ISC FORTRAN

REFERENCE MANUAL

AND

USER'S GUIDE

For

Intecolor and Compucolor

Units with FORTRAN

999275
02/25/80

Intelligent Systems Corp.
Intecolor Drive
225 Technology Park/Atlanta
Norcross, Georgia 30092
Telephone: 404/449-5961
TWX: 810/766-1581

See FORTRAN PLOT LIBRARY
P.21 V.5 #3 COLORCUE.

"SEMANTICALLY" VERY SIMILAR
TO BASIC.

FORTRAN USER PROBLEMS FOUND
P.3 DEC. 82 CUVIC

FORTRAN TUTORIAL —
P.7/8 DEC. 82 CUVIC
AUG/SEPT 82 COLORCUE — OCT/NOV CORRECTION
P.3 CUVIC JAN/FEB 83.

SEE CHIP #61
P.9/12 AUG/SEPT 82 COLORCUE

$350 WHEN FIRST
RELEASED

TABLE OF CONTENTS

	Pages
Notes on ISC FORTRAN	1 — 4
Microsoft FORTRAN-80 Reference Manual	5 — 108
Microsoft FORTRAN-80 User's Manual	3 — 13
LINK-80 Linking Loader Instructions for FORTRAN & MACRO ASSEMBLER	35 — 42

NOTES ON ISC FORTRAN

PROCEDURE FOR THE USE OF ISC FORTRAN:

1) Create a FORTRAN source program module and any other needed modules to be linked on disk using a text editor such as ISC's TEXT EDITOR or SCREEN EDITOR.

2) Compile the modules using F80:

```
FCS>RUN F80
F80>Module name
F80>Module name

- - - - -

F80>Module name
F80>(Control/C)
FCS>
```

Note that the default input file type for F80 is FOR and the output file type is REL.

3) Link the resulting compiled modules together with the library routines needed from FORLIB.REL using L80:

```
FCS>RUN L80
L80>module name. module name.. . .,module name
L80>FORLIB/S
```

At this point, the resulting module may be run:

```
L80>/G
```

or a program (PRG.) file may be generated to be run later:

```
L80>/N:Name for program
L80>/E
FCS>
```

"Locks up" and not saving .PRG use L80>/N:FOR01/E

Note that the default input file type for L80 is REL. and that multiple commands may appear on a single line. OUTPUT FILE is .PRG

EXAMPLE:

Suppose that the program **REGRES.PRG** is to be created on disk from the FORTRAN source program **REG.FOR** and its associated source module **INV.PRG** both of which reside on disk. Assuming at each step that the correct disk is in the drive, the procedure is as follows:

```
FCS>RUN F80
F80>REG
F80>INV
F80>(Control/C)
FCS>RUN L80
L80>REG,INV
L80>FORLIB/S
L80>/N:REGRES/E
FCS>
```

PROGRAMS TOO LARGE TO LINK:

Programs too large to link by the method above may yet be linked by use of the equates file (EQ.REL). At link time. the compiled module(s) of the program are linked to EQ.REL with a program address of AC9D (or beyond) so that the prelinked library may be in memory at runtime:

```
FCS>RUN L80
L80>/P:AC9D (for Compucolor and for Intecolor
             3621; for 8000/8300/8900-Series
             Intecolor use L80>/P:CB30)
L80>Module name, module name,...,module name
L80>EQ
L80>/N:File name/E
FCS>
```

To execute the resulting program. first run LIB.PRG which loads the prelinked library, then run the program:

```
FCS>RUN LIB
FCS>RUN File name
```

For the example above the procedure is

```
FCS>RUN L80
L80>/P:AC9D (for Compucolor and for Intecolor
             3621; for 8000/8300/8900-Series
             Intecolor use L80>/P:CB30)
L80>REGRES,INV
L80>/N:REGRES/E

FCS>RUN LIB
FCS>RUN REGRES
```

OTHER NOTES:

1) One or more of the following 'flags' may be used after a file name in a command line for F80:

   -L List source code on printer
   -T List source code on CRT
   -N Suppress object code
   /A List object code (mnemonics)

   Flags which may be used for L80 are described elsewhere in the reference manual.

2) The logical unit numbers for use in I/O statements are as follows:

   2 - printer
   3 - console
   6-10 - disk files

3) Graphics may be employed by defining LOGICAL (BYTE) variables, assigning the needed plot values to them and using unformatted WRITE statements listing these variables.

4) The form of the OPEN statement is

   CALL OPEN (File name, Logical unit number)

5) The unformatted READ statement is not incorporated.

MICROSOFT

FORTRAN-80

version 3.0

reference manual

MICROSOFT FORTRAN-80
Reference Manual

Contents

Section		Page
1	Introduction	6
2	Fortran Program Form	7
	2.1 Fortran Character Set	7
	2.1.1 Letters	7
	2.1.2 Digits	7
	2.1.3 Alphanumerics	8
	2.1.4 Special Characters	8
	2.2 FORTRAN Line Format	9
	2.3 Statements	13
3	Data Representation/Storage Format	14
	3.1 Data names and types	14
	3.1.1 Names	14
	3.1.2 Types	14
	3.2 Constants	15
	3.3 Variables	19
	3.4 Arrays and Array Elements	20
	3.5 Subscripts	20
	3.6 Data Storage Allocation	21
4	FORTRAN Expressions	25
	4.1 Arithmetic Expressions	25
	4.2 Expression Evaluation	26
	4.3 Logical Expressions	27
	4.3.1 Relational Expressions	28
	4.3.2 Logical Operators	28
	4.4 Hollerith, Literal, and Hexadecimal Constants in Expressions	31
5	Replacement Statements	32
6	Specification Statements	34
	6.1 Specification Statements	34
	6.2 Array Declarators	34
	6.3 Type Statements	35
	6.4 EXTERNAL Statements	37
	6.5 DIMENSION Statements	37

6.6 COMMON Statements 37
6.7 EQUIVALENCE Statements 39
6.8 DATA Initialization Statement 41

7 FORTRAN Control Statements 44

7.1 GOTO Statements 44
7.1.1 Unconditional GOTO 44
7.1.2 Computed GOTO 45
7.1.3 Assigned GOTO 45
7.2 ASSIGN Statement 46
7.3 IF Statement 47
7.3.1 Arithmetic IF 47
7.3.2 Logical IF 47
7.4 DO Statement 48
7.5 CONTINUE Statement 51
7.6 STOP Statement 52
7.7 PAUSE Statement 52
7.8 CALL Statement 53
7.9 RETURN Statement 53
7.10 END Statement 53

8 Input/Output 54

8.1 Formatted READ/WRITE 54
8.1.1 Formatted READ 54
8.1.2 Formatted WRITE 57
8.2 Unformatted READ/WRITE 58
8.3 Auxiliary I/O Statements 59
8.4 ENCODE/DECODE 59
8.5 Input/Output List Specifications 60
8.5.1 List Item Types 60
8.5.2 Special Notes on List Specifications 62
8.6 FORMAT Statements 63
8.6.1 Field Descriptors 63
8.6.2 Numeric Conversions 64
8.6.3 Hollerith Conversions 69

8.6.4 Logical Conversion 71
8.6.5 X Descriptor 72
8.6.6 P Descriptor 72
8.6.7 Special Control Features of FORMAT Statements 73
8.6.7.1 Repeat Specifications 73
8.6.7.2 Field Separators 75
8.6.8 FORMAT Control, List Specifications, and Record Demarcation 76
8.6.9 FORMAT Carriage Control 77
8.6.10 FORMAT Specifications in Arrays . . . 77

9 Functions and Subprograms 80

9.1 PROGRAM Statement 81
9.2 Statement Functions 81
9.3 Library Functions 82
9.4 Function Subprograms 86
9.5 Construction of Function Subprograms 86
9.6 Referencing a Function Subprogram 88
9.7 Subroutine Subprograms 89
9.8 Construction of Subroutine Subprograms 89
9.9 Referencing a Subroutine Subprogram 90
9.10 Return From Function and Subroutine Subprograms 91
9.11 Processing Arrays in Subprograms 92
9.12 BLOCK DATA Subroutine 93

APPENDIX A- Language Extensions and Restrictions . . 96

APPENDIX B- I/O Interface 98

APPENDIX C- Subprogram Linkages 100

APPENDIX D- ASCII Character Codes 102

APPENDIX E- Disk File Access 103

APPENDIX F- FORTRAN-80 Library Subroutines 106

SECTION 1

INTRODUCTION

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

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

This manual defines the FORTRAN source language for the 8080 and Z-80 microcomputers. This language includes the American National Standard FORTRAN language as described in ANSI document X3.9-1966, approved on March 7, 1966, plus a number of language extensions and some restrictions. These language extensions and restrictions are described in the text of this document and are listed in Appendix A.

NOTE

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

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

Section 2 describes the form and components of an 8080 FORTRAN source program. Sections 3 and 4 define data types and their expressional relationships. Sections 5 through 9 describe the proper construction and usage of the various statement classes.

SECTION 2

FORTRAN PROGRAM FORM

8080 FORTRAN source programs consist of one program unit called the Main program and any number of program units called subprograms. A discussion of subprogram types and methods of writing and using them is in Section 9 of this manual.

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

2.1 FORTRAN CHARACTER SET

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

2.1.1 LETTERS

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

NOTE

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

2.1.2 DIGITS

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

NOTE

Strings of digits representing numeric quantities are normally interpreted as decimal numbers. However, in certain statements, the interpretation is in the

Hexadecimal number system in which case the letters A, B, C, D, E, F may also be used as Hexadecimal digits. Hexadecimal usage is defined in the descriptions of statements in which such notation is allowed.

2.1.3 ALPHANUMERICS

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

2.1.4 SPECIAL CHARACTERS

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

NOTES:

1. FORTRAN program lines consist of 80 character positions or columns, numbered 1 through 80. They are divided into four fields.

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

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

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

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

2.2 FORTRAN LINE FORMAT

The sample FORTRAN coding form (Figure 2.1) shows the format of FORTRAN program lines. The lines of the form consist of 80 character positions or columns, numbered 1 through 80, and are divided into four fields.

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

2. Continuation character field- Column 6

3. Statement field- Columns 7 through 72

4. Indentification field- Columns 73 through 80

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

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

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

    1. Column 1 contains the letter C.

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

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

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

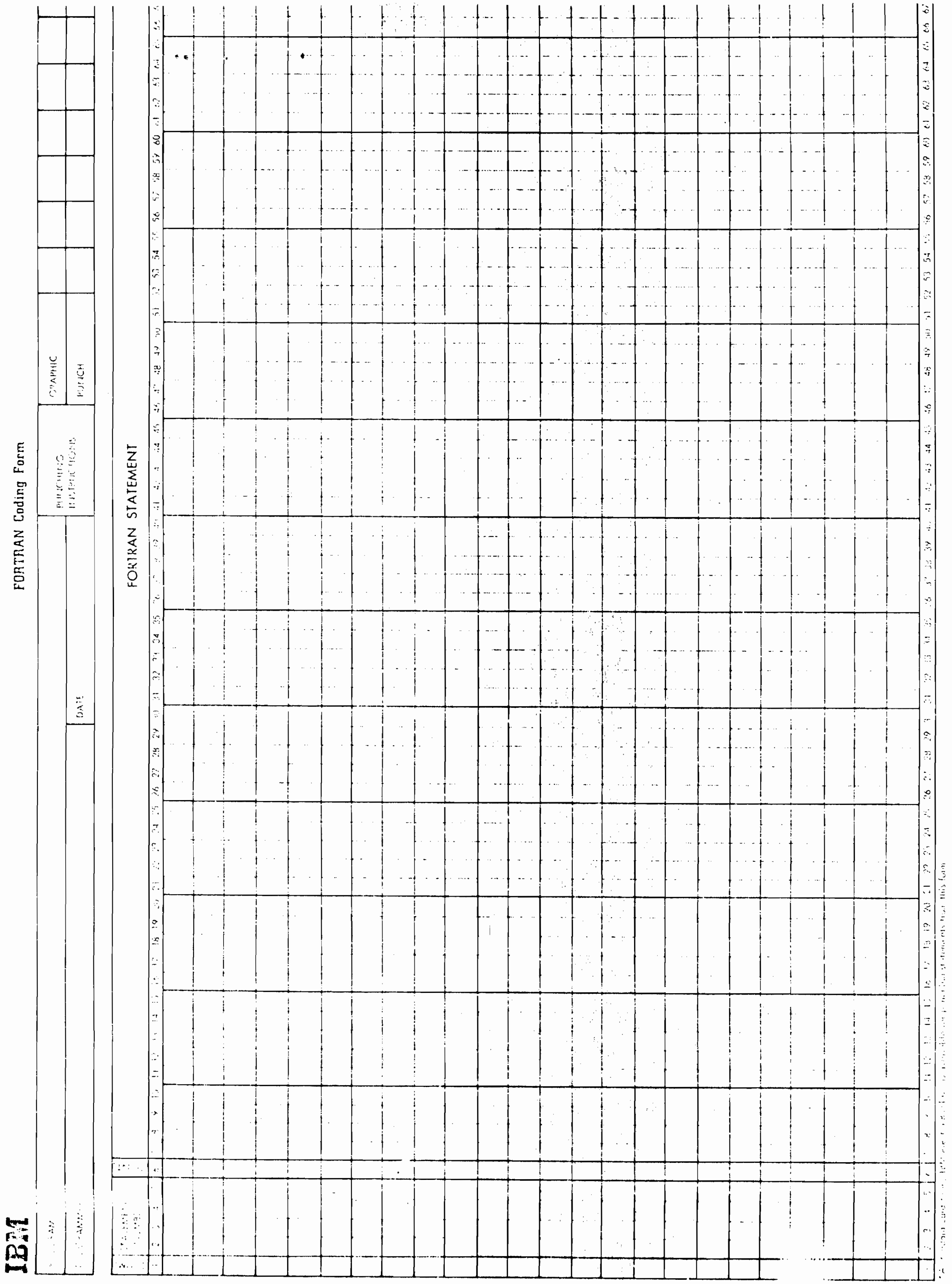

Figure 2.1 FORTRAN Coding Form

Example:

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

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

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

   2. Column 6 must contain a zero or blank.

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

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

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

   Example:

```
      END
```

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

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

   2. Column 6 must contain a zero or blank.

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

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

   Example:

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

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

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

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

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

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

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

   <u>Example:</u>

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

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

The following considerations govern the use of statement labels:

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

2. The numeric value of the label, leading zeros and blanks are not significant.

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

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

Example:

```
C  EXAMPLES OF STATEMENT LABELS
C
   1
   1 01
99999
  763
```

2.3 STATEMENTS

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

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

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

Non-executable statements describe to the processor the nature and arrangement of data and provide information about input/output formats and data initialization to the object 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 usage and construction of the various types of statements are described in Sections 5 through 9.

SECTION 3

DATA REPRESENTATION / STORAGE FORMAT

The FORTRAN Language prescribes a definitive method for identifying data used in FORTRAN programs by name and type.

3.1 DATA NAMES AND TYPES

3.1.1 NAMES

1. Constant - An explicitly stated datum.

2. Variable - A symbolically identified datum.

3. Array - An ordered set of data in 1, 2 or 3 dimensions.

4. Array Element - One member of the set of data of an array.

3.1.2 TYPES

1. Integer -- Precise representation of integral numbers (positive, negative or zero) having precision to 5 digits in the range -32768 to +32767 inclusive (-2**15 to 2**15-1).

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

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

4. Logical -- 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 -1, however any non-zero value will be treated as .TRUE. in a Logical IF statement. In addition, Logical types may be used as one byte signed integers in the

range -128 to +127, inclusive.

5. Hollerith -- A string of any number of characters from the computer's character set. All characters including blanks are significant. Hollerith data require one byte for storage of each character in the string.

3.2 CONSTANTS

FORTRAN constants are identified 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. CONSTANT FORMATS

TYPE	FORMATS AND RULES OF USE	EXAMPLES
INTEGER	1. 1 to 5 decimal digits interpreted as a decimal number.	-763 1 +00672
	2. A preceding plus (+) or minus (-) sign is optional.	-32768 +32767
	3. No decimal point (.) or comma (,) is allowed.	
	4. Value range: -32768 through +32767 (.i.e., -2**15 through 2**15-1).	

REAL	1. A decimal number with precision to 7 digits and represented in one of the following forms:
	a. + or -.f + or -i.f
	b. + or -i.E+ or -e
	    + or -.fE+ or -e
	    + or -i.fE+ or -e
	
	where i, f, and e are each strings representing integer, fraction, and exponent respectively.
		345.
		-.345678
		+345.678
		+.3E3
		-73E4

	2. Plus (+) and minus (-) characters are optional.	
	3. In the form shown in 1 b above, if r represents any of the forms preceding E+ or -e (i.e., rE+ or -e), the value of the constant is interpreted as r times 10**e, where -38<=e<=38.	
	4. If the constant preceding E+ or -e contains more significant digits than	

	the precision for real data allows, truncation occurs, and only the most significant digits in the range will be represented.	
DOUBLE PRECISION	A decimal number with precision to 16 digits. All 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 single precision unless it contains a "D" exponent.	+345.678 +.3D3 -73D4
LOGICAL	.TRUE. generates a non-zero byte (hexadecimal FF) and .FALSE. generates a byte in which all bits are 0. 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. .FALSE.
LITERAL	In the literal form, any number of characters may be enclosed by single quotation marks. The form is as follows: 'X1X2X3...Xn' where each Xi is any character other than '. Two quotation marks in succession may be used to represent the quotation mark character within the string, i.e., if X2 is to be the quotation mark character, the string appears as the following: 'X1''X3...Xn'	
HEXADECIMAL	1. The letter Z or X followed by a single quote, up to 4 hexadecimal	Z'12' X'AB1F'

digits (0-9 and A-F) and a single quote is recognized as a hexadecimal value.	Z'FFFF' X'1F'
2. A hexadecimal constant is right justified in its storage value.	

3.3 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 runtime 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

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 (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 be typed 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 specific statements used in explicitly typing data are described in Section 6.

Variable data receive their numeric value assignments during program execution or, initially, in a DATA statement (Section 6).

Hollerith or Literal data may be assigned to any type variable. Sub-paragraph 3.6 contains a discussion of Hollerith data storage.

3.4 ARRAYS AND ARRAY ELEMENTS

An array is an ordered set of data characterized by the property of dimension. An array may have 1, 2 or 3 dimensions and is identified and typed by a symbolic name in the same manner as a variable except that an array name must be so declared by an "array declarator." Complete discussions of the array declarators appear in Section 6 of this manual. An array declarator also 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.

3.5 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 that govern the use of subscripts are as follows:

1. A subscript contains 1, 2 or 3 subscript expressions (see 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 (Section 6).

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

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

   where C and K are integer constants and V is an integer variable name (see Section 4 for a discussion of expression evaluation).

5. Subscripts themselves may not be subscripted. Examples:

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

3.6 DATA STORAGE ALLOCATION

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

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

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

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

Up to eight Hollerith characters may be associated with Double Precision type storage, up to four with Real, up to two with Integer and one with Logical type storage.

TABLE 3-2. STORAGE ALLOCATION BY DATA TYPES

TYPE	ALLOCATION
INTEGER	2 bytes/ 1/2 storage unit S Binary Value Negative numbers are the 2's complement of positive representations.
LOGICAL	1 byte/ 1/4 storage unit Zero (false) or non-zero (true) A non-zero valued byte indicates true (the logical constant .TRUE. is represented by the hexadecimal value FF). A zero valued byte indicates false. When used as an arithmetic value, a Logical datum is treated as an Integer in the range -128 to +127.
REAL	4 bytes/ 1 storage unit Characteristic S Mantissa Mantissa (continued) The first byte is the characteristic expressed in excess 200 (octal) notation; i.e., a value of 200 (octal) corresponds to a binary exponent of 0. Values less than 200 (octal) correspond to negative exponents, and values greater than 200 correspond to positive exponents. By definition, if the characteristic is zero, the entire number is zero. The next three bytes constitute the mantissa. The mantissa is always normalized such that the high order bit is one, eliminating the need to actually save that bit. The high bit is used instead to indicate the sign of the number. A one indicates a negative number, and zero 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 with that of Real data except Double Precision uses 4 extra bytes for the matissa.

SECTION 4

FORTRAN EXPRESSIONS

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 operands, operators and rules of use for both types are described in the following paragraphs.

4.1 ARITHMETIC EXPRESSIONS

The following rules define all permissible arithmetic expression forms:

1. A constant, variable name, array element reference or FUNCTION reference (Section 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 of the expression are not all of the same type, the type of the expression is determined by the element having the highest type. The type hierarchy (highest to lowest) is as follows: DOUBLE PRECISION, REAL, INTEGER, LOGICAL.

6. Expressions may contain nested parenthesized elements as in the following:

```
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, care should be taken to see that the number of left parentheses and the number of right parentheses are equal.

4.2 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, then the next innermost until the entire expression has been evaluated.

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

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

Examples:

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

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

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

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

4.3 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 (see FUNCTION, Section 9).

2. Two arithmetic expressions separated by a relational operator (i.e., 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.

4.3.1 RELATIONAL EXPRESSIONS

The general form of a relational expression is as follows:

```
e1 r e2
```

where e1 and e2 are arithmetic expressions and r is a relational operator. The six relational operators are as follows:

.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))
```

4.3.2 LOGICAL OPERATORS

Table 4-1 lists the logical operations. U and V denote logical expressions.

Table 4-1. Logical Operations

.NOT.U	The value of this expression is the logical complement of U (i.e., 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 (i.e., 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 (i.e., 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 (i.e., there is a one in the result 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 <u>except</u> when the second operator is .NOT.

That is,

.AND..NOT.

and

.OR..NOT.

are permitted.

Example:

A.AND..NOT.B is permitted

A.AND..OR.B is not permitted

4.4 HOLLERITH, LITERAL, AND HEXADECIMAL CONSTANTS IN EXPRESSIONS

Hollerith, Literal, and Hexadecimal 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 to this are:

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

2. Hollerith, Literal, or Hexadecimal 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.

SECTION 5

REPLACEMENT STATEMENTS

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

v = e

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

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

The following conditions apply to replacement statements:

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

   Example:

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

   The line containing v= must be the 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, then 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 of variable. Y indicates a valid replacement and N indicates an invalid replacement. Footnotes to Y indicate conversion considerations.

Table 5-1. Replacement By Type

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

SECTION 6

SPECIFICATION STATEMENTS

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

6.1 SPECIFICATION STATEMENTS

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

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

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

6.2 ARRAY DECLARATORS

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

Type statements
DIMENSION statements
COMMON statements

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

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 ui is the name of the array, called the declarator name, and the k's are integer constants.

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

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

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

6.3 TYPE STATEMENTS

Variable, array and FUNCTION names are automatically typed 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 pre-defined convention by specifying the type of an item. In addition, these statements may be used to declare arrays.

Type statements have the following general form:

```
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 readability 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.
2. IETA and KLPH 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 pre-defined convention is overridden by their appearance in the INTEGER statement. JMP(15) is a constant array declarator. It redundantly types the array elements as Integer and communicates to the processor the storage requirements and dimensionality of the array.

Example:

```
LOGICAL L1, TEMP
```

NOTE

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

6.4 EXTERNAL STATEMENTS

EXTERNAL statements have the following form:

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

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

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)
```

6.5 DIMENSION STATEMENTS

A DIMENSION statement has the following form:

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

where each ui 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.

6.6 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 the following form:

```
COMMON /Y1/A1/Y2/A2/.../Yn/An
```

where each Yi is a COMMON block storage name and each Ai is a sequence of variable names, array names or constant array declarators, separated by commas. The elements in Ai make up the COMMON block storage area specified by the name Yi. If any Yi is omitted leaving two consecutive slash characters (//), the block of storage so indicated is called blank COMMON. If the first block name (Y1) 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 and thus ZOT (3,3) is an array declarator. ZOT must not have been previously declared. (See "Array Declarators," Paragraph 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 the use of EQUIVALENCE statements. See "EQUIVALENCE Statements," Paragraph 6.7.

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-80 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 the use of EQUIVALENCE statements.

6.7 EQUIVALENCE STATEMENTS

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

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

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

Example:

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

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

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

Example:

If the dimensionaliity 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.

Additonal Considerations:

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

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

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

   Example:

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

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

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

   Example:

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

   The resulting COMMON block will have the following configuration:

Variable	Storage Unit
W = L(1,1)	0
X = R(2,1)	1
Y = R(1,2)	2
R(2,2)	3

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

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

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

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

Example:

```
      DIMENSION XTABLE (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(15).
2. It attempts to EQUIVALENCE two elements of the same COMMON block, A(7) and B(3).
3. Since A is not an array, A(7) is an illegal reference.
4. Making B(3) equivalent to D(5) extends COMMON backwards from its defined starting point.

6.8 DATA INITIALIZATION STATEMENT

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

The statement is of the following form:

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

where "list" represents a list of variable, array or array element names, and the ui are constants corresponding in number to the elements in the list. An exception to the one-for-one correspondence of list items to constants is that an array name (unsubscripted) may appear in the list, and as many constants as necessary to fill the array may appear in the corresponding position between slashes. Instead of ui, it is permissible to write k*ui in order to declare the same constant, ui, k times in succession. 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 ui 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.

Hexadecimal data are stored in a similar fashion. If fewer Hexadecimal characters are used, sufficient leading zeros are added to fill the remainder of the storage unit.

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

```
      DIMENSION HARY (2)
      DATA HARY,B/  4HTHIS, 4H OK.
     1          ,7.86/
```

```
      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'/
```

SECTION 7

FORTRAN CONTROL STATEMENTS

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

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

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

7.1 GO TO STATEMENTS

7.1.1 UNCONDITIONAL GO TO

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

The statement is of the following form:

```
GO TO k
```

where k 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 ahead of statement 310 in the logical flow of the program of which they are a part.

7.1.2 COMPUTED GO TO

Computed GO TO statements are of the form:

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

where the ki are statement labels, and j is an integer variable, $1 \leq j \leq n$.

This statement causes transfer of control to the statement labeled kj. If $j \leq 1$ or $j > n$, control will be passed to the next statement following the Computed GOTO.

Example:

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

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

7.1.3 ASSIGNED GO TO

Assigned GO TO statements are of the following form:

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

  or

GOTO J
```

where J is an integer variable name, and the ki 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.

7.2 ASSIGN STATEMENT

This statement is of the following form:

```
ASSIGN j TO i
```

where 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)
```

7.3 IF STATEMENT

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

Arithmetic IF
Logical IF

7.3.1 ARITHMETIC IF

The arithmetic IF statement is of the form:

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

where e is an arithmetic expression and m1, m2 and m3 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

7.3.2 LOGICAL IF

The Logical IF statement is of the form:

```
IF (u)s
```

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

Control Conditions:

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

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

Examples:

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

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

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

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

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

7.4 DO STATEMENT

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

1) DO k i = m1,m2,m3

or

2) DO k i = m1,m2

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 1, it may be omitted as in 2) above.

The following conditions and restrictions govern the 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 or PAUSE.

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. For example, the loop overhead for a DO loop with a constant limit and an increment of 1 depends upon the type of the index variable as follows:

Index Variable Type	Overhead Microseconds	Bytes
INTEGER*2	35.5	19
INTEGER*1	24	14

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 less than or equal to m2, and then the DO is said to be satisfied. The statements in the DO range will always be executed at least once, even if m1 $\leq$ m2.

   When the DO has been satisfied, control passes to the statement following the terminal statement, otherwise control transfers back to the first executable statement following the DO statement.

Example:

The following example computes

```
100
Sigma  Ai  where a is a one-dimensional array
i=1
```

```
 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 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:

```
          15
     Ck =Sigma    AkjBm, k = 1,2,...,15
         j=1
```

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

7.5 CONTINUE STATEMENT

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

```
     CONTINUE
```

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

Example:

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

7.6 STOP STATEMENT

A STOP statement has one of the following forms:

STOP

or

STOP c

where c is any string of one to six characters.

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

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

7.7 PAUSE STATEMENT

A PAUSE statement has one of the following forms:

PAUSE

or

PAUSE c

where c is any string of up to six characters.

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

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

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

7.8 CALL STATEMENT

CALL statements control transfers into SUBROUTINE subprograms and provide parameters for use by the subprograms. The general forms and detailed discussion of CALL statements appear in Section 9, FUNCTIONS AND SUBPROGRAMS.

7.9 RETURN STATEMENT

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

7.10 END STATEMENT

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

```
END
```

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

SECTION 8

INPUT / OUTPUT

FORTRAN provides a series of statements which define the control and conditions of data transmission between computer memory and external data handling or mass storage devices such as magnetic tape, disk, line printer, punched card processors, keyboard printers, etc.

These statements are grouped as follows:

1. Formatted READ and WRITE statements which cause formatted information to be transmitted between the computer and I/O devices.

2. Unformatted READ and WRITE statements which transmit 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 transmission to provide data conversion and editing information between internal data representation and external character string forms.

8.1 FORMATTED READ/WRITE STATEMENTS

8.1.1 FORMATTED READ STATEMENTS

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

Two forms of the statement are available, as follows:

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

or

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

where:

u - specifies a Physical and Logical Unit Number and may be either an unsigned integer or an integer variable in the range 1 through 255. If an Integer variable is used, an Integer value must be assigned to it prior to execution of the READ statement.

Units 1, 3, 4, and 5 are preassigned to the console Teletypewriter. Unit 2 is preassigned to the Line Printer (if one exists). Units 6-10 are preassigned to Disk Files (see Appendix E). These units, as well as units 11 - 255, may be re-assigned by the user (see Appendix B).

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 the execution time. (See 8.6.10)

L1- is the FORTRAN label on the statement to which the I/O processor will transfer control if an I/O error is encountered.

L2- is the FORTRAN label on the statement to which the I/O processor will transfer control if an End-of-File is encountered.

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

READ (u,f)k is used to input a number of items, corresponding to the names in 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, 8.6). The ERR= and END= clauses are optional. If not specified, I/O errors and End-of-Files cause fatal runtime errors.

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

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

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 8.5.

Examples:

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

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

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

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

   See 8.6 for 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 (see 8.5). All elements of the array ARRY will be 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 (see Hollerith Conversions, 8.6.3).

For example, the statements

```
     READ(I,25)
     .
     .
     .
  25 FORMAT(10HABCDEFGHIJ)
```

cause the next 10 characters of the file on input device I to be read and replace the characters ABCDEFGHIJ in the FORMAT statement.

8.1.2 FORMATTED WRITE STATEMENTS

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

Two forms of the statement are available, as follows:

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

or

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

where:

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.

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, 8.6). 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 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)
```

8.2 UNFORMATTED READ/WRITE

Unformatted I/O (i.e. without data conversion) is accomplished using the statements:

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

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

where:

u - specifies a Logical Unit Number.

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.

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. The total length of the list of variable names in an unformatted READ must not be 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(a)k statement writes one logical record.

5. A logical record may extend across more than one physical record.

8.3 AUXILIARY I/O STATEMENTS

Three auxiliary I/O statements are provided:

```
BACKSPACE u
REWIND u
ENDFILE u
```

Initially, the actions of all three statements are defined as no-ops. They may, however, be redefined (see Appendices B and E).

8.4 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
```

where;

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.

8.5 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 mi are list items separated by commas, as shown.

8.5.1 LIST ITEM TYPES

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

1. 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)
   ```

NOTE

The entry (I,J) defines two items in a list while (26,1) is a subscript.

2. 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 or i = m1,m2

   and enclosed in parentheses.

   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)

   Thus, 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 is:

```
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)
```

8.5.2 SPECIAL NOTES ON LIST SPECIFICATIONS

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

2. Arrays are transmitted by the appearance of the array name (unsubscripted) 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:

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

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

3. 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
```

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

2. In a formatted transmission (READ(u,f)k, WRITE(u,f)k) each item must have the correct type as specified by a FORMAT statement.

8.6 FORMAT STATEMENTS

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

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

The general form of a FORMAT statement is as follows:

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

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

8.6.1 FIELD DESCRIPTORS

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

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

where:

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

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

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

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

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

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

8.6.2 NUMERIC CONVERSIONS

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

F-type 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.

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	bb362.4200
F7.1	-4786.361	-4786.4
F8.4	8.7E-2	bb0.0375
F6.4	4739.76	*.7600
F7.3	-5.6	b-5.600

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

F-Input

(See the description under E-Input below.)

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.

E-Output

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

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

$$w > d + 7$$

Otherwise significant characters may be lost. Some E-Output examples follow:

FORMAT Descriptor	Internal Value	Output (b=blank)
E12.5	76.573	b0.76573Eb02
E14.7	-32672.354	-0.3267235Eb05
E7.3	56.93	* 0.569E
E13.4	-0.0012321	bb-0.1232E-02
E8.2	76321.73	0.76Eb05

E-Input

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

1. Leading spaces (ignored)

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

3. A string of digits

4. A decimal point

5. A second string of digits

6. The character E

7. A + or - sign

8. A decimal exponent

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

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

2. If FORMAT item 8 is present, then 6 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 Section 3 for Real data.

E- and F- and G- Input Examples:

FORMAT Descriptor	Input (b=blank)	Internal Value
E10.3	+0.23756+4	+2375.60
E10.3	bbbbb17631	+17.631
G8.3	b1628911	+1628.911
F12.4	bbbb-6321132	-632.1131

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

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

D-Type Conversions

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

G-Type Conversions

Form: Gw.d

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

G-Input:

(See the description under E-Input)

G-Output:

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

Magnitude	Equivalent Conversion
.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

I-Conversions

Form: Iw

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

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
I3	1234	*34

I-Input:

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

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

Examples:

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

8.6.3 HOLLERITH CONVERSIONS

A-Type Conversion

The form of the A conversion is as follows:

Aw

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

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

A-Output:

If w is greater than 4n (where n is the number of storage units required by the list item), the external output field will consist of w-4n blanks followed by the 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	Type	Output (b=blanks)
A1	A1	Integer	A
A2	AB	Integer	AB
A3	ABCD	Real	ABC
A4	ABCD	Real	ABCD
A7	ABCD	Real	bbbABCD

A-Input:

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

Examples:

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

H-Conversion

The forms of H conversion are as follows:

nHh1h2...hn

'h1h2...hn'

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

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.

H-Output:

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

Examples:

Format Descriptor			Output (b=blanks)
1HA	or	'A'	A
8HbSTRINGb	or	'bSTRINGb'	bSTRINGb
11HX(2,3)=12.0	or	'X(2,3)=12.0'	X(2,3)=12.0
12HIbSHOULDN'T	or	'IbSHOULDN''T'	IbSHOULDN'T

H-Input

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

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

8.6.4 LOGICAL CONVERSIONS

The form of the logical conversion is as follows:

Lw

L-Output:

If the value of an item in an output list corresponding to this descriptor is 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

L-Input

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

8.6.5 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:

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

Input Examples:

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

8.6.6 P DESCRIPTOR

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

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

Effects of Scale Factor on Input:

During E, F, or G input the scale factor takes effect only if no exponent is present in the external representation. In that case, the internal value will be a factor of 10**n less than the external value (the number will be divided by 10**n before being stored).

Effect of Scale Factor on Output:

E-Output, D-Output:

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

F-Output:

The external value will be 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.

8.6.7 SPECIAL CONTROL FEATURES OF FORMAT STATEMENTS

8.6.7.1 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, 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   F6.1,2X)
```

3. Repetition of FORMAT descriptors is also initiated when all descriptors in the FORMAT statement have been used but there are still items in the input/output list that have not been processed. When this occurs the FORMAT descriptors are re-used starting at the opening parenthesis that matches the last closing parenthesis in the FORMAT statement. The parentheses enclosing the entire list of descriptors are not considered unless there are no other parentheses in the list. A repeat count preceding the parenthesized descriptor(s) to be re-used is also active in the re-use. This type of repetitive use of FORMAT descriptors terminates processing of the current record and initiates the processing of a new record each time the re-use begins. Record demarcation under these circumstances is the same as in the paragraph 8.7.6.2 below.

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,KK,LL,MM,K3,LE,
     1   M3
      .
      .
      .
   12 FORMAT (2F9.4,(3I7))
```

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

8.6.7.2 Field Separators

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

Example:

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

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

Each slash terminates a record and sets up the next record for processing. The remainder of an input record is ignored; the remainder of an output record is filled with 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 this example, the data specified by the list of the WRITE statement are output to unit 7 according to the specifications of FORMAT statement 8. Four records are written as follows:

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

Input Example:

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

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

8.6.8 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, 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, G, I, L or A.

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

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

9. If FORMAT control reaches the last right parenthesis of the FORMAT statement but there are more list items to be processed, all or part of the descriptors are reused. (See item 3 in the description of Repeat Specifications, sub-paragraph 8.7.6.1)

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.

8.6.9 FORMAT CARRIAGE CONTROL

The first character of every formatted output record is used to convey carriage control information to the output device, 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

8.6.10 FORMAT SPECIFICATIONS IN ARRAYS

The FORMAT reference, f, of a formatted READ or WRITE statement (See 8.1) 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 (i.e., it begins with a left parenthesis and ends with a right parenthesis).

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

Assume the FORMAT specification

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

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

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

```
C  DECLARE A REAL ARRAY
       DIMENSION A(3), B(3), M(4)
C  INITIALIZE FORMAT WITH DATA STATEMENT
       DATA A/'(3F1','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
       .
       .
       .
```

SECTION 9

FUNCTIONS AND SUBPROGRAMS

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

These procedures are as follows:

1. Statement functions.

2. Library functions.

3. FUNCTION subprograms.

4. SUBROUTINE subprograms.

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

1. Each of these procedures is referenced by its name which, in all cases, is one to six alphanumeric characters of which the first is a letter.

2. The first three are designated as "functions" and are alike in that:

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

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

    3. 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 pre-defined convention.

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

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

9.1 THE PROGRAM STATEMENT

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

```
PROGRAM name
```

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

9.2 STATEMENT FUNCTIONS

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

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

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

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

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

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

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

previously defined statement functions.

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

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

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

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

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

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


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

9.3 LIBRARY FUNCTIONS

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

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

f(a1,a2,...an)

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

In addition to the functions listed in 9-1 and 9-2, four additional library subprograms are provided to enable direct access to the 8080 (or Z80) hardware. These are:

PEEK, POKE, INP, OUT

PEEK and INP are Logical functions; POKE and OUT are subroutines. PEEK and POKE allow direct access to any memory location. PEEK(a) returns the contents of the memory location specified by a. CALL POKE(a1,a2) causes the contents of the memory location specified by a1 to be replaced by the contents of a2. INP and OUT allow direct access to the I/O ports. INP(a) does an input from port a and returns the 8-bit value input. CALL OUT(a1,a2) outputs the value of a2 to the port specified by a1.

Examples:

```
      A1 = B+FLOAT (I7)

      MAGNI = ABS(KBAR)

      PDIF = DIM(C,D)

      S3 = SIN(T12)

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

TABLE 9-1

Intrinsic Functions

Function Name	Definition	Types Argument	Types Function
ABS	|a|	Real	Real
IABS		Integer	Integer
DABS		Double	Double
AINT	Sign of a times lar-	Real	Real
INT	gest integer <= |a|	Real	Integer
IDINT		Double	Integer
AMOD	a1(mod a2)	Real	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 times |a1|	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	Argument Type	Function Type
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

9.4 FUNCTION SUBPROGRAMS

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

A FUNCTION statement has one of the following forms:

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

or

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

where:

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

2. f is the name of the FUNCTION subprogram.

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

9.5 CONSTRUCTION OF FUNCTION SUBPROGRAMS

Construction of FUNCTION subprograms must comply with the following 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 consistant 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 and there must be at least one of them.

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
      DO8 M = 1,J
 8    SUM = SUM + BARY(K,M)
      RETURN
      END
```

9.6 REFERENCING A FUNCTION SUBPROGRAM

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

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

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

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

1. A variable name.

2. An array element name.

3. An array name.

4. An expression.

5. A SUBROUTINE or FUNCTION subprogram name.

6. A Hollerith or Literal constant.

If an ai 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 subprogram references.

If ai 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.

The following examples show references to FUNCTION subprograms.

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

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

9.7 SUBROUTINE SUBPROGRAMS

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

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

or

SUBROUTINE s

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

9.8 CONSTRUCTION OF SUBROUTINE SUBPROGRAMS

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

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

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

4. If a dummy argument is an array name then an array declarator must appear in the subprogram with dimensioning information consistant 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 have 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
```

9.9 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 ai are the actual arguments to be used by the subprogram. The ai must agree in type, order and number with the corresponding dummy arguments in the subprogram-defining SUBROUTINE statement.

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

1. FUNCTION and SUBROUTINE names appearing in the argument list must have 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)
```

9.10 RETURN FROM FUNCTION AND SUBROUTINE SUBPROGRAMS

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

RETURN

The following rules govern the use of the RETURN statement:

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

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

3. RETURN from a SUBROUTINE subprogram is to the 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 available 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.

<u>Example:</u>

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, Z9 and B7 are made available to the calling program when the RETURN occurs.

9.11 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 the folowing:

```
       Calling Program Unit

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

       Called Program Unit

       FUNCTION AVG(ARG,I)
       DIMENSION ARG(50)
       SUM = 0.0
       DO 20 J=1,I
20     SUM = SUM + ARG(J)
       AVG = SUM/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 paramater 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.

9.12 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.

Example:

```
      BLOCK DATA
      LOGICAL A1
      COMMON/BETA/B(3,3)/GAM/C(4)
      COMMON/ALPHA/A1,C,E,D
      DATA B/1.1,2.5,3.8,3*4.96,
     12*0.52,1.1/,C/1.2E0,3*4.0/
      DATA A1/.TRUE/,E/-5.6/
```

APPENDIX A

Language Extensions and Restrictions

The FORTRAN-80 language includes the following extensions to ANSI Standard FORTRAN (X3.9-1966).

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

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

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

4. Statement functions may use subscripted variables.

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

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

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

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

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

FORTRAN-80 places the following restrictions upon Standard FORTRAN.

1. The COMPLEX data type is not implemented. It may be included in a future release.

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

    1. PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA

    2. Type, EXTERNAL, DIMENSION

    3. COMMON

    4. EQUIVALENCE

5. DATA

6. Statement Functions

3. A different amount of computer memory is allocated for each of the data types: Integer, Real, Double Precision, Logical.

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

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

APPENDIX B

I/O Interface

Input/Output operations are table-dispatched to the driver routine for the proper Logical Unit Number. $LUNTB is the dispatch table. It contains one 2-byte driver address for each possible LUN. It also has a one-byte entry at the beginning, which contains the maximum LUN plus one. The initial run-time package provides for 10 LUN's (1 - 10), all of which correspond to the TTY. Any of these may be redefined by the user, or more added, simply by changing the appropriate entries in $LUNTB and adding more drivers. The runtime system uses LUN 3 for errors and other user communication. Therefore, LUN 3 should correspond to the operator console. The initial structure of $LUNTB is shown in the listings following this appendix.

The device drivers also contain local dispatch tables. Note that $LUNTB contains one address for each device, yet there are really seven possible operations per device:

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

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

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

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

2. For a WRITE, the number of bytes to write is in location $BL.

3. For a READ, the number of bytes read should be returned in $BL.

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

   a) CY=1, Z=don't care - I/O error
   b) CY=0, Z=0 - end-of-file encountered
   c) CY=0, Z=1 - normal return

   The runtime system checks the condition codes after calling the driver. If they indicate a non-normal condition, control is passed to the label specified by "ERR=" or "END=" or, if no label is specified, a fatal error results.

5. $IOERR is a global routine which prints an "ILLEGAL I/O OPERATION" message (non-fatal). This routine may be used if there are some operations not allowed on a particular device (i.e. Binary I/O on a TTY).

NOTE

> The I/O buffer has a fixed maximum length of 132 bytes unless it is changed at installation time. If a driver allows an input operation to write past the end of the buffer, essential runtime variables may be affected. The consequences are unpredictable.

The listings following this appendix contain an example driver for a TTY. REWIND, BACKSPACE, and ENDFILE are implemented as No-Ops and Binary I/O as an error. This is the TTY driver provided with the runtime package.

```
                        ;          TTY I/O DRIVER

0000                               EXT      $IOERR,$BL,$BF,$ERR,$TTYIN,$TTYOT
0012                    IRECER     EQU      Ø22       ;INPUT RECORD TOO LONG
0000                               ENTRY    $DRV3
0000   0013 '           $DRV3:     DW       DRV3FR    ;FORMATTED READ
0002   0042 '                      DW       DRV3FW    ;FORMATTED WRITE
0004   0010 '                      DW       DRV3BR    ;BINARY READ
0006   0010 '                      DW       DRV3BW    ;BINARY WRITE
0008   000E '                      DW       DRV3RE    ;REWIND
000A   000E '                      DW       DRV3BA    ;BACKSPACE
000C   000E '                      DW       DRV3EN    ;ENDFILE
000E   AF               DRV3EN:    XRA      A         ;THESE OPERATIONS ARE
                                                      ;NO-OPS FOR TTY
000E                    DRV3RE     EQU      DRV3EN
000E                    DRV3BA     EQU      DRV3EN
000F   C9                          RET
0010   C3 0000 *        DRV3BW:    JMP      $IOERR    ;ILLEGAL OPERATIONS
                                                      ;(PRINT ERROR AND RETURN)
0010                    DRV3BR     EQU      DRV3BW
0013   AF               DRV3FR:    XRA      A         ;READ
0014   32 0000 *                   STA      $BL       ;ZERO BUFFER LENGTH
0017   CD 0000 *        DRV31:     CALL     $TTYIN    ;INPUT A CHAR
001A   E6 7F                       ANI      Ø177      ;AND OFF PARITY
001C   FE 0A                       CPI      1Ø        ;IGNORE LINE FEEDS
001E   CA 0017 '                   JZ       DRV31
0021   F5                          PUSH     PSW       ;SAVE IT
0022   2A 0015 *                   LHLD     $BL       ;GET CHAR POSIT IN BUFFER
0025   26 00                       MVI      H,Ø       ;ONLY 1 BYTE
0027   EB                          XCHG
0028   2A 0000 *                   LHLD     $BF       ;GET BUFFER ADDR
002B   19                          DAD      D         ;ADD OFFSET
002C   F1                          POP      PSW       ;GET CHAR
002D   77                          MOV      M,A       ;PUT IT IN BUFFER
002E   13                          INX      D         ;INCREMENT $BL
002F   EB                          XCHG
0030   22 0023 *                   SHLD     $BL       ;SAVE IT
0033   FE 0D                       CPI      Ø15       ;CR?
0035   C8                          RZ                 ;YES--DONE
0036   7D                          MOV      A,L       ;$BL
0037   FE 80                       CPI      128       ;MAX IS DECIMAL 128
0039   DA 0017 '                   JC       DRV31     ;GET NEXT CHAR
003C   CD 0000 *                   CALL     $ERR
003F   12                          DB       IRECER    ;INPUT RECORD TOO LONG
0040   AF                          XRA      A         ;CLEAR FLAGS
0041   C9                          RET
0042   3A 0031 *        DRV3FW:    LDA      $BL       ;BUFFER LENGTH
0045   B7                          ORA      A
```

```
0046  C8          04900           RZ            ;EMPTY BUFFER
0047  2A 0029 *   05000           LHLD  $BF     ;BUFFER ADDRESS
004A  3D          05100           DCR   A       ;DECREMENT LENGTH
004B  F5          05200           PUSH  PSW     ;SAVE IT
004C  3E 0D       05300           MVI   A,13    ;CR
004E  CD 0000 *   05400           CALL  $TTYOT  ;OUTPUT IT
0051  7E          05500           MOV   A,M     ;GET FIRST CHAR IN BUFFER
0052  FE 2B       05600           CPI   '+'
0054  CA 0079 '   05700           JZ    DR3FW2  ;NO LINE FEEDS
0057  FE 31       05800           CPI   '1'
0059  C2 0064 '   05900           JNZ   DR3FW1  ;NOT FORM FEED
005C  3E 0C       06000           MVI   A,12    ;FORM FEED
005E  CD 004F *   06100           CALL  $TTYOT  ;OUTPUT IT
0061  C3 0079 '   06200           JMP   DR3FW2
0064  3E 0A       06300   DR3FW1: MVI   A,10    ;LF
0066  CD 005F *   06400           CALL  $TTYOT
0069  7E          06500           MOV   A,M     ;GET CHAR BACK
006A  FE 20       06600           CPI   ' '
006C  CA 0079 '   06700           JZ    DR3FW2  ;NO MORE LINE FEEDS
006F  FE 30       06800           CPI   '0'
0071  C2 0079 '   06900           JNZ   DR3FW2  ;NO MORE LINE FEEDS
0074  3E 0A       07000           MVI   A,10    ;LF
0076  CD 0067 *   07100           CALL  $TTYOT
0079  F1          07200   DR3FW2: POP   PSW     ;GET LENGTH BACK
007A  23          07300           INX   H       ;INCREMENT PTR
007B  C8          07400   DRV32:  RZ
007C  F5          07500           PUSH  PSW     ;SAVE CHAR COUNT
007D  7E          07600           MOV   A,M     ;GET NEXT CHAR
007E  23          07700           INX   H       ;INCREMENT PTR
007F  CD 0077 *   07800           CALL  $TTYOT  ;OUTPUT CHAR
0082  F1          07900           POP   PSW     ;GET COUNT
0083  3D          08000           DCR   A       ;DECREMENT IT
0084  C3 007B '   08100           JMP   DRV32   ;ONE MORE TIME
0087              08200           END
```



```
$IOERR  0011*   $BL     0043*   $BF     0048*   $ERR    003D*
$TTYIN  0018*   $TTYOT  0080*   IRECER  0012    $DRV3   0000'
DRV3FR  0013'   DRV3FW  0042'   DRV3BR  0010'   DRV3BW  0010'
DRV3RE  000E'   DRV3BA  000E'   DRV3EN  000E'   DRV31   0017'
DR3FW2  0079'   DR3FW1  0064'   DRV32   007B'
```

```
                     00100   ;COMMENT *
                     00200   ;        DRIVER ADDRESSES FOR LUN'S 1 THROUGH 10
                     00210   ;
0001                 00220   LPT      EQU     1                   ;UNIT 2 IS LPT
0001                 00230   DSK      EQU     1                   ;UNITS 6-10 ARE DSK
0000                 00235   DTC      EQU     0                   ;DTC COMMUNICATIONS UNIT 4
                     00240   ;
                     00300
0000                 00400            ENTRY   $LUNTB
0000                 00500            EXT     $DRV3
0000    0B           00600   $LUNTB:  DB      013                 ;MAX LUN + 1
0001    0000 *        00700           DW      $DRV3      ;THEY ALL POINT TO $DRV3 FOR NOW
0003                 00800            IFF     LPT
                     00900            DW      $DRV3
0003                 01000            ENDIF
0003                 01100            IFT     LPT
0003                 01200            EXT     LPTDRV
0003    0000 *       01300            DW      LPTDRV
0005                 01400            ENDIF
0005    0001 *       01500            DW      $DRV3
0007                 01510            IFF     DTC
0007    0005 *       01600            DW      $DRV3
0009                 01602            ENDIF
0009                 01604            IFT     DTC
                     01605            EXT     $CMDRV
                     01606            DW      $CMDRV
0009                 01608            ENDIF
0009    0007 *       01700            DW      $DRV3
000B                 01800            IFF     DSK
                     01900            DW      $DRV3
                     02000            DW      $DRV3
                     02100            DW      $DRV3
                     02200            DW      $DRV3
                     02300            DW      $DRV3
000B                 02400            ENDIF
000B                 02500            IFT     DSK
000B                 02600            EXT     DSKDRV
000B    0000 *       02700            DW      DSKDRV
000D    000B *       02800            DW      DSKDRV
000F    000D *       02900            DW      DSKDRV
0011    000F *       03000            DW      DSKDRV
0013    0011 *       03100            DW      DSKDRV
0015                 03200            ENDIF
0015                 03300            END
```



```
LPT     0001     DSK      0001     DTC      0000     $LUNTB   0000'
$DRV3   0009*    LPTDRV   0003*    DSKDRV   0013*
```

APPENDIX C

Subprogram Linkages

This appendix defines a normal subprogram call as generated by the FORTRAN compiler. It is included to facilitate linkages between FORTRAN programs and those written in other languages, such as 8080 Assembly.

A subprogram reference with no parameters generates a simple "CALL" instruction. The corresponding subprogram should return via a simple "RET." (CALL and RET are 8080 opcodes - see the assembly manual or 8080 reference manual for explanations.)

A subprogram reference with parameters results in a somewhat more complex calling sequence. Parameters are always passed by reference (i.e., the thing passed is actually the address of the low byte of the actual argument). Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing the parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).

2. If the number of parameters is greater than 3, they are passed as follows:

   1. Parameter 1 in HL.

   2. Parameter 2 in DE.

   3. Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that, with this scheme, the subprogram must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. <u>Neither the compiler nor the runtime system checks for the correct number of parameters.</u>

If the subprogram expects more than 3 parameters, and needs to transfer them to a local data area, there is a system subroutine which will perform this transfer. This argument transfer routine is named $AT, and is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (i.e., the total number of arguments minus 2). The subprogram is responsible for saving the first two parameters before calling $AT. For example, if a subprogram expects 5 parameters, it should look like:

```
SUBR:   SHLD    P1      ;SAVE PARAMETER 1
        XCHG
        SHLD    P2      ;SAVE PARAMETER 2
        MVI     A,3     ;NO. OF PARAMETERS LEFT
        LXI     H,P3    ;POINTER TO LOCAL AREA
        CALL    $AT     ;TRANSFER THE OTHER 3 PARAMETERS
        .
        .
        .
        [Body of subprogram]
        .
        .
        .
        RET             ;RETURN TO CALLER
P1:     DS      2       ;SPACE FOR PARAMETER 1
P2:     DS      2       ;SPACE FOR PARAMETER 2
P3:     DS      6       ;SPACE FOR PARAMETERS 3-5
```

When accessing parameters in a subprogram, don't forget that they are pointers to the actual arguments passed.

NOTE

It is entirely up to the programmer to see to it that the arguments in the calling program match in number, type, and length with the parameters expected by the subprogram. This applies to FORTRAN subprograms, as well as those written in assembly language.

FORTRAN Functions (Section 9) return their values in registers or memory depending upon the type. Logical results are returned in (A), Integers in (HL), Reals in memory at $AC, Double Precision in memory at $DAC. $AC and $DAC are the addresses of the low bytes of the mantissas.

APPENDIX D

ASCII CHARACTER CODES

DECIMAL	CHAR.	DECIMAL	CHAR.	DECIMAL	CHAR.
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093	]
008	BS	051	3	094	^ (or ↑)
009	HT	052	4	095	< (or ←)
010	LF	053	5	096	'
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

LF=Line Feed FF=Form Feed CR=Carriage Return DEL=Rubout

APPENDIX E

DISK FILE ACCESS

FORTRAN-80 provides sequential disk file access via FORTRAN programs. The CP/M and ISIS-II versions of FORTRAN-80 also provide random disk accessing, i.e., a record may be specified with a disk READ or WRITE.

Logical Unit Numbers 6-10 are preassigned to disk files. A READ or WRITE to an LUN automatically OPENs the file for input or output respectively, if it is not already open. The file remains open until closed by an ENDFILE command or until normal program termination. A file that is OPENed by a READ or WRITE statement has a default name that depends upon the LUN and the operating system:

FCS: FORT06.DAT, FORT07.DAT,..., FORT10.DAT

In each case the LUN is incorporated into the default file name.

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 (6,20,REC=I,ERR=50) X, Y, Z
.
.
.
```

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

The record length of any file accessed randomly is assumed to be 128 bytes (1 sector). Therefore, it is recommended that any file you wish to read randomly be created via FORTRAN (or Microsoft BASIC) random access statements.

Random access files may be created via FORTRAN programs, by using either binary or formatted WRITE statements. If the WRITE statement does not cause enough data to be transferred to fill the record (128 bytes), then the end of the record is filled with zeros (NULL characters).

Alternatively, a file may be OPENed using the OPEN subroutine. LUNs 1-5 may also be assigned to disk files with OPEN. The OPEN subrutine allows the program to specify a filename and device to be associated with a LUN, whereas the default specifies a default name and uses the currently selected disk drive.

An OPEN of a non-existent file creates a null file of the appropriate name. An OPEN of an existing file followed by an output deletes the existing file. An OPEN of an existing file followed by an input allows access to the current contents of the file.

The form of an OPEN call under FCS is:

```
CALL OPEN (LUN, Filename)
```

where:

LUN = a Logical Unit Number to be associated with the file (must be an Integer constant or Integer variable with a value between 1 and 10).

Filename = an ASCII name which the operating system will associate with the file. The Filename should be a Hollerith or Literal constant, or a variable or array name where the variable or array contains the ASCII name. The Filename should be in the form normally required by FCS, i.e., an optional device name , followed by a name of up to 6 characters, a period, an extension of up to 3 characters, and a space (or other non-alphanumeric character). The Filename must be terminated by a non-alphanumeric character.

The following are examples of valid OPEN calls under FCS:

```
CALL OPEN (6,'MD1:FOO.DAT ')

CALL OPEN (1,'DF0:TEST.DAT ')

CALL OPEN (10, 'FILE1.RND ')
```

The ENDFILE and REWIND commands allow further program control of disk files. The form of the commands is:

```
ENDFILE L or REWIND L
```

where L is an LUN. ENDFILE L closes the file associated with LUN L. REWIND L closes the file associated with LUN L, then opens it again.

NOTE

Under FCS do not open more than 1 disk file for output on any device.

APPENDIX F

FORTRAN-80 Library Subroutines

The FORTRAN-80 library contains a number of subroutines that may be referenced by the user from FORTRAN or assembly programs. In the following descriptions, $AC refers to the floating accumulator; $AC is the address of the low byte of the mantissa. $AC+3 is the address of the exponent. $DAC refers to the DOUBLE PRECISION accumulator; $DAC is the address of the low byte of the mantissa. $DAC+7 is the address of the DOUBLE PRECISION exponent.

All arithmetic routines (addition, subtraction, multiplication, division, exponentiation) adhere to the following calling conventions.

1. Argument 1 is passed in the registers:
   Integer in [HL]
   Real in $AC
   Double in $DAC

2. Argument 2 is passed either in registers, or in memory depending upon the type:

   a. Integers are passed in [HL], or [DE] if [HL] contains Argument 1.

   b. Real and Double Precision values are passed in memory pointed to by [HL]. ([HL] points to the low byte of the mantissa.)

The following arithmetic routines are contained in the Library:

Function	Name	Argument 1 Type	Argument 2 Type
Addition	$AA	Real	Integer
	$AB	Real	Real
	$AQ	Double	Integer
	$AR	Double	Real
	$AU	Double	Double
Division	$D9	Integer	Integer
	$DA	Real	Integer
	$DB	Real	Real
	$DQ	Double	Integer
	$DR	Double	Real
	$DU	Double	Double
Exponentiation	$E9	Integer	Integer
	$EA	Real	Integer
	$EB	Real	Real
	$EQ	Double	Integer
	$ER	Double	Real
	$EU	Double	Double
Multiplication	$M9	Integer	Integer
	$MA	Real	Integer
	$MB	Real	Real
	$MQ	Double	Integer
	$MR	Double	Real
	$MU	Double	Double
Subtraction	$SA	Real	Integer
	$SB	Real	Real
	$SQ	Double	Integer
	$SR	Double	Real
	$SU	Double	Double

Additional Library routines are provided for converting between value types. Arguments are always passed to and returned by these conversion routines in the appropriate registers:

Logical in [A]

Integer in [HL]

Real in $AC

Double in $DAC

Name	Function
$CA	Integer to Real
$CC	Integer to Double
$CH	Real to Integer
$CJ	Real to Logical
$CK	Real to Double
$CX	Double to Integer
$CY	Double to Real
$CZ	Double to Logical

INDEX

Arithmetic Expression 25-26, 47
Arithmetic IF 44, 47, 49
Arithmetic Operators 8
Array 14, 20, 34-35, 37-38, 40-41, 56, 77, 87-88, 92-93
Array Declarator 20
Array Element 14, 20, 27, 32, 39
ASCII Character Codes 102
ASSIGN 44, 46
Assigned GOTO 44-45

BACKSPACE 59
BLOCK DATA 34, 37, 90, 94

CALL 44, 53, 90
Character Set 7
Characteristic 23
Comment Line 9
COMMON 34, 37, 39-41, 87, 89, 94
Computed GOTO 44-45
Constant 14-15
Continuation 9, 12
CONTINUE 44, 51
Control Statements 44

DATA 34, 41, 87, 89, 94
Data Representation 14
Data Storage 21
DECODE 59
DIMENSION 20, 34, 37, 94
Disk Files 103
DO 44, 47-49
DO Implied List 61
Double precision 14
Dummy 89-91, 93

ENCODE 59
END 53, 87, 90, 94
END Line 11
ENDFILE 59, 103
EQUIVALENCE 34, 39-41, 87, 89, 94
Executable 13, 34, 44
Expression 25-26, 31-32
Extended Range 50
EXTERNAL 34, 37, 88, 91
External Functions 85

Field Descriptors 63
FORMAT 55-57, 63, 67, 69-73, 75-78
Formatted READ 54

Formatted WRITE 57
FUNCTION 34, 37, 80, 86-93

GOTO 44, 49

Hexadecimal 8, 21, 31, 42
Hollerith 9, 15, 20-21, 31, 42, 56, 69-70, 88

I/O 54, 98
I/O List 60
IF 44, 47
Index 49
Initial Line 11
INP 83
Integer 14, 19, 23
Intrinsic Functions 84

Label 9, 12, 44-45, 48
Library Function 80, 82
Library Subroutines 106
Line Format 9
List Item 60
Literal 9, 20-21, 31, 42, 70, 88
Logical 14, 19, 23, 71
Logical Expression 27, 30, 48
Logical IF 44, 47, 49
Logical Operator 28
Logical Unit Number 54, 58, 98, 103
LUN 54, 58, 98, 103

Mantissa 23

Nested 51
Non-executable 13, 34
Numeric Conversions 64

OPEN 103
Operand 25
Operator 25
OUT 83

PAUSE 44, 49, 52
PEEK 83
POKE 83
PROGRAM 34, 81, 90

Random Disk access 103
Range 49

READ 56, 58, 63, 72, 76-78
Real 14, 19, 23
Relational Expression 27-28
Relational Operator 28
Replacement Statement 32, 48
RETURN 44, 49, 53, 87, 90-92
REWIND 59
Rewind 105

Scale Factor 72-73
Specification Statement . . . 34
Statement Function 34, 80-81
STOP 44, 49, 52
Storage 35
Storage Format 14
Storage Unit 21, 23, 39
Subprogram 37, 53, 80, 86-94, 100
SUBROUTINE 34, 37, 53, 80, 87-92
Subscript 20, 27
Subscript Expression 21, 27

Type 94
Type Statement 35

Unconditional GOTO 44
Unformatted I/O 58

Variable 14, 19, 32, 38, 88

WRITE 57-58, 63, 72, 76-78

MICROSOFT

FORTRAN-80

user's manual

Microsoft FORTRAN-80 User's Manual

CONTENTS

SECTION 1 Compiling FORTRAN Programs 5

1.1 FORTRAN-80 Command Scanner 5
1.1.1 Format of Commands 5
1.1.2 FORTRAN-80 Compilation Switches . . 6
1.2 Sample Compilation 9
1.3 FORTRAN-80 Compiler Error Messages 11

SECTION 2 FORTRAN Runtime Error Messages 13

SECTION 1

Compiling FORTRAN Programs

1.1 FORTRAN-80 Command Scanner

To tell the FORTRAN compiler what to compile and with which options, it is necessary to input a "command string," which is read by the FORTRAN-80 command scanner.

1.1.1 Format of Commands

To run FORTAN-80, type RUN F80 in response to the FCS> prompt. FORTRAN-80 will return the prompt F80> indicating that it is ready to accept commands. The general format of a FORTRAN-80 command string is:

sfile1,...,sfileN TO ofile

sfile1,...,sfileN
The FCS filenames of the FORTRAN-80 source files to be compiled. The filenames may specify device and version. The default type is .FOR .

ofile
The FCS filename of the FORTRAN-80 object file. The default type is .REL . If no object file is specified, then the default object filename is the same as sfile1 with .REL for the type.

Listing options are specified by appending a switch to the command string. The following listing options are available:

-T	Listing displayed on the console
-L	Listing printed on the printer
-LS	Listing saved in sfile1.LST

In all cases the error messages are displayed on the console. If both an object file and listing file are requested, the listing file is directed to the other disk unit with the same device type, i.e., if the .REL file is sent to FD1:, the .LST file is sent to FD0:.

If no object file is desired, do not specify the 'TO ofile' in the command. Instead, add a -N switch to the command to suppress the creation of the object file.

Examples:

F80>TEST	Compile TEST.FOR and place the object in TEST.REL.
F80>TEST-T-N	Compile TEST.FOR and list the program on the console. No object is generated.
F80>TEST-L	Compile TEST.FOR giving TEST.REL and list program on the printer.
F80>TEST-N	Compile TEST.FOR without producing an object file or listing.

1.1.2 FORTRAN-80 Compilation Switches

A number of different switches may be given in the command string that will affect the format of the listing file. Each switch should be preceded by a slash (/):

Switch	Action
O	Print all listing addresses, etc. in octal.
A	List generated object code.
S	Each /S allocates an extra 200 bytes of stack space for use during compilation. Use /S if stack overflow errors occur during compilation; otherwise, not needed.
M	Specifies to the compiler that the generated code should be in a form which can be loaded into ROMs. When a /M is specified, the generated code will differ from normal in the following ways: 1. FORMATs will be placed in the program area, with a "JMP" around them. 2. Parameter blocks (for calls with more than 3 parameters) will be initialized at runtime, rather than being initialized by the loader.

Examples:

F80>MYPROG-T Compile MYPROG.FOR giving MYPROG.REL and list the program on the console.

F80>TEST-L/A Compile TEST.FOR giving TEST.REL and list the program with the generated code on the printer.

F80>BIGONE/S Allocate 200 bytes of extra stack space for compilation of BIGONE.FOR into BIGONE.REL .

NOTE

If a FORTRAN program is intended for ROM, the programmer should be aware of the following ramifications:

1. DATA statements should not be used to initialize RAM. Such initialization is done by the loader, and will therefore not be present at execution. Variables and arrays may be initialized during execution via assignment statements, or by READing into them.

2. FORMATs should not be read into during execution.

3. If the standard library I/O routines are used, DISK files should not be OPENed on any LUNs other than 6, 7, 8, 9, 10. If other LUNs are needed for Disk I/O, $LUNTB should be recompiled with the appropriate addresses pointing to the Disk driver routine.

A library routine, $INIT, sets the stack pointer at the top of available memory (as indicated by the operating system) before execution begins.

The calling convention is:

```
LXI     B,<return address>
JMP     $INIT
```

If the generated code is intended for some other machine, this routine should probably be rewritten. The source of the standard initialize routine is provided on the disk as "INIT.MAC". Only the portion of this routine which sets up the stack pointer should ever be modified by the user. The FORTRAN library already contains the standard initialize routine.

1.2 Sample Compilation

```
A>F80

*EXAMPL,TTY:=EXAMPL

FORTRAN-80 Ver. 3.0 Copyright 1978 (C) By Microsoft - Bytes: 4524
00100              PROGRAM EXAMPLE
00200              INTEGER X
00300              I = 2**8 + 2**9 + 2**10
00400              DO 1 J=1,5
*****    0000'     LXI      H,0700
*****    0003'     SHLD     I
00500    C         CIRCULAR SHIFT I LEFT 3 BITS -- RESULT IN X
00600              CALL CSL3(I,X)
*****    0006'     LXI      H,0001
*****    0009'     SHLD     J
00700              WRITE(3,10) I,X
*****    000C'     LXI      D,X
*****    000F'     LXI      H,I
*****    0012'     CALL     CSL3
*****    0015'     LXI      D,10L
*****    0018'     LXI      H,[      03       00]
*****    001B'     CALL     $W2
00800    1         I=X
*****    001E'     LXI      B,X
*****    0021'     LXI      D,I
*****    0024'     LXI      H,[      01       00]
*****    0027'     MVI      A,03
*****    0029'     CALL     $IO
*****    002C'     CALL     $ND
00900    10        FORMAT(2I15)
*****    002F'     LHLD     X
*****    0032'     SHLD     I
*****    0035'     LHLD     J
*****    0038'     INX      H
*****    0039'     MVI      A,05
*****    003B'     SUB      L
*****    003C'     MVI      A,00
*****    003E'     SBB      H
*****    003F'     JP       0009'
01000              END
*****    0042'     CALL     $EX
*****    0045'     0100
*****    0047'     0300

Program Unit Length=0049 (73) Bytes
Data Area Length=000D (13) Bytes

Subroutines Referenced:

$IO                     CSL3                    $W2
$ND                     $EX
```

```
Variables:

X       0001"          I       0003"          J       0005"

LABELS:

1L      002F'          10L     0007"

*^C
A>
```

See Section 1.6 of the Microsoft Utility Software Manual for a listing of the MACRO-80 subroutine CSL3.

1.3 FORTRAN Compiler Error Messages

The FORTRAN-80 Compiler detects two kinds of errors: Warnings and Fatal Errors. When a Warning is issued, compilation continues with the next item on the source line. When a Fatal Error is found, the compiler ignores the rest of the logical line, including any continuation lines. Warning messages are preceded by percent signs (%), and Fatal Errors by question marks (?). The editor line number, if any, or the physical line number is printed next. It is followed by the error code or error message.

Example:

?Line 25: Mismatched Parentheses

%Line 16: Missing Integer Variable

When either type of error occurs, the program should be changed so that it compiles without errors. No guarantee is made that a program that compiles with errors will execute sensibly.

Fatal Errors:

Error Number	Message
100	Illegal Statement Number
101	Statement Unrecognizable or Misspelled
102	Illegal Statement Completion
103	Illegal DO Nesting
104	Illegal Data Constant
105	Missing Name
106	Illegal Procedure Name
107	Invalid DATA Constant or Repeat Factor
108	Incorrect Number of DATA Constants
109	Incorrect Integer Constant
110	Invalid Statement Number
111	Not a Variable Name
112	Illegal Logical Form Operator
113	Data Pool Overflow
114	Literal String Too Large
115	Invalid Data List Element in I/O
116	Unbalanced DO Nest
117	Identifier Too Long
118	Illegal Operator
119	Mismatched Parenthesis
120	Consecutive Operators
121	Improper Subscript Syntax
122	Illegal Integer Quantity
123	Illegal Hollerith Construction
124	Backwards DO reference
125	Illegal Statement Function Name

126	Illegal Character for Syntax
127	Statement Out of Sequence
128	Missing Integer Quantity
129	Invalid Logical Operator
130	Illegal Item Following INTEGER or REAL or LOGICAL
131	Premature End Of File on Input Device
132	Illegal Mixed Mode Operation
133	Function Call with No Parameters
134	Stack Overflow
135	Illegal Statement Following Logical IF

Warnings:

0	Duplicate Statement Label
1	Illegal DO Termination
2	Block Name = Procedure Name
3	Array Name Misuse
4	COMMON Name Usage
5	Wrong Number of Subscripts
6	Array Multiply EQUIVALENCEd within a Group
7	Multiple EQUIVALENCE of COMMON
8	COMMON Base Lowered
9	Non-COMMON Variable in BLOCK DATA
10	Empty List for Unformatted WRITE
11	Non-Integer Expression
12	Operand Mode Not Compatible with Operator
13	Mixing of Operand Modes Not Allowed
14	Missing Integer Variable
15	Missing Statement Number on FORMAT
16	Zero Repeat Factor
17	Zero Format Value
18	Format Nest Too Deep
19	Statement Number Not FORMAT Associated
20	Invalid Statement Number Usage
21	No Path to this Statement
22	Missing Do Termination
23	Code Output in BLOCK DATA
24	Undefined Labels Have Occurred
25	RETURN in a Main Program
26	STATUS Error on READ
27	Invalid Operand Usage
28	Function with no Parameter
29	Hex Constant Overflow
30	Division by Zero
32	Array Name Expected
33	Illegal Argument to ENCODE/DECODE

SECTION 2

FORTRAN Runtime Error Messages

Code	Meaning
Warning Errors:	
IB	Input Buffer Limit Exceeded
TL	Too Many Left Parentheses in FORMAT
OB	Output Buffer Limit Exceeded
DE	Decimal Exponent Overflow (Number in input stream had an exponent larger than 99)
IS	Integer Size Too Large
BE	Binary Exponent Overflow
IN	Input Record Too Long
OV	Arithmetic Overflow
CN	Conversion Overflow on REAL to INTEGER Conversion
SN	Argument to SIN Too Large
A2	Both Arguments of ATAN2 are 0
IO	Illegal I/O Operation
BI	Buffer Size Exceeded During Binary I/O
RC	Negative Repeat Count in FORMAT
Fatal Errors:	
ID	Illegal FORMAT Descriptor
F0	FORMAT Field Width is Zero
MP	Missing Period in FORMAT
FW	FORMAT Field Width is Too Small
IT	I/O Transmission Error
ML	Missing Left Parenthesis in FORMAT
DZ	Division by Zero, REAL or INTEGER
LG	Illegal Argument to LOG Function (Negative or Zero)
SQ	Illegal Argument to SQRT Function (Negative)
DT	Data Type Does Not Agree With FORMAT Specification
EF	EOF Encountered on READ

Runtime errors are surrounded by asterisks as follows:

FW

Fatal errors cause execution to cease (control is returned to the operating system). Execution continues after a warning error. However, after 20 warnings, execution ceases.

CONTENTS

Section 2 LINK-80 Linking Loader For FORTRAN & MACRO ASSEMBLER

Section		Page
2.1	Format of LINK-80 Commands	35
	2.1.1 LINK-80 Command Strings	35
	2.1.2 LINK-80 Switches	36
2.2	Sample Link	38
2.3	Format of LINK Compatible Object Files	38
2.4	LINK-80 Error Messages	40
2.5	Program Break Information	42

SECTION 2

LINK-80 Linking Loader

2.1 Format of LINK-80 Commands

2.1.1 LINK-80 Command Strings

To run LINK-80, type RUN L80 in response to the FCS> prompt. LINK-80 will return with the prompt L80> indicating that it is ready to accept commands. Each command to LINK-80 consists of a string of filenames and switches separated by commas:

```
objdev1:filename.ext/switch1,objdev2:filename.ext,...
```

If the input device for a file is omitted, the default is the currently logged disk. If the extension of a file is omitted, the default is .REL. After each line is typed, LINK will load or search (see /S below) the specified files. After LINK finishes this process, it will list all symbols that remained undefined followed by an asterisk.

Example:

```
*MAIN

 DATA     0100     0200

 SUBR1*        (SUBR1 is undefined)

 DATA     0100     0300

*SUBR1
*/G            (Starts Execution - see below)
```

Typically, to execute a FORTRAN and/or COBOL program and subroutines, the user types the list of filenames followed by /G (begin execution). Before execution begins, LINK-80 will always search the system library (FORLIB.REL or COBLIB.REL) to satisfy any unresolved external references. If the user wishes to first search libraries of his own, he should append the filenames that are followed by /S to the end of the loader command string.

2.1.2 LINK-80 Switches

A number of switches may be given in the LINK-80 command string to specify actions affecting the loading process. Each switch must be preceded by a slash (/). These switches are:

Switch	Action
R	Reset. Put loader back in its initial state. Use /R if you loaded the wrong file by mistake and want to restart. /R takes effect as soon as it is encountered in a command string.
E or E:Name	Exit LINK-80 and return to the Operating System. The system library will be searched on the current disk to satisfy any existing undefined globals. The optional form E:Name (where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program. Use /E to load a program and exit back to the monitor.
G or G:Name	Start execution of the program as soon as the current command line has been interpreted. The system library will be searched on the current disk to satisfy any existing undefined globals if they exist. Before execution actually begins, LINK-80 prints three numbers and a BEGIN EXECUTION message. The three numbers are the start address, the address of the next available byte, and the number of 256-byte pages used. The optional form G:Name (where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program.
N:Name	If /N:file is specified, the program will be saved on disk under the selected name with a default type of .PRG when a /E is done.

P and D

/P and /D allow the origin(s) to be set for the next program loaded. /P and /D take effect when seen (not deferred), and they have no effect on programs already loaded. The form is /P:<address> or /D:<address>, where <address> is the desired origin in the current typeout radix. (Default radix for non-MITS versions is hex. /O sets radix to octal; /H to hex.) LINK-80 does a default /P:<link origin>+3 (i.e., 103H for CP/M and 4003H for ISIS) to leave room for the jump to the start address.

NOTE: Do not use /P or /D to load programs or data into the locations of the loader's jump to the start address (100H to 102H for CPM and 2800H to 2802H for DTC), unless it is to load the start of the program there. If programs or data are loaded into these locations, the jump will not be generated.

If no /D is given, data areas are loaded before program areas for each module. If a /D is given, all Data and Common areas are loaded starting at the data origin and the program area at the program origin. Example:

```
*/P:200,FOO
Data     200     300
*/R
*/P:200 /D:400,FOO
Data     400     480
Program 200     280
```

U

List the origin and end of the program and data area and all undefined globals as soon as the current command line has been interpreted. The program information is only printed if a /D has been done. Otherwise, the program is stored in the data area.

M

List the origin and end of the program and data area, all defined globals and their values, and all undefined globals followed by an asterisk. The program information is only printed if a /D has been done. Otherwise, the program is stored in the data area.

S — Search the filename immediately preceding the /S in the command string to satisfy any undefined globals.

Examples:

*/M — List all globals

*MYPROG,SUBROT,MYLIB/S
Load MYPROG.REL and SUBROT.REL and then search MYLIB.REL to satisfy any remaining undefined globals.

*/G — Begin execution of main program

2.2 Sample Link

```
FCS>RUN L80
L80>EXAMPL,EXMPL1/G
DATA      3000    30AC
[304F     30AC    49]
[BEGIN EXECUTION]

            1792             14336
           14336            -16383
          -16383                14
              14               112
             112               896
FCS>
```

2.3 Format of LINK Compatible Object Files

NOTE

Section 2.3 is reference material for users who wish to know the load format of LINK-80 relocatable object files. Most users will want to skip this section, as it does not contain material necessary to the operation of the package.

LINK-compatible object files consist of a bit stream. Individual fields within the bit stream are not aligned on byte boundaries, except as noted below. Use of a bit stream for relocatable object files keeps the size of object files to a minimum, thereby decreasing the number of disk reads/writes.

There are two basic types of load items: Absolute and Relocatable. The first bit of an item indicates one of these two types. If the first bit is a 0, the following 8 bits are loaded as an absolute byte. If the first bit is a 1, the next 2 bits are used to indicate one of four types of relocatable items:

00	Special LINK item (see below).
01	Program Relative. Load the following 16 bits after adding the current Program base.
10	Data Relative. Load the following 16 bits after adding the current Data base.
11	Common Relative. Load the following 16 bits after adding the current Common base.

Special LINK items consist of the bit stream 100 followed by:

a four-bit control field

an optional A field consisting of a two-bit address type that is the same as the two-bit field above except 00 specifies absolute address

an optional B field consisting of 3 bits that give a symbol length and up to 8 bits for each character of the symbol

A general representation of a special LINK item is:

```
1 00 xxxx     yy     zzz + characters of symbol name
            --------  ------------------------------
             A field             B field
```

xxxx	Four-bit control field (0-15 below)
yy	Two-bit address type field
zzz	Three-bit symbol length field

The following special types have a B-field only:

0	Entry symbol (name for search)
1	Select COMMON block
2	Program name
3	Reserved for future expansion

4 Reserved for future expansion

The following special LINK items have both an A field and a B field:

5 Define COMMON size
6 Chain external (A is head of address chain, B is name of external symbol)
7 Define entry point (A is address, B is name)
8 Reserved for future expansion

The following special LINK items have an A field only:

9 External + offset. The A value will be added to the two bytes starting at the current location counter immediately before execution.
10 Define size of Data area (A is size)
11 Set loading location counter to A
12 Chain address. A is head of chain, replace all entries in chain with current location counter. The last entry in the chain has an address field of absolute zero.
13 Define program size (A is size)
14 End program (forces to byte boundary)

The following special Link item has neither an A nor a B field:

15 End file

2.4 LINK-80 Error Messages

LINK-80 has the following error messages:

?No Start Address	A /G switch was issued, but no main program had been loaded.
?Loading Error	The last file given for input was not a properly formatted LINK-80 object file.
?Out of Memory	Not enough memory to load program.
?Command Error	Unrecognizable LINK-80 command.
?<file> Not Found	<file>, as given in the command string, did not exist.

%2nd COMMON Larger /XXXXXX/

The first definition of COMMON block /XXXXXX/ was not the largest definition. Re-order module loading sequence or change COMMON block definitions.

%Mult. Def. Global YYYYYY

More than one definition for the global (internal) symbol YYYYYY was encountered during the loading process.

%Overlaying [Program / Data] Area

A /D or /P will cause already loaded data to be destroyed.

?Intersecting [Program / Data] Area

The program and data area intersect and an address or external chain entry is in this intersection. The final value cannot be converted to a current value since it is in the area intersection.

?Start Symbol - <name> - Undefined

After a /E: or /G: is given, the symbol specified was not defined.

Origin [Above / Below] Loader Memory, Move Anyway (Y or N)?

After a /E or /G was given, either the data or program area has an origin or top which lies outside loader memory (i.e., loader origin to top of memory). If a Y <cr> is given, LINK-80 will move the area and continue. If anything else is given, LINK-80 will exit. In either case, if a /N was given, the image will already have been saved.

?Can't Save Object File

A disk error occurred when the file was being saved.

2.5 Program Break Information

LINK-80 stores the address of the first free location in a global symbol called $MEMRY if that symbol has been defined by a program loaded. $MEMRY is set to the top of the data area +1.

NOTE

If /D is given and the data origin is less than the program area, the user must be sure there is enough room to keep the program from being destroyed. This is particularly true with the disk driver for FORTRAN-80 which uses $MEMRY to allocate disk buffers and FCB's.

CREATING A FORTRAN PROGRAMME

18/11/84

USE CD0 only!

Insert SCREEN EDITOR disk.

FCS> RUN EDITOR

Remove Screen Editor disk.

Insert "programme" disk.

FILE> FOR01.FOR — If file exists under the same name then it will load, otherwise entry of data will be allowed onto programme disk.

Enter data in correct format - data starts at column 7. The 1 for line continue must be in line 6.

F1 Key — Saves file FOR01.FOR to programme disk.

Remove "[redacted]" disk.

Insert "Compiler/Link" disk.

ERASE PAGE — To clear screen.

FCS> RUN F80

Remove "compiler/link" disk.

Insert "[redacted]" [redacted].

F80> FOR01

F80> Control, C

Remove "Programme" disk.

Insert "Compiler/Link" disk.

FCS> RUN L80

L80> /P:AF00

Remove "compiler/link" disk

Insert "Programme" disk.

L80> FOR01

--- } Produces data on screen

Remove "Programme" disk

Insert "Compiler/Link" disk.

L80> EQ — For Programme too large to link by normal method.

--- } Produces data on screen

Remove "compiler/link" disk.

USE CDO only!

Insert SCREEN EDITOR disk.

FCS>RUN EDITOR

Remove Screen Editor disk.

Insert "programme" disk.

FILE>FOR01.FOR — If file exists under the same name then it will load, otherwise entry of data will be allowed onto programme disk.

Enter data in correct format – data starts at column 7. The 1 for line continuance must be in line 6.

F1 Key — Saves file FOR01.FOR to programme disk.

Remove ~~[illegible]~~" disk.

Insert "Compiler/Link" disk.

ERASE PAGE — To clear screen.

FCS>RUN F80

Remove "compiler/link" disk.

Insert "programme" disk.

F80>FOR01

F80> Control, C

Remove "Programme" disk.

Insert "Compiler/Link" disk.

FCS>RUN L80

L80>/P:AF00

Remove "compiler/link" disk

Insert "Programme" disk.

L80>FOR01

--- } Produces data on screen.

Remove "Programme" disk

Insert "Compiler/link" disk — For programme too large to link by normal method.

L80>EQ

--- } Produces data on screen.

Remove "compiler/link" disk.

Insert "Programme" disk.

L80>/N:FOR01/E

— — ——— } produces data on screen!
— — ———
— — ———

Remove "Programme" disk.
Insert "Library" disk. } may not be necessary if on same disk.

FCS>RUN LIB

Remove "Library" disk.
Insert "Programme" disk.

FCS>RUN FOR01

↓ ↓

THE INTEGER AND FLOATING POINT VALUES APPEAR BELOW

INTERGER	FLOATING POINT
1	1.00
3	3.00
6	6.00
10	10.00
15	15.00
21	21.00
28	28.00
36	36.00
45	45.00
55	55.00

YOU HAVE SEEN THE CCII IN FORTRAN STOP

.REL FILES (AS USED BY FORTRAN F80 & L80 PROGRAMS)

The Fortran/Linker manual describes the format of .REL files (relocatable object files) briefly but to understand them more clearly an example is given below. The following page shows a file (RAN.REL) containing only one program (RAN) & could be created, using a monitor, at say AØØØH–A1ØØH & saved using FCS

SAVE RAN.REL ØØØØ 1Ø1 ØØØØ AØØØ

The programs in a .REL file are stored as a bit stream, ie byte boundaries are ignored except at start & end of programs. Using RAN.REL as an example, the program RAN would look like –

```
|    84    |    D4    |    9Ø    |    53    |    AØ    |    35    |
 10000100  11010100  10010000  01010011  10100000  00110101  etc
| |LINK  |3     |  'R'   |   'A'    |   'N'    | |LINK  |3     |
| |ITEM  |CHARS |        |          |          | |ITEM  |CHARS |
| |No 2. |      |        |          |          | |No. Ø |      |
```

& at the end would look like

```
|    Ø5    |    3A    |    7Ø    |    ØØ    |    ØØ    |    9E    |
 00000101  00111010  01110000  00000000  00000000  10011110
                |   |LINK |<-------- DUMMY ---------->|  |  |LINK |
                    |ITEM |         A FIELD              |  |ITEM |
                    |No 14|                           ^     |No 15|
                    |(END)|                           |     |(EOF)|
                                      FILLER TO BYTE BOUNDARY
```

If another program was to be placed in this file, it would start where the 9E (EOF) is located & the 9E (EOF) placed at the end of it, on a byte boundary.

If you are interested in decoding this program, you will find its structure is – PROGRAM NAME, ENTRY SYMBOL, DATA AREA SIZE, PROGRAM SIZE, PROGRAM, DATA, EXTERNAL SYMBOLS, DEFINITION OF ENTRY POINT & NAME, END. Note that some data areas, considered as program, follow the program area in the decoded .PRG program. The data referred to in DATA AREA SIZE actually precedes the program in the .PRG program. Note also that the CURRENT DATA BASE address refers to the first DATA AREA address & the CURRENT PROGRAM BASE address refers to the first PROGRAM AREA address.

[signature]
7/10/82

```
FORTRAN-80 Ver. 3.44 Copyright 1978-1981 (C) By Microsoft
Created: 10-Dec-81
          PROGRAM TSTRAN
          DO 1 I=1,100
          A=RAN(1.0)
        1 WRITE(2,101) A
      101 FORMAT(1X,F20.18)
          END
```

RANDOM NUMBER GENERATOR
FOR FORTRAN
RAN.REL

```
.935920298099517820
.496393710374832150
.302433401346206670
.359691292047500610
.726209938526153560
.122689582407474520
.847446143627166750
.290751188993453980
.328568428754806520
.027434146031737328
.146319359540939330
.113490380346775050
.943169653415679930
.453304439783096310
.334688931703567500
.771126270294189450
.475711673498153690
.651789367198944090
.606506526470184330
.478847175836563110
.630349695682525630
.803459286689758300
.765748620033264160
.472283989191055300
.540884554386138920
.118236698210239410
.875186800956726070
.795387685298919680
.090225078165531158
.353524476289749150
.846359670162200930
.551600217819213870
.417007893323898320
.116311408579349520
.816287636756896970
.238754004240036010
.597436428070068360
.276686996221542360
.585148990154266360
.018523013219237328
.663575291633605960
.040373716503381729
.867599606513977050
.560356497764587400
.876200854778289790
.340892106294631960
.999435544013977050
.707714736461639400
.293658167123794560
.769314944744110110
.284211724996566770
.698695123195648190
.552061021327972410
.0982801690697[illegible]98
.31153997770[illegible]
.[illegible]39801454[illegible]70
```

```
MON> D,A000-A100
A000 84 D4 90 53 A0 35 24 14
A008 E9 40 38 04 D6 7C 01 9A
A010 00 00 08 70 30 03 EA B5
A018 80 08 70 00 03 D5 9B 40
A020 40 1E 22 1C 0C 00 C8 43
A028 39 80 E0 60 01 DC 43 5C
A030 C0 10 E8 72 78 25 9A 00
A038 00 0E B0 28 00 F1 CC 03
A040 03 00 1F C0 14 40 CB 02
A048 80 08 6C 78 02 1D 0E 4F
A050 04 B3 40 00 01 08 00 00
A058 5E 11 95 84 64 E1 19 18
A060 46 7B 2C BB 89 E4 F1 B2
A068 00 56 46 1B 20 04 38 10
A070 00 68 7E 6B 2A D8 55 84
A078 00 EE 0C 0A 87 19 A0 00
A080 02 BC 43 80 00 18 60 00
A088 01 DC 56 77 15 9D D8 74
A090 F4 00 6A 4A 65 26 47 21
A098 C3 B2 60 44 95 59 A6 01
A0A0 4D D2 3A 60 A5 D1 4C A6
A0A8 41 41 A4 FA 60 CA BC 66
A0B0 A6 1A C7 71 F2 60 D0 81
A0B8 23 1A 13 2E 94 91 A4 20
A0C0 D1 3A 9A 25 C0 00 0F D2
A0C8 50 04 FE 00 00 00 23 27
A0D0 60 0C 91 05 0A 32 7C 00
A0D8 C9 10 50 E3 21 C0 0C 91
A0E0 30 C6 32 46 00 C9 13 50
A0E8 A3 2C 40 10 91 39 49 32
A0F0 32 D2 00 C9 15 0C 63 A0
A0F8 00 0D 49 05 3A 70 00 00
A100 9E .
```

A = RAN (1.Ø)
GIVES NEXT IN SEQUENCE

A = RAN (Ø.Ø)
REPEATS LAST

A = RAN (-1.Ø)
GIVES FIRST IN NEW SEQUENCE.

RANDOM NOS. ARE
GIVEN BETWEEN Ø & 1

7/10/82

RAN. PRC

```
-L0121,01C6
 0121   ??=   FD
 0122   MOV   C,D
 0123   LXI   B,007F      DATA
 0126   NOP
 0127   NOP
 0128   CALL  0420
 012B   LXI   H,0127
 012E   JM    0193        PRC
 0131   LXI   H,0121
 0134   PUSH  PSW
 0135   CALL  0420
 0138   POP   PSW
 0139   LXI   H,0127
 013C   RZ
 013D   ADD   M
 013E   ANI   07
 0140   MVI   B,00
 0142   MOV   M,A
 0143   LXI   H,019B
 0146   ADD   A
 0147   ADD   A
 0148   MOV   C,A
 0149   DAD   B
 014A   CALL  0242
 014D   LDA   0126
 0150   INR   A
 0151   ANI   03
 0153   MVI   B,00
 0155   CPI   01
 0157   ADC   B
 0158   STA   0126
 015B   LXI   H,01B7
 015E   ADD   A
 015F   ADD   A
 0160   MOV   C,A
 0161   DAD   B
 0162   CALL  01CD
 0165   LXI   H,0720
 0168   MOV   E,M
 0169   INX   H
 016A   MOV   D,M
 016B   INX   H
 016C   MOV   C,M
 016D   INX   H
 016E   MOV   B,M
 016F   INX   H
 0170   MOV   A,E
 0171   MOV   E,C
 0172   XRI   4F
 0174   MOV   C,A
 0175   MVI   M,80
 0177   DCX   H
 0178   MOV   B,M
 0179   MVI   M,80
 017B   LXI   H,0125
 017E   INR   M
 017F   MOV   A,M
 0180   SUI   AB
 0182   JNZ   0189
 0185   MOV   M,A
 0186   INR   C
 0187   DCR   D
 0188   INR   E
 0189   CALL  0329
 018C   XRA   A
 018D   LXI   H,0121
 0190   JMP   0411
 0193   MOV   M,A
 0194   DCX   H           PRC
 0195   MOV   M,A
 0196   DCX   H
 0197   MOV   M,A
 0198   JMP   0165
 019B   DCR   M
 019C   MOV   C,D
 019D   JZ    3999
 01A0   INR   E           DATA
 01A1   HLT
 01A2   SBB   B
 01A3   SHLD  B395
 01A6   SBB   B
 01A7   LDAX  B
 01A8   ??=   DD
 01A9   MOV   B,A
 01AA   SBB   B
 01AB   MOV   D,E
 01AC   POP   D
 01AD   SBB   C
 01AE   SBB   C
 01AF   LDAX  B
 01B0   LDAX  D
 01B1   SBB   A
 01B2   SBB   B
 01B3   MOV   H,L
 01B4   CMP   H
 01B5   CALL  D698
 01B8   MOV   M,A
 01B9   MVI   A,98
 01BB   MOV   L,B
 01BC   ORA   C
 01BD   MOV   B,M
 01BE   MOV   L,B
 01BF   SBB   C
 01C0   PCHL
 01C1   SUB   D
 01C2   MOV   L,C
 01C3   ??=   10          DATA
 01C4   POP   D
 01C5   MOV   M,L
 01C6   MOV   L,B
 01C7
```