations in normal mathematical notation. They are of the following form:

```
v = e
```

where v is any variable or array element and e is an expression.

FORTRAN semantics defines the equality sign (=) as meaning to be replaced by rather than the normal is equivalent to. Thus, the object program instructions generated by a replacement statement will, when executed, evaluate the expression on the right of the equality sign and place that result in the storage space allocated to the variable or array element on the left of the equality sign.

The following conditions apply to replacement statements:

1. Both v and the equality sign must appear on the same line. This holds even when the statement is part of a logical IF statement (Section 7).

   Example:

```
C IN A REPLACEMENT STATEMENT THE '='
C     MUST BE IN THE INITIAL LINE.
      A(5,3) =
     1     B(7,2) + SIN(C)
```

   The line containing v= must be the intial line of the statement unless the statement is part of a logical IF statement. In that case the v= must occur no later than the end of the first line after the end of the IF.

2. If the data types of the variable, v, and the expression, e, are different, then the value determined by the expression will be converted, to conform to the typing of the variable. Table 5-1 indicates conversion considerations.

Table 5-1. Replacement By Type

Variable Types	Expression Types			
	Logical	Integer	Real	Double
Logical		d	a	a
Integer	b		a	a
Real	c	c		e
Double	c	c		

a. The REAL expression value is converted to INTEGER, truncated if necessary to conform to the range of INTEGER data. (The value of the low order byte is used for a LOGICAL variable. The value of the two low order bytes is used for an INTEGER variable.)

b. The sign is extended through the second byte.

c. The variable is assigned the real approximation of the integer value of the expression.

d. The variable is assigned the truncated value of the INTEGER expression (the low-order byte is used, regardless of sign).

e. The variable is assigned the rounded value of the REAL expression.

SECTION 6

SPECIFICATION STATEMENTS

Specification statements are non-executable, non-generative statements which define data types of variables and arrays, specify array dimensionality and size, allocate data storage or otherwise supply determinative information to the FORTRAN processor. DATA initialization statements are non-executable, but generate object program data and establish initial values for variable data.

6.1 SPECIFICATION STATEMENTS

There are six kinds of specification statements. They are as follows:

Type, EXTERNAL, and DIMENSION statements
COMMON statements
EQUIVALENCE statements
DATA initialization statements

All specification statements are grouped at the beginning of a program unit and must be ordered as they appear above. Specification statements may be preceded only by a FUNCTION, SUBROUTINE, PROGRAM or BLOCK DATA statement. All specification statements must precede statement functions and the first executable statement.

6.2 ARRAY DECLARATORS

Three kinds of specification statements may specify array declarators. These statements are the following:

Type statements
DIMENSION statements
COMMON statements

Of these, DIMENSION statements have the declaration of arrays as their sole function. The other two serve dual purposes. These statements are defined in subparagraphs 6.3, 6.5 and 6.6.

An array declarator is used to specify the name, dimensionality and size of an array. An array may be declared only once in a program unit.

An array declarator has one of the following forms:

```
ui (k)
ui (k1,k2)
ui (k1,k2,k3)
```

where ui is the name of the array, called the declarator name, and k1, k2, and k3 are integer constants.

Array storage allocation is established upon appearance of the array declarator. Such storage is allocated linearly by the FORTRAN processor where the order of ascendancy is determined by the first subscript varying most rapidly and the last subscript varying least rapidly.

For example, if the array declarator AMAT(3,2,2) appears, storage is allocated for the 12 elements in the following order:

AMAT(1,1,1), AMAT(2,1,1), AMAT(3,1,1), AMAT(1,2,1), AMAT(2,2,1), AMAT(3,2,1), AMAT(1,1,2), AMAT(2,1,2), AMAT(3,1,2), AMAT(1,2,2), AMAT(2,2,2), AMAT(3,2,2)

6.3 TYPE STATEMENTS

A type statement is used to override or confirm the implicit data type of a variable, array, or FUNCTION name.
(If a variable, array, or FUNCTION name does not appear in a Type statement, the name is implicitly typed INTEGER or REAL according to the rules specified in 3.1.2.) A type statement may also specify an array declarator.

A type statement has the form

```
t v1, v2,...,vn
```

where t must be one of the following:

```
BYTE
LOGICAL, LOGICAL*1, LOGICAL*2
INTEGER, INTEGER*1, INTEGER*2
REAL, REAL*4, REAL*8
DOUBLE PRECISION
```

and each vi must be one of the following:

```
variable name
array name
array declarator
FUNCTION name
```

LOGICAL, INTEGER, REAL, and DOUBLE PRECISION conform to standard FORTRAN. The other terms are allowed for readability and compatibility with other FORTRAN implementations, and are equivalent to one of the standard types.

TYPE	EQUIVALENT TYPE
BYTE LOGICAL*1 INTEGER*1	LOGICAL
LOGICAL*2 INTEGER*2	INTEGER
REAL*4	REAL
REAL*8	DOUBLE PRECISION

Example:

```
REAL AMAT(3,3,5),BX,IETA,KLPH
```

NOTE

1. AMAT and BX are redundantly typed.
2. IETA and KLPH are explicitly typed REAL.
3. AMAT(3,3,5) is a constant array declarator specifying an array of 45 elements.

Example:

```
INTEGER M1, HT, JMP(15), FL
```

NOTE

M1 is redundantly typed here. Typing of HT and FL by the pre-defined convention is overridden by their appearance in the INTEGER statement. JMP(15) is a constant array declarator. It redundantly types the array elements as Integer and communicates to the processor the storage requirements and dimensionality of the array.

Example:

```
LOGICAL L1, TEMP
```

NOTE

All variables, arrays, or FUNCTIONs required to be typed LOGICAL must appear in a LOGICAL statement, since no starting letter indicates these types by the default convention.

6.4 EXTERNAL STATEMENT

The EXTERNAL statement has the following form:

```
EXTERNAL u1,u2,...,un
```

where each ui is a SUBROUTINE, BLOCK DATA or FUNCTION name. When the name of a subprogram is used as an argument in a subprogram reference, it must have appeared in a preceding EXTERNAL statement.

For example, if SUM and AFUNC are subprogram names to be used as arguments in the subroutine SUBR, the following statements would appear in the calling program unit:

```
      .
      .
      .
      EXTERNAL SUM, AFUNC
      .
      .
      .
      .
      CALL SUBR(SUM,AFUNC,X,Y)
```

The EXTERNAL statement may also be used to specify that the name of an intrinsic function (a function provided by FORTRAN) refers to some procedure that is external to the program unit. This procedure must be provided by the programmer. The reassignment of the intrinsic function name is effective only within the program unit containing the EXTERNAL statement.

A statement function name must not appear in an EXTERNAL statement.

6.5 DIMENSION STATEMENTS

A DIMENSION statement has the following form:

```
DIMENSION u1,u2,u3,...,un
```

where each ui is an array declarator.

Example:

```
DIMENSION RAT(5,5),BAR(2Ø)
```

This statement declares two arrays - the 25 element array RAT and the 2Ø element array BAR.

6.6 COMMON STATEMENTS

The COMMON statement is a non-executable storage allocating statement. It assigns variables and arrays to an area of storage called COMMON storage that is accessible to all program units in an executable program. This COMMON area is subdivided into blocks called the blank COMMON block and named COMMON blocks. The blank COMMON block is a block of COMMON storage that has no name. A named COMMON

block is a block of COMMON storage that is associated with a specific name supplied by the programmer.

The form of the COMMON statement is:

```
COMMON /C1/LIST1/C2/LIST2/.../Cn/LISTn
```

where

Ci	is either blank or a COMMON block storage name
LISTi	is a sequence of Variable names Array names Constant array declarators separated by commas.

Rules of use:

1. The COMMON block name is formed from 1-6 alphanumeric characters, the first of which must be a letter.

2. A COMMON block name must be different from all subprogram names used in the program.

3. Any COMMON block name may be omitted, leaving two consecutive slash characters (//). If the first COMMON block name is omitted, the two slashes may also be omitted.
   Omitting the block name assigns the variables and arrays in the list to the blank COMMON block.

4. The name of a COMMON block may appear more than once in the same COMMON statement, or in more than one COMMON statement within one program unit.

5. Arrays specified by an array declarator must have not been previously declared (see Section 6.2).

6. The first byte of storage assigned to a particular COMMON block in one program unit is the same as the first byte assigned to the COMMON block with the same name declared in any other program unit.

7. The size of a COMMON block is the number of storage units required to contain the

variables and arrays declared in the COMMON statements in one program unit. The size may be increased by the use of the EQUIVALENCE statement (see Section 6.7).

8. COMMON blocks with the same name need not have the same size in all program units where the name is declared.
If the lengths are different, the program unit containing the COMMON statement specifying the greatest length must be loaded first when the program is LINKed. (See the discussion of LINK in the User's Guide part of this manual.)

9. Storage within a COMMON block is assigned item by item according to the order in which the list items are declared. Subsequent declaration of the same COMMON block name within one program unit causes the first item in the newly declared list to be assigned storage following the last item in the previously declared list.

10. FORTRAN does not require that the list of variables and arrays declared with a COMMON block name be the same in each program unit that declares that COMMON block name, although the lists normally would correspond item by item and type by type.

6.6.1 USE OF COMMON BLOCK STORAGE

Data in COMMON storage may be used by any program unit in the executable program. To access data declared in a COMMON block by one program unit, another program unit must contain a COMMON statement declaring the blank COMMON block or a named COMMON block with the same name. The list of variables and arrays assigned to the COMMON block may or may not be the same in each program unit.

If the lists in two program units do correspond type by type, variable by variable, and array by array, (thus storage unit by storage unit) then changing the value of a variable or array element in one program unit is equivalent to changing the value of the corresponding variable or array element in the other program unit. The symbolic names in the two lists may be the same, for

clarity, but this is not required by FORTRAN.

Examples:

```
     COMMON /AREA/A,B,C/BDATA/X,Y,Z,
    X                         FL,ZAP(3Ø)
```

Two blocks of COMMON storage are allocated:
AREA, with storage for three variables;
BDATA, with storage for four variables and the 3Ø element array ZAP.

```
        COMMON
    X       //
    X       A1,B1
    X       /CDATA/
    X       RATE(3,3)
    X       //
    X       T2,Z3
```

A1,B1,T2,Z3 are assigned to blank COMMON, in that order;
RATE is a nine element array assigned to the COMMON block named CDATA. RATE (3,3) is an array declarator and RATE must not be declared anywhere else within the program unit.

6.7 EQUIVALENCE STATEMENTS

Use of EQUIVALENCE statements permits the sharing of the same storage unit by two or more entities. The general form of the statement is as follows:

```
EQUIVALENCE (u1),(u2),...,(un)
```

where each ui represents a sequence of two or more variables or array elements, separated by commas. Each element in the sequence is assigned the same storage unit (or portion of a storage unit) by the processor. The order in which the elements appear is not significant.

Example:

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

The variables A, B and C will share the same storage unit during object program execution.

If an array element is used in an EQUIVALENCE statement, the number of subscripts must be the same as the number of dimensions established by the array declarator, or it must be one, where the one subscript specifies the array element's number relative to the first element of the array.

Example:

If the dimensionality of an array, Z, has been declared as Z(3,3) then in an EQUIVALENCE statement Z(6) and Z(3,2) have the same meaning.

Additional Considerations:

1. The subscripts of array elements must be integer constants.

2. An element of a multi-dimensional array may be referred to by a single subscript, if desired.

3. Variables may be assigned to a COMMON block through EQUIVALENCE statements.

   Example:

```
COMMON /X/A,B,C
EQUIVALENCE (A,D)
```

   In this case, the variables A and D share the first storage unit in COMMON block X.

4. EQUIVALENCE statements can increase the size of a block indicated by a COMMON statement by adding more elements to the end of the block.

   Example:

```
DIMENSION R(2,2)
COMMON /Z/W,X,Y
EQUIVALENCE (Y,R(3))
```

   The resulting COMMON block will have the following configuration:

Variable	Storage Unit
W = R(1,1)	Ø
X = R(2,1)	1
Y = R(1,2)	2
R(2,2)	3

The COMMON block established by the COMMON statement contains 3 storage units. It is expanded to 4 storage units by the EQUIVALENCE statement.

COMMON block size may be increased only from the last element established by the COMMON statement forward; not from its first element backward.

Note that EQUIVALENCE (X,R(3)) would be invalid in the example. The COMMON statement established W as the first element in the COMMON block and an attempt to make X and R(3) equivalent would be an attempt to make R(1) the first element.

5. It is invalid to EQUIVALENCE two elements of the same array or two elements belonging to the same or different COMMON blocks.

Example:

```
      DIMENSION XTABLE (2Ø), D(5)
      COMMON A,B(4)/ZAP/C,X
         .
         .
         .
      EQUIVALENCE (XTABLE (6),A(7)
     X         B(3),XTABLE(15)),
     X         (B(3),D(5))

         .
         .
         .
```

This EQUIVALENCE statement has the following errors:

1. It attempts to EQUIVALENCE two elements of the same array, XTABLE(6) and XTABLE(15).

2. It attempts to EQUIVALENCE two elements of the same COMMON block, A(7) and B(3).

3. Since A is not an array, A(7) is an illegal reference.

4. Making B(3) equivalent to D(5) extends COMMON backwards from its defined starting point.

6.8 DATA INITIALIZATION STATEMENT

The DATA initialization statement is a non-executable statement which provides a means of compiling data values into the object program and assigning these data to variables and array elements referenced by other statements.

The statement is of the following form:

DATA list/u1,u2,...,un/,...,list/uk,uk+1,...uk+n/

where "list" represents a list of variable, array or array element names, and the ui are constants corresponding in number to the elements in the list. An exception to the one-for-one correspondence of list items to constants is that an array name (unsubscripted) may appear in the list, and as many constants as necessary to fill the array then appear in the corresponding position between slashes. Instead of ui, it is permissible to write k*ui in order to declare the same constant, ui, k times in succession. The k must be a positive integer. Dummy arguments may not appear in the list.

Names of data in COMMON storage may appear in a DATA statement list only when the DATA statement is in a BLOCK DATA subprogram.

Example:

```
      DIMENSION C(7)
      DATA A, B, C(1),C(3)
     X   /14.73,-8.1,2*7.5/
```

This specifies that

A=14.73, B=-8.1, C(1)=7.5, C(3)=7.5

The type of each constant ui must match the type of the corresponding item in the list, except that a Hollerith, literal, or hexadecimal constant may be paired with an item of any type.

When a Hollerith or literal constant is used, the number of characters in its string should be no greater than four times the number of storage units required by the corresponding item, i.e., 1

character for a LOGICAL variable, up to 2 characters for an INTEGER variable, up to 4 characters for a REAL variable, and up to 8 characters for a DOUBLE PRECISION variable.

Hollerith or literal constants are stored left justified in the item's storage. If the specified characters don't fill the item, trailing blanks are added to fill the remainder of the storage.

Hexadecimal constants are stored right justified in the item's storage. If the specified hexadecimal characters don't fill the item, leading zeros are added to fill the remainder of the storage.

The examples below illustrate many of the features of the DATA statement.

```
      DIMENSION ARY(2)
      DATA ARY, B
     1    /4HTHIS, 4H OK., 7.8/


      REAL LIT(2)
      LOGICAL LT,LF
      DIMENSION  H4(2,2),PI3(3)
      DATA A1,B1,K1,LT,LF,
     1     H4(1,1),H4(2,1),H4(1,2),H4(2,2)
     2     PI3
     3      /5.9,2.5E-4,64,.TRUE.,.FALSE.,
     4      1.75E-3,Ø.85E-1,2*75.Ø,
     5      1.,2.,3.14159/
     6     LIT(1)
     7      /'NOGO'/
```

Note that the above examples were arranged for clarity; all elements of a DATA statement may occur on the same line.

SECTION 7

CONTROL STATEMENTS

FORTRAN control statements are executable statements which affect and guide the logical flow of a FORTRAN program. The statements in this category are as follows:

1. GO TO statements:
   a. Unconditional GO TO
   b. Computed GO TO
   c. Assigned GO TO
2. ASSIGN
3. IF statements:
   a. Arithmetic IF
   b. Logical IF
4. DO
5. CONTINUE
6. STOP
7. PAUSE
8. CALL
9. RETURN

When statement labels of other statements are a part of a control statement, such statement labels must be associated with executable statements within the same program unit in which the control statement appears.

7.1 GO TO STATEMENTS

7.1.1 UNCONDITIONAL GO TO

Unconditional GO TO statements are used whenever control is to be transferred unconditionally to some other statement within the program unit.

The statement is of the following form:

```
GO TO k
```

where k must be the statement label of an executable statement in the same program unit.

Example:

```
     GO TO 376
31Ø  A(7) = V1 -A(3)
        .
        .
376  A(2) = VECT
     GO TO 31Ø
```

In these statements, statement 376 is ahead of statement 31Ø in the logical flow of the program unit of which they are a part.

7.1.2 COMPUTED GO TO

Computed GO TO statements are of the form:

```
GO TO (k1,k2,...,kn),j
```

where the ki must be statement labels of executable statements in the same program unit as the computed GO TO, and j is an integer variable, $1 \leq j \leq n$.

This statement causes transfer of control to the statement labeled kj. If $j < 1$ or $j > n$, control will be passed to the next statement following the computed GO TO.

j must not be an integer array element.

Example:

```
     J=3
     .
     .
     .
     GO TO(7, 70, 700, 7000, 70000), J
310  J=5
     GO TO 325
```

When J = 3, the computed GO TO transfers control to statement 700. Changing J to equal 5 changes the transfer to statement 70000. Making J = 0 or J = 6 would cause control to be transferred to statement 310.

7.1.3 ASSIGNED GO TO

Assigned GO TO statements are of the following form:

```
GO TO j,(k1,k2,...,kn)

  or

GO TO J
```

where J is an integer variable name and the ki must be statement labels of executable statements in the same program unit as the Assigned GO TO. This statement causes transfer of control to the statement whose label is equal to the current value of J.

Qualifications

1. The ASSIGN statement must logically precede an assigned GO TO.

2. The ASSIGN statement must assign a value to J which is equal to a statement label included in the list of k's, if the list is specified.

Example:

```
GO TO LABEL, (80,90,100)
```

Only the statement labels 80, 90 or 100 may be assigned to LABEL.

7.2 ASSIGN STATEMENT

This statement is of the following form:

```
ASSIGN j to i
```

where j must be the statement label of an executable statement in the same program unit and i is an integer variable.

This statement is used in conjunction with each assigned GO TO statement that contains the integer variable i. When the assigned GO TO is executed, control will be transferred to the statement labeled j.

Example:

```
ASSIGN 10Ø TO LABEL
.
.
.
ASSIGN 9Ø TO LABEL
GO TO LABEL, (8Ø,9Ø,1ØØ)
```

7.3 IF STATEMENT

IF statements transfer control to one of a series of statements depending upon a condition. Two types of IF statements are provided:

Arithmetic IF
Logical IF

7.3.1 ARITHMETIC IF

The arithmetic IF statement is of the form:

```
IF (e) m1,m2,m3
```

where e is an arithmetic expression and m1, m2 and m3 must be statement labels of executable statements in the same program unit as the arithmetic IF.

Evaluation of expression e determines one of three transfer possibilities:

If e is:	Transfer to:
< Ø	m1
= Ø	m2
> Ø	m3

Examples:

Statement	Expression Value	Transfer to
IF (A) 3,4,5	15	5
IF (N-1) 50,73,9	Ø	73
IF (AMTX(2,1,2))7,2,1	-256	7

7.3.2 LOGICAL IF

The logical IF statement is of the form:

```
IF (u)s
```

where u is a LOGICAL expression and s is any executable statement except a DO statement (see 7.4) or another logical IF statement. The LOGICAL expression u is evaluated as .TRUE. or .FALSE. Section 4 contains a discussion of LOGICAL expressions.

Control Conditions:

If u is FALSE, the statement s is ignored and control goes to the next statement following the logical IF statement. If, however, the expression is TRUE, then control goes to the statement s, and subsequent program control follows normal conditions.

If s is a replacement statement (v = e, Section 5), the variable and equality sign (=) must be on the same line, either immediately following IF (u) or on a separate continuation line with the rest of the line following IF (u) left blank. See examples 4 and 5.

Examples:

```
1.   IF(I.GT.2Ø) GO TO 115

2.   IF(Q.AND.R) ASSIGN 1Ø TO J

3.   IF(Z) CALL DECL(A,B,C)

4.   IF(A.OR.B.LE.PI/2)I=J

5.   IF(A.OR.B.LE.PI/2)
    X    I=J
```

7.4 DO STATEMENT

The DO statement provides a method for repetitively executing a series of statements.
The DO statement has one of the forms:

```
     DO label i=m1,m2,m3
or
     DO label i=m1,m2
```

where

label — must be the label of an executable statement, called the terminal statement of the DO loop, in the same program unit.

i — must be an INTEGER or LOGICAL variable. i is called the DO index.

m1 — must be an INTEGER constant, INTEGER variable, or LOGICAL variable. The value of m1 becomes the initial value of the DO index.

m2 — must be an INTEGER constant, INTEGER variable, or LOGICAL variable. The value of m2 is the limit controlling repetition of the DO loop.

m3 — must be an INTEGER constant, INTEGER variable, or LOGICAL variable. The value of m3 is added to m1 after each execution of the DO loop. If m3 is omitted, its value is assumed to be 1.

RULES OF USE:

1. The DO and the first comma must appear on the intial line.

2. The terminal statement must physically follow the DO statement.

3. The range of a DO loop consists of the executable statements down to and including the terminal statement.

4. The extended range of a DO loop consists of all executable statements which may be executed between the execution of the DO statement and the execution of the terminal statement. Statements within the DO loop range may be physically located such that they are not between the DO statement and its terminal statement (see Example 3).

5. The terminal statement must not be one of:

   Arithmetic IF
   GO TO
   RETURN
   PAUSE
   STOP
   another DO statement

6. The terminal statement may be a logical IF. If the value of the logical expression in the logical IF statement is .FALSE., loop processing continues. If the value of the expression is .TRUE., the statement of the logical IF is executed, then loop processing continues. The statement of the logical IF must not be one of:

   Arithmetic IF
   GO TO
   RETURN
   PAUSE
   STOP

7. The DO index must always be positive.

8. The DO index must never be changed by any statement in the range of the DO loop.

9. Control must never be transferred into the range of a DO loop from a statement that is outside the range of the DO loop.

10. DO loops may be nested.
A statement within the range of a DO loop may be a DO statement. The range of the inner DO loop must be completely included within the range of the outer DO loop.

11. The terminal statement of a nested DO loop may be the same as the terminal statement of the outer DO loop.

12. The DO loop will execute faster and require less object code if m1, m2, and m3 are INTEGER*1 variables or constants, but the maximum number of iterations is limited to 127.

Example: DO loop with constant limit and increment of 1

DO INDEX TYPE	Loop Overhead Microseconds	Bytes
INTEGER*1	24	14
INTEGER*2	35.5	19

EXECUTION OF A DO LOOP:

1. The value of m1 is assigned to i.

2. The statements in the DO loop range are executed once.

3. The value of m3 is added to i.

4. If $i \leq m2$, execution continues at 2.
If $i > m2$, control passes to the statement following the terminal statement.

Examples:

1. The following example computes

$$\sum_{i=1}^{100} A;$$ where A is a one-dimensional array

```
      DIMENSION A(100)
           .
           .
           .
      SUM=A(1)
      DO 100 I=2,100
100   SUM = SUM + A(I)
           .
           .
           .   END
```

2. The following example computes

$$Ck = \sum_{j=1}^{15} AkjBj,\ k=1,2,...,15$$

where A is a two decimal array of 15 rows and 15 columns

B is one-dimensional array with 15 elements

C is one-dimensional array with 15 elements

```
      REAL A(15,15), B(15), C(15)
      INTEGER*1 J,K
      DO 80 K=1,15
      C(K) = 0.0
      DO 80 J=1,15
80    C(K) = C(K) + A(K,J)*B(J)
```

3. The following example illustrates the concept of the extended range of a DO loop:

```
      DIMENSION A(500), B(500)
           .
           .
           .
      DO 100 I=1,500
      IF(C) 50,110,120
50    C=0.
           .
           .
           .
90    A(I) = B(I) + C
100   CONTINUE
           .
           .
           .
```

```
11Ø  C=C+.Ø5
     GO TO 9Ø
12Ø  C=C-.Ø5
     GO TO 9Ø
```

The four statements beginning with one labeled 11Ø are included within the range of the DO loop.

7.5 CONTINUE STATEMENT

CONTINUE is classified as an executable statement. However, its execution does nothing. The form of the CONTINUE statement is as follows:

```
CONTINUE
```

CONTINUE is frequently used as the terminal statement in a DO statement range when the statement which would normally be the terminal statement is one of of those which are not allowed or is only executed conditionally.

Example:

```
  DO 5 K = 1,1Ø
  .
  .
  .
  IF (C2) 5,6,6
6 CONTINUE
  .
  .
  .
  C2 = C2 + .ØØ5
5 CONTINUE
```

7.6 STOP STATEMENT

A STOP statement has one of the following forms:

```
STOP
```

or

```
STOP c
```

where c is any string of one to six characters.

When STOP is encountered during execution of the object program, the word STOP and the character c (if present) are displayed on the operator control console and execution of the program terminates.

The STOP statement, therefore, constitutes a logical end of the program.

7.7 PAUSE STATEMENT

A PAUSE statement has one of the following forms:

```
PAUSE

or

PAUSE C
```

where c is any string of up to six characters.

When PAUSE is encountered during execution of the object program, the word PAUSE and the characters c (if present) are displayed on the operator control console and execution of the program temporarily ceases.

The decision to continue execution of the program is not under control of the program. If execution is resumed through intervention of an operator without otherwise changing the state of the processor, the normal execution sequence, following PAUSE, is continued.

Execution may be terminated by typing a "T" at the operator console. Typing any other character will cause execution to resume.

7.8 CALL STATEMENT

The CALL statement transfers control into a SUBROUTINE subprogram and provides parameters for use by the subprogram. The general forms and detailed discussion of the CALL statement appear in Section 9.9.

The CALL statement may also be used to transfer control into a subprogram that is external to the executable program. This external routine can be written in a language other than FORTRAN. Refer to Appendix C for a more detailed discussion.

7.9 RETURN STATEMENT

The RETURN statement returns control to an originating program unit from a SUBROUTINE or FUNCTION subprogram. The form, use and interpretation of the RETURN statement is described in Section 9.10.

7.10 END STATEMENT

The END statement physically must be the last statement of any FORTRAN program unit. It has the following form:

```
END
```

The END statement is an executable statement and may have a statement label. It causes a transfer of control to be made to the system exit routine $EX, which returns control to CDOS.

SECTION 8

INPUT/OUTPUT

FORTRAN provides a series of statements which define the control and conditions of data transmission between computer memory and external data handling or mass storage devices such as disk, line printer, console, paper tape reader, paper tape punch, etc.

These statements are discussed in the following order:

1. OPEN which makes a disk file available for input or output.

2. READ which has two forms, the FORMATTED READ which inputs ASCII data from an input device and converts it according to the specifications in a separate FORMAT statement, and the UNFORMATTED READ which inputs binary data (in the form used for internal storage) with no conversion.

3. WRITE which has two forms, the FORMATTED WRITE which outputs ASCII data to an output device after conversion from internal storage according to the specifications in a separate FORMAT statement, and the UNFORMATTED WRITE which outputs binary data (data in the form used for internal storage) with no conversion.

4. BACKSPACE, ENDFILE, REWIND which end processing of files or control the next file record to process.

5. ENCODE and DECODE which transfer data between data items according to the specifications in a separate FORMAT statement.

6. I/O LIST SPECIFICATIONS which specify those data items to which data will be input or from which data will be output during the execution of the READ, WRITE, ENCODE and DECODE statements.

7. FORMAT statements which specify the data conversion and editing control to be used during the execution of formatted READ, formatted WRITE, ENCODE, and DECODE statements.

8.1 DISK FILES

Data is stored on disk in groupings of similar records called files. Each of these files has a unique name, assigned either by the programmer in the OPEN statement, or by the FORTRAN compiler if no OPEN statement is used. These records may be accessed sequentially, which means that they are read or written one after the other until the process is complete, or they may be accessed randomly, which means that a record is accessed by specifying the record's position relative to the first record of the file. The first record in a file is record 1. The maximum length is 128 bytes.

Each random WRITE uses 1 complete sector (128 bytes), regardless of the length of the data in the I/O list. Each unformatted WRITE also requires a complete sector, regardless of the length of the data in the I/O list. Sequential formatted WRITEs pack successive records onto disk regardless of sector boundaries, thereby using the disk most efficiently. The maximum number of records in a disk file is currently limited only by the size of the available disk. For 8 inch diskettes this would be a maximum of 1928 sectors, or 241K bytes (1K = 1Ø24 bytes). For 5 1/4 inch diskettes, this would be a maximum of 648 sectors or 81K bytes.

FORMAT OF DISK FILE RECORDS

TYPE OF FILE	RECORD WITHIN FILE	FIRST CHARACTER OF RECORD	LAST CHARACTER OF RECORD	END OF FILE INDICATOR
Sequential Formatted	FIRST OTHERS	Data <C-C>	<CR> <CR>	<CR><LF><^Z>
Sequential Unformatted	ALL	Data	Data	Ret'd by CDOS
Random Formatted	FIRST OTHERS	Data <C-C>	<CR> <CR>	Ret'd by CDOS
Random Unformatted	ALL	Data	Data	Ret'd by CDOS

<C-C> - the control characters inserted according to a control code specified by the user.

See 8.7.9, FORMAT CARRIAGE CONTROL.

<LF> - a Line Feed character; hexadecimal ØA

<CR> - a Carriage Return character; hexadecimal ØD

<^Z> - a Control Z; hexadecimal 1A

These characters are transparent to the user when doing numeric input. However, they are not transparent when doing alphanumeric input; they must be explicitly skipped.

8.2 LOGICAL UNIT NUMBERS

LOGICAL UNIT NUMBERS are numbers in the range 1 through 255 inclusive that are used to specify which I/O device is to be accessed by an I/O statement. Cromemco has assigned LUN's 1-1Ø to specific I/O devices, as follows:

LUN	DEVICE
1	console
2	printer
3-5	console
6-1Ø	disk

Any or all of the LUN's may be permanently reassigned by users to match their specific system configuration (see Appendix B for details.)

FORTRAN also allows the temporary reassignment of LUNs 1-5 to disk files through the use of the CALL OPEN statement. Since only one file may be open on a LUN at one time, this temporary reassignment is useful when it is necessary to have more than 5 files (the normal maximum on LUNS 6-1Ø) open concurrently. The reassignment remains in effect until the file is closed.

8.3 OPEN STATEMENTS

A disk file must be OPENed by a program before its records may be read or new records written.

1. OPENing a non-existent file causes the specified filename to be entered in the disk directory. No disk space is allocated for the file until a WRITE is performed.

2. OPENing an existing file, followed by a READ will allow access of the current records.

3. OPENing an existing file, followed by a sequential WRITE will delete the existing file at the time the first WRITE is performed.

4. Only one file may be OPENed on one LUN at any one time. If the user wishes to access a file using a LUN already assigned to an open file, the ENDFILE statement must first be used to terminate processing of the current file, then the CALL OPEN statement used to initiate processing of the new file.

5. An open file remains open until the execution of an ENDFILE or REWIND statement, or until the executable program terminates.

There are two types of OPENS.

8.3.1 Explicit OPEN

CALL OPEN (LUN, filename, drive)

where

LUN - A logical unit number to be associated with the file. The LUN must be an integer constant or integer variable with a value in the range 1-1Ø inclusive.

filename - An ASCII literal, Hollerith constant, variable, or array containing the ASCII name which will be assigned to the file. The name must be exactly 11 characters long, with any necessary blank fill between the name and extension, and blank

fill following the extension if the extension is not a full 3 characters long. A dot must not separate the name and its extension.

drive - The number of the disk drive containing the diskette on which the file does or will exist. The drive number must be an integer within the range allowed by the operating system.

Ø - the currently selected drive
1 - Drive A
2 - Drive B
3 - Drive C
4 - Drive D

8.3.2 Implicit OPEN

A READ from, or a WRITE to a logical unit automatically OPENs a file for input or output, respectively, if a file is not already open on the logical unit.

1. The disk drive associated with the file defaults to the currently selected disk.

2. The file will have a default name that includes the LUN:

   FORTØ6.DAT, or FORTØ7.DAT, or..., FORT1Ø.DAT

8.4 READ STATEMENTS

The READ statement transfers data from an input device to memory. Records are read sequentially unless the REC=k option is used to read records in random mode.

8.4.1 FORMATTED READ

A formatted READ transfers data from an input device to memory with data conversion and editing control specified in a separate FORMAT statement.

The formatted READ has two forms:

Option 1

```
READ(u, f, REC=k, ERR=L1, END=L2) list
```

Option 2

```
READ(u, f, REC=k, ERR=L1, END=L2)
```

where

u - The logical unit number from which the data will be input. It must be an integer constant or integer variable with a value in the range 1 to 255, inclusive.

f - either 1) the statement label of the FORMAT statement defining the data conversions to be performed for this READ, or 2) an array name specifying the array that contains as data the formatting information. This data is loaded into the array elements with a DATA statement or a previous formatted READ, first type (see 8.9.10 for a detailed discussion of Format specifications in arrays.)

REC=k - Optional
This clause is used with random disk files to read a record from a specific position within the file, relative to the beginning of the file. (The first record in a file is record 1.) k points to the relative position and must be an unsigned integer constant or integer variable with a value greater than or equal to 1.

ERR=L1 - Optional
This clause specifies that control will be transferred to the statement with

label = L1 if an I/O error is encountered when the READ is performed. L1 must be a valid FORTRAN statement label.
If this clause is not included, an I/O error will cause a fatal runtime error.

END=L2 - Optional
This clause specifies that control will be transferred to the statement with label = L2 if an End-of-File is encountered when the READ is performed. L2 must be a valid FORTRAN statement label. The END clause may not be used in the same READ statement with the REC clause.
If this clause is not specified, an End-of-File will cause a fatal runtime error.

list - a list of variable names specifying the receiving items for the input data (see Section 8.8 for a detailed discussion of I/O Lists).

8.4.1.1 Option 1 Notes

READ (u,f) list is used to input a number of items, corresponding to the names in the list, from the device assigned to logical unit u, using the FORMAT statement f to specify the expected external representation of these items.

1. A new record is read from the device each time the READ statement begins execution.

2. Additional records are read and the data is transferred item by item until each item in the list has received data. As many records as necessary are read.

3. If there are more data items in an input record than there are items in the list, each list item receives data and the extra data items are ignored.

Examples

1. Assume that four data entries are punched in a card, with three blank columns separating each and that the data have field widths of 3, 4, 2 and 5 characters respectively starting in column 1 of the card. The statements

```
   READ (5,2Ø)K,L,M,N,
2Ø FORMAT (I3,3X,I4,3X,I2,3X,I5)
```

will read the card (assuming the logical unit number 5 has been assigned to the card reader) and assign the input data to the variables K, L, M and N. The FORMAT statement could also be

```
2Ø FORMAT (I3,I7,I5,I8)
```

See Section 8.9 for a complete description of FORMAT statements.

2. Input the quantities of an array (ARRY):

```
READ (6,21)ARRY
```

All elements of the array ARRY will be read and stored using the appropriate formatting specified by the FORMAT statement labeled 21. (Only the name of the array needs to appear in the list. See Section 8.8).

8.4.1.2 Option 2 Notes

READ (u,f) is used to modify a FORMAT statement during execution by reading Hollerith data (an unconverted ASCII character string) from the device assigned to logical unit u into the Hollerith descriptor within the FORMAT statement with label f.

Example

```
      .
      .
      .
      READ (6, 25)
      .
      .
      .
25    FORMAT (1ØHABCDEFGHIJ)
      .
      .
      .
```

This causes the first 1Ø ASCII characters of the next record of the file on Logical Unit 6 to be read and replace the characters ABCDEFGHIJ in the FORMAT statement.
(See Section 8.9.3 for a more detailed discussion of Hollerith Conversions).

8.4.2 UNFORMATTED READ

An unformatted READ transfers data from an input device to memory without data conversion or editing control.

READ (u, REC=k, ERR=L1, END=L2) list

where u, REC=k, ERR=L1, END=L2, and list have the same meanings as used in formatted READS, discussed in Section 8.2.1.

8.4.2.1 Notes

1. Data is read from the input device and placed into memory with no conversion or editing. An exact image is transferred.

2. The total size, in bytes, of the memory assigned by FORTRAN to the list items must not be larger than the record size.

3. If the total size of the memory assigned to the list items is less than the logical record size, the extra data in the record is ignored.

8.5 WRITE STATEMENTS

The WRITE statement transfers data from memory to an output device. Records are written sequentially unless the REC=k option is used.

8.5.1 FORMATTED WRITE

A formatted WRITE outputs data to a device using a separate FORMAT statement to specify the external representation of the data.

Option 1:

```
WRITE (u,f, REC=k, ERR=L1, END=L2) list
```

Option 2:

```
WRITE (u,f, REC=k, ERR=L1, END=L2)
```

where

u - the logical unit number to which the data will be output. Must be an unsigned integer constant or integer variable with a value in the range 1 to 255 inclusive.

f - the statement label of the FORMAT statement defining the data conversions to be performed for this WRITE.

REC=k - Optional
This clause is used with random disk files to write a record into a specific position in a file relative to the beginning of the file. (The first record in a file is record 1.) k points to the relative position and must be an unsigned integer constant or integer variable with a value greater than or equal to 1.

ERR=L1 - Optional
This clause specifies that control will be transferred to the statement with label = L1 if an I/O error is encountered on the WRITE. L1 must be a valid FORTRAN

statement label.
If this clause is not included, an I/O error will cause a fatal runtime error.

END=L2 - Optional
This clause specifies that control will be transferred to the statement with label = L2 if an End-of-File is encountered when the WRITE is performed. L2 must be a valid FORTRAN statement label. The END clause may not be used in the same WRITE statement with the REC clause.
If this clause is not included, an End-of-File will cause a fatal runtime error.

list - a list of variable names specifying the source items for the output data. (See Section 8.8 for a detailed discussion of I/O lists).

8.5.1.1 Option 1 Notes

1. The first sequential WRITE to an existing disk file deletes the existing file and creates a new one with the same name.

2. More than 1 record may be output by a WRITE, depending on the number of items in the I/O list and the specifications given in the FORMAT statement. The data from each item in the list is transferred one field at a time according to the corresponding Format descriptor. A record is output each time a slash (/) is found in the list of field descriptors, whenever the rightmost parenthesis is found and there are more data items to transfer, and when the last data item has been transferred. (See Sections 8.9.7 and 8.9.8 for a detailed discussion.)

3. If the output is to a device which specifies fixed length records, and the data in the list does not fill the record, the remainder of the record will be filled with null characters (hex ØØ).

4. When a record is written to a disk file using the REC=k option, a record that already exists in the specified position will be overwritten. If no record exists in that position, the file will be extended to create one.

5. A random WRITE to a disk file (using the REC=k option) uses 1 full sector (128 bytes) each time it is performed.

6. If a WRITE to a disk file fails because no disk space is available, the error branch will be taken if the ERR= clause was coded.

Example:

```
WRITE (2,1Ø)A,B,C,D
```

The data assigned to the variables A, B, C and D are output to logical unit number 2, formatted according to the FORMAT statement labeled 1Ø.

8.5.1.2 Option 2 Notes

1. WRITE(u,f) may be used to write alphanumeric information when the characters to be written are specified within the FORMAT statement. In this case a variable list is not required.

   For example, to write the characters 'H CONVERSION' on unit 1,

```
      WRITE(1,26)
      .
      .
      .
   26 FORMAT (12HH CONVERSION)
```

8.5.2 UNFORMATTED WRITE

An unformatted WRITE performs memory-image transmission of data to an output device with no conversion or editing.

```
WRITE (u, REC=k, ERR=L1, END=L2) list
```

where u, REC=k, ERR=L1, END=L2, list have the same meanings as defined in the discussion of Formatted Writes, Section 8.3.1.

1. Unformatted output must be done only to those devices capable of accepting binary data, such as disk. The printer and terminal cannot accept binary data - an **IO** run time error will result when this is attempted.

2. Each record written to a disk file uses a complete sector (128 bytes). If the data does not fill the sector, the remainder of the sector will be filled with null characters (hex ØØ).

8.6 BACKSPACE, ENDFILE, REWIND

These statements are used to position or end processing of files.

8.6.1 BACKSPACE

BACKSPACE u

where u - a logical unit associated with an open file

Initially, the action of BACKSPACE is defined to be a no-op. It may be redefined by the user (see Appendix B.)

8.6.2 ENDFILE

ENDFILE u

where u - a logical unit associated with an open file

ENDFILE closes the file associated with the specified LUN. (The CALL OPEN routine may then be performed to access the file starting at the first record.)

8.6.3 REWIND

REWIND u

where u - a logical unit associated with an open file

REWIND closes the file associated with the specified LUN and reopens it at the beginning.

8.7 DECODE/ENCODE

The DECODE and ENCODE statements transfer data from one section of memory to another using format specifications in separate FORMAT statements.
DECODE moves data from an array to list items, changing it from ASCII representation to the form used for internal storage according to the specifications in a FORMAT statement.
ENCODE moves data from list items to an array, changing data from the form used for internal storage to ASCII representations according to the specifications in a FORMAT statement.

The forms of the statements are:

```
DECODE (array, format) list
ENDODE (array, format) list
```

where

array	is an array name
format	is the label of a FORMAT statement
list	is an I/O list

The action of DECODE is similar to that of READ, except that an array supplies the data instead of an I/O device.

Rules of Use for DECODE:

1. Any legal FORTRAN field descriptor defined in Section 8.9 may be used in the FORMAT statement to specify editing and conversion, subject to the rules specified in Section 8.9.

2. Each list item will receive data - there is no check for overflow.
   DECODE will attempt to process data from storage following an array that is not long enough to satisfy each list item according to the field descriptors, with unpredictable results.

3. The field descriptors in the FORMAT statement will be reused when the rightmost descriptor has been used and there are list items remaining to be processed. The data in the array will be reused each time repeat use of the field descriptors begins, starting with the first character of the array. (See Section 8.9.7 for a complete discussion of repeat specifications.)

The action of ENCODE is similar to that of WRITE, except that an array is the destination of the data instead of an I/O device.

Rules of Use for ENCODE:

1. Any legal FORTRAN field descriptor defined in Section 8.9 may be used in the FORMAT statement to specify editing and conversion, subject to the rules specified in Section 8.9.

2. Each list item will be processed - there is no check for overflow.
   ENCODE will write data into storage following an array that is not long enough to store the data from each list item after the editing and conversion specified by the field descriptors has been performed. Thus, other data or program code will be overwritten.

3. An ENCODE operation that moves data into an array that is larger than required will not affect existing data in the extra storage of the array.

4. The field descriptors in the FORMAT statement will be reused when the rightmost descriptor has been used and there are list items remaining to be processed. The storage allocated to the array will be reused each time repeat use of the field descriptors begins, starting with the first character of

storage allocated to the array. (See Section 8.9.7 for a complete discussion of repeat specifications.)

Examples:

Let A be a 16 element array of type BYTE containing the string of ASCII data

```
'DATA 3 523563.75'
```

The following statements yield the indicated results

```
      BYTE A(16), B(4)
      INTEGER I1,I2,I3,I4
      REAL R1
         DECODE (A,2) B,I1,I2,I3,I4,R1
2        FORMAT (4A1, 4I2, F4.2)
```

Results:

```
B contains 'DATA'
I1 contains the internal representation of 3
I2 contains the internal representation of 5
I3 contains the internal representation of 23
I4 contains the internal representation of 56
R1 contains the internal representation of 3.75
```

Let B be a 4 element array of type BYTE containing the ASCII string 'DATA',
I be a 4 element array of type INTEGER whose elements contain the internal representation of 2,66,84,95 and
R1 be a REAL variable containing the internal representation of 4.321E+12.

The following statements yield the indicated results:

```
      BYTE A(22), B(4)
      INTEGER I(4)
      REAL R1
          ENCODE (A,3) B,I,R1
3         FORMAT (4A1, 4I2, E1Ø.3)
```

Results:

A contains the string of ASCII data:
'DATA 2 68495 .432E+13'

8.8 INPUT/OUTPUT LIST SPECIFICATIONS

An I/O list specifies the data names that will be used as source or destination items in the execution of READ, WRITE, ENCODE, and DECODE statements. The order in which the list items appear must be the same as the order in which the corresponding data fields exist (for input) or will exist (for output) in the external medium.

Lists have the form

item1, item2,...,itemn

The commas are required.

8.8.1 Types of List Items

There are two types of list items. The first type specifies a single datum; the second type specifies multiple data.

1. Single datum.
   A single datum specifier may be a variable name or an array element name.

   Examples:

   A
   B(2,2)
   C(3,3,3)
   D,E,F,(1),G(2,2),H(3,3,3)

2. Multiple data.
   There are two types of multiple data specifiers.

   1) Array Name
      An array name appearing in a list without subscripts is considered equivalent to the listing of each element of the array

in the order specified by array storage allocation (see Section 6.2)

Example:

B is a 2-dimensional array.
The list item B is equivalent to the list.

B(1,1), B(2,1), B(3,1),...,B(1,2), B(2,2), B(3,2),...,B(i,j)

where i,j are the subscript limits of B.

2) Implied-DO lists
An Implied-DO list has the form:

(list, i=m1,m2,m3)

or

(list, i=m1,m2)

where

list	is an Input/Output list
i,m1,m2,m3	have the same meanings as defined for the DO statement (see Section 7.4). The parentheses are required.

When an Implied-DO list is encountered during the execution of an I/O statement, each item in the list is used once, in left to right order, for each iteration of the implied loop. The list is used as many times as the loop is repeated. Implied-DO lists may be nested.

Examples:

Implied-DO List	Equivalent List
(X(I),I=1,4)	X(1),X(2),X(3),X(4)
(Q(J),R(J),J=1,2)	Q(1),R(1),Q(2),R(2)
(G(K),K=1,7,3)	G(1),G(4),G(7)
((A(I,J),I=3,5),J=1,9,4)	A(3,1),A(4,1),A(5,1), A(3,5),A(4,5),A(5,5), A(3,9),A(4,9),A(5,9)
(R(M),M=1,2),I,Z(3)	R(1),R(2),I,Z(3)
(R(3),T(I),I=1,3)	R(3),T(1),R(3),T(2), R(3),T(3)

The elements of an array can be transmitted in an order different from that in which they exist in storage. The array A(2,2) occupies storage in the order A(1,1), A(2,1), A(1,2), A(2,2). By using the Implied-DO list

((A(I,J),J=1,2),I=1,2)

the elements of A will be used in the order

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

8.8.2 Notes on I/O List Specifications

1. Any number of items may appear in a single list, and the items may be of different data types.

2. In formatted I/O (READ(u,f) list, WRITE (u,f) list) the data type of each list item, or the actual data stored in the item (as in the use of LOGICAL, INTEGER, REAL, and DOUBLE PRECISION variables to store ASCII data), must agree with the format descriptor in the FORMAT statement. Unpredictable results may occur when there is a type - descriptor mismatch.

3. List items are used from left to right, with repetition of items within Implied-DO lists controlled by the DO-index parameters.

4. An unsubscripted array name causes all elements of the array to be used in the order specified by array storage allocation (see Section 6.2)

5. Constants must not be used as list items. Constants may appear in a list only when used as subscripts or Implied-DO parameters.

6. The Implied-DO parameters i,m1,m2,m3, when m1,m2,m3 are variables, must not be used as list items within parentheses of an Implied-DO Input list.

Examples:

```
INVALID - READ(1,20)(I,J,A(I),I=1,J,2)
VALID   - READ(1,20)I,J,(A(I),I=1,J,2)
VALID   - WRITE(1,20)(I,J,A(I),I=1,J,2)
```

The following array declarator, assignment statements, and WRITE statement yield the indicated output;

```
DIMENSION A(25)
A(1) = 2.1
A(2) = 0.
A(3) = 2.2
A(4) = 0.
A(5) = 2.3
J=5
WRITE(1,20)J,(I,A(I),I=1,J,2)
```

output results:

```
     5
     1
     2.1
     3
     2.2
     5
     2.3
```

8.9 FORMAT STATEMENTS

FORMAT statements are non-executable, generative statements used in conjunction with formatted READ and WRITE statements. They specify conversion methods and data editing information as the data is transmitted between computer storage and external media representation.

FORMAT statements require statement labels for reference (f) in the READ (u,f)k or WRITE (u,f)k statements.

The general form of a FORMAT statement is as follows:

```
n FORMAT (s1,s2,...,sn/s1',s2',...,sn'/...)
```

where n is the statement label and each si is a field descriptor. The word FORMAT and the parentheses must be present as shown. The slash (/) and comma (,) characters are field separators and are described in paragraph 8.9.7.2. The field is defined as that part of an external record occupied by one transmitted item.

8.9.1 FIELD DESCRIPTORS

Field descriptors describe the sizes of data fields and specify the type of conversion to be exercised upon each transmitted datum. The FORMAT field descriptors may have any of the following forms:

Descriptor	Classification
rFw.d rGw.d rEw.d rDw.d rIw	Numeric Conversion
rLw	Logical Conversion
rAw nHh1h2...hn 'l1l2...ln'	Hollerith Conversion
nX	Spacing Specification
mP	Scaling Factor

where:

1. w and n are positive integer constants defining the field width (including digits, decimal points, algebraic signs) in the external data representation.

2. d is an integer specifying the number of fractional digits appearing in the external data representation.

3. The characters F, G, E, D, I, A and L indicate the type of conversion to be applied to the items in an input/output list.

4. r is an optional, non-zero integer indicating that the descriptor will be repeated r times.

5. The hi and li are characters from the FORTRAN character set.

6. m is an integer constant (positive, negative, or zero) indicating scaling.

8.9.2 NUMERIC CONVERSIONS

Input operations with any of the numeric conversions will allow the data to be represented in a free format; i.e., commas may be used to separate the fields in the external representation.

F-type Conversions

Form: Fw.d

REAL or DOUBLE PRECISION type data are processed using this conversion. Of the total (w) characters processed, d are considered fractional.

F-Output:

Values are converted and output as minus sign (if negative), followed by the integer portion of the number, a decimal point and d digits of the fractional portion of the number. If a value does not fill the field, it is right justified in the field and enough preceding blanks to fill the field are inserted.

A value that requires more field positions than specified by the field descriptor causes FORTRAN to output an asterisk in the leftmost character position of the output field, generate the **FW** run time error, and attempt to recover and continue.

F-Output Examples:

FORMAT Descriptor	Internal Value	Output (b=blank)
F1Ø.4	368.42	bb368.42ØØ
F7.1	-4786.361	-4786.4
F8.4	8.7E-2	bbb.Ø87Ø
F6.4	4739.76	*.76ØØ
F7.3	-5.6	b-5.6ØØ

F-Input:

(See the description under E-Input below.)

E-type Conversions

Form: Ew.d

REAL or DOUBLE PRECISION type data are processed using this conversion. w characters are processed of which d are considered fractional.

E-Output:

Values are converted, rounded to d digits, and output as:

1. minus sign (if negative),
2. a zero and a decimal point,
3. d decimal digits,
4. the letter E,
5. the sign of the exponent (minus or plus),
6. two exponent digits,

in that order. The values as described are right justified in the field w with preceding blanks to fill the field if necessary. The field width w should satisfy the relationship:

w > d + 7

Otherwise significant characters may be lost.

A value that requires more field positions than specified by the field descriptor causes FORTRAN to output an asterisk in the leftmost character position of the output field, generate the **FW** run time error, and attempt to recover and continue.

Examples:

FORMAT Descriptor	Internal Value	Output (b=blank)
E12.5	76.573	bb.76573E+Ø2
E14.7	-32672.354	-Ø.3267235EbØ5
E7.3	56.93	*Ø.569E
E13.4	-Ø.ØØ12321	bbb-.1232E-Ø2
E8.2	76321.73	b.76E+Ø5

E-Input:

Data values which are to be processed under E, F, or G conversion can be a relatively loose format in the external input medium. The format is identical for either conversion and is as follows:

1. Leading spaces (ignored)

2. A + or - sign (an unsigned input is assumed to be positive)

3. A string of digits

4. A decimal point

5. A second string of digits

6. The character E

7. A + or - sign

8. A decimal exponent

Each item in the list above is optional; but the following conditions must be observed:

1. If FORMAT items 3 and 5 (above) are present, then 4 is required.

2. If FORMAT item 8 is present, then 6 and/or 7 are required.

3. All non-leading spaces are considered zeros.

Input data can be any number of digits in length, and correct magnitudes will be developed, but precision will be maintained only to the extent specified in Section 3 for Real data.

E- and F- and G- Input Examples:

FORMAT Descriptor	Input (b=blank)	Internal Value
E1Ø.3	+Ø.23756+4	+2375.6Ø
E1Ø.3	bbbbb17631	+17.631
G8.3	b1628911	+1628.911
F12.4	bbbb-6321132	-632.1131

Note in the above examples that if no decimal point is given among the input characters, the d in the FORMAT specification establishes the decimal point in conjunction with an exponent, if given. If a decimal point is included in the input characters, the d specification is ignored.

The letters E, F, and G are interchangable in the input format specifications. The end result is the same.

D-Type Conversions

D-Input and D-Output are identical to E-Input and E-Output except the exponent may be specified with a "D" instead of an "E".

G-Type Conversions

Form: Gw.d

REAL or DOUBLE PRECISION type data are processed using this conversion. Of the total (w) characters processed, d are considered significant.

G-Input:

(See the description under E-Input)

G-Output:

The method of output conversion is a function of the magnitude of the number being output. Let n be the magnitude of the number. The following table shows how the number will be output:

Magnitude	Equivalent Conversion
$n < .1$	Ew.d
$.1 \leq n < 1$	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).Ø,4X
$10^{d} \leq n$	Ew.d

Examples:

Internal Value	Format	Output
.ØØØØ2	G1Ø.4	b.2ØØØE-Ø4
.2	G1Ø.4	b.2ØØØbbbb
2.	G1Ø.4	b2.ØØØbbbb
2ØØ.	G1Ø.4	b2ØØ.Øbbbb
2ØØØØ.	G1Ø.4	b.2ØØØE+Ø5

I-Conversions

Form: Iw

Only INTEGER data may be converted by this form of conversion. The field width is specified by w.

I-Output:

Values are converted to INTEGER constants. Negative values are preceded by a minus sign. If the value does not fill the field, it is right justified in the field and enough preceding blanks to fill the field are inserted.

A value that requires more field positions than specified by the field descriptor causes FORTRAN to output an asterisk in the leftmost character position of the output field, generate the **FW** run time error, and attempt to recover and continue.

Examples:

FORMAT Descriptor	Internal Value	Output (b=blank)
I6	+281	bbb281
I6	-23261	-23261
I3	126	126
I4	-226	-226
I3	1234	*34

I-Input:

A field of w characters is input and converted to internal INTEGER format. A minus sign may precede the integer digits. If a sign is not present, the value is considered positive.

Integer values in the range -32768 to 32767 are accepted. Non-leading spaces are treated as zeros.

Examples:

Format Descriptor	Input (b=blank)	Internal Value
I4	b124	124
I4	-124	-124
I7	bb6732b	67320
I4	1b2b	1020

8.9.3 HOLLERITH CONVERSIONS

A-Type Conversion

The form of the A conversion is as follows:

Aw

This descriptor causes unmodified Hollerith characters to be read into or written from a specified list item.

The maximum number of actual characters which may be transmitted between internal and external representations using Aw is four times the number of storage units in the corresponding list item (i.e., 1 character for LOGICAL items, 2 characters for INTEGER items, 4 characters for REAL items and 8 characters for DOUBLE PRECISION items).

A-Output:

If w is greater than 4n (where n is the number of storage units required by the list item), the external output field will consist of w-4n blanks followed by the 4 n characters from the internal representation. If w is less than 4n, the external output field will consist of the leftmost w characters from the internal representation.

Examples:

Format Descriptor	Internal	Type	Output (b=blanks)
A1	A1	Integer	A
A2	AB	Integer	AB
A3	ABCD	Real	ABC
A4	ABCD	Real	ABCD
A7	ABCD	Real	bbbABCD

A-Input:

If w is greater than 4n (where n is the number of storage units required by the corresponding list item), the rightmost 4n characters are taken from the external input field. If w is less than 4n, the w characters appear left justified with w-4n trailing blanks in the internal representation.

Examples:

Format Descriptor	Input Characters	Type	Internal (b=blanks)
A1	A	Integer	Ab
A3	ABC	Integer	AB
A4	ABCD	Integer	AB
A1	A	Real	Abbb
A7	ABCDEFG	Real	DEFG

H-Conversion

The forms of H conversion are as follows:

nHh1h2...hn

'h1h2...hn'

These descriptors process Hollerith character strings between the descriptor and the external field, where each h represents any character from the ASCII character set.

Special consideration is required if an apostrophe (') is to be used within the literal string in the second form. An apostrophe character within the string is represented by two successive apostrophes. See the examples below.

H-Output:

The n characters hi, are placed in the external field. In the nHh1h2...hn form the number of characters in the string must be exactly as specified by n. Otherwise, characters from other descriptors will be taken as part of the string. In both forms, blanks are counted as characters.

Examples:

Format Descriptor		Output (b=blank)
1HA	or 'A'	A
8HbSTRINGb	or 'bSTRINGb'	bSTRINGb
11HX(2,3)=12.Ø	or 'X(2,3)=12.Ø'	X(2,3)=12.Ø
12HIbSHOULDN'T	or 'IbSHOULDN''T'	IbSHOULDN'T

H-Input:

The n characters of the string hi are replaced by the next n characters from the input record. This results in a new string of characters in the field descriptor.

FORMAT Descriptor	Input (b=blank)	Resultant Descriptor
4H1234 or '1234'	ABCD	4HABCD or 'ABCD'
7HbbFALSE or 'bbFALSE'	bFALSEb	7HbFALSEb or 'bFALSEb'
6Hbbbbbb or 'bbbbbb'	MATRIX	6HMATRIX or 'MATRIX'

8.9.4 LOGICAL CONVERSIONS

The form of the LOGICAL conversion is as follows:

Lw

L-Output:

If the value of an item in an output list corresponding to this descriptor is Ø, an F will be output; otherwise, a T will be output. If w is greater than 1, w-1 leading blanks precede the letters.

Example:

FORMAT Descriptor	Internal Value	Output (b=blank)
L1	=Ø	F
L1	<>Ø	T
L5	<>Ø	bbbbT
L7	=Ø	bbbbbbF

L-Input:

The external representation occupies w positions. It consists of optional blanks followed by a "T" or "F", followed by optional characters.

8.9.5 X DESCRIPTOR

The form of X conversion is as follows:

nX

This descriptor causes no conversion to occur, nor does it correspond to an item in an input/output list. When used for output, it causes n blanks to be inserted in the output record. Under input, this descriptor causes the next n characters of the input record to be skipped.

Output Examples:

FORMAT Statement	Output (b=blanks)
3 FORMAT (1HA,4X,2HBC)	AbbbbBC
7 FORMAT (3X,4HABCD,1X)	bbbABCDb

Input Examples:

FORMAT Statement	Input String	Resultant Input
1Ø FORMAT (F4.1,3X,F3.Ø)	12.5ABC12Ø	12.5,12Ø
5 FORMAT (7X,I3)	1234567Ø12	Ø12

8.9.6 P DESCRIPTOR

The P descriptor is used to specify a scaling factor for real conversions (F, E, D, G). The form is nP where n is an integer constant (positive, negative, or zero).

The scaling factor is automatically set to zero at the beginning of each formatted I/O call (each READ or WRITE statement). If a P descriptor is encountered while scanning a FORMAT, the scale factor is changed to n. The scale factor remains changed until another P descriptor is encountered or the I/O terminates.

Effects of Scale Factor on Input:

During E, F, or G input the scale factor takes effect only if no exponent is present in the external representation. In that case, the

internal value will be a factor of 1Ø**n less than the external value (the number will be divided by 1Ø**n before being stored).

Effect of Scale Factor on Output:

E-Output, D-Output:

The coefficient is shifted left n places relative to the decimal point, and the exponent is reduced by n (the value remains the same).

F-Output:

The external value will be 1Ø**n times the internal value.

G-Output:

The scale factor is ignored if the internal value is small enough to be output using F conversion. Otherwise, the effect is the same as for E output.

8.9.7 SPECIAL CONTROL FEATURES OF FORMAT STATEMENTS

8.9.7.1 Repeat Specifications

1. The E, F, D, G, I, L and A field descriptors may be indicated as repetitive descriptors by using repeat count r in the form rEw.d, rFw.d, rGw.d, rIw, rLw, rAw. The following pairs of FORMAT statements are equivalent:

```
   66   FORMAT (3F8.3,F9.2)
C   IS EQUIVALENT TO:
   66   FORMAT (F8.3,F8.3,F8.3,F9.2)

   14   FORMAT (2I3,2A5,2E1Ø.5)
C   IS EQUIVALENT TO:
   14   FORMAT (I3,I3,A5,A5,E1Ø.5,E1Ø.5)
```

2. Repetition of a group of field descriptors is accomplished by enclosing the group in parentheses preceded by a repeat count. Absence of a repeat count indicates a count of one. Up to two levels of parentheses, including the parentheses required by the FORMAT statement, are permitted.

Note the following equivalent statements:

```
   22  FORMAT (I3,4(F6.1,2X))
C  IS EQUIVALENT TO:
   22  FORMAT (I3,F6.1,2X,F6.1,2X,F6.1,2X,
      1   F6.1,2X)
```

3. Repetition of FORMAT descriptors is also initiated when all descriptors in the FORMAT statement have been used but there are still items in the input/output list that have not been processed. When this occurs the FORMAT descriptors are re-used starting at the opening parenthesis that matches the last closing parenthesis in the FORMAT statement. The parentheses enclosing the entire list of descriptors are not considered unless there are no other parentheses in the list. A repeat count preceding the parenthesized descriptor(s) to be re-used is also active in the re-use. This type of repetitive use of FORMAT descriptors terminates processing of the current record and initiates the processing of a new record each time the re-use begins. Record demarcation under these circumstances is the same as in the paragraph 8.9.7.2.

Input Example:

```
      DIMENSION A(1ØØ)
      READ (3,13) A
      .
      .
      .
13 FORMAT (5F7.3)
```

In this example, the first 5 quantities from each of 2Ø records are input and assigned to the array elements of the array A.

Output Example:

```
        .
        .
        .
        WRITE (6,12)E,F,K,L,M,KK,LL,MM,K3,L3,
       1   M3
        .
        .
        .
    12 FORMAT (2F9.4,(3I7))
```

In this example, three records are written. Record 1 contains E, F, K, L and M. Because the descriptor 3I7 is reused twice, Record 2 contains KK, LL and MM and Record 3 contains K3, L3 and M3.

8.9.7.2 Field Separators

Two adjacent descriptors must be separated in the FORMAT statement by either a comma or one or more slashes.

Example:

2HOK/F6.3 or 2HOK,F6.3

The slash not only separates field descriptors, but it also specifies the demarcation of formatted records.

Each slash terminates a record and sets up the next record for processing. The remainder of an input record is ignored; the remainder of an output record is filled with null characters (hex ØØ). Successive slashes (///.../) cause successive records to be ignored on input and successive null records to be written on output.

Output example:

```
      DIMENSION A (1ØØ),J(2Ø)
      .
      .
      .
      WRITE (7,8) J,A
  8   FORMAT (1ØI7/1ØI7/5ØF7.3/5ØF7.3)
```

In this example, the data specified by the list of the WRITE statement are output to unit 7 according to the specifications of FORMAT statement 8. Four records are written as follows:

Record 1	Record 2	Record 3	Record 4
J(1)	J(11)	A(1)	A(51)
J(2)	J(12)	A(2)	A(52)
.	.	.	.
.	.	.	.
.	.	.	.
J(10)	J(20)	A(50)	A(100)

Input Example:

```
      DIMENSION B(10)
      .
      .
      .
      READ (4,17) B
   17 FORMAT(F10.2/F10.2///8F10.2)
```

In this example, the two array elements B(1) and B(2) receive their values from the first data fields of successive records (the remainders of the two records are ignored). The third and fourth records are ignored and the remaining elements of the array are filled from the fifth record.

8.9.8 FORMAT CONTROL, LIST SPECIFICATIONS AND RECORD DEMARCATION

The following relationships and interactions between FORMAT control, input/output lists and record demarcation should be noted:

1. Execution of a formatted READ or WRITE statement initiates FORMAT control.

2. The conversion performed on data depends on information jointly provided by the elements in the input/output list and field descriptors in the FORMAT statement.

3. If there is an input/output list, at least one descriptor of types E, F, G, I, L or A must be present in the FORMAT statement.

4. Each execution of a formatted READ statement causes a new record to be input.

5. Each item in an input list corresponds to a string of characters in the record and to a descriptor of the types E, F, G, I, L or A in the FORMAT statement.

6. H and X descriptors communicate information directly between external records and the field descriptors without reference to list items.

7. On input, whenever a slash is encountered in the FORMAT statement or the FORMAT descriptors have been exhausted and re-use of descriptors is initiated, processing of the current record is terminated and the following occurs:

   a. Any unprocessed characters in the record are ignored.

   b. If more input is necessary to satisfy list requirements, the next record is read.

8. A READ statement is terminated when all items in the input list have been satisfied if:

   a. The next FORMAT descriptor is E, F, G, I, L or A.

   b. The FORMAT control has reached the last outer right parenthesis of the FORMAT statement.

   If the input list has been satisfied, but the next FORMAT descriptor is H or X, more data are processed (with the possibility of new records being input) until one of the above conditions exists.

9. If FORMAT control reaches the last right parenthesis of the FORMAT statement but there are more list items to be processed, all or part of the descriptors are reused. (See item 3 in the description of Repeat Specifications, sub-paragraph 8.9.7.1)

1Ø. When a Formatted WRITE statement is executed, records are written each time a slash is encountered in the FORMAT statement or FORMAT control has reached the rightmost right parenthesis. The FORMAT control terminates in one of the two methods described for READ termination in 8 above. Incomplete records are filled with nulls (hexadecimal ØØ) to maintain record lengths.

8.9.9 FORMAT CARRIAGE CONTROL

The first character of every formatted output record is used as a control code to specify the carriage control to be performed for the record. It is interpreted by FORTRAN, and the resulting control character is sent to the device as the first character of the record. If the device is a printer, this character controls action taken before printing. If the device is a disk, the control character occupies the first character position and is transparent to input when numeric Format descriptors are used. When the alphanumeric Format descriptor A is used, these control characters are not transparent and must be explicitly skipped by using the Format descriptor X.

CODE	RESULTING CHARACTER (HEX)	ACTION TAKEN BEFORE PRINTING
Ø	ØAØA (2 line feeds)	Skip 2 lines
1	ØC (form feed)	Insert Form Feed
+	none	No Advance
other	ØA (line feed)	Skip 1 line

8.9.1Ø FORMAT SPECIFICATIONS IN ARRAYS

The FORMAT reference, f, of a formatted READ or WRITE statement (see 8.4.1 and 8.5.1) may be an array name instead of statement label. If such a reference is made, at the time of execution of the READ/WRITE statement the first part of the information contained in the array, taken in natural order, must constitute a valid FORMAT specification. The array may contain non-FORMAT

information following the right parenthesis that ends the FORMAT specification.

The FORMAT specification which is to be inserted in the array has the same form as defined for a FORMAT statement (i.e., it begins with a left parenthesis and ends with a right parenthesis).

The FORMAT specification may be inserted in the array by use of a DATA initialization statement, or by use of a READ statement together with an Aw FORMAT.

Example:

Assume the FORMAT specification

```
(3F1Ø.3,4I6)
```

or a similar 12 character specification is to be stored into an array. The array must allow a minimum of 3 storage units.

The FORTRAN coding below shows the various methods of establishing the FORMAT specification and then referencing the array for a formatted READ or WRITE.

```
C  DECLARE A REAL ARRAY
       DIMENSION A(3), B(3), M(4)
C  INITIALIZE FORMAT WITH DATA STATEMENT
       DATA A/'(3F1','Ø.3,','4I6)'/
       .
       .
       .
C  READ DATA USING FORMAT SPECIFICATIONS
C       IN ARRAY A
       READ(6,A) B, M

C  DECLARE AN INTEGER ARRAY
       DIMENSION IA(4), B(3), M(4)
       .
       .
       .
C  READ FORMAT SPECIFICATIONS
       READ (7,15) IA
C  FORMAT FOR INPUT OF FORMAT SPECIFICATIONS
   15  FORMAT (4A2)
       .
       .
```

```
      .
C  READ DATA USING PREVIOUSLY INPUT
C       FORMAT SPECIFICATION
       READ (7,IA) B,M
      .
      .
      .
```

SECTION 9

FUNCTIONS AND SUBPROGRAMS

The FORTRAN language provides a means for defining and using often needed programming procedures such that the statement or statements of the procedures need appear in a program only once but may be referenced and brought into the logical execution sequence of the program whenever and as often as needed.

These procedures are as follows:

1. Statement functions.
2. Library functions.
3. FUNCTION subprograms.
4. SUBROUTINE subprograms.

Each of these procedures has its own unique requirements for reference and defining purposes. These requirements are discussed in subsequent paragraphs of this section. However, certain features are common to the whole group or two or more of the procedures. These common features are as follows:

1. Each of these procedures is referenced by its name which, in all cases, is one to six alphanumeric characters of which the first is a letter.
2. The first three are designated as functions and are alike in that:
    1. They are always single valued (i.e., they return one value to the program unit from which they are referenced).
    2. They are referred to by an expression containing a function name.
    3. The value returned from a FUNCTION subprogram has a data type indicated either implicitly by the first letter of the FUNCTION name (see Section 3.1.2) or explicitly by the FUNCTION statement.

4. The FUNCTION name itself must appear in a type statement in the calling program unit when the value being returned has been specified, in the body of the function, to be of a type different from the implicit type.

3. FUNCTION subprograms and SUBROUTINE subprograms are considered program units.

In the following descriptions of these procedures, the term calling program means the program unit or procedure in which a reference to a procedure is made, and the term called program means the procedure to which a reference is made.

9.1 THE PROGRAM STATEMENT

The PROGRAM statement provides a means of specifying a name for a main program unit. The form of the statement is:

```
PROGRAM name
```

If present, the PROGRAM statement must appear before any other statement in the program unit. The name consists of 1-6 alphanumeric characters, the first of which must be a letter. If no PROGRAM statement is present in a main program, the compiler assigns a name of $MAIN to that program.

9.2 STATEMENT FUNCTIONS

Statement functions are defined by a single arithmetic or logical replacement statement and are accessable only to statements within the program unit in which the functions are defined.
The general form of a statement function is as follows:

```
f(a1,a2,...,an) = e
```

where f is the function name, the ai are dummy arguments and e is an arithmetic or logical expression.

Rules for ordering, structure and use of statement functions are as follows:

1. Statement function definitions, if they exist in a program unit, must precede all executable statements in the unit and follow all specification statements.

2. The ai are distinct variable names or array elements, but, being dummy variables, they may have the same names as variables of the same type appearing elsewhere in the program unit.

3. The expression e is constructed according to the rules in SECTION 4 and may contain only references to the dummy arguments and non-literal constants, variable and array element references, utility and mathematical function references and references to previously defined statement functions.

4. The type of any statement function name or argument that differs from its pre-defined convention type must be defined by a type specification statement.

5. The relationship between f and e must conform to the replacement rules in Section 5.

6. A statement function is called by its name followed by a parenthesized list of arguments. The expression is evaluated using the arguments specified in the call, and the reference is replaced by the result.

7. The ith parameter in every argument list must agree in type with the ith dummy in the statement function.

The example below shows a statement function and a statement function call.

```
C  STATEMENT FUNCTION DEFINITION
C
      FUNC1(A,B,C,D) = ((A+B)**C)/D

C  STATEMENT FUNCTION CALL
C
      A12=A1-FUNC1(X,Y,Z7,C7)
```

9.3 LIBRARY FUNCTIONS

Library functions are a group of utility and mathematical functions which are built into the FORTRAN system. Their names are pre-defined to the Processor and automatically typed. The functions are listed in Tables 9-1 and 9-2. In the tables, arguments are denoted as a1,a2,...,an, if more than one argument is required; or as a if only one is required.

A library function is called when its name is used in an arithmetic expression. Such a reference takes the following form:

f(a1,a2,...,an)

where f is the name of the function and the ai are actual arguments. The arguments must agree in type, number and order with the specifications indicated in Tables 9-1 and 9-2.

Examples:

```
A1 = B+FLOAT (I7)

MAGNI = ABS (KBAR)

PDIF = DIM (C,D)

S3 = SIN (T12)

ROOT = (-B+SQRT(B**2-4.*A*C))/
1         (2.*A)
```

In addition to the functions listed in Table 9-1 and Table 9-2, four additional library subprograms are provided to enable direct access to the Z8Ø hardware. These are:

PEEK, POKE, INP, OUT

9.3.1 PEEK

PEEK is a function of type INTEGER*1 or BYTE that allows access to any memory location.
The form of this function is:

PEEK(a)

PEEK(a) returns the contents of the memory location specified by a.

a is either an INTEGER constant or INTEGER variable.

9.3.2 POKE

POKE is a subroutine that allows access to any memory location.
The form of the subroutine reference is:

CALL POKE (a1,a2)

CALL POKE (a1,a2) causes the contents of the memory location specified by a1 to be replaced by the value of a2.

a1 is an INTEGER constant or INTEGER variable.
a2 is an INTEGER*1 or BYTE constant or variable.

9.3.3 INP

INP is a function of type INTEGER*1 or BYTE that allows direct access to the input ports.
The form of this function is:

INP(a)

INP(a) does an input from port a and returns the 8 bit value that was input.

a is an INTEGER*1 or BYTE constant or variable.

9.3.4 OUT

OUT is a subroutine that allows direct access to the output ports.
The form of the subroutine reference is:

```
CALL OUT (a1,a2)
```

CALL OUT (a1,a2) outputs the value of a2 to the port specified a1.

a1 is an INTEGER*1 or BYTE constant or variable.
a2 is an INTEGER*1 or BYTE constant or variable.

TABLE 9-1

Intrinsic Functions

Function Name	Definition	Types Argument	Types Function
ABS	|a|	Real	Real
IABS		Integer	Integer
DABS		Double	Double
AINT	Sign of a times lar-	Real	Real
INT	gest integer <=|a|	Real	Integer
IDINT		Double	Integer
AMOD	a1(mod a2)	Real	Integer
MOD		Integer	Integer
DMOD	a1(mod a2)	Double	Double
AMAXØ	Max(a1,a2,...)	Integer	Real
AMAX1		Real	Real
MAXØ		Integer	Integer
MAX1		Real	Integer
DMAX1		Double	Double
AMINØ	Min(a1,a2,...)	Integer	Real
AMIN1		Real	Real
MINØ		Integer	Integer
MIN1		Real	Integer
DMIN1		Double	Double
FLOAT	Conversion from Integer to Real	Integer	Real
IFIX	Conversion from Real to Integer	Real	Integer
SIGN	Sign of a2 times |a1|	Real	Real
ISIGN		Integer	Integer
DSIGN		Double	Double
DIM	a1 - Min(a1,a2)	Real	Real
IDIM		Integer	Integer
SNGL	Conversion from Double to Real	Double	Real
DBLE	Conversion from Real to Double	Real	Double

TABLE 9-2

Basic External Functions

Name	Number of Arguments	Definition	Type Argument	Type Function
EXP	1	e**a	Real	Real
DEXP	1		Double	Double
ALOG	1	ln (a)	Real	Real
DLOG	1		Double	Double
ALOG1Ø	1	log1Ø(a)	Real	Real
DLOG1Ø	1		Double	Double
SIN	1	sin (a)	Real	Real
DSIN	1		Double	Double
COS	1	cos (a)	Real	Real
DCOS	1		Double	Double
TANH	1	tanh (a)	Real	Real
SQRT	1	(a) ** 1/2	Real	Real
DSQRT	1		Double	Double
ATAN	1	arctan (a)	Real	Real
DATAN	1		Double	Double
ATAN2	2	arctan (a1/a2)	Real	Real
DATAN2	2		Double	Double

9.4 FUNCTION SUBPROGRAMS

A program unit which begins with a FUNCTION statement is called a FUNCTION subprogram.

A FUNCTION statement has one of the following forms:

```
t FUNCTION f(a1,a2,...an)
```

or

```
FUNCTION f(a1,a2,...an)
```

where:

1. t is either INTEGER, REAL, DOUBLE PRECISION or LOGICAL or is empty as shown in the second form.

2. f is the name of the FUNCTION subprogram.

3. The ai are dummy arguments of which there must be at least one and which represent variable names, array names or dummy names of SUBROUTINE or other FUNCTION subprograms.

9.5 CONSTRUCTION OF FUNCTION SUBPROGRAMS

Construction of FUNCTION subprograms must comply with the following rules:

1. The FUNCTION statement must be the first statement of the program unit.

2. Within the FUNCTION subprogram, the FUNCTION name must appear at least once on the left side of the equality sign of a replacement statement or as an item in the input list of an input statement. This defines the value of the FUNCTION so that it may be returned to the calling program.

   Additional values may be returned to the calling program through assignment of values to dummy arguments.

3. No name in the dummy argument list may appear in a COMMON
   EQUIVALENCE
   or DATA

   statement in the FUNCTION subprogram.

4. A dummy argument that corresponds to an array requires an array declaration in the FUNCTION subprogram. The dimensioning information must be consistent with that of the array declarator in the calling program.

5. A FUNCTION subprogram must not contain

   a PROGRAM statement
   a BLOCK DATA statement
   a SUBROUTINE statement
   another FUNCTION statement
   or a statement that references the name of the FUNCTION containing it, either directly, or indirectly through a sequence of other subprograms.

   All other FORTRAN statements are allowed.

6. A FUNCTION subprogram must contain at least one RETURN statement. It may contain more than one.
   The RETURN statement provides a logical termination of the execution of a subprogram. It causes control to return to the calling program.

7. The last statement appearing in a FUNCTION subprogram must be the END statement.

Examples:

```
      FUNCTION Z7 (A,B,C)
       .
       .
       .
      Z7 = 5.*(A-B) + SQRT(C)
       .
       .
       .
C     REDEFINE ARGUMENT
      B=B+Z7
       .
       .
       .
      RETURN
       .
       .
       .
      END
```

```
      FUNCTION SUM (BARY,I,J)
      DIMENSION BARY(10,20)
      SUM = 0.0
      DO 8 K=1,I
      DO 8 M = 1,J
    8 SUM = SUM + BARY(K,M)
      RETURN
      END
```

9.6 REFERENCING A FUNCTION SUBPROGRAM

FUNCTION subprograms are called whenever the FUNCTION name, accompanied by an argument list, is used as an operand in an expression. Such references take the following form:

f(a1,a2,...,an)

where f is a FUNCTION name and the ai are actual arguments. Parentheses must be present in the form shown.

The arguments ai must agree in type, order and number with the dummy arguments in the FUNCTION statement of the called FUNCTION subprogram. They may be any of the following:

1. A variable name.

2. An array element name.

3. An array name.

4. An expression.

5. A SUBROUTINE or FUNCTION subprogram name.

6. A Hollerith or Literal constant.

A FUNCTION or SUBROUTINE name appearing in the argument list of a FUNCTION reference must appear in an EXTERNAL statement in the calling program.

A dummy argument corresponding to a FUNCTION or a SUBROUTINE name in the actual argument list must appear in a subprogram reference within the called program.

A dummy argument corresponding to a Hollerith or a Literal constant in the actual argument list should be specified so as to require the same number of storage units as needed to store the constant.

When a FUNCTION subprogram is called, program control goes to the first executable statement following the FUNCTION statement.

The following examples show references to FUNCTION subprograms.

```
Z1Ø = FT1+Z7(D,T3,RHO)

DIMENSION DAT(5,5)
.
.
.
S1 = TOT1 + SUM(DAT,5,5)
```

9.7 SUBROUTINE SUBPROGRAMS

A program unit which begins with a SUBROUTINE statement is called a SUBROUTINE subprogram. The SUBROUTINE statement has one of the following forms:

```
SUBROUTINE s (a1,a2,...,an)
```

or

```
SUBROUTINE s
```

where s is the name of the SUBROUTINE subprogram and each ai is a dummy argument which represents a variable or array name or another SUBROUTINE or FUNCTION name.

9.8 CONSTRUCTION OF SUBROUTINE SUBPROGRAMS

1. The SUBROUTINE statement must be the first statement of the subprogram.

2. The SUBROUTINE subprogram name must not appear in any statement other than the initial SUBROUTINE statement.

3. The dummy argument names must not appear in EQUIVALENCE, COMMON or DATA statements in the subprogram.

4. A dummy argument that corresponds to an array requires an array declarator in the subroutine. The dimensioning information must be consistent with that of the array declarator in the calling program.

5. A subroutine must not contain

   a PROGRAM statement
   a BLOCK DATA statement
   a FUNCTION statement
   another SUBROUTINE statement
   or a statement that references the name of the SUBROUTINE containing it, either directly, or indirectly through a sequence of other subprograms.

   All other FORTRAN statements are allowed.

6. A SUBROUTINE subprogram must contain at least one RETURN statement. It may contain more than one.

   The RETURN statement provides a logical termination of the execution of the subprogram. It causes control to return to the calling program.

7. The last statement appearing in a SUBROUTINE subprogram must be the END statement.

8. Values may be returned to the calling program by redefining the values of the dummy arguments within the called program.
   This may happen in one of three ways:

   1. The dummy argument may appear in a replacement statement to the left of the equal sign.

   2. The dummy argument may appear in the input list of an input statement.

   3. The dummy argument may appear as a parameter in a subprogram reference.

9. A dummy argument that corresponds to an actual argument that is a FUNCTION or SUBROUTINE name must be used as a subprogram reference in the called SUBROUTINE.

Example:

```
C  SUBROUTINE TO COUNT POSITIVE ELEMENTS
C          IN AN ARRAY
       SUBROUTINE COUNTP(ARRAY,I,CNT)
       DIMENSION ARRAY(7)
       CNT = Ø
       DO 9 J=1,I
       IF(ARRAY(J))9,5,5
   9   CONTINUE
       RETURN
   5   CNT = CNT+1.Ø
       GO TO 9
       END
```

9.9 REFERENCING A SUBROUTINE SUBPROGRAM

A SUBROUTINE subprogram is called by using a CALL statement. A CALL statement has one of the following forms:

CALL s(a1,a2,...,an)

or

CALL s

where s is either a SUBROUTINE subprogram name or the name of a subprogram that is external to the executable program, and the ai are the actual arguments to be used by the subprogram. The ai must agree in type, order and number with the corresponding dummy arguments in the subprogram-defining SUBROUTINE statement.

The arguments in a CALL statement must comply with the following rules:

1. FUNCTION and SUBROUTINE names appearing in the argument list must have appeared in an EXTERNAL statement in the calling program.

2. A CALL statement referencing a SUBROUTINE containing a variable array declarator must contain as arguments the actual name of the array and the actual dimension specifications.

3. If an item in the SUBROUTINE subprogram dummy argument list is an array, the corresponding item in the CALL statement argument list must be an array.

When a SUBROUTINE subprogram is called, program control goes to the first executable statement following the SUBROUTINE statement.

Example:

```
      DIMENSION DATA(10)
      .
      .
      .
C  THE STATEMENT BELOW CALLS THE
C       SUBROUTINE IN THE PREVIOUS PARAGRAPH
C
      CALL COUNTP(DATA,10,CPOS)
```

9.10 RETURN FROM FUNCTION AND SUBROUTINE SUBPROGRAMS

The logical termination of a FUNCTION or SUBROUTINE subprogram is a RETURN statement which transfers control back to the calling program. The general form of the RETURN statement is simply the word

RETURN

The following rules govern the use of the RETURN statement:

1. There must be at least one RETURN statement in each SUBROUTINE or FUNCTION subprogram.

2. RETURN from a FUNCTION subprogram is to the instruction sequence of the calling program following the FUNCTION reference.

3. RETURN from a SUBROUTINE subprogram is to the first executable statement in the calling program following the CALL statement.

4. Upon return from a FUNCTION subprogram the single-valued result of the subprogram is available to the evaluation of the expression from which the FUNCTION call was made.

5. Upon return from a SUBROUTINE subprogram any values assigned to the dummy arguments by statements in the SUBROUTINE are available for use by the calling program.

Example:

```
      Calling Program Unit

      .
      .
      .
      CALL SUBR(Z9,B7,R1)
      .
      .
      .

      Called Program Unit

      SUBROUTINE SUBR(A,B,C)
      READ(3,7) B
      A = B**C
      RETURN
  7   FORMAT(F9.2)
      END
```

In this example, new data in Z9 and B7 are made available to the calling program when the RETURN occurs.

9.11 PROCESSING ARRAYS IN SUBPROGRAMS

An array may be passed from a calling program to a subprogram by specifying the array name in the argument list in the CALL statement. The corresponding dummy argument in the argument list of the subprogram must be an array name that appears in an array declarator in the subprogram. The array declarator in the subprogram must specify the dimensioning information necessary to use the array.

An array element may be passed to a subprogram by specifying it in the argument list in the CALL statement. The corresponding dummy argument must not be another array element.

Dimensioning information may be passed to the subprogram by assigning values to INTEGER variables specified in the argument list in the CALL statement. The corresponding dummy arguments must be INTEGER variables, and these variables are used in the variable array declarator that specifies the dimensionality of the array. A variable array

declarator is an array declarator containing INTEGER variables in place of the INTEGER constants used in constant array declarators.
The values of the INTEGER variables used in the variable array declarator must not be changed within the subprogram.
A variable array declarator can be used only when the array name and each INTEGER variable appear as dummy arguments in the argument list of the subprogram.

Examples:

The following subprogram computes the average of the elements of any one dimension array containing at most 50 elements.

Calling program Unit

```
REAL Z1(50), Z2(25)
      .
      .
   A1 = AVG(Z1,50)
   A2 = A1 - AVG(Z2,25)
      .
      .
   END
```

Calling Program Unit

```
REAL FUNCTION AVG (ARG,I)
   REAL
   ARG(50)
   SUM = SUM + ARG(J)
   AVG = SUM/FLOAT(I)
   RETURN
   END
```

The following example illustrates the use of variable array declarators.

Calling Program Unit

```
      REAL A(3,4,5)
           .
           .
         CALL VARDEC (A,3,4,5)
           .
           .
         END
```

Called Program Unit

```
      SUBROUTINE VARDEC (X,I,J,K)
      REAL X(I,J,K)
           .
           .
         RETURN
         END
```

9.12 BLOCK DATA SUBPROGRAMS

A BLOCK DATA subprogram is a program unit that begins with a BLOCK DATA statement. A BLOCK DATA subprogram effects its purpose simply by appearing in an executable program--it is not executable and is not referenced either as a function or as a subroutine. The only purpose of a BLOCK DATA subprogram is to provide initial values for data in COMMON storage at the start of program execution.

The BLOCK DATA statement has the form;

BLOCK DATA subprogram-name

where subprogram-name is optional; when present, subprogram-name is 1-6 characters long, of which the first must be a letter.

The following rules govern the form of a BLOCK DATA subprogram:

1. A BLOCK DATA subprogram must begin with a BLOCK DATA statement.

2. The only statements that may appear in a BLOCK DATA subprogram are:

   Type
   EQUIVALENCE
   DATA
   COMMON
   DIMENSION statements.

3. To initialize one data item of a COMMON block, all elements specified to be in that COMMON block by COMMON statements in other program units must be listed in the COMMON statement in the BLOCK DATA subprogram.

4. One BLOCK DATA subprogram may initialize data items from more than one COMMON block.

5. More than one BLOCK DATA subprogram may appear in a single program. Each BLOCK DATA subprogram must have a unique name.

6. A data item initialized in one BLOCK DATA subprogram should not be initialized in another BLOCK DATA subprogram.

7. The last statement appearing in a BLOCK DATA subprogram must be the END statement.

Example:

```
      BLOCK DATA
      LOGICAL A1
      COMMON/BETA/B(3,3)/GAM/C(4)
      COMMON/ALPHA/A1,C,E,D
      DATA
     -  B
     -     /1.1,2.5,3.8,3*4.96,
     -      2*Ø.52,1.1/
     -  C
     -     /1.2E+Ø2,3*4.Ø1/
      DATA
     -  A1
     -     /.TRUE./
     -  E
     -     /-5.6/
      END
```

APPENDIX A

Language Extensions and Restrictions

Cromemco FORTRAN language includes the following extensions to ANSI Standard FORTRAN (X3.9-1966).

1. Random (direct access) disk files are supported with the REC= clause in the READ and WRITE statements.

2. If c is used in a 'STOP c' or 'PAUSE c' statement, c may be any six ASCII characters.

3. Error and End-of-File branches may be specified in READ and WRITE statements using the ERR= and END= options.

4. The standard subprograms PEEK, POKE, INP, and OUT have been added to the FORTRAN library.

5. Statement functions may use subscripted variables.

6. Hexadecimal constants may be used wherever Integer constants are normally allowed.

7. The literal form of Hollerith data (character string between apostrophe characters) is permitted in place of the standard nH form.

8. Holleriths and Literals are allowed in expressions in place of Integer constants.

9. There is no restriction to the number of continuation lines.

10. Mixed mode expressions and assignments are allowed, and conversions are done automatically.

Cromemco FORTRAN places the following restrictions upon Standard FORTRAN.

1. The COMPLEX data type is not implemented.

2. The specification statements must appear in the following order:

1. PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA
2. Type, EXTERNAL, DIMENSION
3. COMMON
4. EQUIVALENCE
5. DATA
6. Statement Functions

3. A different amount of computer memory is allocated for each of the data types: INTEGER, REAL, DOUBLE PRECISION, LOGICAL.

4. The equal sign of a replacement statement and the first comma of a DO statement must appear on the initial statement line.

Descriptions of these language extensions and restrictions are included at the appropriate points in the text of this document.

APPENDIX B

I/O Interface

Input/Output operations are table-dispatched to the driver routine for the proper Logical Unit Number. $LUNTB is the dispatch table. It contains one 2-byte driver address for each possible LUN. It also has a one-byte entry at the beginning, which contains the maximum LUN plus one. The initial FORTRAN package provides for 10 LUN's (1 - 10), of which numbers 1, 3, 4, and 5 are assigned to the console or CRT, number 2 is assigned to the line printer, and numbers 6 through 10 are assigned to disk files. Any of these may be redefined by the user or more added, simply by changing the appropriate entries in $LUNTB and adding more drivers. The runtime system uses LUN 3 for errors and other user communication; therefore, LUN 3 should always be left to correspond to the operator console ($DRV3). The initial structure of $LUNTB is shown in the listing which follows.

```
CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0001
*** TABLE WITH DRIVER ADDRESSES FOR LUN'S 1 THROUGH 10 ***

                    0002          ENTRY   $LUNTB
                    0003          EXT     $DRV3,LPTDRV,DSKDRV
                    0004 ;
                    0005 ;
0000' 0B            0006 $LUNTB:  DEFB    11              ; MAX. LUN + 1
0001' 0000#         0007          DEFW    $DRV3           ; LOGICAL UNIT NUMBER 1
0003' 0000#         0008          DEFW    LPTDRV          ; LUN 2
0005' 0100#         0009          DEFW    $DRV3           ; LUN 3
0007' 0500#         0010          DEFW    $DRV3           ; LUN 4
0009' 0700#         0011          DEFW    $DRV3           ; LUN 5
000B' 0000#         0012          DEFW    DSKDRV          ; LUN 6
000D' 0B00#         0013          DEFW    DSKDRV          ; LUN 7
000F' 0D00#         0014          DEFW    DSKDRV          ; LUN 8
0011' 0F00#         0015          DEFW    DSKDRV          ; LUN 9
0013' 1100#         0016          DEFW    DSKDRV          ; LUN 10
0015' (0000)        0017          END

Errors              0

Program Length  0015 (21)



CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0002
SYMBOL TABLE

$DRV3   0009#   $LUNTB  0000'   DSKDRV  0013#   LPTDRV  0003#
```

The LUN Table is located in the FORTRAN Library file, FORLIB.REL, under the name $LUNTB, as mentioned above. When linking FORTRAN programs, the Linker will be instructed to search FORLIB to satisfy any undefined symbols. LINK then loads these needed routines into memory. However, if $LUNTB is defined prior to the search of FORLIB, the Linker would then not load it from FORLIB.

This is the procedure which must be followed if the user desires to change the LUN assignments. The new LUN Table is composed using the same name ($LUNTB), it is assembled using the Cromemco Macro Assembler (ASMB) to create a REL object file, and finally it is linked prior to FORLIB. This procedure is demonstrated below; the following listing is a sample alternate LUN Table.

```
CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0001
*** TABLE WITH DRIVER ADDRESSES FOR LUN'S 1 THROUGH 20 ***

                    0002          ENTRY   $LUNTB
                    0003          EXT     $DRV3,LPTDRV,DSKDRV,SPTDRV
                    0004 ;
                    0005 ;
0000' 15            0006 $LUNTB:  DEFB    21              ; 20 LOGICAL UNIT NUMBERS
0001' 0000#         0007          DEFW    SPTDRV          ; LUN 1  (SERIAL PRINTER)
0003' 0000#         0008          DEFW    LPTDRV          ; LUN 2  (PARALLEL   "  )
0005' 0000#         0009          DEFW    $DRV3           ; LUN 3  (CONSOLE/CRT)
0007' 0500#         0010          DEFW    $DRV3           ; LUN 4  (     "     )
0009' 0700#         0011          DEFW    $DRV3           ; LUN 5  (     "     )
000B' 0000#         0012          DEFW    DSKDRV          ; LUN 6  (DISK)
000D' 0B00#         0013          DEFW    DSKDRV          ; LUN 7  ( "  )
000F' 0D00#         0014          DEFW    DSKDRV          ; LUN 8  ( "  )
0011' 0F00#         0015          DEFW    DSKDRV          ; LUN 9  ( "  )
0013' 1100#         0016          DEFW    DSKDRV          ; LUN 10 ( "  )
0015' 1300#         0017          DEFW    DSKDRV          ; LUN 11 ( "  )
0017' 1500#         0018          DEFW    DSKDRV          ; LUN 12 ( "  )
0019' 1700#         0019          DEFW    DSKDRV          ; LUN 13 ( "  )
001B' 1900#         0020          DEFW    DSKDRV          ; LUN 14 ( "  )
001D' 1B00#         0021          DEFW    DSKDRV          ; LUN 15 ( "  )
001F' 1D00#         0022          DEFW    DSKDRV          ; LUN 16 ( "  )
0021' 1F00#         0023          DEFW    DSKDRV          ; LUN 17 ( "  )
0023' 2100#         0024          DEFW    DSKDRV          ; LUN 18 ( "  )
0025' 2300#         0025          DEFW    DSKDRV          ; LUN 19 ( "  )
0027' 2500#         0026          DEFW    DSKDRV          ; LUN 20 ( "  )
0029' (0000)        0027          END

Errors              0

Program Length  0029 (41)


CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0002
SYMBOL TABLE

$DRV3   0009#   $LUNTB  0000'   DSKDRV  0027#   LPTDRV  0003#   SPTDRV  0001#
```

Note the use of the ENTRY statement to define the module to ASMB and LINK. The symbols $DRV3, LPTDRV, and DSKDRV stand for the present FORTRAN CRT driver, line printer driver, and disk driver modules, respectively. In addition the user has added an EXTernal declaration for a new serial line printer driver, SPTDRV. Also note that 1Ø additional disk file LUN's have been added to the list. The first byte of $LUNTB has been changed to allow for these additional Unit Numbers. LUN 3 has been preserved as the console driver as that is the one used by FORTRAN for error messages.

The serial line printer driver declared above might be quite similar to the listing on the following page (note that what is given here is necessary but not sufficient).

```
CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0001
***** FORTRAN SERIAL LINE PRINTER I/O DRIVER *****

                    0002          ENTRY   SPTDRV
                    0003          EXT     $IOERR,$BL,$BF,. . .
                    0004 ;
                    0005 ;
                    0006 ; *** DISPATCH TABLE ***
                    0007 ;
0000' 1300'         0008 SPTDRV: DEFW     SPDVFR        ; FORMATTED READ
0002' 1600'         0009         DEFW     SPDVFW        ; FORMATTED WRITE
0004' 1000'         0010         DEFW     SPDVBR        ; BINARY READ
0006' 1000'         0011         DEFW     SPDVBW        ; BINARY WRITE
0008' 0E00'         0012         DEFW     SPDVRE        ; REWIND
000A' 0E00'         0013         DEFW     SPDVBA        ; BACKSPACE
000C' 0E00'         0014         DEFW     SPDVEN        ; ENDFILE
                    0015 ;
                    0016 ;
                    0017 ; *** REWIND, BACKSPACE, & ENDFILE ***
                    0018 ;
                    0019 SPDVRE:
                    0020 SPDVBA:
000E' AF            0021 SPDVEN: XOR      A             ; THESE OPERATIONS ARE
000F' C9            0022         RET                    ;  NO-OPS FOR SERIAL PRINTER
                    0023 ;
                    0024 ;
                    0025 ; *** BINARY READ & WRITE ***
                    0026 ;
                    0027 SPDVBR:
0010' C30000#       0028 SPDVBW: JP       $IOERR        ; ILLEGAL OPERATIONS
                    0029                                ;  (PRINT ERROR AND RETURN)
                    0030 ;
                    0031 ;
                    0032 ; *** FORMATTED READ ***
                    0033 ;
0013' C31100#       0034 SPDVFR: JP       $IOERR        ; ILLEGAL OPERATION
                    0035                                ;  (PRINT ERROR AND RETURN)
                    0036 ;
                    0037 ;
                    0038 ; *** FORMATTED WRITE ***
                    0039 ;
                    0040 SPDVFW: . . .
                    0041         :
                    0042         :
                    0043         END

Errors              0

Program Length  0016 (22)


CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0002
SYMBOL TABLE

$IOERR  0014#   SPDVBA  000E'   SPDVBR  0010'   SPDVBW  0010'   SPDVEN  000E'
SPDVFR  0013'   SPDVFW  0016'   SPDVRE  000E'   SPTDRV  0000'
```

The LUN Table and additional driver which have been created as a hypothetical example above could now be assembled with the command lines

```
ASMB LUNTB
ASMB SPTDRIVR
```

where LUNTB.Z8Ø and SPTDRIVR.Z8Ø are the names of the source files on disk. ASMB will create REL files for both these modules which can now be linked to the original compiled FORTRAN source program, FORPROG.REL, by the line

```
LINK FORPROG,LUNTB,SPTDRIVR,FORLIB/S/E
```

The Linker will search FORLIB but will ignore the $LUNTB file there because LUNTB.REL was linked first and contained the same name as the original module, $LUNTB.

In addition to the dispatch table represented by $LUNTB, each of the device drivers must contain a local dispatch table (as shown in the serial line printer driver above). Note that $LUNTB contains only one address for each device, yet there are really seven possible operations per device:

1) Formatted Read
2) Formatted Write
3) Binary Read
4) Binary Write
5) Rewind
6) Backspace
7) Endfile

Each device driver contains up to seven routines. The starting addresses of each of these seven routines are placed at the beginning of the driver in the exact order listed above. The entry in $LUNTB then points to this local table, and the runtime system indexes into it to get the address of the appropriate routine to handle the requested I/O operation.

The following conventions apply to the individual I/O routines:

1. Location $BF contains the data buffer address for READs and WRITEs.

2. The number of bytes to be written for a WRITE is contained in location $BL.

3. The number of bytes actually read for a READ should be returned in location $BL.

4. All I/O operations should set the condition codes before exiting to indicate a normal return, an error condition, or an end-of-file condition:

    a) Carry flag reset (Ø) and Zero flag set (1) - normal return
    b) Carry flag set (1) - I/O error
    c) Carry and Zero flags both reset (Ø) - end-of-file encountered

    The runtime system checks the condition codes after calling the driver. If they indicate an I/O error or EOF, control is passed to the label specified by "ERR=" or "END=", respectively, or, if no label is specified, a fatal error results.

5. $IOERR is a global routine which prints the message "ILLEGAL I/O OPERATION" (non-fatal). This routine may be used if there are some operations not to be allowed on a particular device (e.g., binary I/O on a CRT).

6. The I/O buffer has a fixed maximum length of 128 bytes. If a driver allows an input operation to write past the end of the buffer, essential runtime variables may be affected. The consequences of this are unpredictable and undesirable.

The listing which follows contains the CRT driver ($DRV3) provided with the FORTRAN runtime package. REWIND, BACKSPACE, and ENDFILE perform no operation (flags are cleared to indicate EOF) and Binary I/O gives an error message.

```
CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0001
***** FORTRAN CRT I/O DRIVER *****

                    0002          ENTRY   $DRV3
                    0003          EXT     $IOERR,$BL,$BF,$ERR,$TTYIN,$TTYOT
                    0004 ;
      (000A)        0005 LF:      EQU     0AH             ; ASCII LINE FEED
      (000C)        0006 FORM:    EQU     0CH             ; ASCII FORM FEED
      (000D)        0007 CR:      EQU     0DH             ; ASCII CARRIAGE RETURN
      (0012)        0008 IRECER:  EQU     12H             ; INPUT RECORD TOO LONG
                    0009 ;
                    0010 ;
                    0011 ; *** DISPATCH TABLE ***
                    0012 ;
0000' 1300'         0013 $DRV3:   DEFW    DRV3FR          ; FORMATTED READ
0002' 4200'         0014          DEFW    DRV3FW          ; FORMATTED WRITE
0004' 1000'         0015          DEFW    DRV3BR          ; BINARY READ
0006' 1000'         0016          DEFW    DRV3BW          ; BINARY WRITE
0008' 0E00'         0017          DEFW    DRV3RE          ; REWIND
000A' 0E00'         0018          DEFW    DRV3BA          ; BACKSPACE
000C' 0E00'         0019          DEFW    DRV3EN          ; ENDFILE
                    0020 ;
                    0021 ;
                    0022 ; *** REWIND, BACKSPACE, & ENDFILE ***
                    0023 ;
                    0024 DRV3RE:
                    0025 DRV3BA:
000E' AF            0026 DRV3EN:  XOR     A               ; THESE OPERATIONS ARE
000F' C9            0027          RET                     ;   NO-OPS FOR CRT
                    0028 ;
                    0029 ;
                    0030 ; *** BINARY READ & WRITE ***
                    0031 ;
                    0032 DRV3BR:
0010' C30000#       0033 DRV3BW:  JP      $IOERR          ; ILLEGAL OPERATIONS
                    0034                                  ;   (PRINT ERROR AND RETURN)
```

```
CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0002
***** FORTRAN CRT I/O DRIVER *****

                    0036 ;
                    0037 ; *** FORMATTED READ ***
                    0038 ;
0013' AF            0039 DRV3FR: XOR     A
0014' 320000#       0040         LD      ($BL),A       ; ZERO BUFFER LENGTH
0017' CD0000#       0041 DR3FR1: CALL    $TTYIN        ; INPUT A CHARACTER
001A' E67F          0042         AND     7FH           ; STRIP OFF PARITY
001C' FE0A          0043         CP      LF            ; IGNORE LINE FEEDS
001E' CA1700'       0044         JP      Z,DR3FR1
0021' F5            0045         PUSH    AF            ; SAVE THE CHARACTER
0022' 2A1500#       0046         LD      HL,($BL)      ; GET CHAR. POSIT. IN BUFFER
0025' 2600          0047         LD      H,0           ; LIMIT TO VALUE OF 255
0027' EB            0048         EX      DE,HL         ; SAVE OFFSET IN DE
0028' 2A0000#       0049         LD      HL,($BF)      ; GET BUFFER ADDRESS
002B' 19            0050         ADD     HL,DE         ; ADD OFFSET
002C' F1            0051         POP     AF            ; RESTORE CHARACTER
002D' 77            0052         LD      (HL),A        ; PUT IT IN BUFFER
002E' 13            0053         INC     DE            ; INCREMENT $BL
002F' EB            0054         EX      DE,HL
0030' 222300#       0055         LD      ($BL),HL      ; SAVE NEW VALUE
0033' FE0D          0056         CP      CR            ; CHECK FOR CARRIAGE RETURN
0035' C8            0057         RET     Z             ; DONE IF YES
0036' 7D            0058         LD      A,L           ; GET NEXT CHARACTER IF NO
0037' FE80          0059         CP      80H           ; CHECK FOR MAX. LINE LENGTH
0039' DA1700'       0060         JP      C,DR3FR1      ; GET NEXT CHARACTER IF OK
003C' CD0000#       0061         CALL    $ERR          ; ERROR IF LINE TOO LONG
003F' 12            0062         DEFB    IRECER        ; ERROR NO. FOR $ERR ROUTINE
                    0063
0040' AF            0064         XOR     A             ; CLEAR FLAGS AND RETURN
0041' C9            0065         RET
```

```
CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0003
***** FORTRAN CRT I/O DRIVER *****

                    0067 ;
                    0068 ; *** FORMATTED WRITE ***
                    0069 ;
0042' 3A3100#       0070 DRV3FW: LD      A,($BL)       ; GET BUFFER LENGTH
0045' B7            0071         OR      A
0046' C8            0072         RET     Z             ; RETURN IF EMPTY BUFFER
0047' 2A2900#       0073         LD      HL,($BF)      ; GET BUFFER ADDRESS
004A' 3D            0074         DEC     A             ; DECREMENT LINE LENGTH
004B' F5            0075         PUSH    AF            ;   AND SAVE LENGTH
004C' 3E0D          0076         LD      A,CR          ; OUTPUT CARRIAGE
004E' CD0000#       0077         CALL    $TTYOT        ;   RETURN FIRST
0051' 7E            0078         LD      A,(HL)        ; GET FIRST CHARACTER
                    0079                               ;   FROM BUFFER
0052' FE2B          0080         CP      '+'
0054' CA7900'       0081         JP      Z,DR3FW2      ; NO LINE FEEDS COMMAND
0057' FE31          0082         CP      '1'
0059' C26400'       0083         JP      NZ,DR3FW1     ; SKIP IF NOT FORM FEED
005C' 3E0C          0084         LD      A,FORM        ; FORM FEED COMMAND
005E' CD4F00#       0085         CALL    $TTYOT        ; OUTPUT IT
0061' C37900'       0086         JP      DR3FW2        ; LOOP FOR NEXT CHARACTER
                    0087 ;
0064' 3E0A          0088 DR3FW1: LD      A,LF          ; LINE FEED TO NEXT LINE
0066' CD5F00#       0089         CALL    $TTYOT
0069' 7E            0090         LD      A,(HL)        ; GET CHARACTER BACK
006A' FE20          0091         CP      ' '
006C' CA7900'       0092         JP      Z,DR3FW2      ; LEAVE IT AT ONE LF
006F' FE30          0093         CP      '0'
0071' C27900'       0094         JP      NZ,DR3FW2     ; JUMP FOR NO DOUBLE LF'S
0074' 3E0A          0095         LD      A,LF          ; LINE FEED TO NEXT LINE
0076' CD6700#       0096         CALL    $TTYOT
                    0097 ;
0079' F1            0098 DR3FW2: POP     AF            ; RESTORE BUFFER LENGTH
007A' 23            0099         INC     HL            ; INCREMENT POINTER
007B' C8            0100 DR3FW3: RET     Z             ; RETURN IF LINE LENGTH ZERO
007C' F5            0101         PUSH    AF            ; SAVE CHARACTER COUNT
007D' 7E            0102         LD      A,(HL)        ; GET NEXT CHARACTER
007E' 23            0103         INC     HL            ; INCREMENT POINTER
007F' CD7700#       0104         CALL    $TTYOT        ; OUTPUT CHARACTER
0082' F1            0105         POP     AF            ; GET COUNT
0083' 3D            0106         DEC     A             ;   AND DECREMENT IT
0084' C37B00'       0107         JP      DR3FW3        ; LOOP UNTIL ALL
                    0108                               ;   CHARACTERS ARE OUTPUT
0087' (0000)        0109         END

Errors              0

Program Length  0087 (135)


CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0004
SYMBOL TABLE

$BF     0048#   $BL     0043#   $DRV3   0000'   $ERR    003D#   $IOERR  0011#
$TTYIN  0018#   $TTYOT  0080#   CR      000D    DR3FR1  0017'   DR3FW1  0064'
DR3FW2  0079'   DR3FW3  007B'   DRV3BA  000E'   DRV3BR  0010'   DRV3BW  0010'
DRV3EN  000E'   DRV3FR  0013'   DRV3FW  0042'   DRV3RE  000E'   FORM    000C
IRECER  0012    LF      000A
```

```
CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0005
CROSS REFERENCE LISTING

$BF     0003  0049 0073
$BL     0003  0040 0046 0055 0070
$DRV3   0002 #0013
$ERR    0003  0061
$IOERR  0003  0033
$TTYIN  0003  0041
$TTYOT  0003  0077 0085 0089 0096 0104
CR      0007  0056 0076
DR3FR1  0041  0044 0060
DR3FW1  0088  0083
DR3FW2  0098  0081 0086 0092 0094
DR3FW3  0100  0107
DRV3BA  0025  0018
DRV3BR  0032  0015
DRV3BW  0033  0016
DRV3EN  0026  0019
DRV3FR  0039  0013
DRV3FW  0070  0014
DRV3RE  0024  0017
FORM    0006  0084
IRECER  0008  0062
LF      0005  0043 0088 0095
```

APPENDIX C

Subprogram Linkages

This appendix defines a normal subprogram call as generated by the FORTRAN compiler. It is included to facilitate linkages between FORTRAN programs and those written in other languages, such as the assembly language for the Cromemco Macro Assembler.

A subprogram reference with no parameters generates a simple "CALL" instruction. The corresponding subprogram should return via a simple RET (CALL and RET are Z8Ø opcodes - see the Cromemco Macro Assembler Manual or Z8Ø reference manual for explanations.)

A subprogram reference with parameters results in a somewhat more complex calling sequence. Parameters are always passed by reference (i.e., the thing passed is actually the address of the low byte of the actual argument). Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing the parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers.
   Parameter 1 will be in HL,
   Parameter 2 will be in DE,
   Parameter 3 will be in BC.

2. If the number of parameters is greater than 3, they are passed as follows:

   1. Parameter 1 in HL.

   2. Parameter 2 in DE.

   3. Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that, with this scheme, the subprogram must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. Neither the compiler nor the runtime system checks for the correct number of parameters.

If the subprogram expects more than 3 parameters, and needs to transfer them to a local data area, there is a FORTRAN system subroutine which will perform this transfer. This argument transfer routine is named $AT, and is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (i.e., the total number of arguments minus 2). The subprogram is responsible for saving the first two parameters before calling $AT. For example, if a subprogram expects 5 parameters, it should look like:

```
SUBR:   LD      (P1),HL ;SAVE PARAMETER 1 POINTER
        EX      DE,HL
        LD      (P2),HL ;SAVE PARAMETER 2 POINTER
        LD      A,3     ;NO. OF PARAMETERS LEFT
        LD      HL,P3   ;POINTER TO LOCAL AREA
        CALL    $AT     ;TRANSFER THE OTHER 3
                        ;  PARAMETER POINTERS
        .
        .
        .
        [Body of subprogram]
        .
        .
        .
        RET             ;RETURN TO CALLING ROUTINE
P1:     DS      2       ;SPACE FOR PARAMETER 1
P2:     DS      2       ;SPACE FOR PARAMETER 2
P3:     DS      6       ;SPACE FOR PARAMETERS 3-5
```

When accessing parameters in a subprogram, don't forget that they are pointers to the actual arguments passed.

Also, it is entirely up to the programmer to see to it that the arguments in the calling program match in number, type, and length with the parameters expected by the subprogram. This applies to FORTRAN subprograms, as well as those written in assembly language.

FORTRAN Functions (Section 9) return their values in registers or memory depending upon the type. Logical results are returned in (A), Integers in (HL), Reals in memory at $AC, Double Precision in memory at $DAC. $AC and $DAC are the addresses of the low bytes of the mantissas.

APPENDIX D

ASCII CHARACTER CODES

DEC.	HEX	CHAR.	DEC.	HEX	CHAR.	DEC.	HEX	CHAR.
000	00	NUL (CTRL-@)	043	2B	+	086	56	V
001	01	SOH (CTRL-A)	044	2C	,	087	57	W
002	02	STX (CTRL-B)	045	2D	-	088	58	X
003	03	ETX (CTRL-C)	046	2E	.	089	59	Y
004	04	EOT (CTRL-D)	047	2F	/	090	5A	Z
005	05	ENQ (CTRL-E)	048	30	0	091	5B	[
006	06	ACK (CTRL-F)	049	31	1	092	5C	\
007	07	BEL (CTRL-G)	050	32	2	093	5D	]
008	08	BS	051	33	3	094	5E	^
009	09	HT	052	34	4	095	5F	(LA)
010	0A	LF	053	35	5	096	60	'
011	0B	VT	054	36	6	097	61	a
012	0C	FF	055	37	7	098	62	b
013	0D	CR	056	38	8	099	63	c
014	0E	SO (CTRL-N)	057	39	9	100	64	d
015	0F	SI (CTRL-O)	058	3A	:	101	65	e
016	10	DLE (CTRL-P)	059	3B	;	102	66	f
017	11	DC1 (CTRL-Q)	060	3C	<	103	67	g
018	12	DC2 (CTRL-R)	061	3D	=	104	68	h
019	13	DC3 (CTRL-S)	062	3E	>	105	69	i
020	14	DC4 (CTRL-T)	063	3F	?	106	6A	j
021	15	NAK (CTRL-U)	064	40	@	107	6B	k
022	16	SYN (CTRL-V)	065	41	A	108	6C	l
023	17	ETB (CTRL-W)	066	42	B	109	6D	m
024	18	CAN (CTRL-X)	067	43	C	110	6E	n
025	19	EM (CTRL-Y)	068	44	D	111	6F	o
026	1A	SUB (CTRL-Z)	069	45	E	112	70	p
027	1B	ESC (CTRL-[)	070	46	F	113	71	q
028	1C	FS (CTRL-\)	071	47	G	114	72	r
029	1D	GS (CTRL-])	072	48	H	115	73	s
030	1E	RS (CTRL-^)	073	49	I	116	74	t
031	1F	US (CTRL-_)	074	4A	J	117	75	u
032	20	(SPACE)	075	4B	K	118	76	v
033	21	!	076	4C	L	119	77	w
034	22	"	077	4D	M	120	78	x
035	23	#	078	4E	N	121	79	y
036	24	$	079	4F	O	122	7A	z
037	25	%	080	50	P	123	7B	{
038	26	&	081	51	Q	124	7C	|
039	27	'	082	52	R	125	7D	}
040	28	(	083	53	S	126	7E	~
041	29	)	084	54	T	127	7F	DEL
042	2A	*	085	55	U			

LF=Line Feed FF=Form Feed CR=Carriage Return DEL=Rubout
ESC=ESCAPE (LA)=Left Arrow

APPENDIX E

FORTRAN Library Subroutines

The FORTRAN library (FORLIB.REL) contains a number of subroutines that may be referenced by the user from FORTRAN or assembly programs. In the following descriptions, $AC refers to the floating accumulator; $AC is the address of the low byte of the mantissa. $AC+3 is the address of the exponent. $DAC refers to the DOUBLE PRECISION accumulator; $DAC is the address of the low byte of the mantissa. $DAC+7 is the address of the DOUBLE PRECISION exponent.

All arithmetic routines (addition, subtraction, multiplication, division, exponentiation) adhere to the following calling conventions.

1. Argument 1 is passed in the registers:
   Integer in HL
   Real in $AC
   Double in $DAC

2. Argument 2 is passed either in registers or in memory, depending upon the type:

   a. Integers are passed in HL, or DE if HL contains Argument 1.

   b. Real and Double Precision values are passed in memory pointed to by HL. (HL points to the low byte of the mantissa.)

The following arithmetic routines are contained in the Library:

Function	Name	Argument 1 Type	Argument 2 Type
Addition	$AA	Real	Integer
	$AB	Real	Real
	$AQ	Double	Integer
	$AR	Double	Real
	$AU	Double	Double
Division	$D9	Integer	Integer
	$DA	Real	Integer
	$DB	Real	Real
	$DQ	Doulbe	Integer
	$DR	Double	Real
	$DU	Double	Double
Exponentiation	$E9	Integer	Integer
	$EA	Real	Integer
	$EB	Real	Real
	$EQ	Double	Integer
	$ER	Double	Real
	$EU	Double	Double
Multiplication	$M9	Integer	Integer
	$MA	Real	Integer
	$MB	Real	Real
	$MQ	Double	Integer
	$MR	Double	Real
	$MU	Double	Double
Subtraction	$SA	Real	Integer
	$SB	Real	Real
	$SQ	Double	Integer
	$SR	Double	Real
	$SU	Double	Double

Additional Library routines are provided for converting between value types. Arguments are always passed to and returned by these conversion routines in the appropriate registers.

Logical in A

Integer in HL

Real in $AC

Double in $DAC

Function	Name
Integer to Real	$CA
Integer to Double	$CC
Real to Integer	$CH
Real to Logical	$CJ
Real to Double	$CK
Double to Integer	$CX
Double to Real	$CY
Double to Logical	$CZ

PART 2

FORTRAN USER'S MANUAL

SECTION 1

Compiling FORTRAN Programs

The FORTRAN compiler produces object code in relocatable form. This means that any addressing instructions in the object program, such as branches and subprogram calls, use relative addresses, not actual memory addresses. The advantage to this is that the program may be loaded into memory and executed at virtually any address, and that data areas can be separate from program areas.

A program called LINK is used to load the program into memory and change the relative addresses to actual memory addresses so that the program can be executed. LINK contains the facility to save the resulting file on disk, so this loading/linking operation need be done only once. See Section 3 of this part for a complete discussion of LINK.

1.1 FORTRAN Command Scanner

To tell the FORTRAN compiler what to compile and with which options, it is necessary to input a Command String, which is read by the FORTRAN command scanner.

1.1.1 Format of Commands

To run FORTRAN, type FORTRAN followed by a carriage return. FORTRAN will return the prompt "*", indicating it is ready to accept commands. The general format of a FORTRAN command string is:

```
objprog-dev:filename.ext,list-dev:filename.ext=
   source-dev:filename.ext
```

objprog-dev:
The device on which the object program is to be written.

list-dev:
The device on which the program listing is written.

source-dev:

The device from which the source-program input to FORTRAN is obtained. If a device name is omitted, it defaults to the currently selected drive.

filename.ext
The filename and filename extension of the object program file, the listing file, and the source file. If omitted, file name extensions will default as follows:

object file: REL
listing file: PRN
source file: FOR

Either the object file or the listing file or both may be omitted. If neither a listing file nor an object file is desired, place only a comma to the left of the equal sign. If the names of the object file and the listing file are omitted, the default is the name of the source file.

Examples: the asterisk (*) is the prompt from FORTRAN

*=TEST	Compile the program TEST.FOR and place the object in TEST.REL
*,TTY:=TEST	Compile the program TEST.FOR and list program on the terminal. No object code is generated.
*TESTOBJ=TEST.FOR	Compile the program TEST.FOR and put the object code in TESTOBJ.REL
*TEST,TEST=TEST	Compile TEST.FOR, put the object code in TEST.REL and the listing in TEST.PRN
*,=TEST.FOR	Compile TEST.FOR but produce no object or listing file. Useful for checking for errors.

1.1.2 FORTRAN Compilation Switches

A number of different switches may be given in the command string that will affect the format of the listing file. Each switch should be preceded by a slash (/):

Switch	Action
O	Print all listing addresses, etc. in octal.
H	Print all listing addresses, etc. in hexadecimal. (Set by default)
N	Do not list generated code.
R	Force generation of an object file.
L	Force generation of a listing file.
P	Each /P allocates an extra 100 bytes of stack space for use during compilation. Use /P if stack overflow errors occur during compilation, otherwise not needed.
M	Specifies to the compiler that the generated code should be in a form which can be loaded into ROMs. When a /M is specified, the generated code will differ from normal in the following ways: 1. FORMATs will be placed in the program area, with a "JMP" around them. 2. Parameter blocks (for subprogram call with more than 3 parameters) will be initialized at runtime, rather than being initialized by the loader.

Examples:

*,TTY:=MYPROG/N	Compile file MYPROG.FOR and list program on terminal but without generated code.
*=TEST/L	Compile TEST.FOR with object file TEST.REL and listing file TEST.PRN

*=BIGGONE/P/P Compile file BIGGONE.FOR and produce object file BIGGONE.REL. Compiler is allocated 200 extra bytes of stack space.

NOTE

If a FORTRAN program is intended for ROM, the programmer should be aware of the following ramifications:

1. DATA statements should not be used to initialize RAM. Such initialization is done by the loader, and will therefore not be present at execution. Variables and arrays may be initialized during execution via assignment statements, or by READing into them.

2. FORMATs should not be read into during execution.

3. If the standard library I/O routines are used, DISK files should not be OPENed on any LUNs other than 6, 7, 8, 9, 10. If other LUNs are needed for Disk I/O, $LUNTB should be recompiled with the appropriate addresses pointing to the Disk driver routine.

A library routine, $INIT, sets the stack pointer at the top of available memory (as indicated by the operating system) before execution begins.

The calling convention is:

```
LD   BC,<return address>
JP   $INIT
```

The FORTRAN library contains this standard initialize routine.

1.2 Sample Programs

The following listings are listings of sample programs that will be used to demonstrate the use of the FORTRAN compiler and the LINKER in following sections.

SAMPLE 1:

```
      PROGRAM EXAMPLE
C
C     This program demonstrates the use of the READ,
C     WRITE, and FORMAT statements to do input and
C     output to the terminal.  The logical unit
C     number of the terminal is 3, so that number is
C     used in the FORMAT statements.
C     The values entered by the operator in response
C     to the prompts generated by this program must
C     be in the form specified for I format input in
C     Section 8 of the FORTRAN manual.
C
      INTEGER J, K, L
C
C     The next lines write a message on the terminal
C     to prompt the operator to enter the value of J.
C
        WRITE (3, 1Ø)
1Ø      FORMAT (1X, 'J = ')
C
C     The next lines read the value typed by the
C     operator and store it in the integer variable J.
C
        READ (3, 11) J
11      FORMAT (I7)
C
C     The next lines write a message on the terminal
C     to prompt the operator to enter the value of K.
C
        WRITE (3, 12)
12      FORMAT (1X, 'K = ')
C
C     The next line reads the value typed by the
C     operator and stores it in the integer variable
C     K.  A previously used FORMAT statement is used
```

```
C     by the READ statement.
C
         READ (3, 11) K
C
C     The next lines compute the sum of the two
C     variables J and K, and write the value of the
C     result on the terminal.
C
         L = J + K
         WRITE (3, 13) L
13       FORMAT (1X, 'J + K = ', I7, /)
         END
```

SAMPLE 2:

```
       PROGRAM ROOTS
C
C     This program calculates and prints a table of
C     square roots.  Logical unit number 3 is assigned
C     to the terminal, so the number 3 appears in
C     the WRITE statements.
C
C
       REAL A(10)
       INTEGER I, I2, J, J2, K
C
C     These lines print the heading for the table.
C
         WRITE (3, 1)
1        FORMAT (1X, /////, 25X,
     -                'TABLE OF SQUARE ROOTS', /)
         WRITE (3, 2)
2        FORMAT (1X, 8X, '0', 6X, '1', 6X, '2', 6X,
     -                   '3', 6X, '4', 6X, '5', 6X,
     -                   '6', 6X, '7', 6X, '8', 6X,
     -                   '9', /)
C
C     The following lines calculate 10 square roots,
C     load them into an array, and print the elements
C     of the array.
C
         DO 100 I = 1, 10
         I2 = I - 1
         DO 200 J = 1, 10
         J2 = J - 1
         A(J) = SQRT (FLOAT (I2*10 + J2))
200      CONTINUE
100      WRITE (3, 3) I2, (A(K), K = 1, 10)
3        FORMAT (1X, I1, '-', 10(F7.3))
         WRITE (3, 4)
```

```
4       FORMAT (1X, //////)
        END
```

1.3 Sample Compilations

The following execution of FORTRAN compiles Sample 1, using a source file named EXAMPLE.FOR, writing the listing, with no object code included, on the terminal, and producing a relocatable object program on disk with the name EXAMPLE.REL. EXAMPLE.REL is not an executable file. It must be linked and loaded using the Cromemco LINK program (see Part 3 of this manual).

```
A. FORTRAN
* EXAMPLE, TTY:=EXAMPLE/N

 Cromemco FORTRAN 1978 Ver. 3.13 - Bytes: 31740
 Created: 01-Dec-78

 1              PROGRAM EXAMPLE
 2       C
 3       C    This program demonstrates the use of the READ,
 4       C    WRITE, and FORMAT statements to do input and
 5       C    output to the terminal.  The logical unit
 6       C    number of the terminal is 3, so that number is
 7       C    used in the FORMAT statements.
 8       C    The values entered by the operator in response
 9       C    to the prompts generated by this program must
 10      C    be in the form specified for I format input in
 11      C    Section 8 of the FORTRAN manual.
 12      C
 13             INTEGER J, K, L
 14      C
 15      C    The next lines write a message on the terminal
 16      C    to prompt the operator to enter the value of J.
 17      C
 18                WRITE (3, 10)
 19      10        FORMAT (1X, 'J = ')
 20      C
 21      C    The next lines read the value typed by the
 22      C    operator and store it in the integer variable J.
 23      C
 24                READ (3, 11) J
 25      11        FORMAT (I7)
 26      C
 27      C    The next lines write a message on the terminal
 28      C    to prompt the operator to enter the value of K.
```

```
29      C
30              WRITE (3, 12)
31      12      FORMAT (1X, 'K = ')
32      C
33      C     The next line reads the value typed by the
34      C     operator and stores it in the integer variable
35      C     K.  A previously used FORMAT statement is used
36      C     by the READ statement.
37      C
38              READ (3, 11) K
39      C
40      C     The next lines compute the sum of the two
41      C     variables J and K, and write the value of the
42      C     result on the terminal.
43      C
44              L = J + K
45              WRITE (3, 13) L
46      13      FORMAT (1X, 'J + K = ', I7, /)
47              END

Program Unit Length=0075 (117) Bytes
Data Area Length=0035 (53) Bytes

Subroutines Referenced:

$I0                     $INIT                   $W2
$ND                     $R2                     $EX

Variables:

J       0001"           K       0003"           L       0005"

LABELS:

$$L     0006'           10L     0007"           11L     0012"
12L     0016"           13L     0021"
```

The following execution of the FORTRAN compiler demonstrates the appearance of the listing when object code is included.

A. FORTRAN EXAMPLE,TTY:=EXAMPLE

```
Cromemco FORTRAN 1978 Ver.  3.13 -Bytes:  31740
Created:  01-Dec-78

 1              PROGRAM EXAMPLE
*****   0000'    LD      BC,$$L
*****   0003'    JP      $INIT
 2       C
 3       C     This program demonstrates the use of the READ,
 4       C     WRITE, and FORMAT statements to do input and
 5       C     output to the terminal.  The logical unit
 6       C     number of the terminal is 3, so that number is
 7       C     used in the FORMAT statements.
 8       C     The values entered by the operator in response
 9       C     to the prompts generated by this program must
10       C     be in the form specified for I format input in
11       C     Section 8 of the FORTRAN manual.
12       C
13              INTEGER J, K, L
14       C
15       C     The next lines write a message on the terminal
16       C     to prompt the operator to enter the value of J.
17       C
18              WRITE (3, 10)
*****   0006'    LD      DE,10L
*****   0009'    LD      HL,[    03      00]
*****   000C'    CALL    $W2
19      10       FORMAT (1X, 'J = ')
*****   000F'    CALL    $ND
20       C
21       C     The next lines read the value typed by the
22       C     operator and store it in the integer variable J.
23       C
24              READ (3, 11) J
*****   0012'    LD      DE,11L
*****   0015'    LD      HL,[    03      00]
*****   0018'    CALL    $R2
25      11       FORMAT (I7)
*****   001B'    LD      DE,J
*****   001E'    LD      HL,[    01      00]
*****   0021'    LD      A,02
*****   0023'    CALL    $I0
*****   0026'    CALL    $ND
26       C
27       C     The next lines write a message on the terminal
28       C     to prompt the operator to enter the value of K.
```

```
29        C
30                  WRITE (3, 12)
*****     0029'     LD        DE,12L
*****     002C'     LD        HL,[    03      00]
*****     002F'     CALL      $W2
31        12        FORMAT (1X, 'K = ')
*****     0032'     CALL      $ND
32        C
33        C     The next line reads the value typed by the
34        C     operator and stores it in the integer variable
35        C     K.  A previously used FORMAT statement is used
36        C     by the READ statement.
37        C
38                  READ (3, 11) K
*****     0035'     LD        DE,11L
*****     0038'     LD        HL,[    03      00]
*****     003B'     CALL      $R2
39        C
40        C     The next lines compute the sum of the two
41        C     variables J and K, and write the value of the
42        C     result on the terminal.
43        C
44                  L = J + K
*****     003E'     LD        DE,K
*****     0041'     LD        HL,[    01      00]
*****     0044'     LD        A,02
*****     0046'     CALL      $I0
*****     0049'     CALL      $ND
45                  WRITE (3, 13) L
*****     004C'     LD        HL,(K)
*****     004F'     EX        DE,HL
*****     0050'     LD        HL,(J)
*****     0053'     ADD       HL,DE
*****     0054'     LD        (L),HL
*****     0057'     LD        DE,13L
*****     005A'     LD        HL,[    03      00]
*****     005D'     CALL      $W2
46        13        FORMAT (1X, 'J + K = ', I7, /)
*****     0060'     LD        DE,L
*****     0063'     LD        HL,[    01      00]
*****     0066'     LD        A,02
*****     0068'     CALL      $I0
*****     006B'     CALL      $ND
47                  END
*****     006E'     CALL      $EX
*****     0071'     0100
*****     0073'     0300

Program Unit Length=0075 (117) Bytes
Data Area Length=0035 (53) Bytes
```

Subroutines Referenced:

$IØ	$INIT	$W2
$ND	$R2	$EX

Variables:

J	ØØØ1"	K	ØØØ3"	L	ØØØ5"

LABELS:

$$L	ØØØ6'	1ØL	ØØØ7"	11L	ØØ12"
12L	ØØ16"	13L	ØØ21"		

Compiling the program from Sample 2, specifying that no object code is to be included on the listing, produces the following listing.

A. FORTRAN ROOTS,TTY:=ROOTS/N

```
Cromemco FORTRAN 1978 Ver.  3.13 -Bytes:  31740
Created:  01-Dec-78

 1              PROGRAM ROOTS
 2       C
 3       C     This program calculates and prints a table of
 4       C     square roots.  Logical unit number 3 is assigned
 5       C     to the terminal, so the number 3 appears in
 6       C     the WRITE statements.
 7       C
 8       C
 9              REAL A(10)
10              INTEGER I, I2, J, J2, K
11       C
12       C     These lines print the heading for the table.
13       C
14                 WRITE (3, 1)
15       1         FORMAT (1X, /////, 25X,
16              -                 'TABLE OF SQUARE ROOTS', /)
17                 WRITE (3, 2)
18       2         FORMAT (1X, 8X, '0', 6X, '1', 6X, '2', 6X,
19              -                  '3', 6X, '4', 6X, '5', 6X,
20              -                  '6', 6X, '7', 6X, '8', 6X,
21              -                  '9', /)
22       C
23       C     The following lines calculate 10 square roots,
24       C     load them into an array, and print the elements
25       C     of the array.
26       C
27                 DO 100 I = 1, 10
28                 I2 = I - 1
29                 DO 200 J = 1, 10
30                 J2 = J - 1
31                 A(J) = SQRT (FLOAT (I2*10 + J2))
32       200       CONTINUE
33       100       WRITE (3, 3) I2, (A(K), K = 1, 10)
34       3         FORMAT (1X, I1, '-', 10(F7.3))
35                 WRITE (3, 4)
36       4         FORMAT (1X, //////)
37                 END

Program Unit Length=00DD (221) Bytes
Data Area Length=00CE (206) Bytes

Subroutines Referenced:
```

```
$I1                     $IØ                     $INIT
$W2                     $ND                     SQRT
FLOAT                   $M9                     $T1
$EX

Variables:

A       ØØØ1"           I       ØØ29"           I2      ØØ2B"
J       ØØ2D"           J2      ØØ2F"           K       ØØ31"
T:ØØØØØØ        ØØA7"   T:Ø1ØØØØ        ØØA9"
T:ØØØØØ1        ØØAB"

LABELS:

$$L     ØØØ6'           1L      ØØ33"           2L      ØØ5B"
1ØØL    ØØ7B'           2ØØL    ØØ6E'           3L      ØØAF"
4L      ØØC3"
```

Compiling the program from Sample 2, including object code on the listing, produces the following listing.

```
A. FORTRAN
* ROOTS, TTY:=ROOTS

Cromemco FORTRAN 1978 Ver.  3.13 -Bytes:  31740Ø
Created:  Ø1-Dec-78

 1              PROGRAM ROOTS
 2        C
 3        C     This program calculates and prints a table of
 4        C     square roots.  Logical unit number 3 is assigned
 5        C     to the terminal, so the number 3 appears in
 6        C     the WRITE statements.
 7        C
 8        C
 9              REAL A(1Ø)
*****   ØØØØ'    LD      BC,$$L
*****   ØØØ3'    JP      $INIT
1Ø              INTEGER I, I2, J, J2, K
11        C
12        C     These lines print the heading for the table.
13        C
14              WRITE (3, 1)
*****   ØØØ6'    LD      DE,1L
*****   ØØØ9'    LD      HL,[    Ø3      ØØ]
*****   ØØØC'    CALL    $W2
```

```
15        1       FORMAT (1X, /////, 25X,
*****   000F'     CALL    $ND
16              -             'TABLE OF SQUARE ROOTS', /)
17                WRITE (3, 2)
*****   0012'     LD      DE,2L
*****   0015'     LD      HL,[    03      00]
*****   0018'     CALL    $W2
18        2       FORMAT (1X, 8X, '0', 6X, '1', 6X, '2', 6X,
*****   001B'     CALL    $ND
19              -                 '3', 6X, '4', 6X, '5', 6X,
20              -                 '6', 6X, '7', 6X, '8', 6X,
21              -                 '9', /)
22        C
23        C     The following lines calculate 10 square roots,
24        C     load them into an array, and print the elements
25        C     of the array.
26        C
27                  DO 100 I = 1, 10
28                  I2 = I - 1
*****   001E'     LD      HL,0001
*****   0021'     LD      (I),HL
29                  DO 200 J = 1, 10
*****   0024'     LD      HL,(I)
*****   0027'     DEC     HL
*****   0028'     LD      (I2),HL
30                  J2 = J - 1
*****   002B'     LD      HL,0001
*****   002E'     LD      (J),HL
31                  A(J) = SQRT (FLOAT (I2*10 + J2))
32        200       CONTINUE
*****   0031'     LD      HL,(J)
*****   0034'     DEC     HL
*****   0035'     LD      (J2),HL
*****   0038'     LD      HL,(J)
*****   003B'     ADD     HL,HL
*****   003C'     ADD     HL,HL
*****   003D'     LD      DE,A-0004
*****   0040'     ADD     HL,DE
*****   0041'     LD      (T:000000),HL
*****   0044'     LD      HL,(I2)
*****   0047'     EX      DE,HL
*****   0048'     LD      HL,000A
*****   004B'     CALL    $M9
*****   004E'     EX      DE,HL
*****   004F'     LD      HL,(J2)
*****   0052'     ADD     HL,DE
*****   0053'     LD      (T:010000),HL
*****   0056'     LD      HL,T:010000
*****   0059'     CALL    FLOAT
*****   005C'     LD      HL,T:000001
*****   005F'     CALL    $T1
```

```
*****   0062'   LD      HL,T:000001
*****   0065'   CALL    SQRT
*****   0068'   LD      HL,(T:000000)
*****   006B'   CALL    $T1
33      100     WRITE (3, 3) I2, (A(K), K = 1, 10)
*****   006E'   LD      HL,(J)
*****   0071'   INC     HL
*****   0072'   LD      A,0A
*****   0074'   SUB     L
*****   0075'   LD      A,00
*****   0077'   SBC     H
*****   0078'   JP      P,002E'
*****   007B'   LD      DE,3L
*****   007E'   LD      HL,[    03      00]
*****   0081'   CALL    $W2
34      3       FORMAT (1X, I1, '-', 10(F7.3))
*****   0084'   LD      DE,I2
*****   0087'   LD      HL,[    01      00]
*****   008A'   LD      A,02
*****   008C'   CALL    $I0
*****   008F'   LD      HL,0001
*****   0092'   LD      (K),HL
*****   0095'   LD      HL,(K)
*****   0098'   ADD     HL,HL
*****   0099'   ADD     HL,HL
*****   009A'   LD      DE,A-0004
*****   009D'   ADD     HL,DE
*****   009E'   LD      (T:000000),HL
*****   00A1'   LD      HL,(T:000000)
*****   00A4'   EX      DE,HL
*****   00A5'   LD      HL,[    01      00]
*****   00A8'   LD      A,02
*****   00AA'   CALL    $I1
*****   00AD'   LD      HL,(K)
*****   00B0'   INC     HL
*****   00B1'   LD      A,0A
*****   00B3'   SUB     L
*****   00B4'   LD      A,00
*****   00B6'   SBC     H
*****   00B7'   JP      P,0092'
*****   00BA'   CALL    $ND
*****   00BD'   LD      HL,(I)
*****   00C0'   INC     HL
*****   00C1'   LD      A,0A
*****   00C3'   SUB     L
*****   00C4'   LD      A,00
*****   00C6'   SBC     H
*****   00C7'   JP      P,0021'
35              WRITE (3, 4)
*****   00CA'   LD      DE,4L
*****   00CD'   LD      HL,[    03      00]
```

```
*****   00D0'   CALL    $W2
36      4       FORMAT (1X, //////)
*****   00D3'   CALL    $ND
37              END
*****   00D6'   CALL    $EX
*****   00D9'   0100
*****   00DB'   0300

Program Unit Length=00DD (221) Bytes
Data Area Length=00CE (206) Bytes

Subroutines Referenced:

$I1                     $I0                     $INIT
$W2                     $ND                     SQRT
FLOAT                   $M9                     $T1
$EX

Variables:

A       0001"           I       0029"           I2      002B"
J       002D"           J2      002F"           K       0031"
T:000000        00A7"   T:010000        00A9"
T:000001        00AB"

LABELS:

$$L     0006'           1L      0033"           2L      005B"
100L    007B'           200L    006E'           3L      00AF"
4L      00C3"
```

1.4 Sample Executions

The following lines were output when the sample programs were executed.

Sample 1:

A. EXAMPLE

J = 35

K = 35

J + K = 70

Sample 2:

A. ROOTS

TABLE OF SQUARE ROOTS

	Ø	1	2	3	4	5	6	7	8	9
Ø-	Ø.ØØØ	1.ØØØ	1.414	1.732	2.ØØØ	2.236	2.449	2.646	2.828	3.ØØØ
1-	3.162	3.317	3.464	3.6Ø6	3.742	3.873	4.ØØØ	4.123	4.243	4.359
2-	4.47S	4.583	4.69Ø	4.796	4.899	5.ØØØ	5.Ø99	5.196	5.292	5.385
3-	5.477	5.568	5.657	5.745	5.831	5.916	6.ØØØ	6.Ø83	6.164	6.245
4-	6.325	6.4Ø3	6.481	6.557	6.633	6.7Ø8	6.782	6.856	6.928	7.ØØØ
5-	7.Ø71	7.141	7.211	7.28Ø	7.348	7.416	7.483	7.55Ø	7.616	7.681
6-	7.746	7.81Ø	7.874	7.937	8.ØØØ	8.Ø62	8.124	8.185	8.246	8.3Ø7
7-	8.367	8.426	8.485	8.544	8.6Ø2	8.66Ø	8.718	8.775	8.832	8.888
8-	8.944	9.ØØØ	9.Ø55	9.11Ø	9.165	9.22Ø	9.274	9.327	9.381	9.434
9-	9.487	9.539	9.592	9.644	9.695	9.747	9.798	9.849	9.899	9.95Ø

1.5 FORTRAN Compiler Error Messages

The FORTRAN Compiler detects two kinds of errors: Warnings and Fatal Errors. When a Warning is issued, compilation continues with the next item on the source line. When a Fatal Error is found, the compiler ignores the rest of the logical line, including any continuation lines. Warning messages are preceded by percent signs (%), and Fatal Errors by question marks (?). The editor line number, if any, or the physical line number is printed next. It is followed by the error code or error message.

Example:

?Line 25: Mismatched Parentheses

%Line 16: Missing Integer Variable

When either type of error occurs, the program should be changed so that it compiles without errors. No guarantee is made that a program that compiles with errors will execute sensibly.

Fatal Errors:

Error Number	Message
100	Illegal Statement Number
101	Statement Unrecognizable or Misspelled
102	Illegal Statement Completion
103	Illegal DO Nesting
104	Illegal Data Constant
105	Missing Name
106	Illegal Procedure Name
107	Invalid DATA Constant or Repeat Factor
108	Incorrect Number of DATA Constants
109	Incorrect Integer Constant
110	Invalid Statement Number
111	Not a Variable Name
112	Illegal Logical Form Operator
113	Data Pool Overflow
114	Literal String Too Large
115	Invalid Data List Element in I/O
116	Unbalanced DO Nest
117	Identifier Too Long
118	Illegal Operator
119	Mismatched Parentheses
120	Consecutive Operators
121	Improper Subscript Syntax
122	Illegal Integer Quantity
123	Illegal Hollerith Construction
124	Backwards DO reference
125	Illegal Statement Function Name
126	Illegal Character for Syntax
127	Statement Out of Sequence
128	Missing Integer Quantity
129	Invalid Logical Operator
130	Illegal Item Following INTEGER or REAL or LOGICAL
131	Premature End Of File on Input Device (caused by presence of a Control-Z character, or the absence of an END statement)
132	Illegal Mixed Mode Operation
133	Function Call with No Parameters
134	Stack Overflow
135	Illegal Statement Following Logical IF

Warnings:

Ø	Duplicate Statement Label
1	Illegal DO Termination
2	Block Name = Procedure Name
3	Array Name Misuse
4	COMMON Name Usage
5	Wrong Number of Subscripts
6	Array Multiply EQUIVALENCED within a Group
7	Multiple EQUIVALENCE of COMMON
8	COMMON Base Lowered
9	Non-COMMON Variable in BLOCK DATA
1Ø	Empty List for Unformatted WRITE
11	Non-Integer Expression
12	Operand Mode Not Compatible with Operator
13	Mixing of Operand Modes Not Allowed
14	Missing Integer Variable
15	Missing Statement Number on FORMAT
16	Zero Repeat Factor
17	Zero Format Value
18	Format Nest Too Deep
19	Statement Number Not FORMAT Associated
2Ø	Invalid Statement Number Usage
21	No Path to this Statement
22	Missing DO Termination
23	Code Outpuut in BLOCK DATA
24	Undefined Labels Have Occurred
25	RETURN in a Main Program
26	STATUS Error on READ
27	Invalid Operand Usage
28	Function with no Parameter
29	Hex Constant Overflow
3Ø	Division by Zero
31	Missing RETURN in Subprogram
32	Array Name Expected
33	Illegal Argument to ENCODE/DECODE

SECTION 2

FORTRAN Runtime Error Messages

Code	Meaning
Warning Errors:	
A2	Both Arguments of ATAN2 are Ø
BE	Binary Exponent Overflow
BI	Buffer Size Exceeded During Binary I/O
CN	Conversion Overflow on REAL to INTEGER Conversion
DE	Decimal Exponent Overflow (Number in input stream had an exponent larger than 99)
FW	FORMAT Field Width is Too Small
IB	Input Buffer Limit Exceeded
IN	Input Record Too Long
IO	Illegal I/O Operation
IS	Integer Size Too Large
OB	Output Buffer Limit Exceeded
OV	Arithmetic Overflow
RC	Negative Repeat Count in FORMAT
SN	Argument to SIN Too Large
TL	Too Many Left Parentheses in FORMAT
Fatal Errors:	
DT	Data Type Does Not Agree With FORMAT Specification
DZ	Division by Zero, REAL or INTEGER
EF	EOF Encountered on READ
FØ	FORMAT Field Width is Zero
ID	Illegal FORMAT Descriptor
IT	I/O Transmission Error
LG	Illegal Argument to LOG Function (Negative or Zero)
ML	Missing Left Parentheses in FORMAT
MP	Missing Period in FORMAT
SQ	Illegal Argument to SQRT Function (Negative)

Runtime errors are surrounded by asterisks as follows:

FW

Fatal errors cause execution to cease (control is returned to the operating system). Execution continues after a warning error. However, after 2Ø warnings, execution ceases.

SECTION 3

LINK Linking Loader

LINK is a program that loads relocatable object files into memory and changes all relative addresses into actual memory addresses. LINK loads the program area and the data area of the object file into memory starting at addresses specified by the user. After the program is linked and loaded the user may execute the program, or may specify to LINK that the executable object program be saved on disk for future use.

3.1 LINK Commands

3.1.1 LINK Command Strings

To run LINK, type LINK followed by a carriage return. LINK will return the prompt "*", indicating it is ready to accept commands. Each command to LINK consists of a string of filenames and switches separated by commas:

objdev1:filename.ext/switch1,objdev2:filename.ext,...

If the input device for a file is omitted, the default is the currently logged disk. If the extension of a file is omitted, the default is REL. After each line is typed, LINK will load or search (see /S below) the specified files. After LINK finishes this process, it will list all symbols that remained undefined followed by an asterisk.

Example:

```
*MAIN
 DATA      Ø1ØØ      Ø2ØØ
 SUBR*          (SUBR1 is undefined)
 DATA      Ø1ØØ      Ø3ØØ
*SUBR1
*/G             (Starts Execution - see below)
```

Typically, to execute a FORTRAN and/or COBOL program and subroutines, the user types the list of filenames followed by /G (begin execution). Before execution begins, LINK will always search the system library (FORLIB.REL or COBLIB.REL) to satisfy any unresolved external references. If the user wishes first to search libraries of his own, he should append those file names, each followed by /S, to the end of the loader command string.

3.1.2 LINK Switches

A number of switches may be given in the LINK command string to specify actions affecting the loading process. Each switch must be preceded by a slash (/). These switches are:

Switch	Action
R	Reset. Put loader back in its initial state. User /R if you loaded the wrong file by mistake and want to restart. /R takes effect as soon as it is encountered in a command string.
E or E:Name	Exit LINK and return to CDOS. The system library will be searched on the current disk to satisfy any existing undefined globals. The optional form E:Name (where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program. Use /E to load a program and exit back to CDOS.
G or G:Name	Start execution of the program as soon as the current command line has been interpreted. The system library will be searched on the current disk to satisfy any existing undefined globals if they exist. Before execution actually begins, LINK prints three numbers and a BEGIN EXECUTION message. The three numbers are the start address, the address of the next available byte, and the number of 256-byte pages used. The optional form G:Name

(where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program.

N

If a <filename>/N is specified, the program will be saved on disk with the specified name (with a default extension of .COM) when a /E or /G is done. A jump to the start of the program is inserted if needed so the program can run properly at 100H.

P and D

/P and /D allow the program and data origins to be set for the next program loaded. /P and /D take effect when seen (not deferred), and they have no effect on programs already loaded. The form is /P:<address> or /D:<address>, where <address> is the desired origin in the current typeout radix. (Default radix is hex. /O sets radix to octal; /H to hex.) LINK does a default /P:<link origin>+3 (i.e., 103H) to leave room for the jump to the start address.

NOTE: Do not use /P or /D to load programs or data into the locations of the loader's jump to the start address (100H to 102H), unless it is to load the start of the program there. If programs or data are loaded into these locations, the jump will not be generated.

If no /D is given, data areas are loaded before program areas for each module. If a /D is given, all Data and Common areas are loaded starting at the data origin and the program area at the program origin. Example:

```
*/P:200,ROOTS
Data    200    300
*/R
*/P:200 /D:400,ROOTS
Data    400    480
Program 200    280
```

U — List the origin and end of the program and data area and all undefined globals as soon as the current command line has been interpreted. The program information is only printed if a /D has been done. Otherwise, the program is stored in the data area.

M — List the origin and end of the program and data area, all defined globals and their values, and all undefined globals followed by an asterisk. The program information is only printed if a /D has been done. Otherwise, the program is stored in the data area.

S — Search the filename immediately preceding the /S in the command string to satisfy any undefined globals.

Examples:

*/M — List all globals

*MYPROG,SUBRROT,MYLIB/S
Load MYPROG.REL and SUBROT.REL and then search MYLIB.REL to satisfy any remaining undefined globals.

*/G — Begin execution of main program.

3.2 Sample LINK Executions

The following example illustrates the use of the LINK program to load, Link, and execute the program from Sample 1, Part 2.

```
B. LINK
* EXAMPLE/G

Data Ø1Ø3 189D

[Ø138   189D   24]
[BEGIN EXECUTION]

J = 15
```

```
K =    17

J + K =    32
```

The memory addresses printed following "Data" have the following interpretations.

Ø1Ø3	is the address of the first byte allocated to the data area.
Ø138	is the address of the first byte of memory where the program area was loaded.
189D	is the address of the first available byte of memory following the last instruction in the program.
24	is the number of 256 - byte pages required to contain the program. This is roughly 6K of memory.

The following example illustrates the use of LINK to load and link the program from Sample 2, Part 2, save the resulting executable program on disk with the name ROOTS.COM, and return to CDOS.

```
B. LINK
* ROOTS, ROOTS/N/E

Data Ø1Ø3 1EDA

[Ø1D2 1EDA 3Ø]
```

3.3 Format of LINK Compatible Object Files

NOTE

Section 3.3 is reference material for users who wish to know the load format of LINK relocatable object files. Most users will want to skip this section, as it does not contain material necessary to the operation of the package.

LINK-compatible object files consist of a bit stream. Individual fields within the bit stream

are not aligned on byte boundaries, except as noted below. Use of a bit stream for relocatable object files keeps the size of object files to a minimum, thereby decreasing the number of disk reads/writes. There are two basic types of load items: Absolute and Relocatable. The first bit of an item indicates one of these two types. If the first bit is a Ø, the following 8 bits are loaded as an absolute byte. If the first bit is a 1, the next 2 bits are used to indicate one of four types of relocatable items:

ØØ Special LINK item (see below).

Ø1 Program Relative. Load the following 16 bits after adding the current Program base.

1Ø Data Relative. Load the following 16 bits after adding the current Data base.

11 Common Relative. Load the following 16 bits after adding the current Common base.

Special LINK items consist of the bit stream 1ØØ followed by:

a four-bit control field

an optional A field consisting of a two-bit address type that is the same as the two-bit field above except ØØ specifies absolute address

an optional B field consisting of 3 bits that give a symbol length and up to 8 bits for each character of the symbol

A general representation of a special LINK item is:

```
1 ØØ xxxx    yy     zzz + characters of symbol name
             --------  ------------------------------
             A field            B field
```

xxxx Four-bit control field (Ø-15 below)
yy Two-bit address type field
zzz Three-bit symbol length field

The following special types have a B-field only:

Ø	Entry symbol (name for search)
1	Select COMMON block
2	Program name
3	Reserved for future expansion
4	Reserved for future expansion

The following special LINK items have both an A field and a B field:

5	Define COMMON size
6	Chain external (A is head of address chain, B is name of external symbol)
7	Define entry point (A is address, B is name)
8	Reserved for future expansion

The following special LINK items have an A field only:

9	External + offset. The A value will be added to the two bytes starting at the current location counter immediately before execution.
1Ø	Define size of Data area (A is size)
11	Set loading location counter to A
12	Chain address. A is head of chain, replace all entries in chain with current location counter. The last entry in the chain has an address field of absolute zero.
13	Define program size (A is size)
14	End program (forces to byte boundary)

The following special Link item has neither an A nor a B field:

15	End File

3.4 LINK Error Messages

LINK has the following error messages:

?No Start Address	A /G switch was issued, but no main program has been loaded.
?Loading Error	The last file given for input was not a properly formatted LINK object file.

?Out of Memory — Not enough memory to load program.

?Command Error — Unrecognizable LINK command.

?<file> Not Found — <file>, as given in the command string, does not exist.

%2nd COMMON Larger /XXXXXX/
The first definition of COMMON block /XXXXXX/ was not the largest definition. Reorder module loading sequence or change COMMON block definitions.

%Mult. Def. Global YYYYYY
More than one definition for the global (internal) symbol YYYYYY was encountered during the loading process.

%Overlaying [Program] Area
[Data]
A /D or /P will cause already loaded data to be destroyed.

?Intersecting [Program] Area
[Data]
The program and data area intersect and an address or external chain entry is in this intersection. The final value cannot be converted to a current value since it is in the area intersection.

?Start Symbol - <name> - Undefined
After a /E: or /G: is given, the symbol specified was not defined.

Origin [Above] Loader Memory, Move Anyway (Y or N)?
[Below]
After a /E or /G was given, either the data or program area has an origin to top which lies outside loader memory (i.e., loader origin to top of memory). If a Y <cr> is given, LINK will move the area and

continue. If anything else is given, LINK will exit. In either case, if a /N was given, the image will already have been saved.

?Can't Save Object File
A disk error occurred when the file was being saved.

3.5 Separating Data and Program Areas

By default, LINK loads the data area first in memory, followed immediately by the program area. The user may use switches /D and /P to set the beginning addresses of the data and program areas, with the following considerations.

The disk driver module in the FORLIB relocatable library file has declared an ENTRY global symbol called $MEMRY in order to obtain from LINK a pointer to the first available byte following a program's data area. When LINK finds this symbol in the module being loaded, it loads into it the address of the top of the data area + 1.

In the default case, where neither /D nor /P has been used, the program area is considered part of the data area, and LINK loads the two-part data area (data and program), then sets $MEMRY to point to the top of the two-part data area + 1.

When either /D or /P has been used to separate the two areas, $MEMRY will be set to point to the top of the data area + 1.

Thus, when /D is used to load the data area below the program area, the user must be certain that there is enough memory between the top of the data and the bottom of the program area. Otherwise the program area will be overwritten and the program destroyed at runtime.

The FORTRAN disk driver module uses $MEMRY to allocate disk buffers (128 decimal bytes each) and FCB's (33 decimal bytes each). These follow the address pointed to by $MEMRY (top of data area +1), so the user must have an idea of their total length in order to start the program area above their ending address.

INDEX

Cromemco

incorporated

Specialists in computers and peripherals

280 BERNARDO AVE., MOUNTAIN VIEW, CA

023-0038