MICROSOFT FORTRAN-80

version 3.3

reference manual

microsoft

fortran-80 documentation

Microsoft FORTRAN-80 and associated software are accompanied by the following documents:

1. FORTRAN-80 REFERENCE MANUAL provides an extensive description of FORTRAN-80's statements, functions and syntax.

2. FORTRAN-80 USER'S MANUAL describes the FORTRAN-80 compiler commands and error messages.

3. MICROSOFT UTILITY SOFTWARE MANUAL describes the use of the MACRO-80 Assembler, LINK-80 Linking Loader and LIB-80 Library Manager with the FORTRAN-80 compiler.

MICROSOFT FORTRAN-80

Microsoft's FORTRAN-80 package provides new capabilities for users of 8080 and Z80 based microcomputer systems. FORTRAN-80 is comparable to FORTRAN compilers on large mainframes and minicomputers. All of ANSI Standard FORTRAN X3.9-1966 is included except the COMPLEX data type. Therefore, users may take advantage of the many applications programs already written in FORTRAN.

Versions of FORTRAN-80 for the CP/M, TEKDOS, ISIS-II and DTC Microfile floppy disk operating systems are available off the shelf. Other versions will be prepared based upon user demand.

Relocatable Code and Library Features

FORTRAN-80 is unique in that it provides a microprocessor FORTRAN and assembly language development package that generates relocatable object modules. This means that only the subroutines and system routines required to run FORTRAN-80 programs are loaded before execution. Subroutines can be placed in a system library so that users develop a common set of subroutines that are used in their programs. Also, if only one module of a program is changed, it is necessary to recompile only that module.

The standard library of subroutines supplied with FORTRAN-80 includes:

ABS	IABS	DABS	AINT
INT	IDINT	AMOD	MOD
AMAX0	AMAX1	MAX0	MAX1
DMAX1	AMIN0	AMIN1	MIN0
MIN1	DMIN1	FLOAT	IFIX
SIGN	ISIGN	DSIGN	DIM
IDIM	SNGL	DBLE	EXP
DEXP	ALOG	DLOG	ALOG10
DLOG10	SIN	DSIN	COS
DCOS	TANH	SQRT	DSQRT
ATAN	DATAN	ATAN2	DATAN2
DMOD	PEEK	POKE	INP
OUT			

The library also contains routines for 32-bit and 64-bit floating point addition, subtraction, multiplication, division, etc. These routines are among the fastest available for performing these functions on the 8080.

Enhancements

The FORTRAN-80 complier has a number of enhancements of the ANSI Standard:

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

FORTRAN-80 Compiler Characteristics

The FORTRAN-80 compiler can compile several hundred statements per minute in a single pass and needs less than 24K bytes of memory to compile most programs. Any extra available memory will be used by the compiler for extended optimizations.

In spite of its small size, the FORTRAN-80 compiler optimizes the generated object code in several ways:

1. Common subexpression elimination. Common subexpressions are evaluated once, and the value is substituted in later occurrences of the subexpression.
2. Peephole Optimization. Small sections of code are replaced by more compact, faster code in special cases. Example: I=I+1 uses an INX H instruction instead of a DAD.

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

4. Branch Optimizations. The number of conditional jumps in arithmetic and logical IFs is minimized.

Long descriptive error messages are another feature of the compiler. For instance:

? Statement unrecognizable

is printed if the compiler scans a statement that is not an assignment or other FORTRAN statement. The last twenty characters scanned before the error is detected are also printed.

The compiler generates a fully symbolic listing of the machine language being generated. At the end of the listing, the compiler produces an error summary and tables showing the addresses assigned to labels, variables and constants.

Assembler and Linker

A relocating macro assembler (MACRO-80) and relocating linking loader (LINK-80) are included in the FORTRAN-80 package.

The relocating assembler resides in approximately 14K and includes a complete Intel-standard macro facility with IRP, IRPC, REPEAT, local variables and EXITM. MACRO-80 also provides a full set of conditional pseudo-operations plus comment blocks, octal or hex listings and a variable input radix. The assembler accepts both Intel 8080 and Zilog Z80 op codes.

LINK-80, the relocating loader, resolves internal and external references between the object modules loaded. LINK-80 also performs library searches for system subroutines and generates a load map of memory showing the locations of the main program, subroutines and COMMON areas. LINK-80 requires approximately 8K bytes of memory.

Custom I/O Drivers

Users may write non-standard I/O drivers for each Logical Unit Number, making the task of interfacing non-standard devices to FORTRAN programs a straightforward one.

Support

FORTRAN-80 users receive quick turnaround on bug fixes. Updates to FORTRAN-80 are announced regularly and are available for a minimal charge.

Other Products

Microsoft's complete product line includes BASIC for the 6502 and 6800, 8080 BASIC, 8080 COBOL, and EDIT-80, a line-oriented text editor. In addition, Microsoft has development software that runs on the DEC-10 for all of these microprocessors.

Prices

FORTRAN-80 system (including documentation)	\$500.00
FORTRAN-80 documentation only	\$ 20.00

OEM and dealer agreements are available upon request.

Microsoft
10800 NE Eighth, Suite 819
Bellevue, WA 98004
206-455-8080
Telex 328945

MICROSOFT

MICROSOFT FORTRAN-80
Reference Manual

Contents

SECTION 1

INTRODUCTION

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

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

This manual defines the FORTRAN source language for the 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:

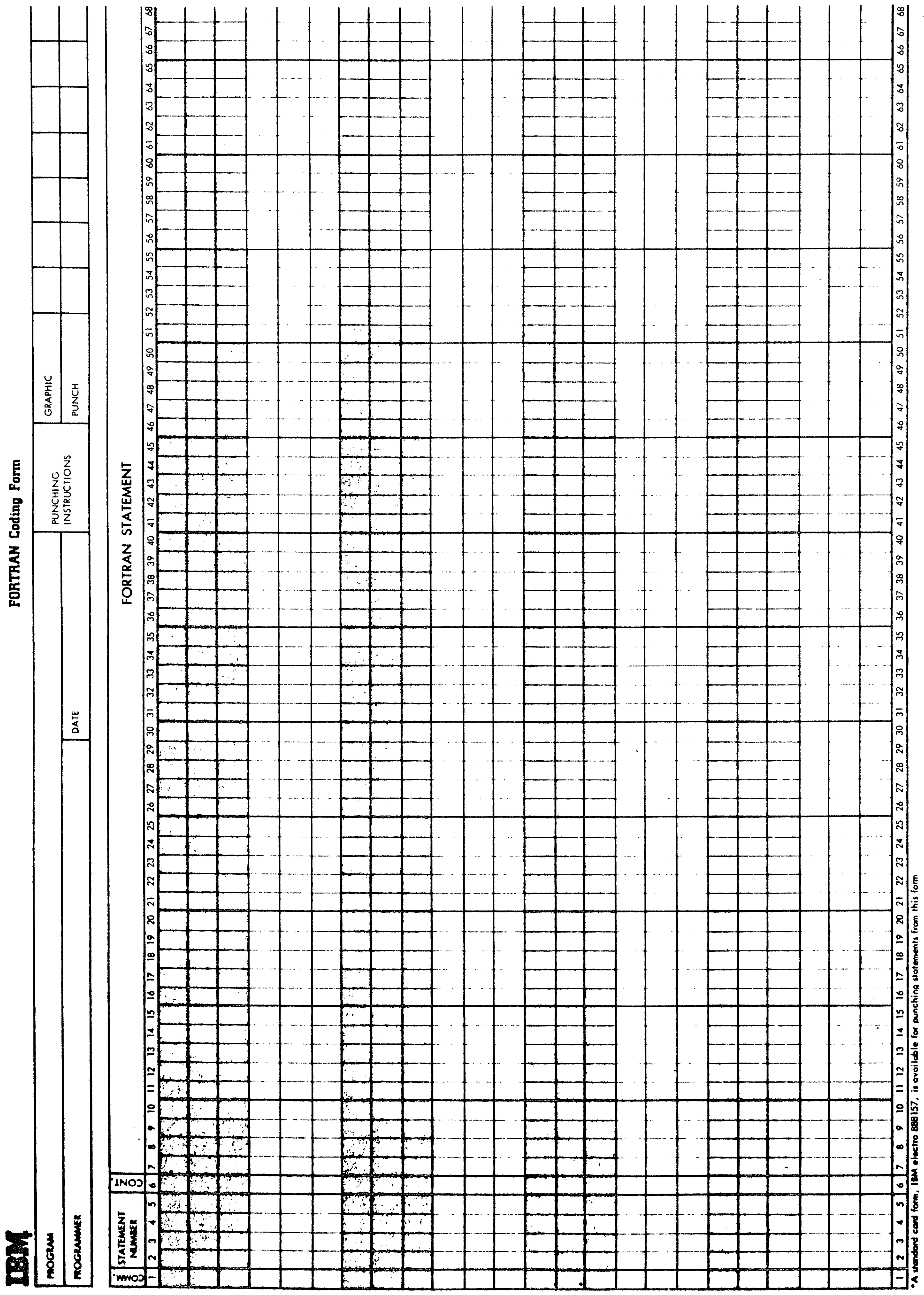
1. Comment line -- used for source program annotation at the convenience of the programmer.

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

    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.



IBM

FORTRAN Coding Form

PROGRAM		PUNCHING INSTRUCTIONS	GRAPHIC	
PROGRAMMER	DATE		PUNCH	

C FOR COMMENT | STATEMENT NUMBER | CONT. | FORTRAN STATEMENT

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68

*A standard card form, IBM electro 888157, is available for punching statements from this form

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 _other than_ zero or blank.

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

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

   Example:

```
C  THE STATEMENTS BELOW ARE AN INITIAL LINE
C       AND 2 CONTINUATION LINES
C
  63  BETA(1,2) =
     1     A6BAR**7-(BETA(2,2)-A5BAR*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
  101
99999
 763
```

2.3 STATEMENTS

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

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

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

Non-executable statements describe to the processor the nature and arrangement of data and provide information about input/output formats and data initialization to the object 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 <u>not</u> be subscripted. Examples:

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

3.6 <u>DATA STORAGE ALLOCATION</u>

Allocation of storage for FORTRAN data is made in numbers of <u>storage units</u>. 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. The expression X**Y**Z is not allowed. It should be written as follows:

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

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

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

The following conditions apply to replacement statements:

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

   Example:

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

   The line containing v= must be the 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 seven kinds of specification statements. They are as follows:

IMPLICIT statements
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 = R(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(15)),
     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'/
```

6.9 IMPLICIT STATEMENT

The IMPLICIT statement is used to redefine default variable types. The syntax is:

```
     IMPLICIT type(range),type(range),...
```

where type is one of the following: INTEGER, REAL, LOGICAL, DOUBLE PRECISION, BYTE, INTEGER*1, INTEGER*2, REAL*4, REAL*8

and range is a list of alphabetic characters separated by commas or hyphens.

Examples:

```
     IMPLICIT INTEGER(A,W-Z),REAL(B-V)
```

All variables (not otherwise declared) starting with the letters A, W, X, Y, Z will be type INTEGER. All variables starting with the letters B through V will be type REAL.

```
     IMPLICIT INTEGER(I-N),REAL(A-H,O-Z)
```

This is the default definition.

Any IMPLICIT statements must appear in a program grouped with the Type and DIMENSION statements. IMPLICIT statements must appear before any other specification statements. If the IMPLICIT statement appears after any Type or DIMENSION statements, the types of the variables already declared will not be affected.

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.

<u>Example:</u>

```
        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 <u>COMPUTED GO TO</u>

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.

<u>Example:</u>

```
        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 <u>ASSIGNED GO TO</u>

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	Overhead	
Type	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 is only 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 User's Manual, Section 3). 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 8.7.10)

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 (see FORMAT statements,8.7). 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 8.6.

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 8.7 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 8.6). 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, 8.7.3).

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 <u>FORMATTED WRITE STATEMENTS</u>

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, 8.7). 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 DISK FILE I/O

A READ or WRITE to a disk file (LUN 6-10) automatically OPENs the file for I/O. The file remains open until closed by an ENDFILE command (see Section 8.4) or until normal program termination.

NOTE

Exercise caution when doing sequential output to disk files. If output is done to an existing file, the existing file will be deleted and replaced with a new file of the same name.

8.3.1 RANDOM DISK I/O

SEE ALSO SECTION 3 OF YOUR MICROSOFT FORTRAN USER'S MANUAL.

Some versions of FORTRAN-80 also provide random disk I/O. For random disk access, the record number is specified by using the REC=n option in the READ or WRITE statement. For example:

```
        I = 10
        WRITE (6,20,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.

In random access files, the record length varies with different versions of FORTRAN. See Section 3 of your Microsoft FORTRAN User's Manual. It is recommended that any file you wish to read randomly be created via FORTRAN (or Microsoft BASIC) random access statements. Files created this way (using either binary or formatted WRITE statements) will zero-fill each record to the proper length if the data does not fill the record.

Any disk file that is OPENed by a READ or WRITE statement is assigned a default filename that is specific to the operating system. See also Section 3 of the FORTRAN User's Manual.

8.3.2 OPEN SUBROUTINE

Alternatively, a file may be OPENed using the OPEN subroutine. LUNs 1-5 may also be assigned to disk files with OPEN. The OPEN subroutine allows the program to specify a filename and device to be associated with a LUN.

An OPEN of a non-existent file creates a null file of the appropriate name. An OPEN of an existing file followed by sequential 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 varies under different operating systems. See your Microsoft FORTRAN User's Manual, Section 3.

8.4 AUXILIARY I/O STATEMENTS

Three auxiliary I/O statements are provided:

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

The actions of all three statements depend on the LUN with which they are used (see Appendix B). When the LUN is for a terminal or line printer, the three statements are defined as no-ops.

When the LUN is for a disk drive, the ENDFILE and REWIND commands allow further program control of disk files. ENDFILE u closes the file associated with LUN u. REWIND u closes the file associated with LUN u, then opens it again. BACKSPACE is not implemented at this time, and therefore causes an error if used.

8.5 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:

```
        1
ENCODE(a,f) k
DECODE(a,f) k
```

INPUT ASCII DATA

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.

8.6 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.

8.6.1 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.

Examples:

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

NOTE

Sublists are not implemented.

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)
```

8.6.2 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.

8.7 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:

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

where m 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.

8.7.1 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.

8.7.2 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	bb368.4200
F7.1	-4786.361	-4786.4
F8.4	8.7E-2	bb0.0870
F6.4	4739.76	*.7600
F7.3	-5.6	b-5.600

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

F-Input

(See the description under E-Input below.)

<u>E-type Conversion</u>

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	bb.76573Eb02
E14.7	-32672.354	-b.3267235Eb05
E13.4	-0.0012321	bb-b.1232E-02
E8.2	76321.73	b.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	-632.1132

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

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

8.7.3 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=blank)
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 hi 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=blank)
1HA	or 'A'	A
8HbSTRINGb	or 'bSTRINGb'	bSTRINGb
11HX(2,3)=12.0	or 'X(2,3)=12.0'	X(2,3)=12.0
11HIbSHOULDN'T	or 'IbSHOULDN''T'	IbSHOULDN'T

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'

8.7.4 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.

8.7.5 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

8.7.6 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.

8.7.7 SPECIAL CONTROL FEATURES OF FORMAT STATEMENTS

8.7.7.1 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 first opening parenthesis in the FORMAT statement. A repeat count preceding the parenthesized descriptor(s) to be re-used is also active in the re-use. This type of repetitive use of FORMAT descriptors terminates processing of 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 8.7.7.2 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.

8.7.7.2 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.

8.7.8 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 8.7.7.1)

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.

8.7.9 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

8.7.10 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 largest integer <= |a|	Real	Real
INT		Real	Integer
IDINT		Double	Integer
AMOD	a1(mod a2)	Real	Real
MOD	MOD(A1,A2)	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	Type Argument	Type Function
EXP	1	e**a	Real	Real
DEXP	1		Double	Double
ALOG	1	ln (a)	Real	Real
DLOG	1		Double	Double
ALOG10	1	log10(a)	Real	Real
DLOG10	1		Double	Double
SIN	1	sin (a)	Real	Real
DSIN	1		Double	Double
COS	1	cos (a)	Real	Real
DCOS	1		Double	Double
TANH	1	tanh (a)	Real	Real
SQRT	1	(a) ** 1/2	Real	Real
DSQRT	1		Double	Double
ATAN	1	arctan (a)	Real	Real
DATAN	1		Double	Double
ATAN2	2	arctan (a1/a2)	Real	Real
DATAN2	2		Double	Double
DMOD	2	a1(mod a2)	Double	Double

9.4 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.

Example:

```
      Calling Program Unit

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

      Called Program Unit

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

In this example, 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,F,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.

5. In Input/Output list specifications, sublists enclosed in parentheses are not allowed.

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     022      ;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     0177     ;AND OFF PARITY
001C  FE 0A                   CPI     10       ;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,0      ;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    $BL      ;SAVE IT
0033  FE 0D                   CPI     015      ;CR?
0035  C8                      RZ               ;YES--DONE
0036  7D                      MOV     A,L      ;$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'
```

```
            MAC80 1.0        PAGE     1

                             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

            MAC80 1.0        PAGE     2

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. Neither the compiler nor the runtime system checks for the correct number of parameters.

If the subprogram expects more than 3 parameters, and needs to transfer them to a local data area, there is a 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

Referencing 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.

1. <u>Referencing Arithmetic Routines</u>

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

2. Referencing Intrinsic Functions

Instrinsic Functions are passed their parameters in H,L and D,E. If there are three arguments, B,C contains the third parameter. If there are more than three arguments, B,C contains a pointer to a block in memory that holds the remaining parameters. Each of these parameters is a pointer to an argument. (See Appendix B.)

For a MIN or MAX function, the number of arguments is passed in A.

NOTE: None of the functions (except INP and OUT) may take a byte variable as an argument. Byte variables must first be converted to the type expected by the function. Otherwise, results will be unpredictable.

3. Formatted READ and WRITE Routines

A READ or WRITE statement calls one of the following routines:

$W2 (2 parameters) $W5 (5 parameters)	Initialize for an I/O transfer to a device (WRITE)
$R2 (2 parameters) $R5 (5 parameters)	Initialize for an I/O transfer from a device (READ)

These routines adhere to the following calling conventions:

1. H,L points to the LUN
2. D,E points to the beginning of the FORMAT statement
3. If the routine has five parameters, then B,C points to a block of three parameters:

    a. the address for an ERR= branch

    b. the address for an EOF= branch

    c. the address for a REC= value

    If one of the parameters is missing, its place in the parameter block is filled with a zero.

The routines that transfer values into the I/O buffer are:

$I0	transfers integers
$I1	transfers real numbers
$I2	transfers logicals
$I3	transfers double precision numbers

Transfer routines adhere to the following calling conventions:

1. H,L points to a location that contains the number of dimensions for the variables in the list
2. D,E points to the first value to be transferred
3. B,C points to the second value to be transferred if there are exactly two values to be transferred by this call. If there are more than two values, B,C points to a block that contains pointers to the second through nth values.

4. Register A contains the number of parameters (including H,L) generated by this call.

The routine $ND terminates the I/O process.

Example:

```
         EXTRN    $W2,$IO,$ND
         ENTRY    TEST
TEST:    LXI      H,LUN
         LXI      D,FORMAT
         CALL     $W2

         LXI      H,DIMENS
         LXI      D,NUMBER
         MVI      A,2
         CALL     $IO

         CALL     $ND

         RET
LUN:     DW       1
FORMAT:  DB       '(11H RESULT IS=,I5)'
DIMENS:  DW       1
NUMBER:  DW       9999

         END      TEST
```

INDEX

MICROSOFT FORTRAN-80

user's manual

Microsoft FORTRAN-80 User's Manual

CONTENTS

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 FORTRAN-80, type F80 followed by a carriage return. FORTRAN-80 will return the prompt "*" (with the DTC operating system, the prompt is ">"), indicating it is ready to accept commands. The general format of a FORTRAN-80 command string is:

```
objprog-dev:filename.ext,list-dev:filename.ext=
   source-dev:filename.ext
```

objprog-dev:
The device on which the object program is to be written.

list-dev:
The device on which the program listing is written.

source-dev:
The device from which the source-program input to FORTRAN-80 is obtained. If a device name is omitted, it defaults to the currently selected drive.

filename.ext
The filename and filename extension of the object program file, the listing file, and the source file. Filename extensions may be omitted. See Section 4 of the Microsoft Utility Software Manual for the default extension supplied by your operating system.

Either the object file or the listing file or both may be omitted. If neither a listing file nor an object file is desired, place only a comma to the left of the equal sign. If the names of the object file and the listing file are omitted, the default is the name of the source file.

Examples:

`*=TEST`	Compile the program TEST.FOR and place the object in TEST.REL
`*,TTY:=TEST`	Compile the program TEST.FOR and list program on the terminal. No object is generated.
`*TESTOBJ=TEST.FOR`	Compile the program TEST.FOR and put object in TESTOBJ.REL
`*TEST,TEST=TEST`	Compile TEST.FOR, put object in TEST.REL and listing in TEST.LST
`*,=TEST.FOR`	Compile TEST.FOR but produce no object or listing file. Useful for checking for errors.

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. (Default for ALTAIR DOS)
H	Print all listing addresses, etc. in hexadecimal. (Default for non-ALTAIR versions)
N	Do not list generated code.
R	Force generation of an object file.
L	Force generation of a listing file.
P	Each /P allocates an extra 100 bytes of stack space for use during compilation. Use /P if stack overflow errors occur during compilation. Otherwise not needed.

M	Specifies to the compiler that the generated code should be in a form which can be loaded into ROMs. When a /M is specified, the generated code will differ from normal in the following ways: 1. FORMATs will be placed in the program area, with a "JMP" around them. 2. Parameter blocks (for subprogram calls with more than 3 parameters) will be initialized at runtime, rather than being initialized by the loader.

Examples:

*,TTY:=MYPROG/N	Compile file MYPROG.FOR and list program on terminal but without generated code.
*=TEST/L	Compile TEST.FOR with object file TEST.REL and listing file TEST.LST
*=BIGGONE/P/P	Compile file BIGGONE.FOR and produce object file BIGGONE.REL. Compiler is allocated 200 extra bytes of stack space.

NOTE

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

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

2. FORMATs should not be read into during execution.

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

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

The calling convention is:

```
   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. 3.2 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 1.8 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 warnikg error. However, after 20 warnings, execution ceases.

SECTION 3

FORTRAN-80 Disk Files

3.1 Random Disk I/O

In the current release of FORTRAN-80, only the CP/M and ISIS-II versions provide random disk I/O capability.

3.2 Default Disk Filenames

A disk file (random or sequential) that is OPENed by a READ or WRITE statement has a default name that depends upon the LUN and the operating system:

'NNNNN' XXX'

CP/M and ISIS II:	FORT06.DAT, FORT07.DAT,..., FORT10.DAT
ALTAIR:	FOR06DAT, FOR07DAT, ..., FOR10DAT
DTC:	FOR06D, FOR07D,..., FOR10D

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

3.3 CALL OPEN

Instead of using READ or WRITE, a disk file may be OPENed using the OPEN subroutine (see the FORTRAN-80 Reference Manual, Section 8.3.2). The format of an OPEN call under CP/M, Altair and DTC is:

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

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 blank-filled to exactly the number of characters allowed by the operating system:

CP/M:	11 characters
ALTAIR:	8 characters
DTC:	6 characters

Drive = the number of the disk drive on which the file exists or will exists (must be an Integer constant or Integer variable within the range allowed by the operating system). If the Drive specified is 0, the currently selected drive is assumed; 1 is drive 0 (or A), 2 is drive 1 (or B), etc.

The form of an OPEN call under ISIS-II 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 ISIS-II, i.e., a device name surrounded by colons, 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). <u>The Filename must be terminated by a non-alphanumeric character.</u>

The following are examples of valid OPEN calls under ISIS-II:

```
CALL OPEN (6, ':F1:FOO.DAT ')

CALL OPEN (1, ':F5:TESTFF.TMP ')

CALL OPEN (4, ':F3:A.B ')
```

3.4 Record Length

The record length of any file accessed randomly under CP/M or ISIS-II 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 created this way (using either binary or formatted WRITE statements) always have 128-byte records. If the WRITE statement does not transfer enough data to fill the record to 128 bytes, then the end of the record is filled with zeros (NULL characters).

MICROSOFT

utility software manual

Microsoft
Utility Software Manual

CONTENTS

SECTION 1

MACRO-80 Assembler

OBJ DEV: FILE [illegible]
LIST DEV: FIL [illegible]
[illegible]

1.1 Format of MACRO-80 Commands

1.1.1 MACRO-80 Command Strings

To run MACRO-80, type M80 followed by a carriage return. MACRO-80 will return the prompt "*" (with the DTC operating system, the prompt is ">"), indicating it is ready to accept commands. The format of a MACRO-80 command string is:

```
objprog-dev:filename.ext,list-dev:filename.ext=
   source-dev:filename.ext
```

objprog-dev:
The device on which the object program is to be written.

list-dev:
The device on which the program listing is written.

source-dev:
The device from which the source-program input to MACRO-80 is obtained. If a device name is omitted, it defaults to the currently selected drive.

filename.ext
The filename and filename extension of the object program file, the listing file, and the source file. Filename extensions may be omitted. See Section 4 for the default extension supplied by your operating system.

Either the object file or the listing file or both may be omitted. If neither a listing file nor an object file is desired, place only a comma to the left of the equal sign. If the names of the object file and the listing file are omitted, the default is the name of the source file.

Examples:

*=SOURCE.MAC	Assemble the program SOURCE.MAC and place the object in SOURCE.REL
*,LST:=TEST	Assemble the program TEST.MAC and list on device LST

*SMALL,TTY:=TEST	Assemble the program TEST.MAC, place the object in SMALL.REL and list on TTY

1.1.2 MACRO-80 Switches

A number of different switches may be given in the MACRO-80 command string that will affect the format of the listing file. Each switch must be preceded by a slash (/):

Switch	Action
O	Print all listing addresses, etc. in octal. (Default for Altair DOS)
H	Print all listing addresses, etc. in hexadecimal. (Default for non-Altair versions)
R	Force generation of an object file.
L	Force generation of a listing file.
C	Force generation of a cross reference file.
Z	Assemble Z80 (Zilog format) mnemonics. (Default for Z80 operating systems)
I	Assemble 8080 mnemonics. (Default for 8080 operating systems)

Examples:

*=TEST/L	Compile TEST.MAC with object file TEST.REL and listing file TEST.LST
*LAST,LAST/C=MOD1	Compile MOD1.MAC with object file LAST.REL and cross reference file LAST.CRF for use with CREF-80 (See Section 1.12)

1.2 Format of MACRO-80 Source Files

In general, MACRO-80 accepts a source file that is almost identical to source files for INTEL compatible assemblers. Input source lines of up to 132 characters in length are acceptable.

MACRO-80 preserves lower case letters in quoted strings and comments. All symbols, opcodes and pseudo-opcodes typed in lower case will be converted to upper case.

NOTE

If the source file includes line numbers from an editor, each byte of the line number must have the high bit on. Line numbers from Microsoft's EDIT-80 Editor are acceptable.

1.2.1 Statements

Source files input to MACRO-80 consist of statements of the form:

```
[label:[:]]  [operator]  [arguments]    [;comment]
```

With the exception of the ISIS assembler $ controls (see Section 1.10), it is not necessary that statements begin in column 1. Multiple blanks or tabs may be used to improve readability.

If a label is present, it is the first item in the statement and is immediately followed by a colon. If it is followed by two colons, it is declared as PUBLIC (see ENTRY/PUBLIC, Section 1.5.10). For exmple:

```
     FOO::    RET
```

is equivalent to

```
     PUBLIC   FOO
     FOO:     RET
```

The next item after the label (or the first item on the line if no label is present) is an operator. An operator may be an opcode (8080 or Z80 mnemonic), pseudo-op, macro call or expression. The evaluation order is as follows:

1. Macro call

2. Opcode/Pseudo operation

3. Expression

Instead of flagging an expression as an error, the assembler treats it as if it were a DB statement (see Section 1.5.4).

The arguments following the operator will, of course, vary in form according to the operator.

A comment always begins with a semicolon and ends with a carriage return. A comment may be a line by itself or it may be appended to a line that contains a statement. Extended comments can be entered using the .COMMENT pseudo operation (see Section 1.5.19).

1.2.2 Symbols

MACRO-80 symbols may be of any length, however, only the first six characters are significant. The following characters are legal in a symbol:

```
A-Z      0-9      $      .      ?      @
```

With the 8080/Z80 assembler, the underline character is also legal in a symbol. A symbol may not start with a digit. When a symbol is read, lower case is translated into upper case. If a symbol reference is followed by ## it is declared external (see also the EXT/EXTRN pseudo-op, Section 1.5.12).

1.2.3 Numeric Constants

The default base for numeric constants is decimal. This may be changed by the .RADIX pseudo-op (see Section 1.5.21). Any base from 2 (binary) to 16 (hexadecimal) may be selected. When the base is greater than 10, A-F are the digits following 9. If the first digit of the number is not numeric (i.e., A-F), the number must be preceded by a zero. <u>This eliminates the use of zero as a leading digit for octal constants, as in previous versions of MACRO-80.</u>

Numbers are 16-bit unsigned quantities. A number is always evaluated in the current radix unless one of the following special notations is used:

nnnnB	Binary
nnnnD	Decimal
nnnnO	Octal
nnnnQ	Octal
nnnnH	Hexadecimal
X'nnnn'	Hexadecimal

Overflow of a number beyond two bytes is ignored and the result is the low order 16-bits.

A character constant is a string comprised of zero, one or two ASCII characters, delimited by quotation marks, and used in a non-simple expression. For example, in the statement

```
        DB      'A' + 1
```

'A' is a character constant. But the statement

```
        DB      'A'
```

uses 'A' as a string because it is in a simple expression. The rules for character constant delimiters are the same as for strings.

A character constant comprised of one character has as its value the ASCII value of that character. That is, the high order byte of the value is zero, and the low order byte is the ASCII value of the character. For example, the value of the constant 'A' is 41H.

A character constant comprised of two characters has as its value the ASCII value of the first character in the high order byte and the ASCII value of the second character in the low order byte. For example, the value of the character constant "AB" is 41H*256+42H.

1.2.4 Strings

A string is comprised of zero or more characters delimited by quotation marks. Either single or double quotes may be used as string delimiters. The delimiter quotes may be used as characters if they appear twice for every character occurrence desired. For example, the statement

```
        DB      "I am ""great"" today"
```

stores the string

```
                I am "great" today
```

If there are zero characters between the delimiters, the string is a null string.

1.3 Expression Evaluation

1.3.1 Arithmetic and Logical Operators

The following operators are allowed in expressions. The operators are listed in order of precedence.

NUL

LOW, HIGH

*, /, MOD, SHR, SHL

Unary Minus

+, -

EQ, NE, LT, LE, GT, GE

NOT

AND

OR, XOR

Parentheses are used to change the order of precedence. During evaluation of an expression, as soon as a new operator is encountered that has precedence less than or equal to the last operator encountered, all operations up to the new operator are performed. That is, subexpressions involving operators of higher precedence are computed first.

All operators except +, -, *, / must be separated from their operands by at least one space.

The byte isolation operators (HIGH, LOW) isolate the high or low order 8 bits of an Absolute 16-bit value. If a relocatable value is supplied as an operand, HIGH and LOW will treat it as if it were relative to location zero.

1.3.2 Modes

All symbols used as operands in expressions are in one of the following modes: Absolute, Data Relative, Program (Code) Relative or COMMON. (See Section 1.5 for the ASEG, CSEG, DSEG and COMMON pseudo-ops.) Symbols assembled under the ASEG, CSEG (default), or DSEG pseudo-ops are in Absolute, Code Relative or Data Relative mode respectively. The number of COMMON modes in a program is determined by the number of COMMON blocks that have been named with the COMMON pseudo-op. Two COMMON symbols are not in the same mode unless they are in the same COMMON block.

In any operation other than addition or subtraction, the mode of both operands must be Absolute.

If the operation is addition, the following rules apply:

1. At least one of the operands must be Absolute.

2. Absolute + <mode> = <mode>

If the operation is subtraction, the following rules apply:

1. <mode> - Absolute = <mode>

2. <mode> - <mode> = Absolute
   where the two <mode>s are the same.

Each intermediate step in the evaluation of an expression must conform to the above rules for modes, or an error will be generated. For example, if FOO, BAZ and ZAZ are three Program Relative symbols, the expression

```
FOO + BAZ - ZAZ
```

will generate an R error because the first step (FOO + BAZ) adds two relocatable values. (One of the values must be Absolute.) This problem can always be fixed by inserting parentheses. So that

```
FOO + (BAZ - ZAZ)
```

is legal because the first step (BAZ - ZAZ) generates an Absolute value that is then added to the Program Relative value, FOO.

1.3.3 Externals

Aside from its classification by mode, a symbol is either External or not External. (See EXT/EXTRN, Section 1.5.12.) An External value must be assembled into a two-byte field. (Single-byte Externals are not supported.) The following rules apply to the use of Externals in expressions:

1. Externals are legal only in addition and subtraction.

2. If an External symbol is used in an expression, the result of the expression is always External.

3. When the operation is addition, either operand (but not both) may be External.

4. When the operation is subtraction, only the first operand may be External.

1.4 Opcodes as Operands

8080 opcodes are valid one-byte operands. Note that only the first byte is a valid operand. For example:

```
        MVI    A,(JMP)
        ADI    (CPI)
        MVI    B,(RNZ)
        CPI    (INX H)
        ACI    (LXI B)
        MVI    C,MOV A,B
```

Errors will be generated if more than one byte is included in the operand -- such as (CPI 5), LXI B,LABEL1) or (JMP LABEL2).

Opcodes used as one-byte operands need not be enclosed in parentheses.

NOTE

Opcodes are not valid operands in Z80 mode.

1.5 Pseudo Operations

1.5.1 ASEG

```
        ASEG
```

ASEG sets the location counter to an absolute segment of memory. The location of the absolute counter will be that of the last ASEG (default is 0), unless an ORG is done after the ASEG to change the location. The effect of ASEG is also achieved by using the code segment (CSEG) pseudo operation and the /P switch in LINK-80. See also Section 1.5.27.

1.5.2 COMMON

```
COMMON /<block name>/
```

COMMON sets the location counter to the selected common block in memory. The location is always the beginning of the area so that compatibility with the FORTRAN COMMON statement is maintained. If <block name> is omitted or consists of spaces, it is considered to be blank common. See also Section 1.5.27.

1.5.3 CSEG

```
CSEG
```

CSEG sets the location counter to the code relative segment of memory. The location will be that of the last CSEG (default is 0), unless an ORG is done after the CSEG to change the location. CSEG is the default condition of the assembler (the INTEL assembler defaults to ASEG). See also Section 1.5.27.

1.5.4 Define Byte

```
DB      <exp>[,<exp>...]

DB      <string>[<string>...]
```

The arguments to DB are either expressions or strings. DB stores the values of the expressions or the characters of the strings in successive memory locations beginning with the current location counter.

Expressions must evaluate to one byte. (If the high byte of the result is 0 or 255, no error is given; otherwise, an A error results.)

Strings of three or more characters may not be used in expressions (i.e., they must be immediately followed by a comma or the end of the line). The characters in a string are stored in the order of appearance, each as a one-byte value with the high order bit set to zero.

Example:

```
0000'   4142            DB      'AB'
0002'   42              DB      'AB' AND 0FFH
0003'   41 42 43        DB      'ABC'
```

1.5.5 Define Character

```
DC      <string>
```

DC stores the characters in <string> in successive memory locations beginning with the current location counter. As with DB, characters are stored in order of appearance, each as a one-byte value with the high order bit set to zero. However, DC stores the last character of the string with the high order bit set to one. An error will result if the argument to DC is a null string.

1.5.6 Define Space

```
DS      <exp>
```

DS reserves an area of memory. The value of <exp> gives the number of bytes to be allocated. All names used in <exp> must be previously defined (i.e., all names known at that point on pass 1). Otherwise, a V error is generated during pass 1 and a U error may be generated during pass 2. If a U error is not generated during pass 2, a phase error will probably be generated because the DS generated no code on pass 1.

1.5.7 DSEG

```
DSEG
```

DSEG sets the location counter to the Data Relative segment of memory. The location of the data relative counter will be that of the last DSEG (default is 0), unless an ORG is done after the DSEG to change the location. See also Section 1.5.27.

1.5.8 Define Word

```
DW      <exp>[,<exp>...]
```

DW stores the values of the expressions in successive memory locations beginning with the current location counter. Expressions are evaluated as 2-byte (word) values.

1.5.9 END

```
END     [<exp>]
```

The END statement specifies the end of the program. If <exp> is present, it is the start address of the program. If <exp> is not present, then no start address is passed to LINK-80 for that program.

1.5.10 ENTRY/PUBLIC

```
ENTRY  <name>[,<name>...]
 or
PUBLIC <name>[,<name>...]
```

ENTRY or PUBLIC declares each name in the list as internal and therefore available for use by this program and other programs to be loaded concurrently. All of the names in the list must be defined in the current program or a U error results. An M error is generated if the name is an external name or common-blockname.

1.5.11 EQU

```
<name> EQU <exp>
```

EQU assigns the value of <exp> to <name>. If <exp> is external, an error is generated. If <name> already has a value other than <exp>, an M error is generated.

1.5.12 EXT/EXTRN

```
EXT    <name>[,<name>...]
 or
EXTRN  <name>[,<name>...]
```

EXT or EXTRN declares that the name(s) in the list are external (i.e., defined in a different program). If any item in the list references a name that is defined in the current program, an M error results. A reference to a name where the name is followed immediately by two pound signs (e.g., NAME##) also declares the name as external.

1.5.13 NAME

NAME ('modname')

NAME defines a name for the module. Only the first six characters are significant in a module name. A module name may also be defined with the TITLE pseudo-op. In the absence of both the NAME and TITLE pseudo-ops, the module name is created from the source file name.

1.5.14 Define Origin

ORG <exp>

The location counter is set to the value of <exp> and the assembler assigns generated code starting with that value. All names used in <exp> must be known on pass 1, and the value must either be absolute or in the same area as the location counter.

1.5.15 PAGE

PAGE [<exp>]

PAGE causes the assembler to start a new output page. The value of <exp>, if included, becomes the new page size (measured in lines per page) and must be in the range 10 to 255. The default page size is 50 lines per page. The assembler puts a form feed character in the listing file at the end of a page.

1.5.16 SET

<name> SET <exp>

SET is the same as EQU, except no error is generated if <name> is already defined.

1.5.17 SUBTTL

SUBTTL <text>

SUBTTL specifies a subtitle to be listed on the line after the title (see TITLE, Section 1.5.18) on each page heading. <text> is truncated after 60 characters. Any number of SUBTTLs may be given in a program.

1.5.18 TITLE

```
TITLE <text>
```

TITLE specifies a title to be listed on the first line of each page. If more than one TITLE is given, a Q error results. The first six characters of the title are used as the module name unless a NAME pseudo operation is used. If neither a NAME or TITLE pseudo-op is used, the module name is created from the source filename.

1.5.19 .COMMENT

```
.COMMENT <delim><text><delim>
```

The first non-blank character encountered after .COMMENT is the delimiter. The following <text> comprises a comment block which continues until the next occurrence of <delimiter> is encountered. For example, using an asterisk as the delimiter, the format of the comment block would be:

```
.COMMENT  *
any amount of text entered
here as the comment block
    .
    .
    .     *
;return to normal mode
```

1.5.20 .PRINTX

```
.PRINTX <delim><text><delim>
```

The first non-blank character encountered after .PRINTX is the delimiter. The following text is listed on the terminal during assembly until another occurrence of the delimiter is encountered. .PRINTX is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches. For example:

```
IF     CPM
.PRINTX /CPM version/
ENDIF
```

NOTE

.PRINTX will output on both passes. If only one printout is desired, use the IF1 or IF2 pseudo-op.

1.5.21 .RADIX

```
.RADIX <exp>
```

The default base (or radix) for all constants is decimal. The .RADIX statement allows the default radix to be changed to any base in the range 2 to 16. For example:

```
LXI    H,0FFH
.RADIX 16
LXI    H,0FF
```

The two LXIs in the example are identical. The <exp> in a .RADIX statement is always in decimal radix, regardless of the current radix.

1.5.22 .REQUEST

```
.REQUEST <filename>[,<filename>...]
```

.REQUEST sends a request to the LINK-80 loader to search the filenames in the list for undefined globals before searching the FORTRAN library. The filenames in the list should be in the form of legal MACRO-80 symbols. They should not include filename extensions or disk specifications. The LINK-80 loader will scpply its default extension and will assume the currently selected disk drive.

1.5.23 .Z80

.Z80 enables the assembler to accept Z80 opcodes. This is the default condition when the assembler is running on a Z80 operating system. Z80 mode may also be set by appending the Z switch to the MACRO-80 command string -- see Section 1.1.2.

1.5.24 .8080

.8080 enables the assembler to accept 8080 opcodes. This is the default condition when the assembler is running on an 8080 operating system. 8080 mode may also be set by appending the I switch to the MACRO-80 command string -- see Section 1.1.2.

1.5.25 Conditional Pseudo Operations

The conditional pseudo operations are:

IF/IFT	True if <exp> is not 0.
IFE/IFF <exp>	True if <exp> is 0.
IF1	True if pass 1.
IF2	True if pass 2.
IFDEF <symbol>	True if <symbol> is defined or has been declared External.
IFNDEF <symbol>	True if <symbol> is undefined or not declared External.
IFB <arg>	True if <arg> is blank. The angle brackets around <arg> are required.
IFNB <arg>	True if <arg> is not blank. Used for testing when dummy parameters are supplied. The angle brackets around <arg> are required.

All conditionals use the following format:

```
IFxx   [argument]
  .
  .
  .
[ELSE
  .
  .
  .     ]
ENDIF
```

Conditionals may be nested to any level. Any argument to a conditional must be known on pass 1 to avoid V errors and incorrect evaluation. For IF, IFT, IFF, and IFE the expression must involve values which were previously defined and the expression must be absolute. If the name is defined after an IFDEF or IFNDEF, pass 1 considers the name to be undefined, but it will be defined on pass 2.

ELSE
Each conditional pseudo operation may optionally be used with the ELSE pseudo operation which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a

given IF, and an ELSE is always bound to the most recent, open IF. A conditional with more than one ELSE or an ELSE without a conditional will cause a C error.

ENDIF
Each IF must have a matching ENDIF to terminate the conditional. Otherwise, an 'Unterminated conditional' message is generated at the end of each pass. An ENDIF without a matching IF causes a C error.

1.5.26 Listing Control Pseudo Operations

Output to the listing file can be controlled by two pseudo-ops:

.LIST and .XLIST

If a listing is not being made, these pseudo-ops have no effect. .LIST is the default condition. When a .XLIST is encountered, source and object code will not be listed until a .LIST is encountered.

The output of cross reference information is controlled by .CREF and .XCREF. If the cross reference facility (see Section 1.12) has not been invoked, .CREF and .XCREF have no effect. The default condition is .CREF. When a .XCREF is encountered, no cross reference information is output until .CREF is encountered.

The output of MACRO/REPT/IRP/IRPC expansions is controlled by three pseudo-ops: .LALL, .SALL, and .XALL. .LALL lists the complete macro text for all expansions. .SALL lists only the object code produced by a macro and not its text. .XALL is the default condition; it is similar to .SALL, except a source line is listed only if it generates object code.

1.5.27 Relocation Pseudo Operations

The ability to create relocatable modules is one of the major features of MACRO-80. Relocatable modules offer the advantages of easier coding and faster testing, debugging and modifying. In addition, it is possible to specify segments of assembled code that will later be loaded into RAM (the Data Relative segment) and ROM/PROM (the Code Relative segment). The pseudo operations that select relocatable areas are CSEG and DSEG. The ASEG pseudo-op is used to generate non-relocatable (absolute) code. The COMMON pseudo-op creates a common data area for every COMMON block that is named in the program.

The default mode for the assembler is Code Relative. That is, assembly begins with a CSEG automatically executed and the location counter in the Code Relative mode, pointing to location 0 in the Code Relative segment of memory. All subsequent instructions will be assembled into the Code Relative segment of memory until an ASEG or DSEG or COMMON pseudo-op is executed. For example, the first DSEG encountered sets the location counter to location zero in the Data Relative segment of memory. The following code is asembled in the Data Relative mode, that is, it is assigned to the Data Relative segment of memory. If a subsequent CSEG is encountered, the location counter will return to the next free location in the Code Relative segment and so on.

The ASEG, DSEG, CSEG pseudo-ops never have operands. If you wish to alter the current value of the location counter, use the ORG pseudo-op.

<u>ORG Pseudo-op</u>
At any time, the value of the location counter may be changed by use of the the ORG pseudo-op. The form of the ORG statement is:

```
        ORG     <exp>
```

where the value of <exp> will be the new value of the location counter in the current mode. All names used in <exp> must be known on pass 1 and the value of <exp> must be either Absolute or in the current mode of the location counter. For example, the statements

```
        DSEG
        ORG     50
```

set the Data Relative location counter to 50, <u>relative to the start of the Data Relative segment of memory</u>.

<u>LINK-80</u>
The LINK-80 linking loader (see Section 2 of this manual) combines the segments and creates each relocatable module in memory when the program is loaded. The origins of the relocatable segments are not fixed until the program is loaded and the origins are assigned by LINK-80. The command to

LINK-80 may contain user-specified origins through the use of the /P (for Code Relative) and /D (for Data and COMMON segments) switches.

For example, a program that begins with the statements

```
        ASEG
        ORG    800H
```

and is assembled entirely in Absolute mode will always load beginning at 800 unless the ORG statement is changed in the source file. However, the same program, assembled in Code Relative mode with no ORG statement, may be loaded at any specified address by appending the /P:<address> switch to the LINK-80 command string.

1.5.28 Relocation Before Loading

Two pseudo-ops, .PHASE and .DEPHASE, allow code to be located in one area, but executed only at a different, specified area.

For example:

```
  0000'                        .PHASE  100H
  0100    CD 0106     FOO:     CALL    BAZ
  0103    C3 0007'             JMP     ZOO
  0106    C9          BAZ:     RET
                               .DEPHASE
  0007'   C3 0005     ZOO:     JMP     5
```

All labels within a .PHASE block are defined as the absolute value from the origin of the phase area. The code, however, is loaded in the current area (i.e., from 0' in this example). The code within the block can later be moved to 100H and executed.

1.6 Macros and Block Pseudo Operations

The macro facilities provided by MACRO-80 include three repeat pseudo operations: repeat (REPT), indefinite repeat (IRP), and indefinite repeat character (IRPC). A macro definition operation (MACRO) is also provided. Each of these four macro operations is terminated by the ENDM pseudo operation.

1.6.1 Terms

For the purposes of discussion of macros and block operations, the following terms will be used:

1. <dummy> is used to represent a dummy parameter. All dummy parameters are legal symbols that appear in the body of a macro expansion.

2. <dummylist> is a list of <dummy>s separated by commas.

3. <arglist> is a list of arguments separated by commas. <arglist> must be delimited by angle brackets. Two angle brackets with no intervening characters (<>) or two commas with no intervening characters enter a null argument in the list. Otherwise an argument is a character or series of characters terminated by a comma or >. With angle brackets that are nested inside an <arglist>, one level of brackets is removed each time the bracketed argument is used in an <arglist>. (See example, Section 1.6.5.) A quoted string is an acceptable argument and is passed as such. Unless enclosed in brackets or a quoted string, leading and trailing spaces are deleted from arguments.

4. <paramlist> is used to represent a list of actual parameters separated by commas. No delimiters are required (the list is terminated by the end of line or a comment), but the rules for entering null parameters and nesting brackets are the same as described for <arglist>. (See example, Section 1.6.5.)

1.6.2 REPT-ENDM

```
REPT   <exp>
 .
 .
 .
ENDM
```

The block of statements between REPT and ENDM is repeated <exp> times. <exp> is evaluated as a 16-bit unsigned number. If <exp> contains any external or undefined terms, an error is generated. Example:

```
SET    0
REPT   10      ;generates DB1-DB10
SET    X+1
DB     X
ENDM
```

1.6.3 IRP-ENDM

```
        IRP     <dummy>,<arglist>
         .
         .
         .
        ENDM
```

The <arglist> must be enclosed in angle brackets. The number of arguments in the <arglist> determines the number of times the block of statements is repeated. Each repetition substitutes the next item in the <arglist> for every occurrence of <dummy> in the block. If the <arglist> is null (i.e., <>), the block is processed once with each occurrence of <dummy> removed. For example:

```
        IRP     X,<1,2,3,4,5,6,7,8,9,10>
        DB      X
        ENDM
```

generates the same bytes as the REPT example.

1.6.4 IRPC-ENDM

```
        IRPC    <dummy>,string (or <string>)
         .
         .
         .
        ENDM
```

IRPC is similar to IRP but the arglist is replaced by a string of text and the angle brackets around the string are optional. The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of <dummy> in the block. For example:

```
        IRPC    X,0123456789
        DB      X+1
        ENDM
```

generates the same code as the two previous examples.

1.6.5 MACRO

Often it is convenient to be able to generate a given sequence of statements from various places in a program, even though different parameters may be required each time the sequence is used. This capability is provided by the MACRO statement. The form is

```
<name> MACRO <dummylist>
          .
          .
          .
        ENDM
```

where <name> conforms to the rules for forming symbols. <name> is the name that will be used to invoke the macro. The <dummy>s in <dummylist> are the parameters that will be changed (replaced) each time the MACRO is invoked. The statements before the ENDM comprise the body of the macro. During assembly, the macro is expanded everytime it is invoked but, unlike REPT/IRP/IRPC, the macro is not expanded when it is encountered.

The form of a macro call is

```
<name> <paramlist>
```

where <name> is the name supplied in the MACRO definition, and the parameters in <paramlist> will replace the <dummy>s in the MACRO <dummylist> on a one-to-one basis. The number of items in <dummylist> and <paramlist> is limited only by the length of a line. The number of parameters used when the macro is called need not be the same as the number of <dummy>s in <dummylist>. If there are more parameters than <dummmy>s, the extras are ignored. If there are fewer, the extra <dummy>s will be made null. The assembled code will contain the macro expansion code after each macro call.

NOTE

A dummy parameter in a MACRO/REPT/IRP/IRPC is always recognized exclusively as a dummmy parameter. Register names such as A and B will be changed in the expansion if they were used as dummy parameters.

Here is an example of a MACRO definition that defines a macro called FOO:

```
FOO     MACRO   X
Y       SET     0
        REPT    X
Y       SET     Y+1
        DB      Y
        ENDM
        ENDM
```

This macro generates the same code as the previous three examples when the call

```
        FOO     10
```

is executed.

Another example, which generates the same code, illustrates the removal of one level of brackets when an argument is used as an arglist:

```
FOO     MACRO   X
        IRP     Y,<X>
        DB      Y
        ENDM
        ENDM
```

When the call

```
        FOO     <1,2,3,4,5,6,7,8,9,10>
```

is made, the macro expansion looks like this:

```
        IRP     Y,<1,2,3,4,5,6,7,8,9,10>
        DB      Y
        ENDM
```

1.6.6 ENDM

Every REPT, IRP, IRPC and MACRO pseudo-op must be terminated with the ENDM pseudo-op. Otherwise, the 'Unterminated REPT/IRP/IRPC/MACRO' message is generated at the end of each pass. An unmatched ENDM causes an O error.

1.6.7 EXITM

The EXITM pseudo-op is used to terminate a REPT/IRP/IRPC or MACRO call. When an EXITM is executed, the expansion is exited immediately and any remaining expansion or repetition is not generated. If the block containing the EXITM is nested within another block, the outer level continues to be expanded.

1.6.8 LOCAL

LOCAL <dummylist>

The LOCAL pseudo-op is allowed only inside a MACRO definition. When LOCAL is executed, the assembler creates a unique symbol for each <dummy> is <dummylist> and substitutes that symbol for each occurrence of the <dummy> in the expansion. These unique symbols are usually used to define a label within a macro, thus eliminating multiply-defined labels on successive expansions of the macro. The symbols created by the assembler range from ..0001 to ..FFFF. Users will therefore want to avoid the form ..nnnn for their own symbols. If LOCAL statements are used, they must be the first statements in the macro definition.

1.6.9 Special Macro Operators and Forms

& The ampersand is used in a macro expansion to concatenate text or symbols. A dummy parameter that is in a quoted string will not be substituted in the expansion unless it is immediately preceded by &. To form a symbol from text and a dummy, put & between them. For example:

```
ERRGEN  MACRO    X
ERROR&X:PUSH     B
        MVI      B,'&X'
        JMP      ERROR
        ENDM
```

In this example, the call ERRGEN A will generate:

```
ERRORA: PUSH     B
        MVI      B,'A'
        JMP      ERROR
```

;; In a block operation, a comment preceded by two semicolons is not saved as part of the expansion (i.e., it will not appear on the listing even under .LALL). A comment preceded by one semicolon, however, will be preserved and appear in the expansion.

! When an exclamation point is used in an argument, the next character is entered literally (i.e., !; and <;> are equivalent).

NUL NUL is an operator that returns true if its argument (a parameter) is null. The remainder of a line after NUL is considered to be the argument to NUL. The conditional

```
IF NUL argument
```

is false if, during the expansion, the first character of the argument is anything other than a semicolon or carriage return. It is recommended that testing for null parameters be done using the IFB and IFNB conditionals.

1.7 Using Z80 Pseudo-ops

When using the 8080/Z80 assembler, the following Z80 pseudo-ops are valid. The function of each pseudo-op is equivalent to that of its 8080 counterpart.

Z80 pseudo-op	Equivalent 8080 pseudo-op
COND	IFT
ENDC	ENDIF
*EJECT	PAGE
DEFB	DB
DEFS	DS
DEFW	DW
DEFM	DB
DEFL	SET
GLOBAL	PUBLIC
EXTERNAL	EXTRN

The formats, where different, conform to the 8080 format. That is, DEFB and DEFW are permitted a list of arguments (as are DB and DW), and DEFM is permitted a string or numeric argument (as is DB).

1.8 Sample Assembly

```
A>M80

*EXMPL1,TTY:=EXMPL1

        MAC80 3.2          PAGE      1


                                     00100    ;CSL3(P1,P2)
                                     00200    ;SHIFT P1 LEFT CIRCULARLY 3 BITS
                                     00300    ;RETURN RESULT IN P2
                                     00400            ENTRY    CSL3
                                     00450    ;GET VALUE OF FIRST PARAMETER
                                     00500    CSL3:
 0000'    7E                         00600            MOV      A,M
 0001'    23                         00700            INX      H
 0002'    66                         00800            MOV      H,M
 0003'    6F                         00900            MOV      L,A
                                     01000    ;SHIFT COUNT
 0004'    06 03                      01100            MVI      B,3
 0006'    AF                         01200    LOOP:   XRA      A
                                     01300    ;SHIFT LEFT
 0007'    29                         01400            DAD      H
                                     01500    ;ROTATE IN CY BIT
 0008'    17                         01600            RAL
 0009'    85                         01700            ADD      L
 000A'    6F                         01800            MOV      L,A
                                     01900    ;DECREMENT COUNT
 000B'    05                         02000            DCR      B
                                     02100    ;ONE MORE TIME
 000C'    C2 0006'                   02200            JNZ      LOOP
 000F'    EB                         02300            XCHG
                                     02400    ;SAVE RESULT IN SECOND PARAMETER
 0010'    73                         02500            MOV      M,E
 0011'    23                         02600            INX      H
 0012'    72                         02700            MOV      M,D
 0013'    C9                         02800            RET
                                     02900            END


        MAC80 3.2          PAGE      S


CSL3      0000I'   LOOP      0006'


No  Fatal error(s)
```

1.9 MACRO-80 Errors

MACRO-80 errors are indicated by a one-character flag in column one of the listing file. If a listing file is not being printed on the terminal, each erroneous line is also printed or displayed on the terminal. Below is a list of the MACRO-80 Error Codes:

A	Argument error Argument to pseudo-op is not in correct format or is out of range (.PAGE 1; .RADIX 1; PUBLIC 1; STAX H; MOV M,M; INX C).
C	Conditional nesting error ELSE without IF, ENDIF without IF, two ELSEs on one IF.
D	Double Defined symbol Reference to a symbol which is multiply defined.
E	External error Use of an external illegal in context (e.g., FOO SET NAME##; MVI A,2-NAME##).
M	Multiply Defined symbol Definition of a symbol which is multiply defined.
N	Number error Error in a number, usually a bad digit (e.g., 8Q).
O	Bad opcode or objectionable syntax ENDM, LOCAL outside a block; SET, EQU or MACRO without a name; bad syntax in an opcode (MOV A:); or bad syntax in an expression (mismatched parenthesis, quotes, consecutive operators, etc.).
P	Phase error Value of a label or EQU name is different on pass 2.
Q	Questionable Usually means a line is not terminated properly. This is a warning error (e.g. MOV A,B,).
R	Relocation Illegal use of relocation in expression, such as abs-rel. Data, code and COMMON areas are relocatable.

U Undefined symbol
A symbol referenced in an expression is not defined. (For certain pseudo-ops, a V error is printed on pass 1 and a U on pass 2.)

V Value error
On pass 1 a pseudo-op which must have its value known on pass 1 (e.g., .RADIX, .PAGE, DS, IF, IFE, etc.), has a value which is undefined. If the symbol is defined later in the program, a U error will not appear on the pass 2 listing.

Error Messages:

'No end statement encountered on input file'
No END statement: either it is missing or it is not parsed due to being in a false conditional, unterminated IRP/IRPC/REPT block or terminated macro.

'Unterminated conditional'
At least one conditional is unterminated at the end of the file.

'Unterminated REPT/IRP/IRPC/MACRO'
At least one block is unterminated.

[xx] [No] Fatal error(s) [,xx warnings]
The number of fatal errors and warnings. The message is listed on the CRT and in the list file.

1.10 Compatibility with Other Assemblers

The $EJECT and $TITLE controls are provided for compatability with INTEL's ISIS assembler. The dollar sign must appear in column 1 only if spaces or tabs separate the dollar sign from the control word. The control

```
$EJECT
```

is the same as the MACRO-80 PAGE pseudo-op.
The control

```
$TITLE('text')
```

is the same as the MACRO-80 SUBTTL <text> pseudo-op.

The INTEL operands PAGE and INPAGE generate Q errors when used with the MACRO-80 CSEG or DSEG pseudo-ops. These errors are warnings; the assembler ignores the operands.

When MACRO-80 is entered, the default for the origin is Code Relative 0. With the INTEL ISIS assembler, the default is Absolute 0.

With MACRO-80, the dollar sign ($) is a defined constant that indicates the value of the location counter at the start of the statement. Other assemblers may use a decimal point or an asterisk. Other constants are defined by MACRO-80 to have the following values:

```
A=7     B=0     C=1     D=2     E=3
H=4     L=5     M=6     SP=6    PSW=6
```

1.11 Format of Listings

On each page of a MACRO-80 listing, the first two lines have the form:

```
[TITLE text]    MAC80 3.2    PAGE x[-y]
[SUBTTL text]
```

where:

1. TITLE text is the text supplied with the TITLE pseudo-op, if one was given in the source program.

2. x is the major page number, which is incremented only when a form feed is encountered in the source file. (When using Microsoft's EDIT-80 text editor, a form feed is inserted whenever a page mark is done.) When the symbol table is being printed, x = 'S'.

3. y is the minor page number, which is incremented whenever the .PAGE pseudo-op is encountered in the source file, or whenever the current page size has been filled.

4. SUBTTL text is the text supplied with the SUBTTL pseudo-op, if one was given in the source program.

Next, a blank line is printed, followed by the first line of output.

A line of output on a MACRO-80 listing has the following form:

```
[crf#]     [error] loc#m    xx    xxxx ...     source
```

If cross reference information is being output, the first item on the line is the cross reference number, followed by a tab.

A one-letter error code followed by a space appears next on the line, if the line contains an error. If there is no error, a space is printed. If there is no cross reference number, the error code column is the first column on the listing.

The value of the location counter appears next on the line. It is a 4-digit hexadecimal number or 6-digit octal number, depending on whether the /O or /H switch was given in the MACRO-80 command string.

The character at the end of the location counter value is the mode indicator. It will be one of the following symbols:

'	Code Relative
"	Data Relative
!	COMMON Relative
<space>	Absolute
*	External

Next, three spaces are printed followed by the assembled code. One-byte values are followed by a space. Two-byte values are followed by a mode indicator. Two-byte values are printed in the opposite order they are stored in, i.e., the high order byte is printed first. Externals are either the offset or the value of the pointer to the next External in the chain.

The remainder of the line contains the line of source code, as it was input.

1.11.1 Symbol Table Listing

In the symbol table listing, all the macro names in the program are listed alphabetically, followed by all the symbols in the program, listed alphabetically. After each symbol, a tab is printed, followed by the value of the symbol. If the symbol is Public, an I is printed immediately after the value. The next character printed will be one of the following:

U	Undefined symbol.
C	COMMON block name. (The "value" of the COMMON block is its length (number of bytes) in hexadecimal or octal.)
*	External symbol.
<space>	Absolute value.
'	Program Relative value.
"	Data Relative value.
!	COMMON Relative value.

1.12 Cross Reference Facility

The Cross Reference Facility is invoked by typing CREF80. In order to generate a cross reference listing, the assembler must output a special listing file with embedded control characters. The MACRO-80 command string tells the assembler to output this special listing file. (See Section 1.5.26 for the .CREF and .XCREF pseudo-ops.) /C is the cross reference switch. When the /C switch is encountered in a MACRO-80 command string, the assembler opens a .CRF file instead of a .LST file.

Examples:

*=TEST/C	Assemble file TEST.MAC and create object file TEST.REL and cross reference file TEST.CRF.
*T,U=TEST/C	Assemble file TEST.MAC and create object file T.REL and cross reference file U.CRF.

When the assembler is finished, it is necessary to call the cross reference facility by typing CREF80. The command string is:

```
*listing file=source file
```

The default extension for the source file is .CRF. The /L switch is ignored, and any other switch will cause an error message to be sent to the terminal. Possible command strings are:

*=TEST	Examine file TEST.CRF and generate a cross reference listing file TEST.LST.
*T=TEST	Examine file TEST.CRF and generate a cross reference listing file T.LST.

Cross reference listing files differ from ordinary listing files in that:

1. Each source statement is numbered with a cross reference number.

2. At the end of the listing, variable names appear in alphabetic order along with the numbers of the lines on which they are referenced or defined. Line numbers on which the symbol is defined are flagged with '#'.

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 L80 followed by a carriage return. LINK-80 will return the prompt "*" (with the DTC operating system, the prompt is ">"), indicating 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	If a <filename>/N is specified, the program will be saved on disk under the selected name (with a default extension of .COM for CP/M) when a /E or /G is done. A jump to the start of the program is inserted if needed so the program can run properly (at 100H for CP/M).

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

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

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

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

xxxx	Four-bit control field (0-15 below)
yy	Two-bit address type field
00	Sixteen-bit value
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	Request library search

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] Area [,Start = xxxx                       ]
            [Data   ]      [,Public = <symbol name>(xxxx)       ]
                           [,External = <symbol name>(xxxx)     ]
```

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

```
?Intersecting [Program] Area
              [Data   ]
```

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

?Start Symbol - <name> - Undefined

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

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

After a /E or /G was given, either the data or program area has an origin 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.

SECTION 3

LIB-80 Library Manager (CP/M Versions Only)

LIB-80 is the object time library manager for CP/M versions of FORTRAN-80 and COBOL-80. LIB-80 will be interfaced to other operating systems in future releases of FORTRAN-80 and COBOL-80.

3.1 LIB-80 Commands

To run LIB-80, type LIB followed by a carriage return. LIB-80 will return the prompt "*" (with the DTC operating system, the prompt is ">"), indicating it is ready to accept commands. Each command in LIB-80 either lists information about a library or adds new modules to the library under construction.

Commands to LIB-80 consists of an optional destination filename which sets the name of the library being created, followed by an equal sign, followed by module names separated by commas. The default destination filename is FORLIB.LIB. Examples:

```
*NEWLIB=FILE1 <MOD2>, FILE3,TEST

*SIN,COS,TAN,ATAN
```

Any command specifying a set of modules concatenates the modules selected onto the <u>end</u> of the last destination filename given. Therefore,

```
*FILE1,FILE2 <BIGSUB>, TEST
```

is equivalent to

```
*FILE1
*FILE2 <BIGSUB>
*TEST
```

3.1.1 Modules

A module is typically a FORTRAN or COBOL subprogram, main program or a MACRO-80 assembly that contains ENTRY statements.

The primary function of LIB-80 is to concatenate modules in .REL files to form a new library. In order to extract modules from previous libraries or .REL files, a powerful syntax has been devised to specify ranges of modules within a .REL file.

The simplest way to specify a module within a file is simply to use the name of the module. For example:

```
SIN
```

But a relative quantity plus or minus 255 may also be used. For example:

```
SIN+1
```

specifies the module after SIN and

```
SIN-1
```

specifies the one before it.

Ranges of modules may also be specified by using two dots:

..SIN means all modules up to and including SIN.

SIN.. means all modules from SIN to the end of the file.

SIN..COS means SIN and COS and all the modules in between.

Ranges of modules and relative offsets may also be used in combination:

```
SIN+1..COS-1
```

To select a given module from a file, use the name of the file followed by the module(s) specified enclosed in angle brackets and separated by commas:

```
FORLIB <SIN..COS>

 or

MYLIB.REL <TEST>

 or

BIGLIB.REL <FIRST,MIDDLE,LAST>

 etc.
```

If no modules are selected from a file, then <u>all</u>

the modules in the file are selected:

```
TESTLIB.REL
```

3.2 LIB-80 Switches

A number of switches are used to control LIB-80 operation. These switches are always preceded by a slash:

/O	Octal - set Octal typeout mode for /L command.
/H	Hex - set Hex typeout mode for /L command (default).
/U	List the symbols which would remain undefined on a search through the file specified.
/L	List the modules in the files specified and symbol definitions they contain.
/C	(Create) Throw away the library under construction and start over.
/E	Exit to CP/M. The library under construction (.LIB) is revised to .REL and any previous copy is deleted.
/R	Rename - same as /E but does not exit to CP/M on completion.

3.3 LIB-80 Listings

To list the contents of a file in cross reference format, use /L:

```
*FORLIB/L
```

When building libraries, it is important to order the modules such that any intermodule references are "forward." That is, the module containing the global reference should physically appear ahead of the module containing the entry point. Otherwise, LINK-80 may not satisfy all global references on a single pass through the library.

Use /U to list the symbols which could be undefined in a single pass through a library. If a module in the library makes a backward reference to a symbol in another module, /U will list that symbol. Example:

```
        *SYSLIB/U
```

NOTE: Since certain modules in the standard FORTRAN and COBOL systems are always force-loaded, they will be listed as undefined by /U but will not cause a problem when loading FORTRAN or COBOL programs.

Listings are currently always sent to the terminal; use control-P to send the listing to the printer.

3.4 Sample LIB Session

```
        A>LIB

        *TRANLIB=SIN,COS,TAN,ATAN,ALOG
        *EXP
        *TRANLIB.LIB/U
        *TRANLIB.LIB/L
           .
           .
           .
        (List of symbols in TRANLIB.LIB)
           .
           .
           .
        */E
        A>
```

3.5 Summary of Switches and Syntax

```
/O  Octal - set listing radix
/H  Hex - set listing radix
/U  List undefineds
/L  List cross reference
/C  Create - start LIB over
/E  Exit - Rename .LIB to .REL and exit
/R  Rename - Rename .LIB to .REL
```

module::=module name {+ or - number}

module sequence ::=

module | ..module | module.. | module1..module2

file specification::=filename {<module sequence> {,<module sequence>}}

command::= {library filename=} {list of file specifications}
    {list of switches}

SECTION 4

Operating Systems

This section describes the use of MACRO-80 and LINK-80 under the different disk operating systems. The examples shown in this section assume that the FORTRAN-80 compiler is in use. If you are using the COBOL-80 compiler, substitute "COBOL" wherever "F80" appears, and substitute the extension ".COB" wherever ".FOR" appears.

4.1 CPM

Create a Source File

Create a source file using the CPM editor. Filenames are up to eight characters long, with 3-character extensions. FORTRAN-80 source filenames should have the extension FOR, COBOL-80 source filenames should have the extension COB, and MACRO-80 source filenames should have the extension MAC.

Compile the Source File

Before attempting to compile the program and produce object code for the first time, it is advisable to do a simple syntax check. Removing syntax errors will eliminate the necessity of recompiling later. To perform the syntax check on a source file called MAX1.FOR, type

```
A>F80 ,=MAX1
```

This command compiles the source file MAX1.FOR without producing an object or listing file. If necessary, return to the editor and correct any syntax errors.

To compile the source file and produce an object and listing file, type

```
A>F80 MAX1,MAX1=MAX1
```

or

```
A>F80 =MAX1/L
```

The compiler will create a REL (relocatable) file called MAX1.REL and a listing file called MAX1.PRN.

Loading, Executing and Saving the Program (Using LINK-80)

To load the program into memory and execute it, type

```
A>L80 MAX1/G
```

To exit LINK-80 and save the memory image (object code), type

```
A>L80 MAX1/E,MAX1/N
```

When LINK-80 exits, three numbers will be printed: the starting address for execution of the program, the end address of the program and the number of 256-byte pages used. For example

```
[210C 401A 48]
```

If you wish to use the CPM SAVE command to save a memory image, the number of pages used is the argument for SAVE. For example

```
A>SAVE 48 MAX1.COM
```

NOTE

CP/M always saves memory starting at 100H and jumps to 100H to begin execution. Do not use /P or /D to set the origin of the program or data area to 100H, unless program execution will actually begin at 100H.

An object code file has now been saved on the disk under the name specified with /N or SAVE (in this case MAX1). To execute the program simply type the program name

```
A>MAX1
```

<u>CPM - Available Devices</u>

A:, B:, C:, D:	disk drives
HSR:	high speed reader
LST:	line printer
TTY:	Teletype or CRT

<u>CPM Disk Filename Standard Extensions</u>

FOR	FORTRAN-80 source file
COB	COBOL-80 source file
MAC	MACRO-80 object file
REL	relocatable object file
PRN	listing file
COM	absolute file

CPM Command Lines
CPM command lines and files are supported; i.e., a COBOL-80, FORTRAN-80, MACRO-80 or LINK-80 command line may be placed in the same line with the CPM run command. For example, the command

```
A>F80 =TEST
```

causes CPM to load and run the FORTRAN-80 compiler, which then compiles the program TEST.FOR and creates the file TEST.REL. This is equivalent to the following series of commands:

```
A>F80
*=TEST
*^C
A>
```

4.2 DTC Microfile

Create a Source File
Create a source file using the DTC editor. Filenames are up to five characters long, with 1-character extensions. COBOL-80, FORTRAN-80 and MACRO-80 source filenames should have the extension T.

Compile the Source File
Before attempting to compile the program and produce object code for the first time, it is advisable to do a simple syntax check. Removing syntax errors will eliminate the necessity of recompiling later. To perform the syntax check on the source file called MAX1, type

```
*F80 ,=MAX1
```

This command compiles the source file MAX1 without producing an object or listing file. If necessary, return to the editor and correct any syntax errors.

To compile the source file MAX1 and produce an object and listing file, type

```
*F80 MAX1,MAX1=MAX1
```
or
```
*F80 =MAX1/L/R
```

The compiler will create a relocatable file called MAX1.O and a listing file called MAX1.L.

Loading, Executing and Saving the Program (Using LINK-80)
To load the program into memory and execute it,

type

```
*L80 MAX1/G
```

To save the memory image (object code), type

```
*L80 MAX1/E
```

which will exit from LINK-80, return to the DOS monitor and print three numbers: the starting addressfor execution of the program, the end address of the program, and the number of 256-byte pages used. For example

```
[210C 401A 48]
```

Use the DTC SAVE command to save a memory image. For example

```
*SA MAX1 2800 401A 2800
```

2800H (24000Q) is the load address used by the DTC Operating System.

NOTE

If a /P:<address> or /D:<address> has been included in the loader command to specify an origin other than the default (2800H), make sure the low address in the SAVE command is the same as the start address of the program.

An object code file has now been saved on the disk under the name specified in the SAVE command (in this case MAX1). To execute the program, simply type

```
*RUN MAX1
```

<u>DTC Microfile - Available Devices</u>

DO:, D1:, D2:, D3:	disk drives
TTY:	Teletype or CRT
LIN:	communications port

<u>DTC Disk Filename Standard Extensions</u>

T	COBOL-80, FORTRAN-80 or MACRO-80 source file
O	relocatable object file
L	listing file

DTC Command Lines
DTC command lines are supported as described in Section 4.1, CPM Command Lines.

4.3 Altair DOS

Create a Source File
Create a source file using the Altair DOS editor. The name of the file should have four characters, and the first character must be a letter. For example, to create a file called MAX1, initialize DOS and type

```
.EDIT MAX1
```

The editor will respond

```
CREATING FILE
00100
```

Enter the program. When you are finished entering and editing the program, exit the editor.

Compile the Source File
Load the compiler by typing

```
.F80
```

The compiler will return the prompt character "*".

Before attempting to compile the program and produce object code for the first time, it is advisable to do a simple syntax check. Removing syntax errors will eliminate the necessity of recompiling later. To perform the syntax check on the source file called MAX1, type

```
*,=&MAX1.
```

(The editor stored the program as &MAX1) Typing ,=&MAX1. compiles the source file MAX1 without producing an object or listing file. If necessary, return to the editor and correct any syntax errors.

To compile the source file MAX1 and produce an object and listing file, type

```
*MAX1R,&MAX1=&MAX1.
```

The compiler will create a REL (relocatable) file called MAX1RREL and a listing file called &MAX1LST. The REL filename must be entered as five characters instead of four, so it is convenient to use the source filename plus R.

After the source file has been compiled and a prompt has been printed, exit the compiler. If the computer uses interrupts with the terminal, type Control C. If not, actuate the RESET switch on the computer front panel. Either action will return control to the monitor.

Using LINK-80

Load LINK-80 by typing

```
.L80
```

LINK-80 will respond with a "*" prompt. Load the program into memory by entering the name of the program REL file

```
*MAX1R
```

Executing and Saving the Program

Now you are ready to either execute the program that is in memory or save a memory image (object code) of the program on disk. To execute the program, type

```
*/G
```

To save the memory image (object code), type

```
*/E
```

which will exit from LINK-80, return to the DOS monitor and print three numbers: the starting address for execution of the program, the end address of the program, and the number of 256-byte pages used. For example

```
[26301 44054 35]
```

Use the DOS SAVE command to save a memory image. Type

```
.SAV MAX1 0 17100 44054 26301
```

17100 is the load address used by Altair DOS for the floppy disk. (With the hard disk, use 44000.)

An object code file h[s now been saved on the disk under the name specified in the SAVE command (in this case MAX1). To execute the program, simply type the program name

```
.MAX1
```

Altair DOS - Available Devices

FO:, F1:, F2:, ...	disk drives
TTY:	Teletype or CRT

Altair DOS Disk Filename Standard Extensions

FOR	FORTRAN-80 source file
COB	COBOL-80 source file
MAC	MACRO-80 source file
REL	relocatable object file
LST	listing file

Command Lines
Command lines are not supported by Altair DOS.

4.4 ISIS-II

Create a Source File
Create a source file using the ISIS-II editor. Filenames are up to six characters long, with 3-character extensions. FORTRAN-80 source filenames should have the extension FOR and COBOL-80 source filenames should have the extension COB. MACRO-80 source filenames should have the extension MAC.

Compile the Source File
Before attempting to compile the program and produce object code for the first time, it is advisable to do a simple syntax check. Removing syntax errors will eliminate the necessity of recompiling later. To perform the syntax check on the source file called MAX1.FOR, type

```
-F80 ,=MAX1
```

This command compiles the source file MAX1.FOR without producing an object or listing file. If necessary, return to the editor and correct any syntax errors.

To compile the source file MAX1.FOR and produce an object and listing file, type

```
-F80 MAX1,MAX1=MAX1
```

or

```
-F80 =MAX1/L/R
```

The compiler will create a REL (relocatable) file called MAX1.REL and a listing file called MAX1.LST.

Loading, Saving and Executing the Program (Using LINK-80)

To load the program into memory and execute it, type

```
-L80 MAX1/G
```

To save the memory image (object code), type

```
-L80 MAX1/E,MAX1/N
```

which will exit from LINK-80, return to the ISIS-II monitor and print three numbers: the starting address for execution of the program, the end address of the program, and the number of 256-byte pages used. For example

```
[210C 401A 48]
```

An object code file has now been saved on the disk under the name specified with /N (in this case MAX1).

ISIS-II - Available Devices

:FO:, :F1:, :F2:, ...	disk drives
TTY:	Teletype or CRT
LST:	line printer

ISIS-II Disk Filename Standard Extensions

FOR	FORTRAN-80 source file
COB	COBOL-80 source file
MAC	MACRO-80 source file
REL	relocatable object file
LST	listing file

ISIS-II Command Lines

ISIS-II command lines are supported as described in Section 4.1, CPM Command Lines.

Index

Microsoft Utility Software
ADDENDUM TO: Section 4, Operating Systems
10/78

FORTRAN-80 under TEKDOS

FORTRAN-80 and MACRO-80

The FORTRAN-80 compiler and MACRO-80 assembler accept commands of the same format as TEKDOS assembler commands; i.e., three filename or device name parameters plus optional switches.

```
F80 [object-output][list-output]{source-input}[sw1][sw2]...
```

The object and listing file parameters are optional. These files will not be created if the parameters are omitted, however any error messages will still be displayed on the console. The available switches are as described in the FORTRAN-80 User's Manual and Microsoft Utility Software Manual, except that the switches are delimited by commas or blanks instead of slashes.

LINK-80

The LINK-80 loader accepts interactive commands only. When LINK-80 is invoked, and whenever it is waiting for input, it will prompt with an asterisk. Commands are lists of filenames and/or devices separated by commas and optionally interspersed with switches. The input to LINK-80 must be Microsoft relocatable object code (not the same as TEKDOS loader format).

Switches to LINK-80 are delimited by hyphens under TEKDOS, instead of slashes. All LINK-80 switches (as documented in the Microsoft Utility Software Manual) are supported, except "G" and "N", which are not implemented at this time.

Examples:

1. Compile a Fortran program named FTEST, creating an object file called FREL and a listing file called FLST:

```
>F80 FREL FLST FTEST
```

2. Load FTEST, link in the required library routines, and save the loaded module:

```
>L80
*FREL-E
[04AD  22B8]
*DOS*ERROR 46
L80 TERMINATED
>M FMOD 400 22B8 04AD
```

Note that "-E" exits via an error message due to execution of a Halt instruction. The memory image is intact, however, and the "Module" command may be used to save it. Once a program is saved in module format, it may then be executed directly without going through LINK-80 again. "-E" searches the system library (FORLBREL), if necessary, before exiting.

The bracketed numbers printed by LINK-80 before exiting are the entry point address and the highest address loaded, respectively. The loader default is to begin loading at 400H. However, the loader also places a jump to the start address in location 0, thereby allowing execution to begin at 0.

The memory locations between 0003 and 0400H are reserved for SRB's and I/O buffers at runtime. If you wish to load a program below 400H, then the I/O drivers should be altered. The modules that must potentially be modified for custom I/O are:

DSKDRV, TEKIO, INIT, LUNTB, IOINIT, EXIT

These source modules are provided on the standard distribution disks and may be modified and assembled using MACRO-80. If the modified I/O routines are then force-loaded before the library search, the standard library routines will not be loaded.

Disk I/O and LUN Assignments

(See FORTRAN-80 Reference Manual, Section 8.3.)

Logical units 1-4 are assigned to the console and may be used for either input or output.

Logical units 5-10 go through DSKDRV. They default to sequential disk files with the names

FOR05DAT,...,FOR10DAT.

These units may be re-assigned to any filename or device using an OPEN call. The form of an OPEN call is:

```
     CALL OPEN(LUN, filename)
```

where LUN is a logical unit number (Integer variable or constant between 5 and 10), and filename is a Hollerith or Literal constant or variable containing the ASCII filename and/or device. The filename cannot have more than 11 characters, and it must be terminated by a blank or null character.

Examples:

```
     CALL OPEN(8,'TSTFIL/1 ')
```

opens TSTFIL on drive 1 and associates it with LUN8.

```
     CALL OPEN(5,'REMO ')
```

opens LUN5 for device REMO.

ADDENDUM
FORTRAN-80 under TEKDOS

CREF80

The form of commands to CREF80 is:

```
     C80 {list-output}{cref-input}[sw1][sw2]...
```

Both filename parameters are required; switches are optional.

Example:

Create a CREF file using MACRO-80:

```
     M80 ,, TSTCRF TSTMAC C
```

Create a cross reference listing from the CREF file:

```
     C80 TSTLST TSTCRF
```