# DOS FORTRAN COMPILER
# FORTRAN/FORT55

User's Guide

Version 1

March, 1981

Document No. 50526

# PREFACE

Datapoint FORTRAN contains features and enhancements that render it comparable to FORTRAN compilers used on large mainframes and minicomputer systems. All of ANSI Standard FORTRAN X3.9-1966 is included except the COMPLEX data type.

This user's guide describes Datapoint FORTRAN for the programmer. Chapter 1 is a short overview that includes a discussion of notational conventions used throughout the text and example material. Chapter 2 describes the form and components of a FORTRAN source program, and Chapters 3 and 4 define data types and their expressional relationships as they are used in the program. Chapters 5 through 9 describe the proper construction and use of the various statement classes.

Appendix A is a listing of the error messages that can occur during compilation. Appendices B and C give compilation procedures for the 1500 FORTRAN and the 5500 FORT55 and show a sample test program compile input; Appendix D contains an explanation of the ARC Enqueue/Dequeue Subsystem.

## TABLE OF CONTENTS

# CHAPTER 1. INTRODUCTION TO DOS FORTRAN

## 1.1 Overview of FORTRAN

FORTRAN is a universal, problem-oriented programming language. The name of the language is an acronym for FORmula TRANslator. The Datapoint FORTRAN compiler is designed to simplify the preparation and checkout of programs written in the FORTRAN language.

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

## 1.2 Features of Datapoint FORTRAN

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. The language extensions and restrictions are described in the text of this user's guide.

NOTE

This FORTRAN differs from the American National Standard in that it does not include the COMPLEX data type.

Datapoint FORTRAN is unique in that it provides an assembly language development package that generates relocatable object modules. Only the subroutines and system routines required to run FORTRAN programs need to be loaded before execution. Users are able to develop a common set of subroutines for their programs and place these in the system library. Then, if one module of a program is changed, it is necessary to recompile only that one module.

Enhancements:

The FORTRAN compiler contains a number of enhancements to the ANSI Standard:

- LOGICAL variables which can be used as integer quantities in the range +127 to -128.
- LOGICAL DO loops for tighter, faster execution of small valued integer loops.
- Mixed mode arithmetic.
- Octal constants.
- Literals and Holleriths allowed in expressions.
- Logical operations on integer data .AND., .OR., .NOT., .XOR. can be used for 16-bit or 8-bit Boolean operations.
- READ/WRITE End-of-File or Error Condition transfer. END=n and ERR=n (where n is the statement number) can be included in READ or WRITE statements to transfer control to the specified statement on detection of an error or end-of-file condition.
- ENCODE/DECODE for FORMAT operations to memory.

Characteristics:

The FORTRAN compiler can compile several hundred statements per minute in a single pass. All extra available memory will be used by the compiler for extended optimizations.

In spite of its small size, the FORTRAN compiler optimizes the generated object code in several ways:

- Common subexpressions elimination. Common subexpressions are evaluated once, and the value is substituted in later occurrences of the subexpression.
- Peephole optimization. Small sections of code are replaced by more compact, faster code in special cases.

- Constant folding. Integer constant expressions are evaluated at compile time.

- Branch optimizations. The number of conditional jumps in arithmetic and logical IFs is minimized.

Descriptive error messages are another feature of the compiler. For example, the following message is printed if the compiler scans a statement that is not an assignment or other FORTRAN statement:

```
? STATEMENT UNRECOGNIZABLE
```

The last twenty characters scanned before the error is detected are also printed.

## 1.3 Notation Used in This Guide

The syntax of FORTRAN is displayed using the following fundamental conventions:

- Words in all capital letters stand for themselves. Words in lower case letters are the names of other constructs.

- Corner brackets (< >) are placed around lower-case words which name a class of items that can be substitutes for the corner-bracketed name. For example, if manual instructions specify the item <digit>, this means that any digit may be entered. The instruction might look like this:

  ```
  <letter><digit>
  ```

  This means "any letter followed by any digit."

- Square brackets ([ ]) enclose optional items or groups of items. If the digit in the example above were to be optionally included, the notation would be:

  ```
  <letter>[<digit>]
  ```

  If the word "TO" may be optionally included, the notation would be: [TO]

- Ellipsis, represented by three consecutive periods (...), means "more of the same." For example:

  ```
  BYTE <string>[<string>...]
  ```

  This instruction means that the operation BYTE is to be performed on the characters contained in a string followed by an optional series of successive strings.

- The vertical bar (¦) indicates an either-or choice. Thus:

  ```
  <letter>[<letter> ¦ <digit>]...
  ```

  In this example, a <letter> is optionally followed by either a letter (or a series of letters) or a digit (or series of digits).

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

# CHAPTER 2. FORTRAN PROGRAM FORM

## 2.1 Introduction

FORTRAN source programs consist of one program unit called the MAIN PROGRAM and unlimited program units called SUBPROGRAMS. Subprogram types, and methods of writing and using them, are discussed in Chapter 9.

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

## 2.2 FORTRAN Character Set

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

### 2.2.1 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, $

NOTE:

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

### 2.2.2 Digits

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

## 2.2.3 Alphanumerics

This is a subset of characters made up of all letters and all digits.

## 2.2.4 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 classed 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.

## 2.3 FORTRAN Line Format

The FORTRAN coding form contains the format of FORTRAN program lines. The lines of the form consist of 80 character positions or columns, numbered 1 through 80, divided into four fields.

### 2.3.1 Fields

1. Statement Label (or Number) field: Columns 1 through 5. (Statement labels are discussed in Section 2.3.3.)

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 desired and is ignored by the FORTRAN compiler.

### 2.3.2 Lines

The lines of a FORTRAN statement are placed in Columns 1 through 72, where they are formatted according to line types. There are four line types; definitions and column formats for each are given below.

INITIAL LINE-- the first or only line of each statement.

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

b. Column 6 must contain a zero or blank.

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

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

<u>Example</u>:

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

CONTINUATION LINE-- A continuation line is used when additional lines of coding are required to complete a statement originating with an initial line.

a. Column 6 must contain a character <u>other than</u> zero or blank.

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

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

<u>Example</u>:

```
C  THE STATEMENTS BELOW ARE AN INITIAL
C   LINE AND 2 CONTINUATION LINES
C
   63  BETA(1,2) =
     1   A6BAR**7-(BETA(2,2)-A5BAR*50
     2   +SQRT (BETA(2,1)))
```

COMMENT LINE-- A comment line is used for source program annotation at the programmer's convenience.

a. Column 1 contains the letter <u>C</u>.

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

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

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

<u>Example</u>:

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

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

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

b. Column 6 must contain a zero or blank.

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

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

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

Example:

```
      END
```

## 2.3.3 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 use of statement labels:

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

b. For calculating the numeric value of the label, leading zeros and blanks are not significant.

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

d. A label on a continuation line is ignored by the FORTRAN compiler.

Example:

```
C  EXAMPLES OF STATEMENT LABELS
C
  1
  1 0
99999
  763
```

NOTE:

Embedded blanks within statement labels are discarded; thus, 1 blank 0 is evaluated as 10 (ten).

### 2.3.4 Statements

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

#### 2.3.4.1 Executable Statements

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

1. Replacement statements.
2. Control statements.
3. Input/Output statements.

#### 2.3.4.2 Non-Executable 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 program 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 use and construction of the various types of statements are described in Chapters 5 through 9.

# CHAPTER 3. FORMAT FOR DATA REPRESENTATION AND STORAGE

## 3.1 Introduction

The FORTRAN language prescribes a definitive method for identifying data by name and type.

## 3.2 Data Names

FORTRAN defines four data names: constant, variable, array, and array element.

### 3.2.1 Constant

A constant is an explicity stated datum.

### 3.2.2 Variable

A variable is a symbolically identified datum.

### 3.2.3 Array

An array is an ordered set of data in 1, 2, or 3 dimensions.

### 3.2.4 Array Element

An array element is one member of the set of data in an array.

## 3.3 Data Types

There are five defined data types in FORTRAN: integer, real, double precision, logical, and Hollerith.

### 3.3.1 Integer

An integer is a whole number which can be positive, negative, or zero, having precision to 5 digits in the range -32768 to +32767 (-2**15 to 2**15-1).

## 3.3.2 Real

Approximations of real numbers (positive, negative or zero) are represented in computer storage in 4-byte, floating-point form. Real data are precise to 7 or more significant digits and their magnitude may lie between the approximate limits of 10**-38 and 10**38 (2**-127 and 2**127).

## 3.3.3 Double Precision

Double precision approximations of real numbers (positive, negative, or zero) are represented in computer storage in 8-byte, floating-point form. Double precision data are precise to 16 or more significant digits in the same magnitude range as real data.

## 3.3.4 Logical

Logical data types are one-byte representations of the truth values TRUE or FALSE, with FALSE defined to have an internal representation of zero. The constant .TRUE. has the value of -1; however any non-zero value will be treated as .TRUE. in a Logical IF statement. In addition, logical type data may be used as one-byte signed integers in the range of -128 to +127, inclusive.

## 3.3.5 Hollerith

A string of characters from the ASCII character set used to represent the alphabet, integers 0 through 9, and symbols. All characters including blanks are significant. Hollerith data requires one byte for storage of each character in the string.

## 3.4 Constants

FORTRAN constants are indentified explicitly by stating their actual value. The plus (+) character need not precede positive-valued constants.

Formats for writing constants are shown in Table 3-1.

TABLE 3-1. FORMATS OF CONSTANTS

| TYPE | FORMATS AND RULES FOR USE | EXAMPLES |
|---|---|---|
| INTEGER | 1. 1 to 5 decimal digits interpreted as a decimal number | +763<br>-763<br>1 |
| | 2. A preceding plus sign or minus sign is optional | +32767<br>-32768 |
| | 3. No decimal point (.) or comma (,) is allowed. | |
| | 4. Value range: -32768 through through +32767 (-2**15 through 2**15-1). | |
| REAL | 1. A decimal number with precision to 7 digits and represented in one of the following forms:<br>a. Plus or minus .f ; or plus or minus i.f.<br>b. Plus or minus i.Ee, where e is plus or minus e; or plus or minus .fEe , where e is plus or minus e. Plus or minus i.fEe , where e is plus or minus e.<br>where i, f, and e are each strings representing integer, fraction, and exponent respectively. | 345.<br>-.345678<br>+345.678<br>+.3E3<br>-73E4 |
| | 2. Use of the plus character (+) with real data is optional. | |
| | 3. In the form shown in 1.b above, if r represents any of the forms preceding Ee, where e is plus or minus e , the value of the constant is interpreted as r times 10**e , where the range of the exponent e is always plus or minus 38. | |

| | | |
|---|---|---|
| | 4. If the constant preceding E, plus or minus 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 presented. | |
| DOUBLE PRECISION | A decimal number with precision to 16 digits. All formats and rules are identical to those for REAL constants, except D is used in place of E . Note that a real constant is assumed to be single precision unless it contains a D exponent. | +345.678<br>+.3D3<br>-73D4 |
| LOGICAL | .TRUE. generates a non-zero byte (octal 377) and .FALSE. generates a byte in which all bits are 0.<br><br>If logical values are used as one-byte integers, the rules for use are the same as for type INTEGER, except that the range allowed is -128 to +127, inclusive. | .TRUE. |
| LITERAL | In the literal form, any number of characters may be enclosed by apostrophes. The form is as follows:<br><br>'$x_1x_2x_3...x_n$'<br><br>where each $x_i$ is any character other than '.'(the period). Two apostrophes in succession may be used to represent the character within the string; i.e., if $x_2$ is to be the apostrophied character, the string appears as the following:<br><br>'$x_1$''$x_3...x_n$' | 'EXAMPLE' |

| | | | |
|---|---|---|---|
| OCTAL | 1. | A literal where the first character is the upper-case letter O and followed by three digits enclosed by apostrophes is recognized as an octal value. | O'101' |
| | 2. | An octal constant is right-justified in storage. | |

## 3.5 Variables

Variable data are identified in FORTRAN statements by symbolic names. The names are unique strings of from 1 to 6 alphanumeric characters of which the first is a letter.

NOTE:

System variable names and run time subprogram names are distinguished from other variable names in that they begin with the dollar sign character ($). It is therefore strongly recommended that in order to avoid conflicts, symbolic names in FORTRAN source programs begin with some letter other than "$".

Examples:

I5, TBAR, B23, ARRAY, XFM79, MAX, A1$C

NOTE: Blanks embedded within variables are discarded; thus, T BAR is evaluated as TBAR.

### 3.5.1 Variable Types

Variable data are classified into four types: INTEGER, REAL, DOUBLE PRECISION and LOGICAL. The specification of type is accomplished in one of the following ways:

1. Implicit typing, in which the first letter of the symbolic name specifies Integer or Real type. Unless explicitly typed (see 3.5.2. below), symbolic names beginning with I, J, K, L, M or N represent Integer variables, and symbolic names beginning with letters other than I, J, K, L, M or N represent Real variables.

Integer Variables:

```
ITEM
J1
MODE
K123
N2
```

Real Variables:

```
BETA
H2
ZAP
AMAT
XID
```

2. Variables may have their type specified explicitly. That is, they may be given a particular type without reference to the first letters of their names. Variables may be explicitly typed as INTEGER, REAL, DOUBLE PRECISION or LOGICAL. The statements used in explicitly specifying data type are described in Chapter 6.

## 3.5.2 Variable Value Assignments

Variable data receive their numeric value assignments during program execution or, initially, in a DATA statement (Sec. 6.2).

Hollerith or Literal data may be assigned to any type variable. Section 3.8 contains a discussion of Hollerith data storage.

## 3.6 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 discussion of the array declarators appears in Chapter 5 of this manual. An array declarator also indicates the dimensionality and size 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. 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.

## 3.7 Subscripts

A subscript follows an array name to uniquely identify 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 for the use of subscripts are as follows:

1. A subscript contains 1, 2, or 3 subscript expressions (see item 4 below) 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 (see Chapter 6).

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

   ```
   k    c*v    v-k    v+k    v    c*v+k    c*v-k
   ```

   where c and k are integer constants and v is an integer variable name. See Chapter 5 for a discussion of expression evaluation.

   Examples:

   ```
   X(2*J-3,7)   A(I,J,K)    I(20)    C(L-2)  Y(I)
   ```

5. Subscripts themselves may not be subscripted.

## 3.8 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 storage unit formats of the three data types.

Octal 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. See Chapter 7.

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 |
|---|---|
| INTEGER | 2 bytes/ 1/2 storage unit<br><br>Sign Binary value<br><br>Negative numbers are the 2's complement of positive representations. |
| LOGICAL | 1 byte/ 1/4 storage unit<br><br>Zero (false) or non-zero (true)<br><br>A non-zero-valued byte indicates true (the logical constant). .TRUE. is represented by the octal value 0377. A zero-valued byte indicates false.<br><br>When used as an arithmetic value, a Logical datum is treated as an Integer in the range -128 to +127. |

| | | | | |
|---|---|---|---|---|
| REAL | 4 bytes/ 1 storage unit | | | |

| Character-istic | Sign | Mantissa | Mantissa |
|---|---|---|---|

The first byte is the characteristic expressed in excess 0200 (octal) notation; i.e., a value of 0200 corresponds to a binary exponent of 0. Values less than 0200 correspond to negative exponents, and values greater than 0200 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 its 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, a zero indicates 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 to that of Real data except Double Precision uses 4 extra bytes for the mantissa.

# CHAPTER 4. FORTRAN EXPRESSIONS

## 4.1 Introduction

A FORTRAN expression is composed of a single operand or a string of operands connected by operators. Two expression types--Arithmetic and Logical--are provided by FORTRAN. The operand, operators, and rules for using both types are described in the following paragraphs.

## 4.2 Arithmetic Expressions

The following rules define all permissible arithmetic expression forms:

1. A constant, variable name, array element reference, or FUNCTION reference (Chapter 9) standing alone is an expression.

   Examples:

   ```
   S(I)
   JOBNO
   217
   17.26
   SQRT(A+B)
   ```

2. If e is an expression whose first character is not an operator, then +e and -e are called signed expressions.

   Examples:

   ```
   -S
   +JOBNO
   -217
   +17.26
   -SQRT(A+B)
   ```

3. If e is an expression, then (e) means the quantity resulting when e is evaluated.

   Examples:

   ```
   (-A)
   -(JOBNO)
   -(X+1)
   (A-SQRT(A+B))
   ```

4. If e is an unsigned expression and f is any expression, then f+e, f-e, f*e, f/e and f*e are all expressions.

   Examples:

```
-(B(I,J)+SQRT(A+B(K,L)))
1.7E-2**(X+5.0)
-(B(I+3,3*J+5)+A)
```

5. An evaluated expression may be Integer, Real, Double Precision, or Logical. The type is determined by the data types of the elements of the expression. If the elements are not all of the same type, the expression's type is determined by the element having the highest type. The type hierarchy, from highest to lowest, is: DOUBLE PRECISION, REAL, INTEGER, LOGICAL.

6. Expressions may contain nested parenthesized elements as in this example:

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

   where Y+X is the innermost element, (Y+X)/T is the next innermost, Z-((Y+X)/T) the next. In such expressions, the number of left and right parentheses must be equal, and the number of nested parentheses cannot exceed 14.

## 4.3 Expression Evaluation

Arithmetic expressions are evaluated according to the following rules:

1. Parenthesized expression elements are evaluated first. If parenthesized elements are nested, the innermost elements are evaluated first, the next innermost next, and so forth until the entire expression has been evaluated.

2. Within parentheses and/or wherever parentheses do not govern the order of evaluation, the hierarchy of operations in order of precedence is:

   a. FUNCTION evaluation
   b. Exponentiation
   c. Multiplication and Division
   d. Addition and Subtraction

Example:

The expression

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

is evaluated in the following sequence:

```
Y+R       = e1
(e1)/T    = e2
Z-e2      = e3
e3**J     = e4
A*e4      = e5
e5+VAL    = e6
```

3. The expression X**Y**Z is not allowed. It should be written as:

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

4. Use of an array element reference requires the evaluation of its subscript. Subscript expressions are evaluated under the same rules as other expressions.

## 4.4 Logical Expressions

A Logical Expression may be any of the following:

1. A single Logical Constant (i.e., .TRUE. or .FALSE.), a Logical Variable, Logical Array Element or Logical FUNCTION reference (discussed in Chapter 9).

2. Two arithmetic expressions separated by a relational operator, that is, a relational expression.

3. Logical operators acting upon logical constants, logical variables, logical array elements, logical FUNCTIONS, relational expressions or other logical expressions. The value of a logical expression is always either .TRUE. or .FALSE.

### 4.4.1 Relational Expressions

The general form of a relational expression is:

```
<e1><r><e2>
```

In this relational expression, e1 and e2 are arithmetic expressions and r is a relational operator. There are six relational operators:

| | |
|---|---|
| .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 |

The value of the relational expression is .TRUE. if the condition defined by the operator is met. Otherwise, the value is .FALSE.

Examples:

```
A.EQ.B
(A**J).GT.(ZAP*(RHO*TAU-ALPH)
C.LT.'Z'
```

### 4.4.2. Logical Operators

Table 4-1 lists the logical operations. Logical operations are denoted by u and v.

## TABLE 4-1. LOGICAL OPERATIONS

| | |
|---|---|
| .NOT.u | The value of this expression is the logical complement of u, that is, 1 bits become 0 and 0 bits become 1. |
| u.AND.v | The value of this expression is the logical product of u and v, that is, there is a 1 bit in the result only where the corresponding bits in both u and v are 1. |
| u.OR.v | The value of this expression is the logical sum of u and v, that is, there is a 1 in the result if the corresponding bit in u or v is 1 or if the corresponding bits in both u and v are 1. |
| u.XOR.v | The value of this expression is the exclusive OR of u and v, that is, there is a one in the reult if the corresponding bits in u and v are 1 and 0 or 0 and 1 respectively. |

Examples:

If u = 01101100 and v = 11001001, then

```
.NOT.u    =    10010011
u.AND.v   =    01001000
u.OR.v    =    11101101
u.XOR.v   =    10100101
```

The following are additional considerations for construction of Logical Expressions:

1. Any Logical Expression may be enclosed in parentheses. However, a Logical Expression to which the .NOT. operator is applied must be enclosed in parentheses if it contains two or more elements.

2. In the hierarchy of operations, parentheses may be used to specify the ordering of the expression evaluation. Within parentheses, and where parentheses do not dictate evaluation order, the order is understood to be as follows:

   a. FUNCTION Reference
   b. Exponentiation (**)
   c. Multiplication and Division (* and /)
   d. Addition and Subtraction (+ and -)
   e. .LT., .LE., .EQ., .NE., .GT., .GE.
   f. .NOT.
   g. .AND.
   h. .OR., .XOR.

   Examples:

   The expression

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

   is evaluated as:

```
e1 = B(3,2).GT.Z
e2 = X .AND. Y
e3 = e2 .OR. e1
```

   The expression

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

   is evaluated as:

```
e1 = B(3,2) .GT. Z
e2 = Y .OR. e1
e3 = X .AND. e2
```

3. It is invalid to have two contiguous logical operators except where the second operator is .NOT.

   That is, .AND..NOT. and .OR..NOT. are valid.

   Example:

   | | |
   |---|---|
   | A.AND..NOT.B | is permitted |
   | A.AND..OR. B | is invalid and not permitted |

## 4.5 Hollerith, Literal, and Octal Constants in Expressions

Hollerith, Literal, and Octal constants are allowed in expressions in place of Integer constants. These special constants always evaluate to an Integer value and are therefore limited to a length of two bytes. The only exceptions are:

1. Hollerith or Literal constants may be used as subprogram parameters.

2. Hollerith, Literal, or Octal constants may be up to four bytes long in DATA statements when associated with Real variables, or up to eight bytes long when associated with Double Precision variables.

# CHAPTER 5. REPLACEMENT STATEMENTS

## 5.1 Introduction

Replacement statements define computations and are used in much the same way as equations in normal mathematical notation. They have 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.

## 5.2 Replacement Statements

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--see Chapter 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 initial 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, the value determined by the expression will be converted, if possible, to conform to the typing of the variable. Table 5-1 shows which type expressions may be equated to which type variable. Y indicates a valid replacement. Footnotes to Y (the lower-case letters) indicate conversion considerations.

TABLE 5-1. REPLACEMENT BY TYPE

| | Expression Types (e) | | | |
|---|---|---|---|---|
| Variable Types | Integer | Real | Logical | Double Precision |
| Integer | Y | Ya | Yb | Ya |
| Real | Yc | Y | Yc | Ye |
| Logical | Yd | Ya | Y | Ya |
| Double Prec. | Yc | Y | Yc | Y |

a. The Real expression value is converted to Integer, truncated if necessary to conform to the range of Integer data.

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.

# CHAPTER 6. SPECIFICATION STATEMENTS

## 6.1 Introduction

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 compiler. DATA initialization statements are non-executable, but generate object program data and establish initial values for variable data.

## 6.2 Specification Statements

The seven kinds of specification statements are:

Type, IMPLICIT, 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.

## 6.3 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 Sections 6.4, 6.7, and 6.8.

Array declarators are used to specify the name, dimensionality, and sizes of arrays. 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 $u_i$ is the name of the array, called the declarator name, and the k elements are integer constants.

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

For example, if the array declarator AMAT(3,2,2) appears, storage is allocated for its 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)
```

## 6.4 Type Statements

Variable, array, and FUNCTION names are automatically typed as Integer or Real by the 'predefined' convention unless they are changed by Type statements. For example, the type is Integer if the first letter of an item is I, J, K, L, M, or N. Otherwise, the type is Real.

Type statements provide for overriding or confirming the 'predefined' convention by specifying the type of an item. In addition, these statements may be used to declare arrays.

The general form of Type statements is:

```
<t> <v1>,<v2>,...<vn>
```

where t represents one of the terms INTEGER, INTEGER*1, INTEGER*2, REAL, REAL*4, REAL*8, DOUBLE PRECISION, LOGICAL, LOGICAL*1, LOGICAL*2, or BYTE.

Each v is an array declarator or a variable, array or FUNCTION name.

The INTEGER*1, INTEGER*2, REAL*4, REAL*8, LOGICAL*1, and LOGICAL*2 types are allowed for readibility and compatibility with other FORTRANs.

BYTE, INTEGER*1, LOGICAL*1, and LOGICAL are all equivalent; INTEGER*2, LOGICAL*2, and INTEGER are equivalent; REAL and REAL*4 are equivalent; DOUBLE PRECISION and REAL*8 are equivalent.

Example:

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

NOTE:

1. AMAT and BX are redundantly typed, since under the convention they are Real already.
2. IETA and KLPH, which under the convention would be Integer, are unconditionally declared 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 'predefined' 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 or double-precision <u>must</u> appear in a type statement, since no starting letter indicates these types by the default convention.

## 6.5 IMPLICIT Statements

IMPLICIT statements have this form:

```
IMPLICIT <t> (a[,a]...) [,<t>(a[,a]...)]
```

where <t> represents one of the terms INTEGER, REAL, DOUBLE PRECISION, or LOGICAL;

and where a is either a single letter or a range of single letters in alphabetical order. A range is denoted by the first and last letter of the range separated by a minus sign. Writing a range of letters a1 - a2 has the same effect as writing a list of the single letters a1 through a2.

An IMPLICIT statement specifies a type for all variables, arrays, symbolic names of constants, external functions, and statement functions that begin with any letter that appears in the specification, either as a single letter or included in a range of letters. IMPLICIT statements do not change the type of any intrinsic functions. An IMPLICIT statement only applies to the program unit that contains it.

Type specification by an IMPLICIT statement may be overridden or confirmed for any particular variable, array, symbolic name of a constant, external, external function, or statement function name by the appearance of that name in a type-statement. An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement for the name of that function subprogram.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements. A program unit may contain more than one IMPLICIT statement.

The same letter must not appear as a single letter, or be included in a range of letters, more than once in all of the IMPLICIT statements in a program unit.

## 6.6 External Statements

EXTERNAL statements have this form:

```
EXTERNAL <u1>,<u2>,...<un>
```

where each $u_i$ 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.

When a BLOCK DATA subprogram is to be included in a program load, its name must have appeared in an EXTERNAL statement within the main program unit.

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)
```

## 6.7 Dimension Statements

A DIMENSION statement has the following form:

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

where each $u_i$ is an array declarator.

Example:

```
DIMENSION RAT(5,5),BAR(20)
```

This statement declares two arrays--the 25-element array RAT and the 20-element array BAR.

## 6.8 Common Statements

COMMON statements are non-executable, storage-allocating statements which assign variables and arrays to a storage area called COMMON storage and provide the facility for various program units to share the use of the same storage area.

COMMON statements are expressed in this form:

```
COMMON /<y1>/<a1>/<y2>/<a2>.../<yn>/<an>
```

Each $y_i$ is a <u>COMMON block storage name</u> and each $a_i$ is a sequence of variable names, array names or constant array declarators, separated by commas. The elements in $a_i$ make up the <u>COMMON block storage area</u> specified by the name $y_i$. If any $y_i$ is omitted, leaving two consecutive slash characters (//), the block of storage so indicated is called <u>blank COMMON</u>. If the first block name ($y_i$) is omitted, the two slashes may be omitted.

Example:

```
      COMMON /AREA/A,B,C/BDATA/X,Y,Z,
     X    FL,ZAP(30)
```

In this example, two blocks of COMMON storage are allocated--AREA with space for three variables, and BDATA, with space for four variables and the 30 element array, ZAP.

Example:

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

In this example, A1, B1, T2 and Z3 are assigned to blank COMMON in that order. The pair of slashes preceding A1 could have been omitted.

CDATA names COMMON block storage for the nine-element array, ZOT. Thus ZOT(3,3) is an array declarator. ZOT <u>must not have been previously declared</u>. See Array Declarators, Section 6.3.

Additional considerations:

1. The name of a COMMON block may appear more than once in the same COMMON statement, or in more than one COMMON statement.

2. A COMMON block name is made up of from 1 to 6 alphanumeric characters, the first of which must be a letter.

3. A COMMON block name must be different from any subprogram names used throughout the program.

4. The size of a COMMON area may be increased by use of EQUIVALENCE statements. See EQUIVALENCE Statements, Section 6.9.

5. The lengths of COMMON blocks of the same name need not be identical in all program units where the name appears. However, if the lengths differ, the program unit specifying the greatest length must be loaded first--see the discussion of LINK in the User's Guide. The length of a COMMON area is the number of storage units required to contain the variables and arrays declared in the COMMON statement or statements, unless expanded by EQUIVALENCE statements.

## 6.9 EQUIVALENCE Statements

EQUIVALENCE statements permit sharing the same storage unit by two or more entities. The general form of the statement is this:

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

where each $u_i$ represents a sequence of two or more variable or array elements, separated by commas. Each element in the sequence is assigned the same storage unit--or portion of one--by the compiler. The order in which 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 the number must be one, where the one subscript specifies the array element's number relative to the first element of the array.

As an 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.

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

   The COMMON block established by the following example establishes 3 storage units via the COMMON statement. It is expanded to 4 storage units by the EQUIVALENCE statement.

   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) | 0 |
   | X = R(2,1) | 1 |
   | Y = R(1,2) | 2 |
   | R(2,2) | 3 |

Note that EQUIVALENCE (X,R(3)) would be invalid in this 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(20), D(5)
   COMMON A,B(4)/ZAP/C,X
     .
     .
     .
   EQUIVALENCE (XTABLE(6), A(7)
  X      B(3),XTABLE(5)),
  Y      (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(5).
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.

## 6.10 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 has this form:

DATA list/$u_1$,$u_2$,...,$u_n$/,list/$u_k$,$u_{k+1}$,...$u_{k+n}$/

where "list" represents a list of variable, array, or array element names, and the $u_i$ 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 may appear in the corresponding position between slashes. Instead of $u_i$, it is permissible to write k*ui in order to declare the same constant, $u_i$, k times in succession. k must be a positive integer. Dummy arguments may not appear in the list.

Example:

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

This implies that:

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

The type of each constant $u_i$ must match the type of the corresponding item in the list, except that a Hollerith or Literal 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, and 4 or fewer characters for a Real variable.

If fewer Hollerith or Literal characters are specified, trailing blanks are added to fill the remainder of storage.

The example below illustrates many of the features of the DATA statement:

```
       REAL  LIT(2)
       LOGICAL LT,LF
       DIMENSION  H4(2,2),PI3(3)
       DATA A1,B1,K1,LT,LF,H4(1,1),H4(2,1)
      1    H4(1,2),H4(2,2),PI3/5.9,2.5E-4,
      2    64,.FALSE.,.TRUE.,1.75E-3,
      3    0.85E-1,2*75.0,1.,2.,3.14159/
      4    LIT(1)/'NOGO'/
```

This implies that:

```
A1 = 5.9                 H4(1,2) = 75.0
B1 = 2.5E-4              H4(2,2) = 75.0
K1 = 64                  PI3(1) = 1.
LT = .FALSE.             PI3(2) = 2.
LF = .TRUE.              PI3(3) = 3.14159
H4(1,1) = 1.75E-3        LIT(1) = NOGO
H4(2,1) = 0.85E-1
```

# CHAPTER 7. FORTRAN CONTROL STATEMENTS

## 7.1 Introduction

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.

## 7.2 GO TO Statements

### 7.2.1 Unconditional GO TO

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

Unconditional GO TO statements have this form:

```
        GO TO <k>
```

In this statement, k is the statement label of an executable statement in the same program unit.

Example:

```
              GO TO 376
        310   A(7) = V1 - A(3)
                .
                .
        376   A(2) = VECT
              GO TO 310
```

In these statements, statement 376 is executed prior to statement 310 in the logical flow of the program of which they are a part.

7.2.2 Computed GO TO

Computed GO TO statements have this form:

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

where $k_i$ are statement labels, and j is an integer variable greater than or equal to 1 and less than or equal to n.

This statement causes transfer of control to the statement labeled $k_i$. If j is less than or equal to 0 or if j is greater than n, control will be passed to the next statement following the Computed GOTO.

Example:

```
              J = 3
              .
              .
              .
              GO TO (7, 709, 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 5 changes the transfer to statement 70000. Making J = 0 or J = 6 would cause control to be transferred to statement 310.

## 7.2.3 Assigned GO TO

Assigned GO TO statements have this form:

```
GO TO <j>[(<k1>,<k2>,...,<kn>)]
```

In these examples, j is an integer variable name, and the $k_i$ are statement labels of executable statements. 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 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.

## 7.3 ASSIGN Statement

The ASSIGN statement has the following form:

```
ASSIGN <j> TO <i>
```

In this ASSIGN statement, j is a statement label of an executable statement and i is an integer variable.

The 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 100 TO LABEL
.
.
.
ASSIGN 90 TO LABEL
GO TO LABEL (80, 90, 100)
```

## 7.4 IF Statement

IF statements transfer control to one of a series of statements depending upon a condition. Two types of IF statements may be made:

Arithmetic IF
Logical IF

### 7.4.1 Arithmetic IF

Arithmetic IF statements have the form:

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

In this statement, e is an arithmetic expression and m1, m2, and m3 are statement labels.

Evaluation of expression e determines one of three transfer possibilities:

| If e is: | Transfer to: |
|---|---|
| <0 | m1 |
| =0 | m2 |
| >0 | m3 |

Examples:

| Statement: | Expression Value: | Transfer to: |
|---|---|---|
| IF (A)3,4,5 | 15 | 5 |
| IF (N-1)50,73,9 | 0 | 73 |
| IF (AMTX(2,1,2))7,2,1 | -256 | 7 |

## 7.4.2 Logical IF

The Logical IF statement has this form:

```
IF (<u>)<s>
```

In this Logical IF, u is a Logical expression and s is any executable statement except a DO statement (see Section 7.5, item 4) or another Logical IF statement. The Logical expression u is evaluated as .TRUE. or .FALSE. See Chapter 4 for a discussion of Logical expressions.

### 7.4.2.1 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, see Chapter 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 line spaces following IF(u) left blank. See example 4 below.

Examples:

1. `IF(I.GT.20) GO TO 115`

   If the evaluation of I is greater than 20, program control will be transferred to label 115.

2. `IF(Q.AND.R) ASSIGN 10 TO J`

   If the evaluation of (Q.AND.R) is "true", the variable J will be assigned the number 10.

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

   If the evaluation of Z is "true", program control will be passed to the subroutine DECL.

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

   or

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

## 7.5 DO Statement

The DO statement, as implemented in FORTRAN, provides a method for repetitively executing a series of statements. The DO statement takes the following form:

```
DO <k>,<i> = <m1>,<m2>[,<m3>]
```

where k is a statement label, i is an integer or logical variable, and m1, m2, and m3 are integer constants or integer or logical variables. If m3 is not specified, it is assumed to be a constant 1.

The elements of the above DO statement are defined as follows:

| | |
|---|---|
| k | is the terminal statement. |
| i | is the index or controlling variable. |
| m1 | is the initial index value. |
| m2 | is the terminal index value. |
| m3 | is the index increment value. |

The following conditions and restrictions govern use of DO statements:

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

2. The statement labeled k, called the terminal statement, must be an executable statement.

3. The terminal statement must physically follow its associated DO, and the executable statements following the DO, up to and including the terminal statement, constitute the range of the DO statement.

4. The terminal statement may not be an Arithmetic IF, GO TO, RETURN, STOP, PAUSE, or another DO.

5. If the terminal statement is a Logical IF and its expression is .FALSE., then the statements in the DO range are reiterated.

   If the expression is .TRUE., the statement of the Logical IF is executed and then the statements in the DO range are reiterated. The statement of the Logical IF may not be a GO TO, Arithmetic IF, RETURN, STOP, PAUSE, or DO statement.

6. The controlling integer variable, i, is called the index of the DO range. The index must be positive and may not be modified by any statement in the range.

7. If m1, m2, and m3 are Integer*1 variables or constants, the DO loop will execute faster and be shorter, but the range is limited to 127 iterations.

8. During the first execution of the statements in the DO range, i is equal to m1; the second execution, i = m1+m3; the third, i=m1+2*m3, etc., until i is equal to the highest value in this sequence which is less than or equal to m2, at which time the DO is said to be satisfied. The statements in the DO range will always be executed at least once, even if m1 is greater than or equal to m2.

   When the DO has been satisfied, control passes to the statement following the terminal statement. If the DO is not yet satisfied, control transfers back to the first executable statement following the DO statement.

   Example:

```
100 DIMENSION A(100)
    .
    .
    .
    SUM = A(1)
    DO 31 I = 2,100
 31 SUM = SUM + A(I)

    END
```

9. The range of a DO statement may be extended to include all statements which may logically be executed between the DO and its terminal statement. Thus, parts of the DO range may be situated such that they are not physically between the DO statement and its terminal statement, but are executed logically in the DO range. This is called the extended range.

Example:

```
   DIMENSION A(500), B(500)
   .
   .
   .
   DO 50 I = 10, 327, 3
   .
   .
   .
   .
   IF (V7 - C*C) 20, 15, 31
30
   .
   .
   .
50 A(I) = B(I) + C
   .
   .
   .
20 C =C - .05
   GO TO 50
31 C = C + .0125
   GO TO 30
```

10. It is invalid to transfer control into the range of a DO statement from a statement not itself in the range or extended range of the same DO statement.

11. Within the range of a DO statement, there may be other DO statements, in which case the DO's must be nested. That is, if the range of one DO contains another DO, then the range of the inner DO must be entirely included in the range of the outer DO.

    The terminal statement of the inner DO may also be the terminal statement of the outer DO.

    For example, given a two-dimensional array A of 15 rows and 15 columns, and a 15 element one-dimensional array B, the following statements compute the 15 elements of array C to the formula:

Formula:

$$C_k = \sum_{j=1}^{15} A_{kj}B_m, \; k=1,2,...,15$$

Code:

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

NOTE: In the above example, the inner DO loop will be completed before the next iteration of the outer DO loop.

## 7.6 CONTINUE Statement

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

```
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 those which are not allowed or is only executed conditionally.

Example:

```
      DO 5 K =1,10
      .
      .
      .
      IF (C2) 5,6,6
 6    CONTINUE
      .
      .
      .
      C2 = C2 + .005
 5    CONTINUE
```

## 7.7 STOP Statement

A STOP statement has the form:

```
STOP [<c>]
```

As shown here, c is any string of one to six characters.

When STOP is encountered during execution of the object program, the characters c (if present) are displayed on the processor screen and program execution terminates.

The STOP statement, therefore, constitutes the logical end of the program, and must be located before the END statement in a program.

## 7.8 PAUSE Statement

PAUSE statement has the following form:

```
PAUSE [<c>]
```

In the above PAUSE statement, c is any string of up to six characters.

When PAUSE is encountered during execution of the object program, the characters c (if present) are displayed on the processor screen and program execution is suspended.

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

During a PAUSE, the operator may resume execution of the program by pressing the DISPLAY key.

## 7.9 CALL Statement

CALL statements control transfers into SUBROUTINE subprograms and provide parameters for use by the subprograms. The general form of CALL statements and a detailed discussion of their structure appear in Chapter 9, FUNCTIONS AND SUBPROGRAMS.

## 7.10 RETURN Statement

The form, use, and interpretation of the RETURN statement is described in Chapter 9.

## 7.11 END Statement

The END statement must physically be the last statement in any FORTRAN program. Its form is:

```
          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, which returns control to the DOS.

# CHAPTER 8. INPUT / OUTPUT

## 8.1 Introduction

FORTRAN provides a series of statements which define the control and conditions of data transfer between computer memory and external devices.

These statements are grouped as follows:

1. Formatted READ and WRITE statements which cause formatted information to be transferred between the computer and I/O devices.
2. Unformatted READ and WRITE statements which transfer unformatted binary data in a form similar to internal storage.
3. Auxiliary I/O statements for positioning and demarcation of files.
4. ENCODE and DECODE statements for transferring data between memory locations.
5. FORMAT statements used in conjunction with formatted record transfer to provide data conversion and editing information between internal data representation and external character string forms.

## 8.2 Formatted READ/WRITE Statements

### 8.2.1 Formatted READ Statements

A formatted READ statement is used to transfer information from an input device to the processor.

The form of a READ statement is as follows:

```
READ (<u>,<f>[,END=<l1>][,ERR=<l2>][,REC=<n> | KEY=<xx>])[<k>]
```

The formatting elements are defined as follows:

u - specifies a Physical and Logical Unit Number and may be either an unsigned integer or an integer variable in the range 1 through 8. If an Integer variable is used, an Integer value must be assigned to it prior to execution of the READ statement.

Logical Unit Number (LUN) 9 is preassigned to the console. LUN 10 is preassigned to the Local Printer--if one is attached to the processor and online.

LUNs 1-8 are assumed to reference disk files. If not specifically opened to a named file, default is to a name in the form of "FORnnDAT" where the "nn" represents the LUN. The default assignments may be overridden by the OPEN (to a file name) subroutine, permitting LUNs 1-8 to be reassigned by the user.

When a formatted file is opened without specifying the file name, the extension defaults to /TXT. When an unformatted file is opened without the filename specified, the assumed default extension is /TMP.

f - is the statement label of the FORMAT statement describing the type of data conversion to be used within the input transmission, or it may be an array name, in which case the formatting information may be input to the program at execution time. Section 8.9.10 has more details.

l1 - is the FORTRAN label on the statement to which the I/O program routine will transfer control if an End-of-File is encountered.

l2 - is the FORTRAN label on the statement to which the I/O program routine will transfer control if an I/O error is encountered.

k - is a list of variable names, separated by commas, specifying the input data.

n - defines the record number to be used in a READ statement using the Random Access method.

xx - defines the Key to be used in a READ statement using Indexed Sequential Access method (ISAM).

READ (<u>,<f>)<k> is used to input a number of items, corresponding to the names on the list k, from the file on logical unit u, and using the FORMAT statement f to specify the external representation of these items (FORMAT Statements, Sec. 8.9). The ERR = and END = clauses are optional. If not specified, I/O and End-of-File errors cause fatal run-time errors.

The following notes further define the function of the READ (<u>,<f>)<k> statement:

1. Each time execution of the READ statement begins, the program reads a new record from the input file.

2. The number of records to be input by a single READ statement is determined by the list, k, and format specifications.

3. The list k specifies the number of items to be read from the input file and the locations into which they are to be stored.

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

5. If there are more quantities in an input record than there are items in the list, only the number of quantities equal to the number of items in the list are transmitted. Remaining quantities are ignored.

6. Exact specifications for the list k are described in 8.8.

Examples:

1. Assume that four data entries are in a disk record, with three blanks separating each, and that the data have field widths of 3, 4, 2 and 5 characters respectively, starting in column 1 of the record. The statements:

```
      READ(5,20)K,L,M,N
    20  FORMAT(I3,3X,I4,3X,I2,3X,I5)
```

   read the record (assuming the Logical Unit Number 5 has been assigned to the file via an OPEN) and assign the input data to the variables K, L, M, and N. The FORMAT statement could also be:

```
    20  FORMAT(I3,I7,I5,I8)
```

   Section 8.9 contains a complete description of FORMAT statements.

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

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

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

READ(u,k) may be used in conjunction with a FORMAT statement to read H-type alphanumeric data into an existing H-type field. (Hollerith Conversions are discussed in Section 8.9.3).

## 8.2.2 Formatted WRITE Statements

A formatted WRITE statement is used to transfer information from the processor to an output device.

The form of the statement is:

```
WRITE (<u>,<f>[,ERR=<l1>][,END=<l2>][,REC=<n> | KEY=<xx>])[<k>]
```

The format elements of these statements are defined as follows:

| | |
|---|---|
| u - | specifies a Logical Unit Number. |
| f - | is the statement label of the FORMAT statement describing the type of data conversion to be used with the output transmission. |
| l1 - | specifies an I/O error branch. |
| l2 - | specifies an EOF branch. |
| k - | is a list of variable names separated by commas, specifying the output data. |
| n - | defines the record number to be used in a WRITE statement using the Random Access method. |
| xx - | defines the Key to be used in a WRITE statement using the Indexed Sequential Access method (ISAM). |

WRITE (<u>,<f>)<k> is used to output the data specified in the list k to a file on logical unit u using the FORMAT statement f to specify the external representation of the data (see FORMAT statements, Sec. 8.9). The following notes further define the function of the WRITE statement:

1. Several records may be output with a single WRITE statement, with the number determined by the list and FORMAT specifications.

2. Successive data are output until the data specified in the list are exhausted.

3. If output is to a device which specifies fixed length records and the data specified in the list do not fill the record, the remainder of the record is filled with blanks.

Example:

```
    WRITE(2,10)A,B,C,D
```

The data assigned to the variables A, B, C, and D are output to the Logical Unit Number 2, formatted according to the FORMAT statement labeled 10.

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)
```

## 8.3 Unformatted READ/WRITE

Unformatted I/O--that which does not have data conversion--is accomplished with READ and WRITE statements.

The following notes define the functions of unformatted I/O statements.

1. Unformatted READ/WRITE statements perform memory-image transmission of data with no data conversion or editing.
2. The amount of data transmitted corresponds to the number of variables in the list k.
3. An error will result if the total length of the list of variable names in an unformatted READ is longer than the record length. If the logical record length and the length of the list are the same, the entire record is read. If the length of the list is shorter than the logical record length, the unread items in the record are skipped.
4. The WRITE(u)k statement writes one logical record.
5. A logical record may extend across more than one physical record.
6. Unformatted files may only be processed sequentially.

## 8.3.1 Unformatted READ

The unformatted READ has the following form:

```
READ (<u>[,ERR=<l1>])(<k>)
```

In the above example:

u - specifies a Logical Unit Number.

l1 - specifies an I/O error branch.

k - is a list of variable names, separated by commas, specifying the I/O data.

## 8.3.2 Unformatted WRITE

The unformatted WRITE has the following form:

```
WRITE (<u>[,ERR=<l1>])[<k>]
```

In this example:

u - specifies a Logical Unit Number.

l1 - specifies an I/O error branch.

k - is a list of variable names, separated by commas, specifying the I/O data.

## 8.4 File Formats

FORTRAN deals with two different types of files: formatted and unformatted. While it is possible to write formatted records into an unformatted file and unformatted records into an formatted file, such unorthodox operations would result in a non-standard file, the structure of which would be generally unintelligible. Programs attempting to read such a file will generate I/O errors. Format types should not be mixed within a single file.

### 8.4.1 Formatted Files

FORTRAN formatted I/O reads and writes DOS standard text files (described in the DOS User's Guide, in the chapter titled SYSTEM STRUCTURE). All access methods use record-compressed format. Space compression can be turned on or off by the SPCON/SPCOFF routine after a file has been opened. Sequential and ISAM access default to space compression on; random access defaults to space compression off. Once space compression has been set on or off for a file, it remains in the set-on or set-off condition until specifically changed by the SPCON/SPCOFF routine, or until the file is closed through ENDFILE, or until FORTRAN program termination.

Maximum record length is determined by the logical record length specified or assumed when the file is opened. Sequential and ISAM files use variable length records; random access files use fixed-length records. When writing a random file, if the format specified writes too few characters to satisfy the specified record length, the record is padded with blanks. Using any access method, if the format specified writes more characters than fit in the specified record length, the record is truncated and trailing characters are discarded.

The text end-of-file (EOF) is automatically written when using sequential or ISAM access. A sequential EOF is written when the file is closed; an ISAM EOF is maintained throughout all file modification. FORTRAN does not write an EOF to a random file. If a random file is to be accessed sequentially (by FORTRAN or by any other Datapoint facility), an EOF must be written to it by some other means to avoid a record format error. The problem may be avoided in two ways:

1. Do not attempt to read a FORTRAN random access file sequentially.

2. Pre-allocate the random file, and write the EOF to it using FORTRAN sequential access or using another language.

For example, the file could be opened for sequential access and the desired number of fixed-length records, all blanks, written to it; then an EOF can be written when the file is closed. If this procedure is followed, the file cannot grow past its original pre-allocated size without destroying the EOF. The file can only increase in size by repeating the pre-allocation procedure to obtain more space.

Space compression may be turned on for a random file, but doing so serves no purpose. Since all records of a random file must be physically the same length, when space compression is on, a string of spaces will be written in normal compressed format, then the number of bytes required for the compressed spaces will be padded out with deleted data characters (032). As an example, a random file with space compression on the string 040 040 040 040 would have the string written as 011 004 032 032. Both strings will produce the same result (four spaces) when read.

### 8.4.2 Unformatted Files

FORTRAN unformatted files are binary image files, not DOS text files. Unformatted files are readable only by FORTRAN unformatted I/O statements and are not compatible with other Datapoint languages or utilities. An unformatted file uses fixed-length records. There are no logical end-of-record or end-of-sector marks. When the physical end of a sector is reached on an unformatted read or write, the I/O operation continues with the first data byte of the next sector. Because of the lack of compatibility and the lack of format testing on input, unformatted files should not normally be used for data storage and, if used, should always be accessed sequentially.

Unformatted files typically are used only for temporary storage of working data during a program, as implied by the default extension /TMP. Unformatted I/O is somewhat faster than formatted I/O since no data translation is performed; and the binary representation is more compact than ASCII representation, so fewer bytes are needed.

## 8.5 Disk File I/O

A READ or WRITE to a disk file (Logical Unit Numbers 1 through 8) automatically OPENs the file for I/O. The file remains open until closed by an ENDFILE command (discussed in Section 8.6) or until normal program termination.

NOTE

Exercise caution when doing sequential output to disk files. If output is done to an existing file, the existing file will be overwritten by the new file data.

## 8.5.1 Random Disk I/O

For random disk access, the record number is specified by using the REC=<n> option in the READ or WRITE statement. For example:

```
        I = 10
        WRITE (6,15,REC=I,ERR=50) X, Y, Z
   15   FORMAT (I7,I4,I6)
```

This program segment writes a record 10 (REC=I) on Logical Unit Number 6. If a previous record 10 exists, it is written over. If no previous record 10 exists, the file is extended to create one. Any attempt to read a non-existent record results in an I/O error.

For the specific case of reading a random file sequentially, the file must first be built sequentially; otherwise, when no EOF is found, the read terminates in a record format error. Such a file is created by an initial prepararation of writing sequentially the maximum number of records the random file is expected to contain and filling them with blanks. For example (where b=blank):

```
        WRITE (6,10)
   10   FORMAT ('50b')
```

This creates a sequential file that can be written and read randomly, as well as a random access file that can be read sequentially.

Any disk file OPENed by a READ or WRITE statement is assigned a default filename in the form of: FORnnDAT/xxx. In this form, nn=LUN used in the READ/WRITE statement that references the file, and /xxx=filename extension where /TXT if the default for a formatted file and /TMP is the default for an unformatted file. The logical record length of a default file is 128 bytes (127 characters plus end-of-record mark for formatted files, 32 storage units for unformatted). A file may be default-opened only once in a program; a file may be opened more than once, but subsequent opens can only be done by using the OPEN subroutine.

## 8.5.2 OPEN Subroutine

Alternatively, a file may be OPENed using the OPEN subroutine. Logical Unit Numbers 1 through 8 may be assigned to disk files with OPEN. The OPEN subroutine allows the program to specify a filename and device to be associated with a Logical Unit Number.

Once a Logical Unit Number is associated with a file through the OPEN subroutine, that LUN may not be associated with another file in the same program. This limits the number of files allowed in one program to 8.

When attempting to create a new file with the OPEN subroutine, the file does not exist until written to. An OPEN of an existing file followed by sequential output overwrites the existing file. An OPEN of an existing file followed by an input allows access to the current contents of the file.

The OPEN subroutine call has this form:

```
CALL OPEN (<lun>,<filename>,<lrl>)
```

where lun = the Logical Unit Number assigned to the file; filename is the file to be opened; and lrl is the maximum length of a record in the file.

Logical Unit Number may specify a disk or printer. <filename> is any constant or variable that may include the extension and drive specification for the file.

The file specification must be terminated by a non-alphanumeric character that is neither "/" nor ":" The default extension for the file specification is /TXT.

Example: CALL OPEN(1,'FILEINb',50)

Where: b=blank and 50=lrl

lrl = logical record length. The first time a LUN is opened, lrl must be the maximum that will be used with that LUN. If a file is closed, then opened again, lrl must be less than or equal to the value of the first OPEN. The logical record length must <u>always</u> be specified when the OPEN subroutine is called.

When a file is opened with the OPEN subroutine, lrl of formatted files must be logical record length plus one byte more for the end-of-record mark; for unformatted files, lrl must be logical record length only.

If LUN 1, 2, or 3 is OPENed with a filename of "b" (blank), the filename will be taken from the COMMAND line default.

## 8.5.3 Updating in Place with REWRITE

It is frequently necessary to update a logical record in place on the disk. The REWRITE statement provides this capability.

### 8.5.3.1 Form of the REWRITE Statement

The REWRITE statement has the same syntax as a WRITE statement. For example:

```
REWRITE (<u>,<f>[,END=<l1>][,ERR=<l2>][,REC=<n> ¦ KEY=<xx>])[<k>]
```

In this example:

u - specifies a Logical Unit Number.

f - is the statement label of the FORMAT statement describing the type of data conversion to be used with the output transmission.

l1 - specifies an I/O error branch.

l2 - specifies an EOF branch.

k - is a list of variable names, separated by commas, specifying the I/O data.

n - defines the record number to be used in a REWRITE statement using the Random Access method.

xx - defines the Key to be used in a REWRITE statement using Indexed Sequential Access method (ISAM).

Example of Random REWRITE:

```
I=10
READ (6,100,REC=I,ERR=50) X, Y, Z
REWRITE (6,100,REC=I,ERR=50) X, Y, Z
```

This program segment reads record 10 on Logical Unit Number 6. It then rewrites record 10 on Logical Unit Number 6. The previous record 10 must exist and it is overwritten.

If no previous record 10 exists, an I/O error occurs and control will be transferred to label 50.

The effect of this is to replace the data read from the file in the last READ statement with the data from the REWRITE statement.

NOTE: The READ must have been a random READ.

An ISAM REWRITE is identical to the random REWRITE shown above, except that KEY= option replaces REC= option to read the file.

### 8.5.3.2 Considerations of Updating In Place

The size of the record or records named in the READ statement and the subsequent REWRITE statement must be identical.

## 8.5.4 Indexed Sequential Files

A file with an extension of /ISI is assumed to be an index file. It must first be created by the DOS INDEX utility before ISAM READs or WRITEs are performed and may be either a null file or a file created sequentially. The index file contains the name of the indexed data file. When a file is opened with the extension /ISI, both the index file and data file are associated with the LUN.

READ and WRITE operations are normally performed on the data file in physical order by logical records. If a "KEY=" option is specified in a READ statement, the data file is positioned to the logical record associated with the specified key, and then the READ operation is performed. The key value must be terminated by an octal character less than 040.

NOTE: Set the key array to zero initially.

A null key value is used to specify that the data file is to be read in key sequence. If the specified key is not in the index file, the data file is positioned to the end of the file in preparation for adding a new key and the associated data. The "END=" option is used to specify a "key not found" exit.

A WRITE statement with the "KEY=" option specified is used to insert a new key into the index and to begin writing the data associated with the key. A null key is not allowed in this case.

### 8.5.5 Deleting or Inserting Multiple ISAM Keys

In Datapoint's file structure, ISAM files may have more than one index (/ISI) file associated with them. For deleting keys and inserting keys in an index file when an indexed file has more than one index, the following routines are provided:

```
CALL DELKEY (<lun>,<key>)

CALL INSKEY (<lun>,<key>)
```

DELKEY deletes the specified key from the index. It also marks the corresponding data record as deleted by overstoring it with 032 (octal) codes.

INSKEY inserts the specified key in the index. The key points to the end of the file where the new data will be written.

When multiple indices are used, each index file must be opened as a separate LUN. All but one of the keys must be inserted in the corresponding index file with INSKEY. The first key must be inserted in its index file with the "KEY=" option of the WRITE statement.

## 8.6 Auxiliary I/O Statements

Four auxiliary I/O statements are provided:

```
REWIND <u>
ENDFILE <u>
CALL SPCON <u>
CALL SPCOFF <u>
```

The actions of these statements depend on the Logical Unit Number with which they are used. When the LUN is for a local or Servo printer, the REWIND and ENDFILE statements cause no action.

When the LUN specifies a disk file, the ENDFILE and REWIND commands allow further program control of disk files. ENDFILE <u> closes the file associated with Logical Unit Number u and writes an EOF. REWIND <u> closes the file associated with LUN u, then opens it again. It does not write an EOF.

Space compression default for random files is OFF and must be specified ON with SPCON <u> if desired.
For sequential files, space compression default is ON, and must be specified OFF by CALL SPCOFF <u>. Files should not be mixed OFF and ON.

## 8.7 ENCODE/DECODE

ENCODE and DECODE statements transfer data, according to format specifications, from one section of memory to another. DECODE changes data from ASCII format to the specified format. ENCODE changes data of the specified format into ASCII format. The two statements are of the form:

```
ENCODE(<a>,<f>)<k>
DECODE(<a>,<f>)<k>
```

a, f, and k are defined as follows:

a is an array name
f is FORMAT statement number
k is an I/O list

DECODE is analogous to a READ statement, since it causes conversion from ASCII to internal format. ENCODE is analogous to a WRITE statement, causing conversion from internal formats to ASCII.

NOTE:

Care should be taken that the array a is always large enough to contain all of the data being processed. There is no check for overflow. An ENCODE operation which overflows the array will probably wipe out important data following the array. A DECODE operation which overflows will attempt to process the data following the array.

## 8.8 INPUT/OUTPUT List Specifications

Most forms of READ/WRITE statements may contain an ordered list of data names which identify the data to be transmitted. The order in which the list items appear must be the same as that in which the corresponding data exists (INPUT), or will exist (OUTPUT), in the external I/O medium.

Lists have the following form:

```
<m1>,<m2>,...,<mn>
```

where the $m_i$ are list items separated by commas, as shown.

### 8.8.1 List Item Types

A list item may be a single datum identifier or a multiple data identifier.

#### 8.8.1.1. Single Datum Identifier

A single datum identifier item is the name of a variable or array element. One or more of these items may be enclosed in parentheses without changing their intended meaning.

Examples:

```
A
C(26,1),R,K,D,I,J
B,I(10,10),S,(R,K),F(1,25)
```

#### 8.8.1.2. Multiple Data Identifiers

Multiple data identifier items are in two forms:

a. An array name appearing in a list without subscript(s) is considered equivalent to the listing of each successive element of the array.

Example:

If B is a two-dimensional array, the list item B is equivalent to:

```
B(1,1),B(2,1),B(3,1)...,B(1,2),B(2,2)...,B(j,k)
```

where j and k are the subscript limits of B.

b. DO-implied items are lists of one or more single datum identifiers or other DO-implied items followed by a comma character and an expression of the form:

```
<i> = <m1>,<m2>[,<m3>]
```

DO-implied items are enclosed in parentheses as shown below.

The elements i, m1,m2,m3 have the same meaning as defined for the DO statement. The DO implication applies to all list items enclosed in parentheses with the implication.

Examples:

| DO-Implied Lists | Equivalent Lists |
|---|---|
| (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,ZAP(3) | R(1),R(2),I,ZAP(3) |
| (R(3),T(I),I=1,3) | R(3),T(1),R(3),T(2), R(3),T(3) |

In this way the elements of a matrix, for example, may be transmitted in an order different from the order in which they appear in storage. The array A(3,3) occupies storage in the order:

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

By specifying the transmission of the array with the DO-implied list item ((A(I,J),J=1,3),I=1,3), the order of transmission becomes:

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

## 8.8.2 Special Notes On List Specifications

1. The ordering of a list is from left to right with repetition of items, other than those as subscripts, enclosed in parentheses when accompanied by controlling DO-implied index parameters.

2. Arrays are transmitted by the appearance of the unsubscripted array name in an input/output list.

3. Constants may appear in an input/output list only as subscripts or as indexing parameters.

4. For input lists, the DO-implying elements i, m1,

m2 and m3, may not appear within the parentheses as list items.

Examples:

```
READ (1,20)(I,J,A(I),I=1,J,2) is not allowed.
```

```
READ (1,20)I,J,(A(I),I=1,J,2) is allowed, but the
read value of I will be lost.
```

```
WRITE(1,20)(I,J,A(I),I=1,J,2) is allowed.
```

Consider the following examples:

```
   DIMENSION A(25)

   A(1) = 2.1
   A(3) = 2.2
   A(5) = 2.3
   J = 5

   WRITE (1,20) J,(I,A(I),I=1,J,2)
   .
   .
   .
```

The output of this WRITE statement is:

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

5. Any number of items may appear in a single list.

6. In a formatted READ or WRITE, each item of the list must have the correct type as specified by a FORMAT statement.

## 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 this:

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

In this statement, n is the statement label, each $s_i$ is a field descriptor, and xxx is a literal string. The word FORMAT and the parentheses must be present as shown. The slash (/) and the comma (,) characters are field separators and are described in Section 8.9.7.1. 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: |
|---|---|
| <r>F<w>.<d> | Numeric Conversion |
| <r>G<w>.<d> | Numeric Conversion |
| <r>E<w>.<d> | Numeric Conversion |
| <r>D<w>.<d> | Numeric Conversion |
| <r>I<w> | Numeric Conversion |
| <r>L<w> | Logical Conversion |
| <r>A<w> | Hollerith Conversion |
| $nHh_1h_2...h_n$ | Hollerith Conversion |
| $'l_1l_2...l_n'$ | Hollerith Conversion |
| nX | Spacing Specification |
| mP | Scaling Factor |

Elements of the Field Descriptors are defined:

1. w and n are positive integer constants defining the field width--including digits, decimal points, and 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, H I, A, and L indicate the type of conversion to be applied to the items in an input/output list. They are discussed in the following sections.

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

5. The $h_i$ and $l_i$ are characters from the FORTRAN character set.

6. m is an integer constant, either 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, spaces, or any nonnumeric characters, may be used to separate the fields in the external representation.

#### 8.9.2.1 F-type conversion

Form: Fw.d

Real or Double Precision type data are processed using this conversion. w characters are processed, of which d are considered fractional.

#### 8.9.2.2 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. If a value requires more field positions than allowed by w , the first w-1 digits of the value are output, preceded by an asterisk.

F-Output Examples:

| FORMAT Descriptor | Internal Value | Output (b=blank) |
|---|---|---|
| F10.4 | 368.42 | bb368.4200 |
| F7.1 | -4786.361 | -4786.4 |
| F8.4 | 8.7E-2 | bb0.0870 |
| F6.4 | 4739.76 | * .7600 |
| F7.3 | -5.6 | b-5.600 |

* Note the loss of leading digits in the 4th line above.

#### 8.9.2.3 F-Input

See the description under E-Input, Section 8.9.2.6.

## 8.9.2.4 E-type Conversion

Form: Ew.d

Real or Double Precision type data are processed using this conversion. w characters are processed, of which d are considered fractional.

## 8.9.2.5 E-Output

Values are converted, rounded to d digits, and output in the following order as:

1. a 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 blank)
6. two exponent digits

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. Some E-Output examples follow:

| FORMAT Descriptor | Internal Value | Output (b=blank) |
|---|---|---|
| E12.5 | 76.573 | bb.76573Eb02 |
| E14.7 | -32672.354 | -b.3267235Eb05 |
| E13.4 | -0.0012321 | bb-b.1232E-02 |
| E8.2 | 76321.73 | b.76Eb05 |

## 8.9.2.6 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 each conversion and is as follows:

1. Leading spaces (ignored)
2. A plus or a minus 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 plus or a minus 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 items 3 and 5 (above) are present, then 6 or 7 or both 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 Chapter 3 for Real data.

E- and F- and G-Input Examples:

| FORMAT Descriptor | Input (b=blank) | Internal Value |
|---|---|---|
| E10.3 | +0.23756E+4 | +2375.600 |
| E10.3 | bbbbb17631 | +17.631 |

| | | |
|---|---|---|
| G8.3 | b1628911 | +1628.911 |
| F12.4 | bbbb-632113 | -632.1130 |

Note in the above examples that if no decimal point is given among the input characters, the fractional length specification 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 fractionl length specification is ignored.

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

### 8.9.2.7 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".

### 8.9.2.8 G-Type Conversions

Form: Gw.d

Real or Double Precision type data are processed using this conversion. w characters are processed, of which d are considered significant. Note that G-type conversions differ from E-type and F-type in that d refers to significant digits, not to fractional digits.

### 8.9.2.9 G-Input:

See the description under E-Input

### 8.9.2.10 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: |
|---|---|
| .1 <= n< 1 | F(w-4).d,4X |
| 1 <= n< 10 | F(w-4).d-1),4X |
| . | . |
| . | . |
| . | . |
| $10^{d-2}$ <= n < $10^{d-1}$ | F(w-4).1,4X |

| | |
|---|---|
| $10^{d-1} <= n < 10^d$ | F(w-4).0,4X |
| Otherwise | Ew.d |

## 8.9.2.11 I-Type Conversions

Form: Iw

Only Integer data may be converted by this form of conversion. w specifies field width.

## 8.9.2.12 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. If the value exceeds the field width, only the least significant w-1 characters are output preceded by an asterisk.

Examples:

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

## 8.9.2.13 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

### 8.9.3.1 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.

### 8.9.3.2 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 4n 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 Value | Type | Output (b=blanks) |
|---|---|---|---|
| A1 | A1 | Integer | A |
| A2 | AB | Integer | AB |
| A3 | ABCD | Real | ABC |
| A4 | ABCD | Real | ABCD |
| A7 | ABCD | Real | bbbABCD |

### 8.9.3.3 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 | BC |
| A4 | ABCD | Integer | CD |
| A1 | A | Real | Abbb |
| A7 | ABCDEFG | Real | DEFG |

### 8.9.3.4 H-Type Conversion

The forms of H conversion are as follows:

$nHh_1h_2...h_n$

$'h_1h_2...h_n'$

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

NOTE:

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.

### 8.9.3.5 H-Output

The n characters $h_i$ are placed in the external field. In the $nHh_1h_2...h_n$ 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=blanks) |
|---|---|---|
| 1HA | or 'A' | A |
| 8HbSTRING | or 'bSTRINGb' | bSTRINGb |

| | | |
|---|---|---|
| 11HX(2,3)=12.0 or 'X(2,3)=12.0' | | X(2,3)=12.0 |
| 11HIbSHOULDN'T or 'IbSHOULDN''T' | | IbSHOULDN'T |

### 8.9.3.6 H-Input

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

Examples:

| 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

### 8.9.4.1 L-Output

If the value of an item in an output list corresponding to this descriptor is 0, an F will be output; otherwise, a T will be output. If w is greater than 1, w-1 leading blanks precede the letters.

Examples:

| FORMAT Descriptor | Internal Value | Output (b=blank) |
|---|---|---|
| L1 | =0 | F |
| L1 | <>0 | T |
| L5 | <>0 | bbbbT |
| L7 | =0 | bbbbbbF |

### 8.9.4.2 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 circumstances, this descriptor causes the next n characters of the input record to be skipped.

Output Examples:

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

Input Examples:

| FORMAT Statement | Input String | Resultant Input |
|---|---|---|
| 10 FORMAT (F4.1,3X,F3.0) | 12.5ABC120 | 12.5,120 |
| 5 FORMAT (7X,I3) | 1234567012 | 012 |

### 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.

#### 8.9.6.1 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 10**n less than the external value; that is, the number will be divided by 10**n before being stored.

#### 8.9.6.2 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 10**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 Repeat Specifications

1. The E, F, D, G, I, L, and A field descriptors may be indicated as repetitive descriptors by using a repeat count r in the form rEw.d, rFw.d, rGw.d, rIw, rLw, or 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 (2I3,2A5,2E10.5)
C    IS EQUIVALENT TO:
    14 FORMAT (I3,I3,A5,A5,E10.5,E10.5)
```

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         F.6.1,2X)
```

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 first opening parenthesis

in the FORMAT statement. 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 a new record and initiates the processing of a new record each time re-use begins. Record demarcation under these circumstances is the same as in the Section 8.9.7.1.

Input Example:

```
      DIMENSION A(100)
      READ (3,13) A
      .
      .
      .
   13 FORMAT (5F7.3)
```

In this example, the first 5 quantities from each of 20 records are input and assigned to the array elements of the array A.

Output Example:

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

In this example, two records are written. Record 1 contains E, F, K, L, and M. Because the FORMAT statement has been exhausted, the FORMAT statement is re-used starting at the first left parenthesis and record 2 contains AA, BB, K3, L3, and M3.

## 8.9.7.1 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 also specifies the demarcation of formatted records.

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

Output Example:

```
    DIMENSION A(100),J(20)
    .
    .
    .
    WRITE (7,8) J,A
8   FORMAT (10I7/10I7/50F7.3/50F7.3)
```

In the above 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   J(1), J(2), ... J(10)

Record 2   J(11), J(12) ... J(20)

Record 3   A(1), A(2), ... A(50)

Record 4   A(51), A(52), ... 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 arrays 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, D, 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, D, G, I, L, or A in the FORMAT statement.
6. H and X descriptors communicate information directly between the external record 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, D, 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 re-used. (See item 3 in the description of Repeat Specifications, Sec. 8.8.7.)

10. 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 blanks to maintain record lengths.

## 8.9.9 FORMAT Carriage Control

The first character of every print-formatted output record is used to convey carriage control information to the printer, and is therefore never printed. The carriage control character determines what action will be taken before the line is printed. The options are as follows:

| Control Character | Action Taken Before Printing |
|---|---|
| 0 | Skip 2 lines |
| 1 | Insert Form Feed |
| + | No advance |
| Other | Skip 1 line |

## 8.9.10 FORMAT Specifications in Arrays

The FORMAT reference, f, of a formatted READ or WRITE statement (Sec. 8.2) may be an array name instead of a statement label. If such 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, that is, it begins with a left parenthesis and ends with a right parenthesis.

The FORMAT specifications 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. As an example, assume that the FORMAT specification:

```
(3F10.3,4I6)
```

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

The FORTRAN coding below shows the various methods of establishing the above-example 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','0.3,','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
           .
           .
           .
```

# CHAPTER 9. FUNCTIONS AND SUBPROGRAMS

## 9.1 Introduction

The FORTRAN language provides a means for defining and using often-needed programming procedures in such a way that the statement or statements of the procedures need appear only once in a program, but may be referenced and brought into the logical execution sequence whenever 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 definition purposes. These requirements are discussed in subsequent sections of this chapter. However, certain features are common to the whole group, or to two or more of them. These common features are:

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 two procedures are designated "functions" and are alike in that:

   a. They are always single-valued--i.e., they return one value to the program unit from which they are referenced.

   b. They are referred to by an expression containing a function name.

   c. They must be typed by type-specification statements if the data type of the single-valued result is to be different from that indicated by the predefined convention.

3. FUNCTION 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.

## 9.2 The PROGRAM Statement

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

```
PROGRAM name
```

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

## 9.3 Statement Functions

Statement functions are defined by a single arithmetic or logical assignment statement and are relevant only to the program unit in which they appear. The general form of a statement function is:

$$f(a_1,a_2,\ldots,a_n) = e$$

where f is the function name, the $a_i$ 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 $a_i$ are distinct variable names or array elements, but being dummy variables 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 Chapter 5 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 predefined convention type must be defined by a type-specification statement.

5. The relationship between f and e must conform to the replacement rules in Chapter 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 paramenter in every argument list must agree in type with the ith dummy in the statement function.

The following example shows a statement function definition 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 designed into the FORTRAN system. Their names are predefined to the FORTRAN compiler 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 $a_i$ are actual arguments. The arguments must agree in type, number, and order with the specifications indicated in Tables 9-1 and 9-2.

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-largest integer <=¦a¦ | Real | Real |
| INT | | Real | Integer |
| IDINT | | Double | Integer |
| AMOD | a1 (mod a2) | Real | Real |
| MOD | | Integer | Integer |
| AMAX0 | Max(a1,a2,...) | Integer | Real |
| AMAX1 | | Real | Real |
| MAX0 | | Integer | Integer |
| MAX1 | | Real | Integer |
| DMAX1 | | Double | Double |
| AMIN0 | Min(a1,a2,...) | Integer | Real |
| AMIN1 | | Real | Real |
| MIN0 | | 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 * ¦a¦ | Real | Real |
| ISIGN | | Integer | Integer |
| DSIGN | | Double | Double |
| DIM | a1 - Min(a1,a2) | Real | Real |
| IDIM | | Integer | Integer |
| SNGL | | Double | Real |
| DBLE | | 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 |
| ALOG10 | 1 | log10(a) | Real | Real |
| DLOG10 | 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 |
| DMOD | 2 | a1(mod a2) | Double | Double |

## 9.5 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 $a_i$ are dummy arguments, of which there must be at least one, representing variable names, array names, or dummy names of SUBROUTINE or other FUNCTION subprograms.

## 9.6 Construction of FUNCTION Subprograms

Construction of FUNCTION subprograms must comply with the following restrictions:

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 an assignment 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.

Example:

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

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

4. If a dummy argument is an array name, then an array declarator must appear in the subprogram with dimensioning information consistent with that in the calling program.

5. A FUNCTION subprogram may contain any defined FORTRAN statements other than BLOCK DATA statements, SUBROUTINE statements, another FUNCTION statement or any statement which references either the FUNCTION being defined or another subprogram that references the FUNCTION being defined.

6. The logical termination of a FUNCTION subprogram is a RETURN statement--there must be at least one of these.

7. A FUNCTION subprogram must physically terminate with an END statement.

Example:

```
          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
```

## 9.7 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)
```

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

The arguments $a_i$ 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.

If an $a_i$ is a subprogram name, that name must have previously been distinguished from ordinary variables by appearing in an EXTERNAL statement and the corresponding dummy arguments in the called FUNCTION subprograms must be used in subpgrogram references.

If $a_i$ is a Hollerith or Literal constant, the corresponding dummy variable should encompass enough storage units to correspond exactly to the amount of storage needed by the constant.

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

Examples:

```
Z10 = FT1+Z7(D,T3,RHO)

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

## 9.8 SUBROUTINE Subprograms

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

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

        or

SUBROUTINE s
```

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

## 9.9 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. If a dummy argument is an array name then an array declarator must appear in the subprogram with dimensioning information consistent with that in the calling program.
5. If any of the dummy arguments represent values that are to be determined by the SUBROUTINE subprogram and returned to the calling program, these dummy arguments must appear within the subprogram on the left side of the equality sign in a replacement statement, in the input list of an input statement, or as a parameter within a subprogram reference.
6. A SUBROUTINE may contain any FORTRAN statements other than BLOCK DATA statements, FUNCTION statements, another SUBROUTINE statement, a PROGRAM statement or any statement which references the SUBROUTINE subprogram being defined or another subprogram which references the SUBROUTINE subprogram being defined.
7. A SUBROUTINE subprogram may contain any number of RETURN statements. It must contain at least one.

8. The RETURN statement(s) is the logical termination point of the subprogram.

9. The physical termination of a SUBROUTINE subprogram is an END statement.

10. If an actual argument transmitted to a SUBROUTINE subprogram by the calling program is the name of a SUBROUTINE or FUNCTION subprogram, the corresponding dummy argument must be used in the called SUBROUTINE subprogram as a subprogram reference.

Example:

```
C     SUBROUTINE TO COUNT POSITIVE ELEMENTS
C             IN AN ARRAY
          SUBROUTINE COUNT P(ARRY,I,CNT)
          DIMENSION ARRY(7)
          CNT = 0
          DO 9 J = 1,I
          IF (ARRY(J))9,5,5
      9   CONTINUE
          RETURN
      5   CNT = CNT + 1.0
          GO TO 9
          END
```

## 9.10 Referencing a SUBROUTINE Subprogram

A SUBROUTINE subprogram may be 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 a SUBROUTINE subprogram name and the $a_i$ are the actual arguments to be used by the subprogram. The $a_i$ 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 previously appeared in an EXTERNAL statement.

2. If the called SUBROUTINE subprogram contains a variable array declarator, then the CALL statement must contain the actual name of the array and the actual dimension specifications as arguments.

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)
```

## 9.11 RETURN from FUNCTION and SUBROUTINE Subprograms

The logial 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 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 next executable statement in the calling program which would logically follow the CALL statement.

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

5. Upon return from a SUBROUTINE subprogram the values assigned to the arguments 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, the new values for Z9 and B7 are made available to the calling program when the RETURN occurs.

## 9.12 Processing Arrays in Subprograms

If a calling program passes an array name to a subprogram, the subprogram must contain the dimension information pertinent to the array. A subprogram must contain array declarators if any of its dummy arguments represent arrays or array elements.

For example, a FUNCTION subprogram designed to compute the average of the elements of any one dimension array might be as follows:

```
              Calling Program Unit:

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

```
                 .
                 .

                 Called Prorgam Unit:

                 FUNCTION AVG(ARG,I)
                 DIMENSION ARG(50)
                 SUM = 0.0
                 DO 20 J = 1,I
       20        SUM = SUM + ARG(J)
                 AVG = SUB/FLOAT(I)
                 RETURN
                 END
```

Note that actual arrays to be processed by the FUNCTION subprogram are dimensioned in the calling program and the array names and their actual dimensions are transmitted to the FUNCTION subprogram by the FUNCTION subprogram reference. The FUNCTION subprogram itself contains a dummy array and specifies an array declarator.

Dimensioning information may also be passed to the subprogram in the parameter list. For example:

```
                 Calling Program Unit:

                 DIMENSION A(3,4,5)
                 .
                 .
                 .
                 CALL SUBR(A,3,4,5)
                 .
                 .
                 .
                 END


                 Called Program Unit:

                 SUBROUTINE SUBR(X,I,J,K)
                 DIMENSION X(I,J,K)
                 .
                 .
                 .
                 RETURN
                 END
```

It is valid to use variable dimensions only when the array name and all of the variable dimensions are dummy arguments. The variable dimensions must be type Integer. It is invalid to change the values of any of the variable dimensions within the called program.

## 9.13 BLOCK DATA Subprograms

A BLOCK DATA subprogram has as its only purpose the initialization of data in a COMMON block during loading of a FORTRAN object program. BLOCK DATA subprograms begin with a BLOCK DATA statement of the following form:

```
BLOCK DATA [subprogram-name]
```

and end with an END statement. Such subprograms may contain only Type, EQUIVALENCE, DATA, COMMON, and DIMENSION statements and are subject to the following considerations:

1. If any element in a COMMON block is to be initialized, all elements of the block must be listed in the COMMON statement even though they might not all be initialized.

2. Initialization of data in more than one COMMON block may be accomplished in one BLOCK DATA subprogram.

3. There may be more than one BLOCK DATA subprogram loaded at any given time.

4. Any particular COMMON block item should only be initialized by one program unit.

5. A BLOCK program must be INCluded at link time.

Example:

```
      BLOCK DATA INITIT
      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,
     12*0.52,1.1/,C/1.2E0,3*4.0/
      DATA A1/.TRUE./,E/-5.6/
```

# APPENDIX A. FORTRAN ERROR MESSAGES

The FORTRAN compiler detects two kinds of errors: Warnings and Fatal Errors. When a Warning issues, compilation continues with the next item on the source line. A Fatal Error, however, causes the compiler to ignore the rest of the logical line, including any continuation lines.

Warning messages are preceded by percent signs (%), while Fatal Errors are preceded by question marks (?). The physical line number follows the question mark and percent sign, and the error code or message follows the line number.

Example:

```
?Line 25: Mismatched Parentheses

%Line 16: Missing Integer Variable
```

When either type of error occurs, the program should be modified so that it will compile without errors.

## A.1 Fatal Compilation Errors

<u>Message</u>

Array Name Misuse
Backwards DO Reference
Consecutive Operators
Data Pool Overflow
Function Call with No Parameters
Identifier Too Long
Illegal Character for Syntax
Illegal Data Constant
Illegal DO Nesting
Illegal Hollerith Construction
Illegal Integer Quality
Illegal Item Following INTEGER or REAL or LOGICAL
Illegal Logical Form Operator
Illegal Mixed Mode Operation
Illegal Operator
Illegal Procedure Name
Illegal Statement Completion
Illegal Statement Following Logical IF
Illegal Statement Function Name
Illegal Statement Number
Improper Subscript Syntax
Incorrect Integer Constant
Incorrect Number of DATA Constants
Invalid DATA Constant or Repeat Factor
Invalid Data List Element in I/O
Invalid Logical Operator
Invalid Operand Usage
Invalid Statement Number
Literal String Too Large
Missing Integer Quantity
Missing Name
Not a Variable Name
Premature End of File on Input Device
Stack Overflow
Statement Out of Sequence
Statement Unrecognizable or Misspelled
Too Many Parentheses. 14 Allowed
Unbalanced DO Nest

## A.2 Compilation Error Warnings

Array Multiply EQUIVALENCEd within a Group
Array Name Expected
Block Name = Procedure Name
Code Output in BLOCK DATA
COMMON Base Lowered
COMMON Name Usage
Division by Zero
Duplicate Statement Label
Empty List for Unformatted WRITE
Format Nest Too Deep
Function with No Parameter
Hex Constant Overflow
Illegal Argument for ENCODE/DECODE
Illegal DO Termination
Invalid Statement Number Usage
Missing DO Termination
Missing Integer Variable
Missing Statement Number on FORMAT
Mixing of Operand Modes Not Allowed
Multiple EQUIVALENCE of COMMON
No Path to this Statement
Non-COMMON Variable in BLOCK DATA
Non-Integer Expression
Operand Mode Not Compatible with Operator
RETURN in a Main Program
Statement Number Not FORMAT Associated
STATUS Error on READ
Undefined Labels Have Occurred
Wrong Number of Subscripts
Zero Format Value
Zero Repeat Factor

## A.3 FORTRAN Runtime Error Messages

Runtime error messages carry a leading and trailing double asterisk, in this manner:

```
**FW**
```

### A.3.1 Warning Errors

After a warning error, execution continues; after 20 warnings, however, execution ceases.

| Code | Meaning |
|---|---|
| A2 | Both Arguments of ATAN2 are 0 |
| 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 for SIN Too Large |
| TL | Too Many Left Parentheses in FORMAT |

### A.3.2 Fatal Errors

Fatal errors cause execution to cease; control returns to the DOS.

| | |
|---|---|
| DT | Data Type Does Not Agree with FORMAT Specification |
| DZ | Division by ZERO, REAL, or INTEGER |
| EF | EOF Encountered on READ |
| FO | 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 Parenthesis in FORMAT |
| MP | Missing Period in FORMAT |
| SQ | Illegal Argument to SQRT Function (Negative) |

# APPENDIX B. 1500 FORTRAN

## B.1 1500 Operating Environment

In the 1500-series processor operating environment, FORTRAN requires a 1500 with at least 64K of memory operating under DOS.H.

## B.2 Installation

The FORTRAN release consists of FORTRAN/CMD and FORLIB/REL which reside on a single diskette. To install, boot the 1500 processor from a DOS.H diskette in drive 0, and insert the FORTRAN diskette in drive 1, 2, or 3.

## B.3 Compilation

A FORTRAN source program that is contained in an existing text file can be compiled to an output code file and later linked with LINK15 by a command to the operating system. The user keys in the command to DOS in the following format:

```
FORTRAN <sourcefile>[,<relfile>][,<printfile>][;<options>]
```

### B.3.1 Command Line Files

Up to three (3) files may be specified on the command line:

| | |
|---|---|
| <sourcefile> | Filename of the file containing the FORTRAN program. This file name is required; the default extension is /TXT. |
| <relfile> | Filename of the Relocatable object file that will be created by this compilation. This name is not required and its default value is <sourcefile>/REL. |
| <printfile> | Filename of the printer output file of the compilation. This name is not required; its default value is <sourcefile>/PRT. |

## B.3.2 Command Line Options

The command line options indicate disposition of the output file; they are indicated by the semicolon (;) following the file specifications. Options available are mutually exclusive: only one option letter may be given on the command line. The choices for output file disposition are designated by four different alpha characters: L, P, Q, and D. The action instituted by each of these characters is described below.

L - List on Local printer.

The L option causes a printer listing of the source program that includes internal references.

P - List on disk file.

The P option causes a listing file to be produced on the disk. If <printfile> is specified on the command line, this disk file will have the name of the source file with the extension of /PRT.

Q - Append to existing disk file.

The Q option appends the listing file to an existing listing file. There is no default filename for the Q option, so the <printfile> name must be specified on the command line when Q option is selected.

D - Display listing on terminal's video screen.

The D option causes the listing file to be displayed on the screen.

## B.4 Compilation Example

FORTRAN compiles and LINKs with a set of instructions in the following form.

```
FORTRAN DEBTEST
LINK15;N
SEGMENT DEBTEST/CMD
INCLUDE DEBTEST/REL
LIBRARY FORLIB/REL
LIBRARY DOSGUP/REL
*
```

# APPENDIX C.  5500 FORT55

## C.1  5500 Operating Environment

FORT55 executes on a processor using the 5500 instruction set. These are the 1800, 3800, 5500, and 6600 series of Datapoint computers. Execution is standalone on a processor with 48K of memory; with 56K of memory, FORT55 executes under ARC. The operating system required is:

| | | |
|---|---|---|
| DOS.D, version 2.6.1 | - | 3800, 1800, 5500, 6600 |
| DOS.E, version 2.6 | - | 5500, 6600 |
| DOS.G, version 2.6 | - | 1800 |

## C.2  Installation

FORT55 is released on 3 cassette tapes; it is installed using the MIN utility program (refer to the Disk Operating System DOS User's Guide, document number 50432). After the tapes are input, build the FORLIB/REL by keying in the chain command:

```
CHAIN F55RTL/TXT
```

## C.3  Compilation Procedure

A FORTRAN source program that is contained in an existing text file can be compiled to an output code file and later linked with LINK15 by a command to the operating system. The user keys in the command to DOS in the following format:

```
FORT55 <sourcefile>[,<relfile>][,<printfile>][;<options>]
```

### C.3.1  Command Line Files

Up to three (3) files may be specified on the command line:

| | |
|---|---|
| <sourcefile> | Filename of the file containing the FORTRAN program. This file name is required; the default extension is /TXT. |

| | |
|---|---|
| <relfile> | Filename of the Relocatable object file that will be created by this compilation. This name is not required and its default value is <sourcefile>/REL. |
| <printfile> | Filename of the printer output file of the compilation. This name is not required; its default value is <sourcefile>/PRT. |

## C.3.2 Command Line Options

The command line options indicate disposition of the utput file; they are indicated by the semicolon (;) llowing the file specifications. Options available are ually exclusive: only one option letter may be given on the command line. The choices for output file disposition are designated by four different alpha characters: L, P, Q, and D. The action instituted by each of these characters is described below.

L - List on Local printer.

The L option causes a printer listing of the source program that includes internal references.

P - List on disk file.

The P option causes a listing file to be produced on the disk. If <printfile> is specified on the command line, this disk file will have the name of the source file with the extension of /PRT.

Q - Append to existing disk file.

The Q option appends the listing file to an existing listing file. There is no default filename for the Q option, so the <printfile> name must be specified on the command line when Q option is selected.

D - Display listing on terminal's video screen.

The D option causes the listing file to be displayed on the screen.

## C.4 Compilation Example

FORT55 compiles and LINKs with a set of instructions in the following form. A starting load address of 017000 must be specified for FORT55.

Example:

```
FORT55  DEBTEST
LINK;N
SEGMENT DEBTEST/CMD,017000
INCLUDE DEBTEST/REL
LIBRARY FORLIB55/REL
LIBRARY DOSEPT/REL
*
```

NOTE: If the program does not use ISAM files, the following line may be included to save room in the object code:

```
INCLUDE FORLIB55.ISIGONE
```

This prevents the inclusion of generalized ISAM subroutines that would otherwise be brought in with the FORLIB55/REL library.

# APPENDIX D. DOS ARC ENQUEUE AND DEQUEUE

This discussion follows closely the material presented in the Attached Resource Computing System ARC User's Guide (document number 50299-2), Chapter 5, titled 'Updating of Shared Files.' Reviewing the detailed description of the updating of shared files presented in the ARC user's guide will aid the user in understanding the philosphy and implementation of enqueue and dequeue requests.

## D.1 Enqueuing Shared Files

In the ARC system, data files are shared among multiple processors and are accessed and updated concurrently by the several processors. There must be a method of coordinating simultaneous requests for file access and update. Determining which processor gets access to what file and whether or not an update should be made often depends on many other data items, as well as on a precise, instantaneous knowledge of the state of all files before a valid update is possible.

The portion of ARC that resolves this dilemma is called the Enqueue/Dequeue Subsystem. It is now possible to have several programs executing simultaneously, written variously in FORTRAN, DATASHARE, COBOL and assembler, that have the capability to access and update the same file. This ability to do simultaneous operations is not inherent in the FORTRAN, DATASHARE, COBOL, and assembler code programs; the enqueueing and dequeueing is effected by the convention of including specific program code to invoke the Enqueue/Dequeue Subsystem. The enqueue and dequeue instructions are coded into any program that is to update a file, or to read or write to a shared file.

Under ARC, enqueues may be implemented at several different levels. These levels correspond to the naturally nested functions occurring as a result of requests made by the applications programs. FORTRAN allows only level 3 enqueue requests. Level 3 is called the Multiaccess Transaction Level and corresponds to DATASHARE PI statements. Level 3 supports transactions which may include a number of individual accesses, some of which may result in DOS disk storage management functions being performed.

## D.2 Invoking Enqueue

To acquire the exclusive use of a resource (a file, or a volume of files), the applications program must contain two separate steps: the request for exclusive use, called enqueueing; and the release of the resource for its use by others, called dequeueing. The enqueue request is made before beginning to look at the data items, which must not change during the course of the update. Then, when the update is complete, dequeue releases the file, making it available for update requests from other processors and programs.

FORTRAN issues level 3 enqueue requests. (A complete description of enqueue levels is contained in the ARC user's guide.) An ISAM file may be asociated with many ISAM index (/ISI) files. However, to enqueue any ISAM file, only the related data file needs to be enqueued. This has the effect of enqueueing all related ISAM index files. The /ISI file(s) should not be specifically enqueued; FORTRAN performs this function internally.

The FORTRAN calling sequence is:

```
CALL ARCNQ (n, lun <,lun ,...,lun>)
```

where n is the number of logical units in the list and lun is the logical unit number to be enqueued.

The sequence CALL ARCDQ performs a level 3 dequeue. As an example of the enqueue/dequeue coding convention:

```
OPEN (4, 'SCRATCH ', 80)
CALL ARCNQ (1, 4)
READ (4, 15) T
CALL ARCDQ
```

## D.3 Enqueueing Errors

Errors that can occur are:

ARC NQ/DQ ERR:1 - Bad logical unit number (1..8)

ARC NQ/DQ ERR:2 - Wrong number of luns (1..8)

ARC NQ/DQ ERR:3 - ARC NQ failure (logical)

ARC NQ/DQ ERR:4 - ARC NQ failure (RIM interconnect)