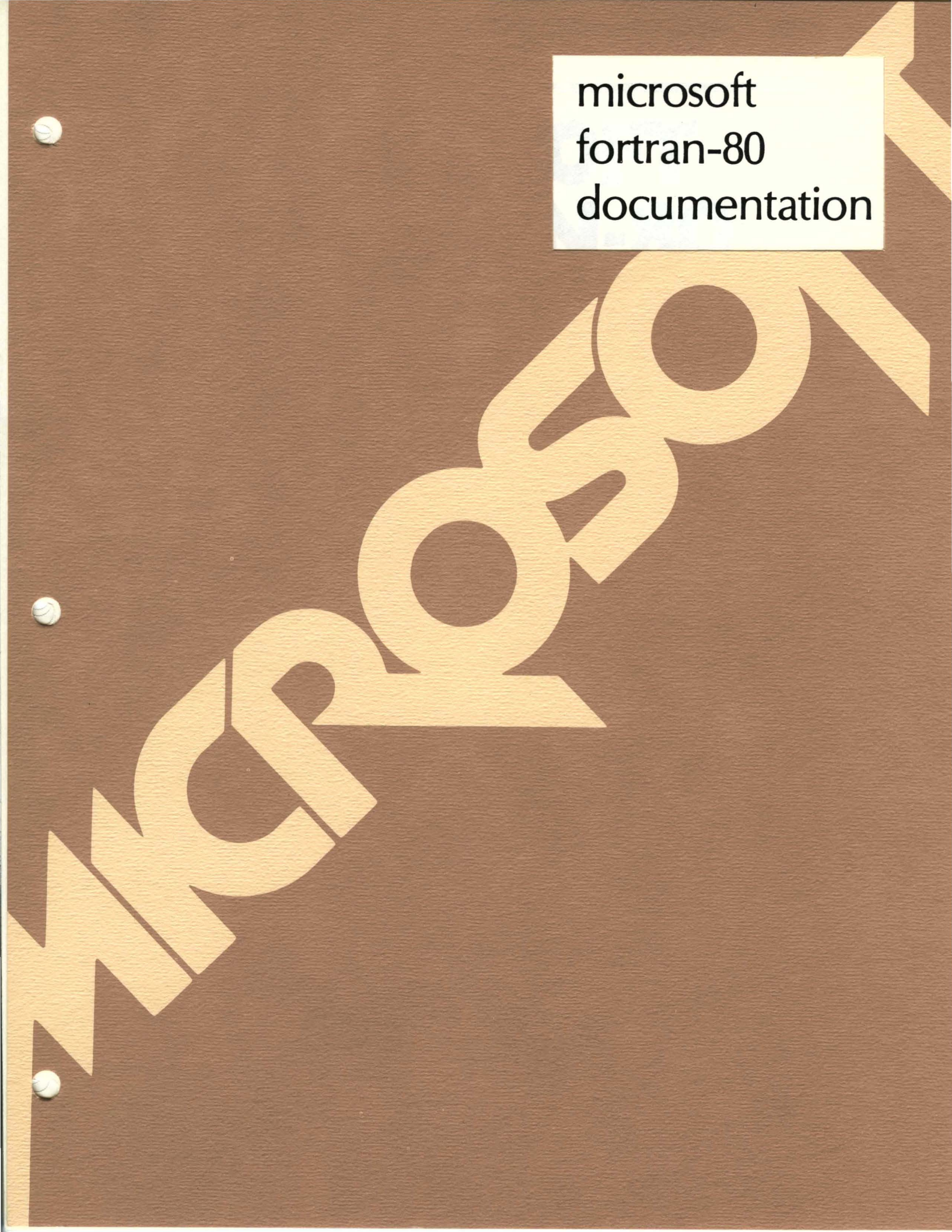# microsoft
fortran-80
documentation

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.

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft. The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of the agreement.

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.

MICROSOFT
FORTRAN-80

THE MOST COMPLETE IMPLEMENTATION OF THE FORTRAN LANGUAGE FOR 8080 AND Z80 MICROPROCESSORS.

MICROSOFT

FORTRAN-80

Microsoft FORTRAN-80 brings the most popular science and engineering programming language to 8080/Z80 microcomputers. FORTRAN-80 is comparable to FORTRAN compilers on large mainframes and mini-computers. The FORTRAN-80 package includes full ANSI Standard FORTRAN X3.9-1966 except the COMPLEX data type.

FORTRAN-80 ENHANCES THE 1966 ANSI STANDARD IN SEVERAL WAYS:

■ Single byte LOGICAL variables which can be used as integer quantities in the range +127 to −127.

■ DO loops which use LOGICAL variables for tighter, faster execution of small loops.

■ Mixed mode arithmetic expressions.

■ Hexadecimal constants.

■ Hollerith (character) literals accepted.

■ Logical operations on integer data. . AND., . OR., .NOT., .XOR. can be used for 8-bit, 16-bit, or 32-bit Boolean operations.

■ READ/WRITE End of File or Error Condition transfer. END=n and ERR=n (where n is the statement number) can be included in READ or WRITE statements to transfer control to the statement specified by n when an error or file end is detected.

■ ENCODE/DECODE for FORMAT operations to memory.

■ IMPLICIT statement for redefining default variable types by specifying a type and a range of initial letters.

■ INCLUDE statement for including commonly used subroutines, code, or declarations from another file.

■ INTEGER*4 variables and constants using 32 bits in the range of +2,147,483,647 to −2,147,483,648.

■ Support for CP/M version 2.x providing access to a maximum of 65,536 records in a file as large as 8 megabytes.

FORTRAN-80 COMPILER

FORTRAN-80 compiles several hundred statements per minute in a single pass. It requires no more than 27K bytes of memory to compile most programs. Additional memory, when available, is used for symbol table storage and optimizations. Compiled programs are relocatable modules that are linked and loaded at runtime.

The FORTRAN-80 compiler optimizes the generated object code as follows:

Common Subexpression Elimination.
Common subexpressions are evaluated once; the value is automatically substituted in subsequent occurrences of the subexpression.

Peephole Optimization.
In special cases, small sections of code can be replaced by more compact code. Example: I = I + 1 uses an INX H instruction instead of a DAD.

Constant Folding.
Integer constant expressions are evaluated at compile time.

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

The compiler also provides diagnostic output. Descriptive error messages include the preceding twenty characters. At program's end, the compiler generates an error summary. A fully symbolic listing of the generated machine language is also produced. This is supplemented by tables of addresses assigned to labels, variables and constants.

SUBROUTINE LIBRARY

FORTRAN-80 supplies an extensive library of efficient subroutines. Only the necessary subroutines are loaded at runtime. The LIB-80 standard library includes:

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

The library also contains efficient routines for 16-bit and 32-bit integer arithmetic and 32-bit and 64-bit floating point arithmetic.

ASSEMBLER AND LINKER

The FORTRAN-80 package includes the MACRO-80 relocating macro assembler and LINK-80 relocating linking loader.

MACRO-80 relocating assembler resides in approximately 19K. It 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, conditional listing control, comment blocks, octal or hex listings and a variable input radix. The assembler accepts both Intel 8080 and Zilog Z80 mnemonics.

LINK-80 relocating loader resolves external references between object modules. LINK-80 performs library searches for system subroutines. It also generates a memory load map displaying the locations of the main program, subroutines and COMMON areas. LINK-80 requires approximately 10K bytes of memory.

CUSTOM I/O DRIVERS

Users may write non-standard I/O drivers for each Logical Unit Number, so interfacing non-standard devices to FORTRAN programs is straightforward.

THE FORTRAN-80 SYSTEM

Versions of FORTRAN-80 are available for various operating systems including Digital Research CP/M®, Tektronix TEKDOS, Intel™ ISIS II, Radio Shack TRSDOS™ Model I and Model II, GenRad RDOS and many others. Microsoft welcomes the opportunity to adapt FORTRAN-80 to OEM systems.

Updates to FORTRAN-80 are announced periodically and are available for a minimal charge.

FORTRAN-80 system (including documentation): $500.00

FORTRAN-80 documentation only: $ 20.00

OEM and dealer agreements are available upon request.

CP/M is a registered trademark of Digital Research
Intel is a trademark of the Intel Corporation
TRSDOS is a trademark of Radio Shack

Addendum to: FORTRAN-80 Reference Manual

Random Number Generator

A new function, RAN, has been added to the FORTRAN
library routines listed in Table 9-2. RAN is a
random number generator that is compatible with
Microsoft's   BASIC   Compiler   and   BASIC-80
Interpreter. The random number generated is a REAL
decimal number between 0 and 1.

The random number generator is called by a
statement of the following form:

      <variable> = RAN(x)

If x < 0, the first value of a new sequence of
random numbers is returned.

If X = 0, the last random number generated is
returned again.

If x > 0, the next random number in the sequence is
generated.

ADDENDUM TO: FORTRAN-80   Reference Manual
               INTEGER*4 Variables

There are the following restrictions on the use of the new
INTEGER*4 extended integer data type:

   1.   INTEGER*4 variables cannot be used as Logical Unit
        Numbers in READ or WRITE statements.

   2.   INTEGER*4 variables cannot be used   as   array
        indices.

   3.   INTEGER*4 variables cannot be used as indices in
        implied DO loops.

   4.   INTEGER*4 variables cannot be used in computed or
        assigned GOTO statements.

fortran-80
user's
manual
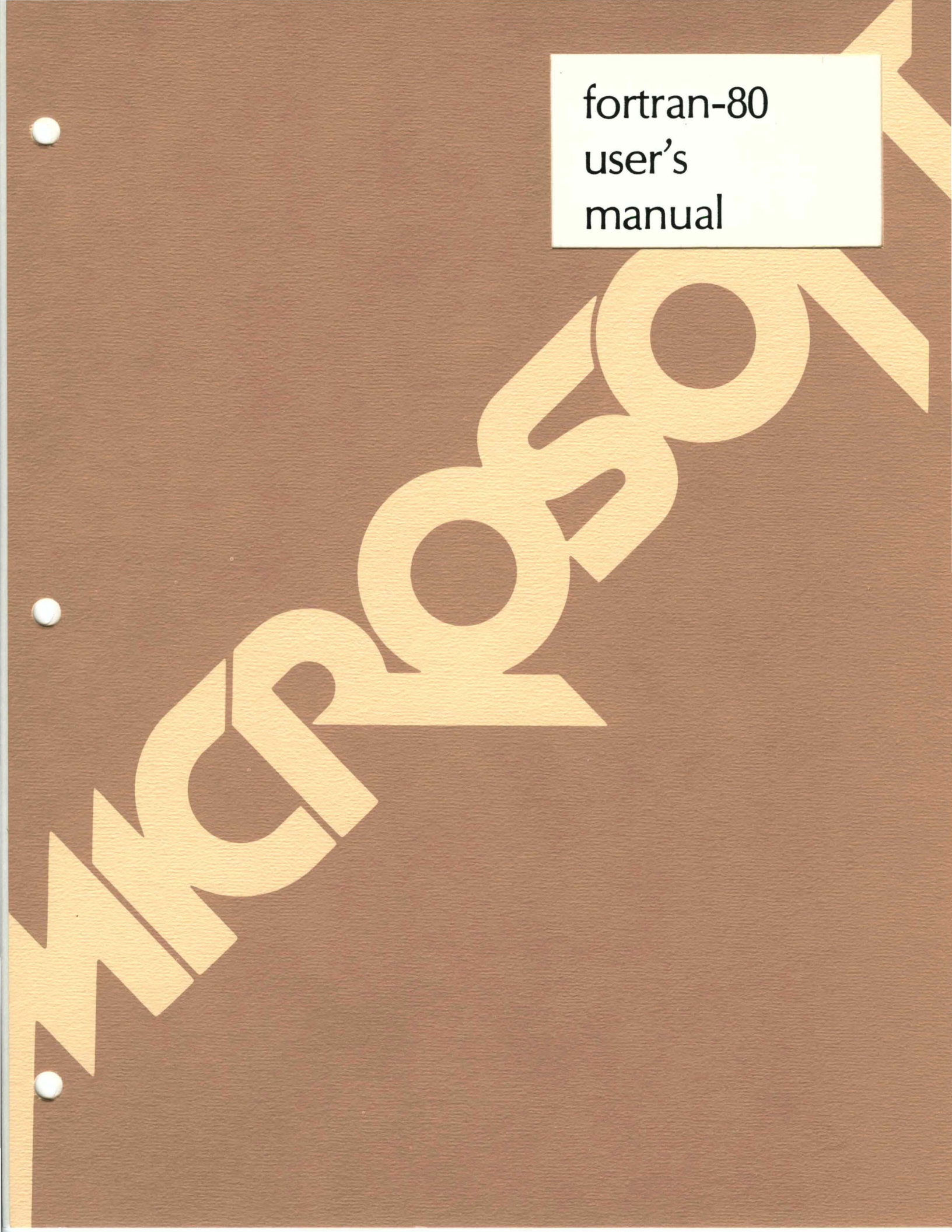
fortran-80
user's
manual

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft. The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of the agreement.

8202-342-02

Addendum to: FORTRAN-80 User's Manual
               (for CP/M Operating Systems)

<u>Add</u> <u>to</u> <u>Section</u> <u>1.2,</u> <u>Command</u> <u>Format</u>

The FORTRAN-80 Compiler now accepts lower case filenames and
device codes at the asterik prompt level.

<u>Add</u> <u>to</u> <u>Section</u> <u>2.2,</u> <u>FORTRAN</u> <u>Runtime</u> <u>Error</u> <u>Messages</u>

The fatal run-time error **FN** has been removed.   This
error occured when the program attempted to read a
non-existant file.   The new DSKDRV.MAC now allows the
programmer to use the "ERR=" switch in the READ statement to
trap this error. If the "ERR=" switch is not used in the
READ statement, and a read is attempted on a non existent
file, a fatal error, **IO**, will occur.

ADDENDUM TO: Microsoft FORTRAN-80 User's Manual
                Section 4
                TEKDOS Release 3.1

If you are using TEKDOS version 3.1 and wish to have line
feeds inserted in compiler print listings that are sent
directly to the printer, make the following patch in the
FORTRAN version 3.4 compiler:

        Using the TEKDOS DEBUG program, insert 00 hex (NOP)
in locations 2A95 hex, 2A96 hex, and 2A97 hex.

This patch will enable users of printers without an
auto-linefeed function to print compiled listings properly.

Microsoft
FORTRAN-80 User's Manual

CONTENTS

SECTION 1

Compiling FORTRAN Programs

1.1   Running the FORTRAN-80 Compiler

The command to run FORTRAN-80 is

F80

FORTRAN-80 returns the prompt "*", indicating it is
ready to accept commands.

NOTE

If you are using the TEKDOS operating
system, see Section 4 for proper command
formats.

1.2   Command Format

A command to FORTRAN-80 consists of a string of
filenames with optional switches. All filenames
should follow the operating system's conventions
for filenames and extensions. The default
extensions supplied by Microsoft software are as
follows:

File	CP/M	ISIS-II
FORTRAN source file	FOR	FOR
COBOL source	COB	COB
MACRO-80 source file	MAC	MAC
Relocatable object file	REL	REL
Listing file	PRN	LST
Absolute file	COM	

A command to FORTRAN-80 conveys the name of the
source file to be compiled, the names of the
file(s) to be created, and which options are
desired. The format of an FORTRAN-80 command is:

objfile,lstfile=source file

Only the equal sign and the source file field are
required to create a relocatable object file with
the default (source) filename and the default
extension REL.

Otherwise, an object file is created only if the

objfile field is filled, and a listing file is created only if the lstfile field is filled.

To compile the source file without producing an object file or listing file, place only a comma to the left of the equal sign. This is a handy procedure that lets you check for syntax errors before compiling to an object file.

Examples:

```
*=TEST              Compile the program TEST.FOR
                    and place the object file
                    in TEST.REL

*TESTOBJ=TEST.FOR   Compile the program TEST.FOR
                    and place the object file
                    in TESTOBJ.REL

*TEST,TEST=TEST     Compile TEST.FOR, placing the
                    object file in TEST.REL and
                    the listing file in TEST.PRN.
                    (With ISIS-II, the listing file
                    is TEST.LST.)

*,=TEST.FOR         Compile TEST.FOR without creating
                    an object or listing file. Useful
                    for error check.
```

1.2.1   Devices

Any field in the FORTRAN-80 command string can also specify a device name. The default device name with the CP/M operating system is the currently logged disk. The default device name with the ISIS-II operating system is disk drive 0. The command format is:

dev:objfile,dev:lstfile=dev:source file

The device names are as follows:

Device	CP/M	ISIS-II
Disk drives	A:, B:, C:,...	:F0:, :F1:, :F2:, ...
Line printer	LST:	LST:
Console	TTY:	TTY:
High speed reader	HSR	

Examples:

*,TTY:=TEST   Compile the source file TEST.FOR
                         and list the program on the
                         console. No object code is
                         generated. Useful for error check.

*SMALL,TTY:=B:TEST   Compile TEST.FOR (found
                         on disk drive B), place
                         the object file in SMALL.REL,
                         and list the program on the console.

1.2.2   FORTRAN-80 Compilation Switches

A switch is a letter that is appended to the
command string, preceded by a slash. It specifies
an optional task to be performed during
compilation. More than one switch can be used, but
each must be preceded by a slash. (With the TEKDOS
operating system, switches are preceded by commas
or spaces. See Section 4.)
All switches are optional.

The available switches are:

Switch   Action

O   Print all listing addresses, etc.
                         in octal.

H   Print all listing addresses, etc.
                         in hexadecimal. (Default)

N   Do not list generated code.

R   Force generation of an object file.

L   Force generation of a listing file.

P   Each /P allocates an extra 100
                         bytes of stack space for use during
                         compilation. Use /P if stack
                         overflow   errors   occur   during
                         compilation. Otherwise not needed.

M   Specifies to the compiler that the
                generated code should be in a form
                which can be loaded into ROMs.
                When a /M is specified, the
                generated code will differ from
                normal in the following ways:
                1. FORMATs will be placed in the
                program area, with a "JMP" around
                them.
                2. Parameter blocks (for
                subprogram 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 to create
                object file TEST.REL and
                listing file TEST.PRN. (With
                ISIS-II, the listing file
                is TEST.LST.)

*=BIGGONE/P/P   Compile file BIGGONE.FOR
                and produce object file
                BIGGONE.REL. Compiler is
                allocated 200 extra bytes
                of stack space.

1.3   Programs in ROM

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.

1.4   Sample Compilation

A>F80

*EXAMPL,TTY:=EXAMPL

```
FORTRAN-80 Ver. 3.3 Copyright 1979 (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 2.9 of the Microsoft Utility Software Manual for
a listing of the MACRO-80 subroutine CSL3.

SECTION 2

Error Messages

2.1   <u>FORTRAN</u> <u>Compiler</u> <u>Error</u> <u>Messages</u>

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
and the last 20 characters scanned at the time the
error was detected.

Example:

?Line 25: Mismatched Parentheses: I=(I+J))
%Line 16: Missing Integer Variable: I(2*2,

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
136    Wrong Number of Subscripts
137    File Not Found
```

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
```

2.2   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
FW   FORMAT Field Width is Too Small

Fatal Errors:

ID   Illegal FORMAT Descriptor
F0   FORMAT Field Width is Zero
MP   Missing Period in FORMAT
IR   Real Number Written to INTEGER Format Field
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)
S   QIllegal 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 at address XXXX**

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

NOTE

It is possible, in rare cases, to get a
FORTRAN-80 internal error, as designated by
the error code "??". This indicates an
internal malfunction of the runtime. If
you get the "??" error code, contact
Microsoft and report the conditions under
which the message appeared.

SECTION 3

FORTRAN-80 Disk Files

NOTE

If you are using the TEKDOS
operating system, see Section
4.2, Disk I/O and LUN
Assignments.

3.1   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.
For CP/M and ISIS-II, the default filenames are:

    FORT06.DAT, FORT07.DAT,..., FORT10.DAT

(For the default filenames used with TEKDOS, see
Section 4.)

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

3.2   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 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 eleven 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 A, 2 is drive 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). The Filename must be terminated by a non-alphanumeric character.

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 ')

3.3   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).

SECTION 4

FORTRAN-80 with TEKDOS

4.1   Command Format

The command format for FORTRAN-80 differs slightly
under the TEKDOS operating system.

The FORTRAN-80 compiler accepts command lines only.
A prompt is not displayed and interactive commands
are not accepted. Commands have the same format as
TEKDOS assembler commands; i.e., three filename or
device name parameters plus optional switches.

F80 [objfile] [lstfile] sourcefile [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 Section
1 of this manual, except that the switches are
delimited by commas or spaces instead of slashes.

4.2   Disk I/O and LUN Assignments

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

FORTRAN-80 under TEKDOS does not support random
access disk files. Only sequential files are
supported.

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.
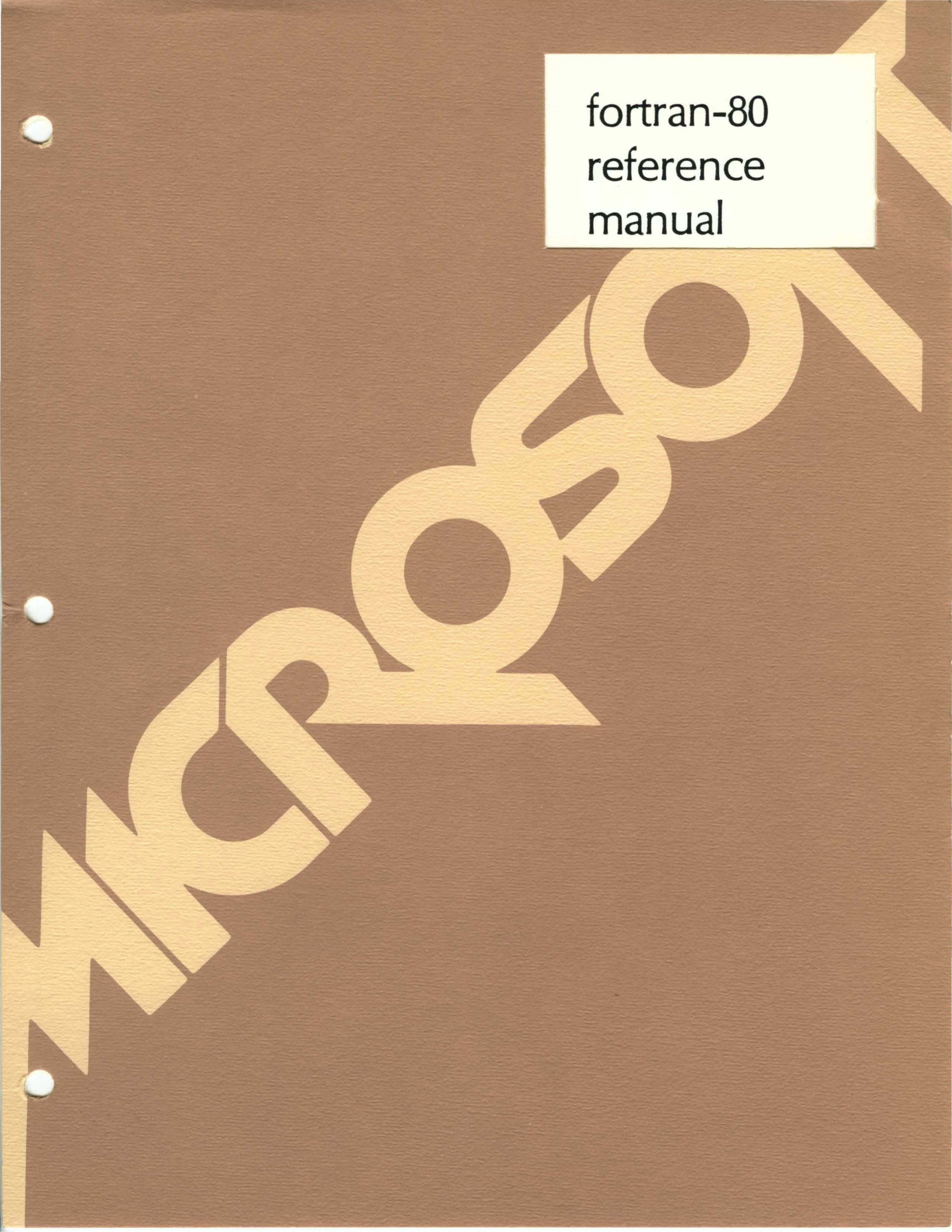
Examples:

        CALL OPEN(8,'TSTFIL/1 ')

        opens TSTFIL on drive 1 and associates it
        with LUN8.

        CALL OPEN(5,'REMO ')

        opens LUN5 for device REMO.

fortran-80
reference
manual

fortran-80
reference
manual

8201-342-03

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 FORTRAN-80 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

FORTRAN-80 source programs consist of one program unit called the <u>Main program</u> and any number of program units called <u>subprograms</u>. 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   <u>FORTRAN CHARACTER SET</u>

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

2.1.1   <u>LETTERS</u>

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,$

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   <u>DIGITS</u>

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

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   <u>ALPHANUMERICS</u>

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

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

A FORTRAN program line consists of up to 80 columns, 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. 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 be 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)
      ```

2. 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)))
```

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

   1. Column 1 contains the letter C.

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

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

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

      Example:

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

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

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

   2. Column 6 must be blank.

   3. Columns 7-72 contain the END statement.

   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.

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.

2.4   INCLUDE STATEMENT

The INCLUDE statement causes the compiler to bring an outside FORTRAN source program into the current program. The format of the statement is

     INCLUDE<filename>

Use of INCLUDE eliminates the need to repeat an often-used sequence of statements in the current source file.

SECTION 3

DATA REPRESENTATION / STORAGE FORMAT

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

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.

6. Extended Integer -- INTEGER*4 is an extended precision representation using 32-bit two's complement (four 8-bit bytes) for 9+ significant digits in the range -2147483648 to +2147483647 inclusive (-2**31 to 2**31-1).

Integer*4 variables cannot be used as Logical Unit Numbers, array indices, implied DO loop indices, or as the control variable in computed or assigned GOTO statements.

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 but spaces are permitted in the source program.	- 32 767
	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.	

TYPE	FORMATS AND RULES OF USE	EXAMPLES

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 rep-
   resented.

DOUBLE   A decimal number with   +345.678

PRECISION   precision to 16 digits. All   +.3D3
                formats and rules are identi-   -73D4
                cal to those for REAL con-
                stants, except D is used in
                place of E. Note that a real
                constant is assumed single pre-
                cision unless it contains a
                "D" exponent.

LOGICAL   .TRUE. generates a non-zero   .TRUE.
                byte (hexadecimal FF) and   .FALSE.
                .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.

TYPE	FORMATS AND RULES OF USE	EXAMPLES

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 digits (0-9 and A-F) and a single quote is recognized as a hexadecimal value.

Z'12'

X'AB1F'
Z'FFFF'

X'1F'

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

INTEGER*4

1. One to ten decimal digits interpreted as a decimal number.

1234567890
0

2. A preceding plus or minus sign is optional.

-2147483647

3. No deciml point or comma is allowed, but spaces are permitted in the source program.

- 2 147 483 647

4. Value range:
-2147483648 through 2147483647
(i.e., -2**31 through 2**31-1).

3.3   <u>VARIABLES</u>

Variable data are identified in FORTRAN statements by <u>symbolic names.</u> 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 ($). It is therefore strongly recommended that in order to avoid conflicts, symbolic names in FORTRAN source programs begin with some letter other than "$".

<u>Examples:</u>

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.

    <u>Integer Variables</u>

    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   DATA STORAGE ALLOCATION

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

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

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

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

Up to eight Hollerith characters may be associated with Double Precision type storage, up to four with Real or Integer*4, up to two with Integer*2, 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 two's complement of positive representations. The storage order is reversed. The least significant byte is followed by the most significant byte.

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 (hi)
Mantissa(mid)		Mantissa (low)

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. The format of the mantissa is "signed magnitude." The bytes are stored in

reverse order: mantissa low order, followed
by mid order, high order, and characteristic.

DOUBLE   8 bytes/ 2 storage units
PRECISION
                  The internal form of Double Precision data is
                  identical with that of Real data except
                  Double Precision uses 4 extra bytes for the
                  matissa.

INTEGER*4   4 bytes/ 1 storage unit

                  Negative numbers are represented in two's
                  complement form. The bytes are stored in
                  reverse order, least significant to most
                  significant.

TABLE 3-3.   EQUIVALENT DATA TYPES AND SIZES

Equivalent Representations   Size in Bytes

        BYTE   1
        INTEGER*1   1
        LOGICAL   1

        INTEGER   2
        INTEGER*2   2
        LOGICAL*2   2

        INTEGER*4   4
        LOGICAL*4   4

        REAL   4
        REAL*4   4

        DOUBLE PRECISION   8
        REAL*8   8

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, Extended 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*4, 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:

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

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:

    el r e2

where el and e2 are arithmetic expressions and r is a <u>relational</u> <u>operator.</u>   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.

<u>Examples:</u>

    A.EQ.B
    (A**J).GT.(ZAP*(RHO*TAU-ALPH))

4.3.2   LOGICAL OPERATORS

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

Table 4-1.   Logical Operations

.NOT.U   The value of this expression is the
                      logical complement of U (i.e., 1
                      bits become 0 and 0 bits become 1).

U.AND.V   The value of this expression is the
                      logical product of U and V (i.e.,
                      there is a 1 bit in the result only
                      where the corresponding bits in both
                      U and V are 1.

U.OR.V   The value of this expression is the
                      logical sum of U and V (i.e., there
                      is   a   1 in the result if the
                      corresponding bit in U or V is 1 or
                      if the corresponding bits in both U
                      and V are 1.

U.XOR.V   The value of this expression is the
                      exclusive OR of U and V (i.e., there
                      is a one in the result if the
                      corresponding bits in U and V are 1
                      and 0 or 0 and 1 respectively.

Examples:

If U = 01101100 and V = 11001001 , then

        .NOT.U   = 10010011
        U.AND.V = 01001000
        U.OR.V   = 11101101
        U.XOR.V = 10100101

The following are additional considerations for construction of Logical expressions:

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

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

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

Examples:

The expression

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

is evaluated as

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

The expression

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

is evaluated as

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

3. It is invalid to have two contiguous logical
    operators <u>except</u> when the second operator is
    .NOT. That is,

        .AND..NOT.

and

        .OR..NOT.

are permitted.

<u>Example:</u>

        A.AND..NOT.B   is permitted

        A.AND..OR.B   is not permitted

4.4   <u>HOLLERITH, LITERAL, AND HEXADECIMAL CONSTANTS IN
        EXPRESSIONS</u>

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	Ext Int
Integer	Y	Ya	Yb	Ya	Yg
Real	Yc	Y	Yc	Ye	Yc
Logical	Yd	Ya	Y	Ya	Yd
Double	Yc	Y	Yc	Y	Yc
Ext Int	Yf	Yh	Yb,f	Yh	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.

f.   The sign is extended through the third and fourth bytes.

g.   The variable is assigned the truncated value of the Extended Integer expression.

h.   The expression value is converted to Extended Integer and truncated to conform to the range of Extended Integer data.

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 (kl,k2)
    ui (kl,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 vl,v2,...vn

where t represents one of the terms INTEGER, INTEGER*1, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, DOUBLE PRECISION, LOGICAL, LOGICAL*1, LOGICAL*2, LOGICAL*4, or BYTE. Each v is an array declarator or a variable, array or FUNCTION name. The INTEGER*1, INTEGER*2, INTEGER*4, REAL*4, REAL*8, LOGICAL*1, LOGICAL*2, LOGICAL*4 types are allowed for readability and compatibility with other FORTRANs.

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   <u>EXTERNAL STATEMENTS</u>

EXTERNAL statements have the following form:

           EXTERNAL ul,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   <u>DIMENSION STATEMENTS</u>

A DIMENSION statement has the following form:

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

where each ui is an array declarator.

<u>Example:</u>

           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 /yl/al/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 (yl) 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 //Al,Bl/CDATA/ZOT(3,3)
         X   //T2,Z3

In this example, Al, Bl, T2 and Z3 are assigned to blank COMMON in that order. The pair of slashes preceding Al 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/ul,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, INTEGER*4, 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

10. END

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   <u>GO</u> <u>TO</u> <u>STATEMENTS</u>

7.1.1   <u>UNCONDITIONAL</u> <u>GO</u> <u>TO</u>

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   COMPUTED GO TO

          Computed GO TO statements are of the form:

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

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

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

          Example:

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

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

7.1.3   ASSIGNED GO TO

Assigned GO TO statements are of the following form:

        GO TO j,(kl,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   <u>LOGICAL</u> <u>IF</u>

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.

<u>Examples</u>:

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

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

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

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

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

7.4   DO STATEMENT

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

    1)   DO k i = m1,m2,m3

    or

    2)   DO k i = m1,m2

where k is a statement label, i is an integer or
logical variable, and m1, m2 and m3 are integer
constants or integer or logical variables.

If m3 is 1, it may be omitted as in 2) above.

The following conditions and restrictions govern
the use of DO statements:

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

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

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

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

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

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

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

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

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

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

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

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

<u>Example:</u>

The following example computes

$$\sum_{i=1}^{100} A_i \quad \text{where a is a one-dimensional array}$$

```
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:

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

```
      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 10. 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 Section 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 Integer data entries are stored in a sequential disk file and that the values have field widths of 3, 4, 2 and 5 respectively. The statements

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

    will read the file and assign the input data to the variables K, L, M and N.

    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   FORMATTED WRITE STATEMENTS

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

Two forms of the statement are available, as follows:

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

       or

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

where:

u - specifies a Logical Unit Number.

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

L1- specifies an I/O error branch.

L2- specifies an EOF branch.

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

WRITE (u,f)k is used to output the data specified in the list k to a file on logical unit u using the FORMAT statement f to specify the external representation of the data (see FORMAT statements, 8.7). 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 ('H 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.

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.

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:

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

where:

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

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

                        NOTE

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

8.6   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	Numeric Conversion
rDw.d	
rIw	
rLw	Logical Conversion
rAw	
nHh1h2...hn	Hollerith Conversion
'1l12...ln'	
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 | Internal | Output |
Descriptor	Value	(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.)

E-type Conversion

Form: Ew.d

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

E-Output

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

1. a minus sign (if negative),

2. a zero and a decimal point,

3. d decimal digits,

4. the letter E,

5. the sign of the exponent (minus or blank),

6. two exponent digits,

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

$$w > d + 7$$

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

FORMAT Descriptor	Internal Value	Output (b=blank)
E12.5	76.573	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   Internal   Output
Descriptor   Value   (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. Note
that 1X is required, as X alone will not work.

Output Examples:

FORMAT Statement   Output
                                    (b=blank)

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,L3,M3
        .
        .
        .
  12  FORMAT (2F9.4,3(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   <u>FORMAT CONTROL, LIST SPECIFICATIONS AND RECORD
         DEMARCATION</u>

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

Formatted I/O to a console or printer uses the first character of each record for carriage control. The carriage control character is 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

Formatted I/O to disk does not require the first character of each record to be a carriage control character. Records are terminated by a carriage return character (X'0D'). There are no line-feeds (X'0A') in the file unless written there explicitly.

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. <u>Example:</u>

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(al,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 are pre-defined to the
Processor and automatically typed. The functions
are listed in Tables 9-1 and 9-2. In the tables,
arguments are denoted as a1,a2,...,an, if more than
one argument is required; or as a if only one is
required. A library function is called when its
name is used in an arithmetic expression. Such a
reference takes the following form:

f(a1,a2,...an)

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

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

For the following:
b, bl, and b2 are BYTE constants or variables
i is an INTEGER constant or variable

PEEK and INP are Logical functions; POKE and OUT are subroutines. PEEK and POKE allow direct access to any memory location. PEEK(i) returns the contents of the memory location specified by i. CALL POKE(i,b) causes the contents of the memory location specified by i to be replaced by the contents of b. INP and OUT allow direct access to the I/O ports. INP(b) does an input from port b and returns the 8-bit value input. CALL OUT(bl,b2) outputs the value of b2 to the port specified by bl.

RAN is another function in the FORTRAN library. RAN is a random number generator that is compatible with Microsoft's BASIC Compiler and BASIC-80 interpreter. The random number generated is a REAL decimal number between 0 and 1. The random number generator is called by a statement of the following form:

    <variable> = RAN(x)

If x < 0, the first value of a new sequence of random numbers is returned.

If x = 0, the last random number generated is returned again.

If x > 0, the next random number in the sequence is generated.

Examples using library functions:

      Al = 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	Function
ABS	|a|	Real	Real
IABS		Integer	Integer
DABS		Double	Double
AINT	Sign of a times lar-	Real	Real
INT	gest integer <=|a|	Real	Integer
IDINT		Double	Integer
AMOD	al(mod a2)	Real	Real
MOD		Integer	Integer
AMAX0	Max(al,a2,...)	Integer	Real
AMAX1		Real	Real
MAX0		Integer	Integer
MAX1		Real	Integer
DMAX1		Double	Double
AMIN0	Min(al,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 |al|	Real	Real
ISIGN		Integer	Integer
DSIGN		Double	Double
DIM	al - Min(al,a2)	Real	Real
IDIM		Integer	Integer
SNGL		Double	Real
DBLE		Real	Double

TABLE 9-2

Basic External Functions

Name	Number of Arguments	Type Definition	Argument	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 (al/a2)	Real	Real
DATAN2	2		Double	Double
DMOD	2	al(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(al,a2,...an)

or

FUNCTION f(al,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(al,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. There
is no type conversion of arguments. There must be
at least one argument. Arguments 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(al,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   <u>RETURN</u> <u>FROM</u> <u>FUNCTION</u> <u>AND</u> <u>SUBROUTINE</u> <u>SUBPROGRAMS</u>

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

RETURN

The following rules govern the use of the RETURN statement:

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

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

3. RETURN from a SUBROUTINE subprogram is to the next executable statement in the calling program which would logically follow the CALL statement.

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

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

<u>Example:</u>

        Calling Program Unit
        .
        .
        .
            CALL SUBR(Z9,B7,R1)
        .
        .
        .

        Called Program Unit

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

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

9.11   PROCESSING ARRAYS IN SUBPROGRAMS

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

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

Calling Program Unit

DIMENSION Z1(50),Z2(25)
.
.

.
A1 = AVG(Z1,50)
.

.

.
A2 = A1-AVG(Z2,25)
.

.

.

Called Program Unit

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

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

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

          Calling Program Unit

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

          Called Program Unit

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

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

9.12   BLOCK DATA SUBPROGRAMS

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

BLOCK DATA [subprogram-name]

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

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

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

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

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

Example:

        BLOCK DATA
        LOGICAL Al
        COMMON/BETA/B(3,3)/GAM/C(4)
        COMMON/ALPHA/Al,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 Al/.TRUE./,E/-5.6/

9.13   PROGRAM CHAINING

Programs may be loaded and executed (CHAINed) by a FORTRAN program through the CALL FCHAIN facility. The general syntax is:

    CALL FCHAIN ('filename')

where filename is a valid operating-system-dependent file specification of a machine executable file. The exact syntax varies under different operating systems. Refer to the "Microsoft FORTRAN-80 User's Manual", Section 3.

RULES:

1.   'filename' must be valid according to your operating system's rules.

2.   The program CHAINed must be a "MAIN" program. That is, one having an ENTRY Point. FORTRAN, COBOL, and assembly language subroutines do not conntain a "MAIN" entry point.

3.   Parameters may not be passed to CHAINed programs.

4.   Illegal filename, Illegal drive specification, File not found, Out of memory, and Disk read errors will result in a fatal **IO** Error.

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.

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). Units 1, 3, 4, and 5 are preassigned to the console (TTY). Unit 2 is preassigned to the Line Printer. Units 6-10 are preassigned to Disk Files (see User's Manual, Section 3).

Any of these may be redefined by the user simply by changing the appropriate entries in $LUNTB. 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.

It is also possible to add LUNs to $LUNTB. If you do this, change the MAXLUN+1 byte at the label $LUNTB, and make sure you also change the value of MAXLUN in the DSKDRV.MAC module.

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)   Unformatted Read
        4)   Unformatted 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. Unformatted I/O on a TTY).

                          NOTE

              The I/O buffer has a fixed
              maximum length of 132 bytes.
              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 Unformatted I/O as an error. This is the TTY driver provided with the runtime package.

```
                              00100    ;         TTY I/O DRIVER
                              00200
0000                          00300            EXT      $IOERR,$BL,$BF,$ERR,$TTYIN,$TTYOT
0012                          00400    IRECER  EQU      022         ;INPUT RECORD TOO LONG
0000                          00500            ENTRY    $DRV3
0000     0013 '              00600    $DRV3:   DW       DRV3FR      ;FORMATTED READ
0002     0042 '              00700            DW       DRV3FW      ;FORMATTED WRITE
0004     0010 '              00800            DW       DRV3BR      ;BINARY READ
0006     0010 '              00900            DW       DRV3BW      ;BINARY WRITE
0008     000E '              01000            DW       DRV3RE      ;REWIND
000A     000E '              01100            DW       DRV3BA      ;BACKSPACE
000C     000E '              01200            DW       DRV3EN      ;ENDFILE
000E     AF                 01300    DRV3EN:  XRA      A           ;THESE OPERATIONS ARE
                              01400                                 ;NO-OPS FOR TTY
000E                         01500    DRV3RE   EQU      DRV3EN
000E                         01600    DRV3BA   EQU      DRV3EN
000F     C9                 01700            RET
0010     C3 0000 *          01800    DRV3BW:  JMP      $IOERR      ;ILLEGAL OPERATIONS
                              01900                                 ; (PRINT ERROR AND RETURN)
0010                         02000    DRV3BR   EQU      DRV3BW
0013     AF                 02100    DRV3FR:  XRA      A           ;READ
0014     32 0000 *          02200            STA      $BL         ;ZERO BUFFER LENGTH
0017     CD 0000 *          02300    DRV31:   CALL     $TTYIN      ;INPUT A CHAR
001A     E6 7F              02400            ANI      0177        ;AND OFF PARITY
001C     FE 0A              02500            CPI      10          ;IGNORE LINE FEEDS
001E     CA 0017 '          02600            JZ       DRV31
0021     F5                 02700            PUSH     PSW         ;SAVE IT
0022     2A 0015 *          02800            LHLD     $BL         ;GET CHAR POSIT IN BUFFER
0025     26 00              02900            MVI      H,0         ;ONLY 1 BYTE
0027     EB                 03000            XCHG
0028     2A 0000 *          03100            LHLD     $BF         ;GET BUFFER ADDR
002B     19                 03200            DAD      D           ;ADD OFFSET
002C     F1                 03300            POP      PSW         ;GET CHAR
002D     77                 03400            MOV      M,A         ;PUT IT IN BUFFER
002E     13                 03500            INX      D           ;INCREMENT $BL
002F     EB                 03600            XCHG
0030     22 0023 *          03700            SHLD     $BL         ;SAVE IT
0033     FE 0D              03800            CPI      015         ;CR?
0035     C8                 03900            RZ                   ;YES--DONE
0036     7D                 04000            MOV      A,L         ;$BL
0037     FE 80              04100            CPI      128         ;MAX IS DECIMAL 128
0039     DA 0017 '          04200            JC       DRV31       ;GET NEXT CHAR
003C     CD 0000 *          04300            CALL     $ERR
003F     12                 04400            DB       IRECER      ;INPUT RECORD TOO LONG
0040     AF                 04500            XRA      A           ;CLEAR FLAGS
0041     C9                 04600            RET
0042     3A 0031 *          04700    DRV3FW:  LDA      $BL         ;BUFFER LENGTH
0045     B7                 04800            ORA      A
```

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

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

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

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

APPENDIX C

Subprogram Linkages

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

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

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

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

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

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

1. Parameter 1 in HL.

2. Parameter 2 in DE.

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

Note that, with this scheme, the subprogram must know how
many parameters to expect in order to find them.
Conversely, the calling program is responsible for passing
the correct number of parameters. 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). Extended
integers and Reals return results in memory at $AC. Double
Precision Reals return results 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	^
009	HT	052	4	095	_
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. Referencing Arithmetic Routines

    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	$AY	Integer*4	Integer
	$A1	Integer*4	Integer*4
	$AA	Real	Integer
	$AE	Real	Integer*4
	$AB	Real	Real
	$AQ	Double	Integer
	$AV	Double	Integer*4
	$AR	Double	Real
	$AU	Double	Double
Division	$D9	Integer	Integer
	$DY	Integer*4	Integer
	$D1	Integer*4	Integer*4
	$DA	Real	Integer
	$DE	Real	Integer*4
	$DB	Real	Real
	$DQ	Double	Integer
	$DV	Double	Integer*4
	$DR	Double	Real
	$DU	Double	Double
Exponentiation	$E9	Integer	Integer
	$EY	Integer*4	Integer
	$E1	Integer*4	Integer*4
	$EA	Real	Integer
	$EE	Real	Integer*4
	$EB	Real	Real
	$EQ	Double	Integer
	$EV	Double	Integer*4
	$ER	Double	Real
	$EU	Double	Double
Multiplication	$M9	Integer	Integer
	$MY	Integer*4	Integer
	$M1	Integer*4	Integer*4
	$MA	Real	Integer
	$ME	Real	Integer*4
	$MB	Real	Real
	$MQ	Double	Integer
	$MV	Double	Integer*4
	$MR	Double	Real
	$MU	Double	Double
Subtraction	$SY	Integer*4	Integer
	$S1	Integer*4	Integer*4
	$SA	Real	Integer
	$SE	Real	Integer*4
	$SB	Real	Real
	$SQ	Double	Integer
	$SV	Double	Integer*4
	$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]

Extended Integer in $AC

Real in $AC

Double Precision REAL in $DAC

Name	Function
$CD	Integer to Integer*4
$CA	Integer to Real
$CC	Integer to Double
$C4	Integer*4 to Integer
$C5	Integer*4 to Real
$C6	Integer*4 to Logical
$C7	Integer*4 to Double
$CH	Real to Integer
$CL	Real to Integer*4
$CJ	Real to Logical
$CK	Real to Double
$CX	Double to Integer
$C0	Double to Integer*4
$CY	Double to Real
$CZ	Double to Logical

2. Referencing Intrinsic Functions

Intrinsic 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)   Initialize for an I/O transfer
$W5 (5 parameters)   to a device (WRITE)

$R2 (2 parameters)   Initialize for an I/O transfer
$R5 (5 parameters)   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

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
$I4   transfers extended integers (4 bytes)

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=,15)'
DIMENS: DW        1
NUMBER: DW        9999

        END       TEST
```

4. Loading and Storing Floating Accumulator

In the following definitions, $AC refers to the floating accumulator and $DAC refers to the DOUBLE PRECISION accumulator.

To Load Floating Accumulator:
(H,L points to value to be loaded.)

Name	Function
$L1	Loads into $AC, 4 bytes
$L3	Loads into $DAC, 8 bytes

To Store Floating Accumulator:
(H,L points to memory where value is to be stored)

Name	Function
$T1	Stores 4 bytes from $AC
$T3	Stores 8 bytes from $DAC

INDEX

**utility
software
package
reference manual**

utility
software
package
reference manual

for 8080 microprocessors

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft, Inc. The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy The Utility Software Package on cassette tape, disk, or any other medium for any purpose other than purchaser's personal use.

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.

CP/M is a registered trademark of Digital Research.
The Utility Software Package, MACRO-80, LINK-80, CREF-80, and LIB-80 are trademarks of Microsoft, Inc.

8401-343-04

Contents

Contents

CHAPTER 1

INTRODUCTION

Welcome to the world of Utility Software Package programming. During the course of this manual, we will learn what the Utility Software Package is, why you use it, and how to use it.

1.1 CONTENTS OF THE UTILITY SOFTWARE PACKAGE

One diskette with the following files:

```
M80.COM - MACRO-80 Macro Assembler program
L80.COM - LINK-80 Linking Loader program
CREF80.COM - Cross-Reference Facility
LIB.COM - Library Manager program
          (CP/M versions only)
```

One Manual

The Utility Software Package Reference Manual

IMPORTANT

Always make backup copies of your diskettes before using them.

1.2 SYSTEM REQUIREMENTS

MACRO-80 requires about 19K of memory, plus about 4K for buffers. LINK-80 requires about 14K of memory. CREF-80 requires about 6K of memory. LIB-80 requires about 5K of memory. The operating system usually requires about 6K bytes of memory. So a minimum system requirement for the Utility Software Package is 29K bytes of memory. While it is possible to run Utility Software Package programs with only one disk drive, we recommend strongly that you have two disk drives available.

1.3 WHOM IS THE UTILITY SOFTWARE PACKAGE FOR?

The Utility Software Package is a complete assembly language development system with powerful features that support advanced assembly language programming skills. This manual describes the Utility Software Package thoroughly, but the descriptions assume that the reader understands assembly language programming and has experience with an assembler.

If you have never programmed in assembly language, we suggest that you gain some experience on a simpler assembler.

Books on Assembly Language Programming

We can also recommend the following books for basic instruction in assembly language programming:

Leventhal, Lance A. 8080A/8085 Assembly Language Programming. Berkeley: Osborne/McGraw-Hill, 1978.

Leventhal, Lance A. Z80 Assembly Language Programming. Berkeley: Osborne/McGraw-Hill, 1979.

Zaks, Rodnay. Programming the Z80. Second edition. Berkeley: Sybex, 1980.

1.4 A WORD ABOUT THIS MANUAL

Organization

In front of each chapter is a contents page that expands the
entries on the contents page at the beginning of the manual.
Chapter 1 gives introductory, background, and overview
information about the Utility Software Package. Chapters
2-8 describe the use and operation of the Utility Software
Package programs. The manual concludes with several
appendices which contain some helpful reference information.

Syntax Notation

The following notation is used throughout this manual in
descriptions of command and statement syntax:

[ ]   Square brackets indicate that the enclosed entry is
        optional.

< >   Angle brackets indicate user entered data. When
        the angle brackets enclose lower case text, the
        user must type in an entry defined by the text;
        for example, <filename>. When the angle brackets
        enclose upper case text, the user must press the
        key named by the text; for example, <RETURN>.

{ }   Braces indicate that the user has a choice between
        two or more entries. At least one of the entries
        enclosed in braces must be chosen unless the
        entries are also enclosed in square brackets.

...   Ellipses indicate that an entry may be repeated as
        many times as needed or desired.

CAPS   Capital letters indicate portions of statements or
        commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks,
and equal signs, must be entered exactly as shown.

1.5 OVERVIEW

The Utility Software Package is an assembly language
programming system that parallels the design and programming
power of assemblers and related software on big computers.
Consequently, the design and use of the Utility Software
Package involves traits and methods that may be new to you.
As explained earlier, we assume that you have some
experience in assembly language programming. Your knowledge
of when and why to use particular operation codes and
pseudo-operations is the base on which you can build your
knowledge of the Utility Software Package.

One word of caution: some terms used in this manual may be
familiar to you from other sources. Be sure to notice
especially how familiar terms are used in the Utility
Software Package so that you are not confused or misled.

The Utility Software Package programming relies on two
important software programs -- an assembler and a linking
loader. To develop an assembly language program that runs
on your computer, you must use both the assembler and the
linking loader. The whole process is diagrammed on the
facing page. The numbers on the diagram correspond to the
numbers in the explanations below.

1. You create an assembly language source program using
some editor.

2. You assemble your source program using the MACRO-80
macro assembler. The result is a file that contains
intermediate object code. This intermediate code is closer
to machine code than your source code, but cannot be
executed.

3. You link and load separately assembled file(s) into a
single program file using the LINK-80 linking loader.
LINK-80 converts the file(s) of intermediate code into a
single file of true machine code which can be executed from
the operating system.

These are only the basics of the whole process. This two
step process of converting a source file to an executable
program allows you to manipulate your programs to save you
time and to extend your programs' usefulness in the
following ways:

1.

EDITOR

source file

2.

MACRO–80

assembled file

3.

LINK–80

executable file
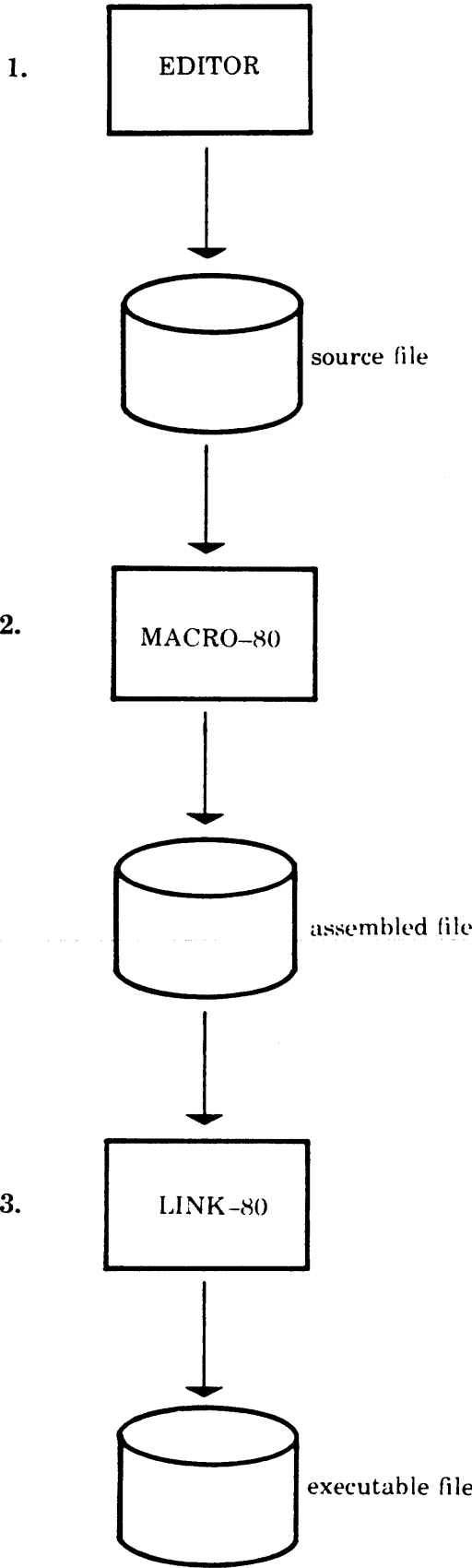
Figure 1.1:   Developing Assembly Language Programs

First, you can break your program in convenient parts called
modules. You can manipulate these modules at will. You can
assemble the modules individually. You fix only those that
do not work right and reassemble them. This saves you time.

Second, you can manipulate the placement of modules in
memory, subject to certain restrictions; or allow LINK-80
to place modules for you. (This trait is described below
under the fourth trait.)

Third, you can use assembled modules in other programs or in
variations of the original program because there is no
permanent connection among the modules. This saves you
recoding time if a part of a program performs some useful,
often-repeated task.

Whenever you want to combine assembled modules into an
executable program, you use the LINK-80 linking loader. If
you simply tell LINK-80 the modules you want combined, it
loads them end-to-end in memory. But you have an additional
choice. You can set up a direct connection between a
statement in one module and a statement inside another
module. This direct connection (or "link") means that a
value (usually a program address) in one module can be used
in another module exactly at the point required.

LINK-80 creates the links between modules. You give LINK-80
the signals it needs to create these links. The signals are
called symbols, specifically EXTERNAL symbols and PUBLIC
symbols. An EXTERNAL symbol signals LINK-80 that you want
it to link a value from another module into this point in
the program. The value to be linked-in is defined by a
PUBLIC symbol, which is a signal that directs LINK-80 to the
correct module and statement line. LINK-80 then links the
PUBLIC symbol's value to the EXTERNAL symbol, then continues
loading the module with the EXTERNAL symbol. The diagram
below suggests this process.

;loading a module with
;an EXTERNAL symbol

;here LINK-80 looks for
;the PUBLIC symbol
;and links its value
;then LINK-80
;continues to load
;the module with an
;EXTERNAL symbol
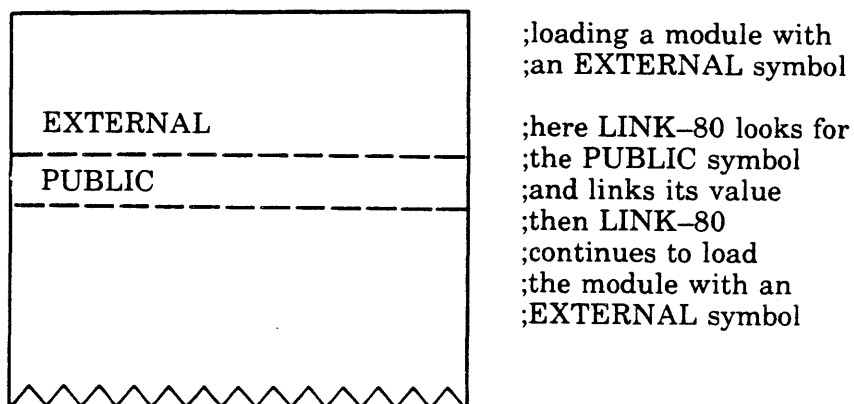
Figure 1.2: PUBLIC symbol linked into module at EXTERNAL

Fourth, modules can be assembled into different modes, even
within a single module. The four modes are Absolute,
Data-relative, Code-relative, and COMMON-relative. The
absolute mode is similar the code produced by most small
system assemblers. The code is assembled at fixed addresses
in memory. The other three modes are very different and are
the reason you can place modules anywhere in memory. Each
of the three relative modes assembles to a separate segment.
The addresses within each segment are relative addresses.
This means the first instruction byte of a segment is given
a relative address of 0, the second byte is given relative
address 1, and so on. When LINK-80 loads the module, it
changes the relative addresses in the segments to fixed
addresses in memory. The relative addresses are offsets
from some fixed address that LINK-80 uses. For the first
module loaded, this address is 103H under the CP/M operating
system. Thus, relative addresses in the first module are
offsets from 103H. The second module is loaded at the end
of the first, and the relative addresses are offsets from
the last address in the first module. Subsequent modules
are loaded (and offset) similarly. You can change the
default start address for the first module at link time.
Then, the relative addresses become offsets from the fixed
address you specify.

```
   relative                          fixed
   address                           address
      0  ┌──────────────┐            103H    ;start address
      .  │ │MOD1         │   .
         │ │             │   .
      .  │ │             │   .
      .  │ │             │   .
      .  │ │             │   .
    100  │ │             │   .        203H
         │ ─────────────── │                 ;end MOD1, begin MOD2
      0  │ MOD2           │            204H
      .  │ │             │   .
      .  │ │             │   .
      .  │ │             │   .
    250  │ │             │   .        454H
         │ ─────────────── │
                                              ;end of MOD2
      .  │ │             │   .
      .  │ │             │   .
      .  │ │             │   .
         └──────────────┘
```
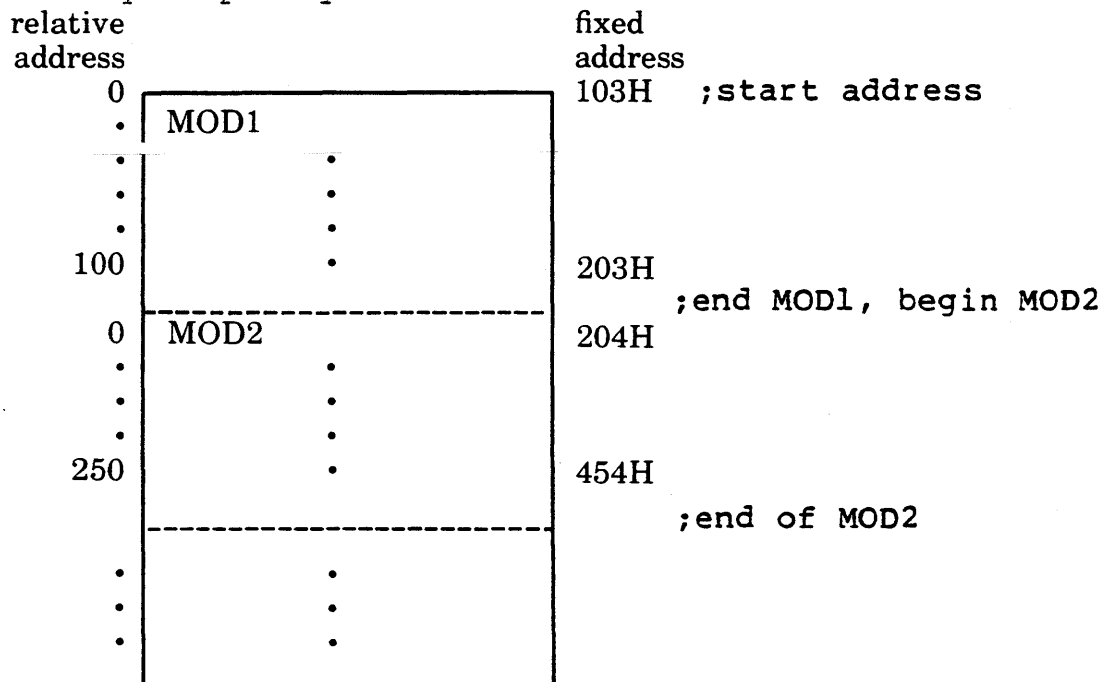
   Figure 1.3: Loading Changes Relative Addresses to Fixed

One effect of this relative addressing method is that ORG
statements become very different creatures. For the
relative segments, the ORG statement specifies an offset
rather than a fixed address (as most assemblers do -- ORG
specifies a fixed address in the absolute segment). Thus, a
relative segment with an ORG statement would skip over a
specified number of addresses before beginning to load the
rest of the code in that segment.

```
relative                                    fixed
address                                     address
   0   ┌─────────────────────────────┐      103H    ;start address
   •   │ MOD1                         │
   •   │                   •          │
   •   │                   •          │
   •   │                   •          │
   •   │                   •          │
 100   │                   •          │      203H
       ├─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─┤              ;end MOD1, begin MOD2
   0   │ MOD2                         │      204H
  50   │ ORG 50                       │      254H    ;skips 50 addresses
   •   │                   •          │
   •   │                   •          │
   •   │                   •          │
 300   │                   •          │      504H
       ├─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─┤              ;end of MOD2
   •   │                   •          │
   •   │                   •          │
   •   │                   •          │
       └─────────────────────────────┘
```

You should read carefully the description of ORG found in Chapter 4.

The ability to manipulate the placement of modules in memory, with some restrictions (see Chapter 6), derives from the assembler giving relative addresses instead of absolute addresses. This ability to manipulate module placement in memory is called relocatability; the modules are relocatable; the intermediate code produced by the assembler for the three relative segments is called relocatable code. That is why assembled modules are given the filename extension .REL, and these assembled files are called REL files.

Each mode serves a different purpose. The absolute mode contains code you want placed in specific memory addresses. Each relative mode is loaded into memory as a separate segment. The data-relative segment contains data items and any code that may change often and should only be placed in RAM. The code-relative segment contains code that will not change and therefore is suitable for ROM and PROM. The COMMON-relative segment contains data items that can be shared by more than one module.

Source statements in these modes take on the traits of their mode. The symbols and expressions in statements are evaluated by the assembler according to the mode in which they are found and the type of data and other entries that define the symbol or make up the parts of an expression. The mode traits attributed to a symbol or expression are called, appropriately, its Mode; that is, a symbol or expression is absolute, data-relative, code-relative, or COMMON-relative. This concept of mode is important because it is the source of both flexibility and complexity. If all

source statements are assembled in absolute mode, symbols
and expressions always have absolute values, and using
absolute symbols and expressions is not complex. The
problem with absolute mode is that relocatability is
possible only through the most complex and time consuming of
techniques. Absolute mode effectively reduces your ability
to reuse code in a new program.

The relative modes (data, code, and COMMON) are the basis of
relocatability and, therefore, of the flexibility to
manipulate modules. The complexity is that relative symbols
and relative expressions are much more difficult to
evaluate. In fact, the assembler must pass through the
source statements twice to assemble a module. During the
first pass, the assembler evaluates the statements and
expands macro call statemnts, calculates the amount of code
it will generate, and builds a symbol table where all
symbols and macros are assigned values. During the second
pass, the assembler fills in the symbol and expression
values from the symbol table, expands macro call statements,
and emits the intermediate code into a REL file.

When the REL files are given to LINK-80, the segments are
linked together and loaded into fixed memory addresses. The
relative addresses are converted to absolute addresses. The
fixed addresses are assigned to the relative segments in the
order: COMMON-relative and data-relative, then
code-relative. The relative segments are loaded relative to
default address 103H under CP/M. (The addresses 100H-102H
are used for a jump to the start address of the first
program instruction, which is normally the first address
following the COMMON and data area.)

When LINK-80 is finished linking modules together and
assigning addresses, the result can be saved in a file that
is executable from the operating system. Executing the
program is then as simple as entering an operating system
command, so these linked and loaded files are called command
files.

This short overview should give you a general idea of the
workings and processes of the Utility Software Package.
Short descriptions of all the Utility Software Package
programs are given in the next chapter. Detailed
descriptions are given in the rest of this manual.
Therefore, the information contained in this overview will
be repeated in fuller detail elsewhere in this manual.

As an aid to the description in the next chapter and the
rest of this manual, the next page contains an expanded
version of the diagram at the beginning of this overview.
The expanded diagram shows the relationships among all the
programs in the Utility Software Package.
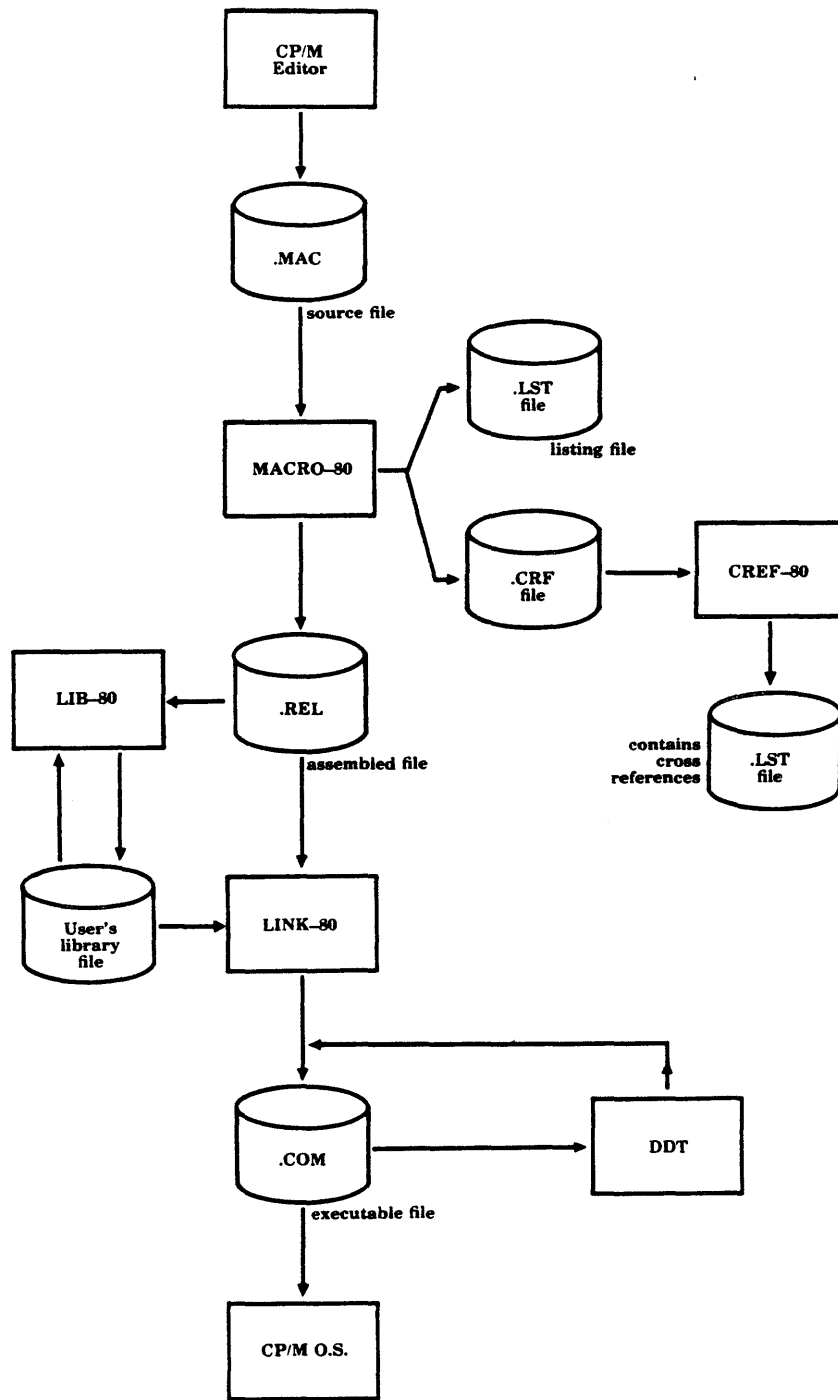
Figure 1.5:   Relationships among programs

Contents

CHAPTER 2

FEATURES OF THE UTILITY SOFTWARE PACKAGE

The Utility Software Package is an Assembly Language Development System that assembles relocatable code from two assembly languages, supports a macro facility and conditional assembly, and provides several utility programs that enhance program development.

WHAT IS AN UTILITY SOFTWARE PACKAGE?

An Utility software package is more than an assembler. An Utility Software Package is a series of related utility programming tools:

> for assembling an assembly language source file,
>
> for linking several assembled modules into one program,
>
> for creating library files of subroutines (also assembled modules),
>
> for creating cross-reference listings of program symbols,
>
> for testing and debugging binary (machine executable) program files,

Microsoft's Utility Software Package provides versions of these tools that make the Utility Software Package extremely powerful and useful as a program development system. Each tool in the Utility Software Package is described in detail in its own chapter.

2.1   TWO ASSEMBLY LANGUAGES

The assembler in your Utility Software Package supports two
assembly languages.   Microsoft's MACRO-80 macro assembler
supports both 8080 and Z80 mnemonics.

2.2   RELOCATABILITY

MACRO-80 can produce modules of relocatable code.   Also,
like many assemblers, the MACRO-80 assembler can produce
absolute code. The key advantage of relocatability is that
programs can be assembled in modules. Then, within certain
restrictions described in Chapter 6, the modules can then be
located almost anywhere in memory.

Relocatable modules also 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 or into ROM/PROM.

Relocatability will be discussed further under Section 3.2,
Symbols.

2.3   MACRO FACILITY

The MACRO-80 assembler supports a complete, Intel standard
macro facility.   The macro facility allows a programmer to
write blocks of code for a set of instructions used
frequently.   The   need   for   recoding these instructions is
eliminated.

The programmer gives this block of code a name, called a
macro.   The instructions are the macro definition. Each
time the set of instructions is needed, instead of recoding
the set of instructions, the programmer simply "calls" the
macro. MACRO-80 expands the macro call by assembling the
block of instructions into the program automatically. The
macro call also passes parameters to the assembler for use
during macro expansion. The use of macros reduces the size
of a source module because the macro definitions are stored
in disk files and come into the module only when needed
during assembly.

Macros can be nested, that is, a macro can be called from
inside another macro. Nesting of macros is limited only by
memory.

2.4 CONDITIONAL ASSEMBLY

MACRO-80 also supports conditional assembly. The programmer can determine a condition under which portions of the program are either assembled or not assembled. Conditional assembly capability is enhanced by a complete set of conditional pseudo operations that include testing of assembly pass, symbol definition, and parameters to macros. Conditionals may be nested up to 255 levels.

2.5 UTILITY PROGRAMS

Three utility programs provide the additional support needed to develop powerful and useful assembly language programs: LINK-80 Linking Loader, LIB-80 Library Manager, and CREF-80 Cross Reference Facility.

LINK-80 Linking Loader

The Microsoft LINK-80 Linking Loader is used to convert the assembled module (.REL file) into an executable module (.COM file). The .REL file is not an executable file.

LINK-80 can also be used to:

> load, link, and run one or more modules

> load relocatable programs at user-specified locations

> load program areas and data areas into separate memory locations

While performing these tasks, LINK-80 resolves external references between modules (that is, any program that calls an external value, something defined in a different program or module, will have the outside references filled at link time by LINK-80), and saves the executable object (.COM) file on disk, so it can be run from the operating system.

These load capabilities mean that the assembled program may be linked with the user's library to add routines to one of the high-level langauge runtime libraries. Assembled programs can be linked to high-level language programs -- COBOL-80 and FORTRAN-80, for example -- as well as to MACRO-80 programs.

CREF-80 Cross Reference Facility

The CREF-80 Cross Reference Facility processes a cross
reference file generated by MACRO-80. The result is a cross
reference listing that can aid in the debugging of your
program.

LIB-80 Library Manager (CP/M versions only)

LIB-80 is designed as a runtime library manager for CP/M
versions of the Utility Software Package. LIB-80 may also
be used to create your own library of assembly language
subroutines.

LIB-80 creates runtime libraries from assembly language
programs that are subroutines to COBOL, FORTRAN, and other
assembly language programs. The programs collected by
LIB-80 may be special modules created by the programmer or
modules from an existing library. With LIB-80, you can
create specialized runtime libraries for whatever execution
requirements you design.

Contents

CHAPTER 3

PROGRAMMING WITH THE UTILITY SOFTWARE PACKAGE

This chapter describes what the user needs to know to create MACRO-80 macro assembler source files. Source files are created using a text editor, such as CP/M ED. The Utility Software Package does not include a text editor program.

Source files are assembled using the procedures described in Chapter 4.

3.1   SOURCE FILE ORGANIZATION

File Organization

A MACRO-80 macro assembler source file is a series of lines written in assembly language. The last line of the file must be an END statement. Matching statements (such as IF...ENDIF) must be entered in the proper sequence. Otherwise, lines may appear in any order the programmer designs.

Statement Line Format

Source files input to the MACRO-80 macro assembler consist of statement lines divided into parts or "fields."

```
        BUF:          DS          1000H          ;create a buffer
         ↑            ↑            ↑               ↑
         |            |            |               |
       SYMBOL     OPERATION     ARGUMENT        COMMENT
```

SYMBOL   field contains one of the three types of symbol
           (LABEL, PUBLIC, and EXTERNAL), followed by a colon
           unless it is part of a SET, EQU, or MACRO
           statement.

OPERATION field contains an OPCODE, a PSEUDO-OP, a MACRO
           name, or an expression.

ARGUMENT   field contains expressions (specific values,
           variables, register names, operands and operators).

;COMMENT   field contains comment text always preceded by a
           semicolon.

All fields are optional. You may enter a completely blank
line.

Statement lines may begin in any column. Multiple blanks or
tabs may be inserted between fields to improve readability,
but at least one space or tab is required between each
field.

Comments

A MACRO-80 macro assembler source line is basically an
Operation and its Argument. Therefore, the MACRO-80 macro
assembler requires that a COMMENT always begin with a
semicolon. A COMMENT ends with a carriage return.

For long comments, you may want to use the .COMMENT
pseudo-op to avoid entering a semicolon for every line. See
the File Related Pseudo-ops section of Chapter 4 for the
description of .COMMENT.

3.2 SYMBOLS

Symbols are simply names for particular functions or values. Symbol names are created and defined by the programmer.

Symbols in the Utility Software Package belong to one of three types, according to their function. The three types are LABEL, PUBLIC, and EXTERNAL. All three types of symbols have a MODE attribute that corresponds to the segment of memory the symbol represents. Refer to the section on modes following the description of symbol types.

All three types of symbols have the following characteristics:

1. Symbols may be any length, but the number of significant characters passed to the linker varies with the type of symbol:

    a. for LABELs, only the first sixteen characters are significant.

    b. for PUBLIC and EXTERNAL symbols, only the first six characters are passed to the linker.

    Additional characters are truncated internally.

2. A legal symbol name may contain the characters:

    A-Z   0-9 $.   ?   @   _

3. A symbol may not start with a digit or an underline

4. When a symbol is read, lower case is translated into upper case, so you may enter the name using either case or both.

LABEL:

A LABEL: is a reference point for statements inside the program module where the label appears. A LABEL: sets the value of the symbol LABEL to the address of the data that follows. For example, in the statement:

        BUF: DS   1000H

BUF: equals the first address of the 1000H byte reserved space.

Once a label is defined, the label can be used as an entry in the ARGUMENT field. A statement with a label in its argument loops to the statement line with that label in its SYMBOL field, which is where the label is defined. The label's definition replaces the label used in an ARGUMENT field. For example,

        STA   BUF

sends the value in the accumulator to the area in memory represented by the label BUF.

A LABEL may be any legal symbol name, up to 16 characters long.

If you want to define a LABEL, it must be the first item in the statement line. 8080 and Z80 labels must be followed immediately by a single colon (no space), unless the LABEL is part of a SET or EQU statement. (If two colons are entered, the "label" becomes a PUBLIC symbol. See PUBLIC Symbols below.)

PUBLIC

A PUBLIC symbol is defined much like a LABEL. The difference is that a PUBLIC symbol is available as a reference point for statements in other program modules, too.

A symbol is declared PUBLIC by:

        two colons (::) following the name. For example,

            FOO::   RET

        one of the pseduo-ops PUBLIC, ENTRY, or GLOBAL. For example,

            PUBLIC   FOO

        See the Data Definition and Symbol Definition Pseudo-ops section in Chapter 4 for descriptions of how to use these pseudo-ops.

The result of both methods of declaration is the same. Therefore,

        FOO::   RET

is equivalent to

        PUBLIC   FOO
        FOO: RET

EXTERNAL

An EXTERNAL symbol is defined outside the program module where it appears. An EXTERNAL symbol is defined as a PUBLIC symbol in another, separate program module. At link time (when the LINK-80 Linking Loader is used), the EXTERNAL symbol is given the value of the PUBLIC symbol in the other program module. For example:

        MOD1

                FOO::   DB   7   ;PUBLIC FOO = 7

        MOD2

                        BYTE EXT   FOO   ;EXTERNAL FOO

                At link time, LINK-80 goes to the address of PUBLIC
                FOO and uses the value there (7) for EXTERNAL FOO.

A symbol is declared EXTERNAL by:

    1. two pound signs (##) following a reference to a
        symbol name. For example:

            CALL   FOO##

        declares FOO as a two-byte symbol defined in
        another program module.

    2. one of the pseudo-ops EXT, EXTRN, or EXTERNAL for
        two-byte values. For example:

            EXT   FOO

        declares FOO as a two-byte value defined in another
        program module.

    3. one of the pseudo-ops BYTE EXT, BYTE EXTERN, or
        BYTE EXTERNAL for one-byte values. For example:

            BYTE EXT   FOO

        declares FOO as a one-byte value defined in another
        program module.

        See the Symbol Definition Pseudo-ops section in
        Chapter 4 for descriptions of how to use these
        pseudo-ops.

As for PUBLIC symbols, the result of both methods of declaration is the same. Therefore,

        CALL   FOO##

is equivalent to

        EXT   FOO
        CALL   FOO

MODES

A symbol is referenced by entering its name in the ARGUMENT field of a statement line. When a symbol is referenced, the value of the symbol (derived from the instruction which defines the symbol) is substituted for the symbol name and used in the operation.

The value of a symbol is evaluated according to its program counter (PC) mode. The PC mode determines whether a section of a program will be loaded into memory at addresses predetermined by the programmer (absolute mode), or at relative addresses that change depending on the size and number of programs (code relative mode) and amount of data (data relative mode), or at addresses shared with another program module (COMMON mode). The default mode is Code Relative.

Absolute Mode: Absolute mode assembles non-relocatable code. A programmer selects Absolute mode when a block of program code is to be loaded each time into specific addresses, regardless of what else is loaded concurrently.

Data Relative Mode: Data Relative mode assembles code for a section of a program that may change and therefore must be loaded into RAM. This applies to program data areas especially. Symbols in Data Relative Mode are relocatable.

Code Relative Mode: Code (program) Relative mode assembles code for sections of programs that will not be changed and therefore can be loaded into ROM/PROM. Symbols in Code Relative Mode are relocatable.

COMMON Mode: COMMON mode assembles code that is loaded into a defined common data area. This allows program modules to share a block of memory and common values.

To change mode, use a PC mode pseudo-op in a statement line. The PC mode pseudo-ops are:

        ASEG   Absolute mode
        DSEG   Data Relative mode
        CSEG   Code Relative mode--default mode
        COMMON   COMMON mode

These pseudo-ops are described in detail in the PC Mode Pseudo-ops section of Chapter 4.

This PC mode capability in the MACRO-80 macro assembler allows a programmer to develop assembly language programs that can be relocated. Many assembly language programmers may have learned always to set an Origin statement at the beginning of every module, subroutine, or main assembly language program. Under MACRO-80 this mode of addressing is called Absolute mode because hard (or actual addresses) are specified beginning, especially, with the Origin statement.

MACRO-80 has two other, "relative" modes of addressing available, called Code (Program) relative and Data relative. Segments of code written in these two modes are relocatable. Relocatable means the program module can be loaded starting at <u>any</u> address in available memory, using the /P and /D switches (special commands) in LINK-80.

3.3   OPCODES AND PSEUDO-OPS

Opcodes are the mnemonic names for the machine instructions.
Pseudo-ops are directions to the assembler, not the
microprocessor.

MACRO-80 supports two instruction sets: 8080 and Z80. A
list of the opcodes with brief summaries of their functions
is included as Appendix F. To program with the opcodes of
the different languages, the user must first enter the
pseudo-op which tells the assembler which language is being
coded. Refer to the Language Set Selection Pseudo-ops
section of Chapter 4 for details.

MACRO-80 also supports a large variety of pseudo-ops that
direct the assembler to perform many different functions.
The pseudo-ops are described extensively in Chapter 4 and
are summarized in Appendix E.

Opcodes and pseudo-ops are (usually) entered in the
OPERATION field of a statement line. (A program statement
line usually has an entry in the operation field, unless the
line is a Comment line only. The Operation field will be
the first field filled if no label is entered.) An
Operation may be any 8080 or Z80 mnemonic; or a MACRO-80
macro assembler pseudo-op, macro call, or expression.

The OPERATION field entries are evaluated in the following
order:

1.   Macro call

2.   Opcode/Pseudo-op

3.   Expressions

MACRO-80 compares the entry in the OPERATION filed to an
internal list of macro names. If the entry is found, the
macro is expanded. If the entry is not a macro, MACRO-80
tries to evaluate the entry as an opcode. If the entry is
not an opcode, MACRO-80 tries to evaluate the entry as a
pseudo-op. If the entry is not a pseudo-op, MACRO-80
evaluates the entry as an expression. If an expression is
entered as a statement line without an opcode, pseudo-op, or
macro name in front of it, the MACRO-80 macro assembler does
not return an error. Rather, the assembler assumes that a
define byte pseudo-op belongs in front of the expression and
assembles the line.

Because of the order of evaluation, a macro name that is the
same as an opcode prevents you from using the opcode again,
except as a macro call. For example, if you give a block of
macro code the name ADD in your program, you cannot use ADD
as an opcode in that program.

3.4 ARGUMENTS: EXPRESSIONS

Arguments for the opcodes and pseudo-ops are usually called expressions because they resemble mathematical expressions, such as 5+4*3. The parts of an expression are called operands (5, 4, and 3 in the mathematical expression) and operators (the + and * are examples). Expressions may contain one operand or more than one. One operand expressions are probably the form most commonly used as arguments. If the expression contains more than one operand, the operands are related to each other by an operator. For example:

          5+4   6-3   7*2   8/7   9>8

and so on. In MACRO-80, operands are numeric values represented by numbers, characters, symbols, or 8080 opcodes. Operators may be arithmetic or logical.

You are probably familiar with the various forms of expressions that can be used as arguments, but you may want to review the details given below for characteristics unique to MACRO-80.

The following sections define the forms of operands and operators MACRO-80 supports.

3.4.1 Operands

Operands may be numbers, characters, symbols, or 8080 opcodes.

Numbers

The default base for numbers is decimal. The base may be changed by the .RADIX pseudo-op. Any base from 2 (binary) to 16 (hexadecimal) may be selected. When the radix is greater than 10, A-F are used for the digits following 9. If the first digit of a number is not numeric, the number must be preceded by a zero.

A number is always evaluated in the current radix unless one of the following special notations is used:

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

Numbers are 16-bit unsigned binary quantities. Overflow of a number beyond two bytes (16 bits -- that is, 65535 decimal) is ignored, and the result is the low order 16 bits.

ASCII Strings

A string is composed of zero or more characters delimited by quotation marks. Either single (') or double (") quotation marks may be used as string delimiters. When a quoted string is entered as an argument, the values of the characters are stored in memory one after the other. For example:

        DB   "ABC"

stores the ASCII value of A at the first address, B at the second address, and C at the third.

The delimiter quotes may be used as characters if they appear twice for every character occurrence desired. For example, the statement

        "I am ""great"" today"

stores the string

        I am "great" today

If no characters are placed between the quotation marks, the string is evaluated as a null string.

Character Constants

Like strings, character constants are composed of zero, one, or two ASCII characters, delimited by quotation marks. Either single or double quotation marks may be used as delimiters. The delimiter quotes may be used as characters if they appear twice for every character occurrence desired.

The differences are:

1. A character constant is only zero, one, or two characters.

2. Quoted characters are a character constant only if the expression has more than one operand. If the characters are entered as the only operand, they are evaluated and stored as a string. For example:

        'A'+1 is a character constant, but

        'A' is a string.

3. The value of a character constant is calculated, and the result is stored with the low-byte in the first address and the high-byte in the second. For example:

3. The value of a character constant is calculated, and the result is stored with the low-byte in the first address and the high-byte in the second. For example:

    DW   'AB'+0

    evaluates to 4142H and stores 42 in the first address and 41 in the second.

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'+0 is 41H*256+42H+0.

The ASCII decimal and hexadecimal values for characters are listed in Appendix C.

Symbols in Expressions

A symbol may be used as an operand in an expression. The symbol is evaluated, and the value is substituted for the symbol. The Operation is performed using the symbol's value.

The benefit of using symbols as operands is that the programmer need not remember the exact value each time it is needed; rather, the symbol name can be used. The name is usually easier to remember, especially if the symbol name is made mnemonic. The use of symbols as operands becomes more attractive, of course, as the number of symbols in a program increases.

Rules Governing the Use of EXTERNALS in expressions:

1. EXTERNAL symbols may be used in expressions with the following operators only:

    +   -   *   /   MOD   HIGH   LOW

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

MODE Rules affecting SYMBOLS in expressions:

1. In any operation, except AND, OR, or XOR, the operands may be any mode.

2. For AND, OR, XOR, SHL, and SHR, both operands must be absolute and internal.

3. When an expression contains an Absolute operand and an operand in another mode, the result of the expression will be in the other (not Absolute) mode.

4. When subtracting two operands in different modes, the result will be in Absolute mode. Otherwise, the result will be in the mode of the operands.

5. When adding a data relative symbol and a code relative symbol, the result will be unknown, and MACRO-80 passes the expression to LINK-80 as an unknown, which LINK-80 resolves.

Current Program Counter Symbol

One additional symbol for the Argument field only must be noted: the current program counter symbol. The current program counter is the address of the next instruction to be assembled. The current program counter is often a convenient reference point for calculating new addresses. Instead of remembering or calculating the current program address, the programmer uses a symbol that tells the assembler to use the value of the current program address.

The current program counter symbol is $.

8080 Opcodes as Operands

8080 opcodes are valid one-byte operands in 8080 mode only. During assembly, the opcode is evaluated to its hexadecimal value.

To use 8080 opcodes as operands, first set the .8080 pseudo-op. See the Language Set Selection Pseudo-ops section of Chapter 4 for a description of how to use the .8080 pseudo-op.

Only the first byte is a valid operand. Use parentheses to direct the assembler to generate one byte for opcodes that normally generate more than one. For example:

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

The assembler returns an error if more than one byte is included in the operand (inside the parentheses) -- such as (CPI 5), (LXI B,LABEL1), or (JMP LABEL2).

Opcodes that generate one byte normally may be used as operands without being enclosed in parentheses.

3.4.2 Operators

MACRO-80 allows both arithmetic and logical operators. Operators which return true or false conditions return true if the result is any non-zero value and false if the result is zero.

The following arithmetic and logical operators are allowed in expressions.

Operator   Definition

NUL   Returns true if the argument (a parameter) is null. The remainder of the line after NUL is taken as the argument to NUL. The conditional

                        IF NUL <argument>

                    is false if the first character of the argument is anything other than a semicolon or carriage return. Note that IFB and IFNB perform the same functions but are simpler to use. (Refer to the Conditional Assembly Facility section in Chapter 4.)

TYPE   The TYPE operator returns a byte that describes two characteristics of its argument: 1) the mode, and 2) whether it is External or not. The argument to TYPE may be any expression (string, numeric, logical). If the expression is invalid, TYPE returns zero.

                    The byte that is returned is configured as

follows:

The lower two bits are the mode. If the
lower two bits are:

```
0       the mode is Absolute
1       the mode is Program Relative
2       the mode is Data Relative
3       the mode is Common Relative
```

The high bit (80H) is the External bit. If
the high bit is on, the expression contains
an External. If the high bit is off, the
expression is local (not External).

The Defined bit is 20H. This bit is on if
the expression is locally defined, and it is
off if the expression is undefined or
external. If neither bit is on, the
expression is invalid.

TYPE is usually used inside macros, where an
argument type may need to be tested to make a
decision regarding program flow; for
example, when conditional assembly is
involved.

EXAMPLE:

```
FOO     MACRO    X
        LOCAL    Z
Z       SET TYPE X
IF      Z...
```

TYPE tests the mode and type of X. Depending
on the evaluation of X, the block of code
beginning with IF Z... may be assembled or
omitted.

LOW   Isolates the low order 8 bits of an absolute
                 16-bit value.

HIGH   Isolates the high order 8 bits of an absolute
                 16-bit value.

*   Multiply

/   Divide

MOD   Modulo. Divide the left operand by the right
                 operand and return the value of the remainder
                 (modulo).

SHR   Shift Right. SHR is followed by an integer
                        which specifies the number of bit positions
                        the value is to be right shifted.

SHL   Shift Left. SHL is followed by an integer
                        which specifies the number of bit positions
                        the value is to be left shifted.

- (Unary Minus)   Indicates that following value is negative,
                        as in a negative integer.

+   Add

-                       Subtract the right operand from the left
                        operand.

EQ   Equal. Returns true if the operands equal
                        each other.

NE   Not Equal. Returns true if the operands are
                        not equal to each other.

LT   Less Than. Returns true if the left operand
                        is less than the right operand.

LE   Less than or Equal. Returns true if the left
                        operand is less than or equal to the right
                        operand.

GT   Greater Than. Returns true if the left
                        operand is greater than the right operand.

GE   Greater than or Equal. Returns true if the
                        left operand is greater than or equal to the
                        right operand.

NOT   Logical NOT. Returns true if left operand is
                        true and right is false or if right is true
                        and left is false. Returns false if both are
                        true or both are false.

AND   Logical AND. Returns true if both operators
                        are true. Returns false if either operator
                        is false or if both are false. Both operands
                        must be absolute values.

OR   Logical OR. Returns true if either operator
                        is true or if both are true. Returns false
                        if both operators are false. Both operands
                        must be absolute values.

XOR   Exclusive OR.   Returns   true   if   either
                       operator is true and the other is false.
                       Returns false if both operators are true or
                       if both operators are false. Both operands
                       must be absolute values.

The order of precedence for the operators is:

        NUL, TYPE

        LOW, HIGH

        *, /, MOD, SHR, SHL

        Unary Minus

        +, -

        EQ, NE, LT, LE, GT, GE

        NOT

        AND

        OR, XOR

Subexpressions involving operators of higher precedence than
an expression are computed first. The order of precedence
may be altered by using parentheses around portions of an
expression you wish to give higher precedence.

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

The byte isolation operators (HIGH and LOW) isolate the
high- or low-order 8 bits of a 16-bit value.

Contents

CHAPTER 4

ASSEMBLER FEATURES

The MACRO-80 macro assembler features three general facilities: single-function pseudo-ops, a macro facility, and a conditional assembly facility.

4.1   SINGLE-FUNCTION PSEUDO-OPS

Single-function pseudo-ops involve only their own statement line and direct the assembler to perform only one function. (Macros and conditionals involve more than one line of code, so they may be thought of as block pseudo-ops.)

The Single-Function Pseudo-ops are divided into five types: Instruction Set Selection, Data Definition and Symbol Definition, PC Mode, File Related, and Listing Control.

INSTRUCTION SET SELECTION

The default instruction set mode is 8080.   If the correct
instruction set selection pseudo-op is not given, the
assembler will return fatal errors for opcodes that are not
valid for the current instruction set selection mode.   That
is, .Z80 assembles Z80 opcodes only; .8080 assembles 8080
opcodes only.   Therefore, if you have written any assembly
language programs for Z80, you need to insert the .Z80
instruction set pseudo-op at the beginning of the program
file.

Note that all the pseudo-ops listed in this chapter will
assemble in both instruction set modes.

.Z80

>   .Z80 takes no arguments.   .Z80 directs MACRO-80 to
>   assemble Z80 opcodes.

.8080

>   .8080 takes no arguments.   .8080   directs   MACRO-80
>   to assemble 8080 opcodes.   (default)

>   All opcodes entered following an Instruction Set
>   Selection pseudo-op will be assembled as that type
>   of code until a different Instruction Set Selection
>   pseudo-op is encountered.

>   If you enter an opcode not belonging to the
>   selected instruction set, MACRO-80 will return an
>   Objectionable Syntax error (letter O).

DATA DEFINITION AND SYMBOL DEFINITION

All of the data definition and symbol definition pseudo-ops
are supported in both instruction set modes. (The one
notable exception is SET, which is   illegal   in   .Z80   mode.
For your information, The following notation has been placed
before the pseudo-op syntax to indicate which microprocessor
the pseudo-op is usually associated with:

   *   indicates a Z80 pseudo-op

         No asterisk indicates an Intel 8080 pseudo-op

<u>Define</u> <u>Byte</u>

```
    DB <exp>[,<exp>...]
 *  DEFB <exp>[,<exp>...]
    DB <string>[<string>...]
 *  DEFM <string>[,<string>...]
```

The arguments to DB are either expressions or strings. The arguments to DEFB are expressions. The arguments to DEFM are strings. Strings must be enclosed in quotes, either single or double.

NOTE: DB is used throughout the following explanation to represent all the Define Byte pseudo-ops.

DB is used to store a value (string or numeric) in a memory location, 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 8080 or Z80 string are stored in the order of appearance, each as a one-byte value with the high order bit set to zero.

EXAMPLE:

```
                                     DB      'AB'
                                     DB      'AB' AND 0FFH
                                     DB      'ABC'
```

assembles as:

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

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.

>EXAMPLE:

>   FOO: DC   "ABC"

>assembles to:

>0000' 41 42 C3 FOO: DC   "ABC"

Define Space

```
    DS   <exp>[,<val>]
*   DEFS <exp>[,<val>]
```

        The define space pseudo-ops reserve an area of memory. The value of <exp> gives the number of bytes to be reserved.

        To initialize the reserved space, set <val> to the value desired. If <val> is nul (that is, omitted), the reserved space is left as is (uninitialized); the reserved block of memory is not automatically initialized to zeros. As an alternative to setting <val> to zero, when you want the define space block initialized to zeros, you may use the /M switch at assembly time. See the Switches section in Chapter 5, Running MACRO-80, for a description of the /M switch.

        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 define space pseudo-op generated no code on pass 1.

EXAMPLE:

```
    DS        100H
```

reserves 100H bytes of memory, uninitialized (whatever values were in those bytes before the program was loaded will still be there). Use the /M switch at assembly time to initialized the 100H bytes to zero, if you want. Or, use the following statement to initialize a reserved space to zero or any other value:

```
    DS        100H,2
```

reserves 100H bytes, each initialized to a value of 2.

Define Word

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

The define word pseudo-ops store the values of the expressions in successive memory locations beginning with the current location counter. Expressions are evaluated as 2-byte (word) values. Values are stored low-order byte first, then high-order byte.

Contrast with DDB.

EXAMPLE:

```
                        FOO:    DW              1234H
```

assembles as:

```
0000'  1234      FOO:      DW      1234H
```

Note: The bytes are shown on the listing in the order entered, not the order stored.

Equate

<name> EQU <exp>

        EQU assigns the value of <exp> to <name>. The
        <name> may be a label, a symbol, or a variable, and
        may be used subsequently in expressions. <name>
        may not be followed by colon(s).

        If <exp> is External, an error is generated. If
        <name> already has a value other than <exp>, an M
        error is generated.

        If you will want to redefine <name> later in the
        program, use the SET or ASET pseudo-op to define
        <name> instead of EQU.

        Contrast with SET.

        EXAMPLE:

           BUF   EQU   0F3H

External Symbol
───────────

        EXT   <name>[,<name>...]
        EXTRN <name>[,<name>...]
   *   EXTERNAL <name>[,<name>...]
        BYTE EXT <symbol>
        BYTE EXTRN <symbol>
        BYTE EXTERNAL <symbol>

        The External symbol pseudo-ops declare that the
        name(s) in the list are External (i.e., defined in
        a different module). If any item in the list
        refers to a name that is defined in the current
        program, an M error results. A reference to a name
        where the name is followed immediately be two pound
        signs (e.g., NAME##) also declares the name as
        External.

        Externals may evaluate to either one or two bytes.
        For all External symbol names, only the first 6
        characters are passed to the linker. Additional
        characters are truncated internally.

        EXAMPLE:

            EXTRN   ITRAN   ;tranf init rtn

        MACRO-80 will generate no code for this statement
        when this module is assembled. When ITRAN is used
        as an argument to a CALL statement, the CALL ITRAN
        statement generates the code for CALL but a zero
        value (0000*) for ITRAN. At link time, LINK-80
        will search all modules loaded for a PUBLIC ITRAN
        statement and use the definition of ITRAN found in
        that module to define ITRAN in the CALL ITRAN
        statement.

Public Symbol

ENTRY <name>[,<name>...]
GLOBAL <name>[,<name>...]
PUBLIC <name>[,<name>...]

> The Public symbol pseudo-ops declare each name in
> the list as internal and therefore available for
> use by this program and other programs to be loaded
> concurrently and linked with LINK-80. 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
> block name.
>
> Only the first 6 characters of a Public symbol name
> are passed to the linker. Additional characters
> are truncated internally.
>
> EXAMPLE:
>
>   PUBLIC   ITRAN   ;tranf init rtn
>   •
>   •
>   •
>   ITRAN: LD   HL,PASSA   ;store addr of
>   ;reg pass area
>
> MACRO-80 assembles the LD statement as usuall but
> generates no code for the PUBLIC ITRAN statement.
> When LINK-80 sees EXTRN ITRAN in another module, it
> knows to search until it sees this PUBLIC ITRAN
> statement. Then, LINK-80 links the value of ITRAN:
> LD HL,PASSA statement to the CALL ITRAN statement
> in the other module(s).

Set

```
       <name> SET <exp>        (Not in .Z80 mode)
  *    <name> DEFL <exp>
       <name> ASET <exp>
```

> The Set pseudo-ops assign the value of <exp> to
> <name>. The <name> may be a label, a symbol, or a
> variable, and may be used subsequently in
> expressions. <name> may not be followed by
> colon(s). If <exp> is External, an error is
> generated.
>
> The SET pseudo-op may <u>not</u> be used in .Z80 mode
> because SET is a Z80 opcode. Both ASET and DEFL
> may be used in both instruction set modes.
>
> Use one of the SET pseudo-ops <u>instead</u> of EQU to
> define and redefine <name>s you may want to
> redefine later. <name> may be redefined with any
> of the Set pseudo-ops, regardless of which
> pseudo-op was used to define <name> originally (the
> prohibition against SET in .Z80 mode still applies,
> however).
>
> Contrast with EQU.
>
> EXAMPLE:
>
> ```
>     FOO ASET BAZ+1000H
> ```
>
> Whenever FOO is used as an expression (operand),
> the ALDS assembler will evaluate BAZ+1000H and
> substitute the value for FOO. Later, if you want
> FOO to represent a different value, simply reenter
> the FOO ASET statement with a different expression.
>
> ```
>     FOO ASET BAZ+1000H
>         .
>         .
>         .
>     FOO ASET 3000H
>         .
>         .
>         .
>     FOO DEFL 6CDEH
> ```

PC MODE

Many of the pseudo-ops operate on or from the current
location counter, also known as the program counter or PC.
The current PC is the address of the next byte to be
generated.

In MACRO-80, the PC has a mode, which gives symbols and
expressions their modes. (Refer again to the Overview in
Chapter 1 and the Symbols section in Chapter 3, if
necessary.) Each mode is given a segment of memory by
LINK-80 for the instructions assembled to each mode.

The four modes are Absolute, Data Relative, Code Relative,
and COMMON Relative.

If the PC mode is absolute, the PC is an absolute address.
If the PC mode is relative, the PC is a relative address and
may be considered an offset from the absolute address where
the beginning of that relative segment will be loaded by
LINK-80.

The PC mode pseudo-ops are used to specify in which PC mode
a segment of a program will be assembled.

Absolute Segment

ASEG

ASEG never has operands. ASEG generates non-relocatable code.

ASEG sets the location counter to an absolute segment (actual address) of memory. The ASEG will default to 0, which could cause the module to write over part of the operating system. We recommend that each ASEG be followed with an ORG statement set at 103H or higher.

Code Segment

CSEG

> CSEG never has an operand. Code assembled in Code
> Relative mode can be loaded into ROM/PROM.
>
> CSEG resets 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.
>
> Note, however, that the ORG statement does not set
> a hard (absolute) address under CSEG mode. An ORG
> statement under CSEG causes the assembler to add
> the number of bytes specified by the <exp> argument
> in the ORG statement to the last CSEG address
> loaded. If, for example, ORG 50 is given, MACRO-80
> will add 50 bytes to the current CSEG location then
> begin loading the CSEG. The clearing effect of the
> ORG statement following CSEG (and DSEG as well) can
> be used to give the module an offset. The
> rationale for not allowing ORG to set an absolute
> address for CSEG is to keep the CSEG relocatable.
>
> To set an absolute address for the CSEG, use the /P
> switch in LINK-80.
>
> CSEG is the default mode of the assembler.
> 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, DSEG, or COMMON pseudo-op is
> executed. CSEG is then entered to return the
> assembler to Code Relative mode, at which point the
> location counter returns to the next free location
> in the Code Relative segment.

Data Segment

DSEG

The DSEG pseudo-op never has operands. DSEG
specifies segments of assembled relocatable code
that will later be loaded into RAM only.

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.

Note, however, that the ORG statement does not set
a hard (absolute) address under DSEG mode. An ORG
statement under DSEG causes the assembler to add
the number of bytes specified by the <exp> argument
in the ORG statement to the last DSEG address
loaded. If, for example, ORG 50 is given, MACRO-80
will add 50 bytes to the last DSEG address loaded
then begin loading the DSEG. The clearing effect
of the ORG statement following DSEG (and CSEG as
well) can be used to give the module an offset.
The rationale for not allowing ORG to set an
absolute address for DSEG is to keep the DSEG
relocatable.

To set an absolute address for the DSEG, use the /D
switch in LINK-80.

Common Block

COMMON /<block name>/

> The argument to COMMON is the common block name. COMMON creates a common data area for every COMMON block that is named in the program. If <block name> is omitted or consists of spaces, the block is considered to be blank common.
>
> COMMON statements are non-executable, storage allocating statements. .COMMON assigns variables, arrays, and data to a storage area called COMMON storage. This allows various program modules to share the same storage area. Statements entered following the .COMMON statement are assembled to the COMMON area under the <block name>. The length of a COMMON area is the number of bytes required to contain the variables, arrays, and data declared in the COMMON block, which ends when another PC mode pseudo-op is encountered. COMMON blocks of the same name may be different lengths. If the lengths differ, then the program module with the longest COMMON block must be loaded first (that is, must be the first module name given in the LINK-80 command line; see Chapter 6 for the description of LINK-80).
>
> 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.

EXAMPLE:

```
                COMMON  /DATABIN/
        ANVIL   EQU     100H
                DB      0FFH
                DW      1234H
                DCI     'FORGE'
                CSEG

                  .
                  .
                  .
```

Set Origin

ORG <exp>

At any time, the value of a location counter may be
changed by use of ORG. Under the ASEG PC mode, the
location counter is set to the value of <exp>, and
the assembler assigns generated code starting with
that value. Under the CSEG, DSEG, and COMMON PC
modes, the location counter for the segment is
incremented by the value of <exp>, and the
assembler assigns generated code starting with the
value of that last segment address loaded plus the
value of <exp>.   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.

EXAMPLE:

    DSEG
    ORG   50

sets the Data Relative location counter to 50,
relative to the start of the Data Relative segment
of memory. This means that the first 50H addresses
will be filled with 0.   This method provides
relocatability. The ORG <exp> statement does not
specify a fixed address in CSEG or DSEG mode;
rather, LINK-80 loads the segment at a flexible
address appropriate to the modules being loaded
together.

On the other hand, a program that begins with the
statements

    ASEG
    ORG   800H

and is assembled entirely in Absolute mode will
always load beginning at 800H, unless the ORG
statement is changed in the source file. That is,
ORG <exp> following ASEG originates the segment at
a fixed (i.e., absolute) address specified by
<exp>.   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.
(For details, see Section 6.3, Switches.)

Relocate

.PHASE <exp>
   .
   .
   .
.DEPHASE

> .PHASE allows code to be located in one area, but executed only at a different area with a start address specified by <exp>. The <exp> must be an absolute value. .DEPHASE is used to indicate the end of the relocated block of code.
>
> The PC mode within a .PHASE block is absolute, the same as the mode of the <exp> in the .PHASE statement. The code, however, is loaded in the area in effect when the .PHASE statement is encountered. The code within the block is later moved to the address specified by <exp> for execution.

> EXAMPLE:

```
                          .PHASE      100H
              FOO:        CALL        BAZ
                          JMP         ZOO
              BAZ:        RET
                          .DEPHASE
              ZOO:        JMP         5
```

> assembles as:

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

> .PHASE....DEPHASE blocks are a way to execute a block of code at a specific absolute address.

FILE RELATED

The file related pseudo-ops insert long comments in the program, give the module a name, end the module, or move other files into the current program.

Comment

.COMMENT <delim><text><delim>

The first non-blank character encountered after
.COMMENT is taken as the delimiter. The <text>
following the delimiter becomes a comment block
which continues until the next occurrence of
<delimiter>.

Use the .COMMENT pseudo-op to make long comments.
It is not necessary to enter the semicolon to
indicate a COMMENT. Indeed, the main reason for
using .COMMENT is to override the need to begin
each comment line with a semicolon. During
assembly, .COMMENT blocks are ignored and not
assembled.

EXAMPLE:

    .COMMENT * any amount of text
    entered here
        .
        .
    .* ;return to normal assembly

End of Program

END [<exp>]

The END statement specifies the end of the module. If the END statement is not included, a %No END statement warning error message results.

The <exp> may be a label, symbol, number, or any other legal argument that LINK-80 can load as the starting point into the first address to be loaded. If <exp> is present, LINK-80 will place an 8080 JMP instruction at 0100H to the address of <exp>. If <exp> is not present, then no start address is passed to LINK-80 for that program, and execution begins at the first module loaded. (Also, if <exp> is not specified, the LINK-80 /G switch will not work for the module.)

The <exp> tells LINK-80 that the program is a main program. Without <exp>, LINK-80 takes assembly language programs as subroutines. If you link only assembly language programs and none contains an END statement with <exp>, LINK-80 will ask for a main program. If you link two or more programs with END <exp> statements, LINK-80 cannot distinguish which should be the main program.

If you want to link two or more main programs, use the /G:Name or /E:Name switches in LINK-80 (see Section 6.2.2, Switches). The "Name" will be the <exp> of the END statement for the program you want to serve as the main program.

If any high-level language program is loaded with assembly language modules, LINK-80 takes the high-level language program as the main program automatically. Therefore, if you want an assembly language module executed before the high-level language program, use the /G:Name or /E:Name switch in LINK-80 to set the assembly language module as the beginning of the program.

As an alternative, we recommend that you place a CALL or INCLUDE statement at the beginning of the high-level language program, and call in the assembly language program for execution prior to execution of the high-level language program.

Include

```
INCLUDE <filename>
$INCLUDE <filename>
MACLIB <filename>
```

All three pseudo-ops are synonomous.

These Include pseudo-ops insert source code from an alternate assembly language source file into the current source file during assembly. Use of an Include pseudo-op eliminates the need to repeat an often-used sequence of statements in the current source file.

The <filename> is any valid file specification for the operating system. If the filename extension and/or device designation are other than the default, source filename specifications must include them. The default filename extension for source files is .MAC. The default device designation is the currently logged drive or device.

The included file is opened and assembled into the current source file immediately following the Include pseudo-op statement. When end-of-file is reached, assembly resumes with the next statement following Include pseudo-op.

Nested Includes are not allowed. If encountered, they will result in an objectionable syntax error, O.

The file specified in the operand field must exist. If the file is not found, the error V (value error) is returned, and the Include is ignored. The V error is also returned if the Include filename extension is not .MAC.

On a MACRO-80 listing, the letter C is printed between the assembled code and the source line on each line assembled from an included file. See the Listing Control Pseudo-op section below for a description of listing file formats.

Name Module

NAME ('modname')

> Name defines a name for the module. The parentheses and quotation marks around modname are required. 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 filename.

Radix

.RADIX <exp>

> The <exp> in a .RADIX statement is always a decimal numeric constant, regardless of the current radix.
>
> The default input radix (or base) for all constants is decimal. The .RADIX pseudo-op allows you to change the input radix to any base in the range 2 to 16.
>
> .RADIX does not change the radix of the listing; rather, it allows you to input numeric values in the radix you choose without special notation. (Values in other radixes still require the special notations described in Section 3.4.1.) Values in the generated code remain in hexadecimal radix.

EXAMPLE:

```
                    DEC:        DB      20
                                .RADIX 2
                    BIN:        DB      00011110
                                .RADIX 16
                    HEX:        DB      0CF
                                .RADIX 8
                    OCT:        DB      73
                                .RADIX 10
                    DECI:       DB      16
                    HEXA:       DB      0CH
```

assembles as:

```
0000'   14          DEC:        DB      20
0002                             .RADIX 2
0001'   1E          BIN:        DB      00011110
0010                             .RADIX 16
0002'   CF          HEX:        DB      0CF
0008                             .RADIX 8
0003'   3B          OCT:        DB      73
000A                             .RADIX 10
0004'   10          DECI:       DB      16
0005'   0C          HEXA:       DB      0CH
```

Request

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

>   When you run LINK-80, .REQUEST sends a request to
>   the LINK-80 linking loader to search the filenames
>   in the list for undefined external symbols. If
>   LINK-80 finds any undefined external symbols
>   (external symbols for which a corresponding PUBLIC
>   symbol is not currently loaded), you will know that
>   you need to load one or more additional modules to
>   complete linking.
>
>   The filenames in the list should be in the form of
>   legal symbols. <filename> should not include a
>   filename extension or device designation. LINK-80
>   assumes the default extension (.REL) and the
>   currently logged disk drive.

>   EXAMPLE:

>   .
>   .
>   .
>   .REQUEST   SUBR1
>   .
>   .
>   .

>   LINK-80 will search SUBR1 for external symbols
>   which do not have corresonding PUBLIC symbol
>   definitions declared among the currently loaded
>   modules.

LISTING

Listing pseudo-ops perform two general functions: format
control and listing control.   Format control pseudo-ops
allow the programmer to insert page breaks and direct page
headings.   Listing control pseudo-ops turn on and off the
listing of all or part of the assembled file.

Format Control

These pseudo-ops allow you to direct page breaks, titles, and subtitles on your program listings.

Form Feed

*   *EJECT [<exp>]
    PAGE <exp>
    $EJECT

>   The form feed pseudo-ops cause the assembler to
>   start a new output page. The assembler puts a form
>   feed character in the listing file at the end of
>   the 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.
>
>
>   EXAMPLE:
>
>
>   .
>   .
>   .
>   *EJECT 58
>   .
>   .
>   .
>
>
>   The assembler causes the printer to start a new
>   page every time 58 lines of program have been
>   printed.

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-op is used. (If neither a TITLE nor a
> NAME pseudo-op is used, the module name is created
> from the source filename.)

> EXAMPLE:

>> TITLE PROG1
>> .
>> .
>> .

> The module name is now PROG1. The module may be
> called by this name, which will be printed at the
> top of every listing page.

Subtitle

```
SUBTTL <text>
$TITLE ('<text>')
```

> SUBTTL specifies a subtitle to be listed in each
> page heading on the line after the title. The
> <text> is truncated after 60 characters.
>
> Any number of SUBTTLs may be given in a program.
> Each time the assembler encounters SUBTTL, it
> replaces the <text> from the previous SUBTTL with
> the <text> from the most recently encountered
> SUBTTL. To turn off SUBTTL for part of the output,
> enter a SUBTTL with a null string for <text>.
>
> EXAMPLE:
>
>   SUBTTL SPECIAL I/O ROUTINE
>   .
>   .
>   .
>   SUBTTL
>   .
>   .
>   .
>
> The first SUBTTL causes the subtitle SPECIAL I/O
> ROUTINE to be printed at the top of every page.
> The second SUBTTL turns off subtitle (the subtitle
> line on the listing is left blank).

General Listing Control

.LIST   - List all lines with their code
.XLIST - Suppress all listing

>   .LIST is the default condition. If you specify a
>   listing file in the command line, the file will be
>   listed.
>
>   When .XLIST is encountered in the source file,
>   source and object code will not be listed. .XLIST
>   remains in effect until a .LIST is encountered.
>
>   .XLIST overrides all other listing control
>   pseudo-ops. So, nothing will be listed, even if
>   another listing pseudo-op (other than .LIST) is
>   encountered.
>
>   EXAMPLE:
>
>   .
>   .
>   .
>   .XLIST   ;listing suspended here
>   .
>   .
>   .
>   .LIST   ;listing resumes here

Print At Terminal

.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.
>
> .PRINTX will output on both passes. If only one
> printout is desired, use the IF1 or IF2 pseudo-op,
> depending on which pass you want displayed. See
> the Conditional pseudo-ops for IF1 and IF2.
>
>
> EXAMPLE:
>
>   .PRINTX *Assembly half done*
>
> The assembler will send this message to the
> terminal screen when encountered.
>
> IF1
> .PRINTX *Pass 1 done*   ;pass 1 message only
> ENDIF
>
> IF2
> .PRINTX *Pass 2 done*   ;pass 2 message only
> ENDIF

Conditional Listing Control

The three conditional listing control pseudo-ops are used to specify whether or not you wish statements contained within a false conditional block to appear on the listing. See also the description of the /X switch in Chapter 5.

Suppress False Conditionals

.SFCOND

> .SFCOND suppresses the portion of the listing that contains conditional expressions that evaluate as false.

List False Conditionals

.LFCOND

> .LFCOND assures the listing of conditional expressions that evaluate false.

Toggle False Listing Conditional

.TFCOND

> .TFCOND toggles the current setting. .TFCOND operates independently from .LFCOND and .SFCOND. .TFCOND toggles the default setting, which is set by the presence or absence of the /X switch in the assembler command line. When /X is present, .TFCOND will cause false conditionals to list. When /X is not given, .TFCOND will suppress false conditionals.

Macro Expansion Listing Control

Expansion listing pseudo-ops control the listing of lines
inside macro and repeat pseudo-op (REPT, IRP, IRPC) blocks,
and may be used only inside a macro or repeat block.

Exclude Non-code Macro Lines

.XALL

> .XALL is the default.

> .XALL lists source code and object code produced by
> a macro, but source lines which do not generate
> code are not listed.

List Macro Text

.LALL

> .LALL lists the complete macro text for all
> expansions, including lines that do not generate
> code.

Suppress Macro Listing

.SALL

> .SALL suppresses listing of all text and object
> code produced by macros.

CREF Listing Control Pseudo-ops

You may want the option of generating a cross reference listing for part of a program but not all of it. To control the listing or suppressing of cross references, use the cross reference listing control pseudo-ops, .CREF and .XCREF, in the source file for MACRO-80. These two pseudo-ops may be entered at any point in the program in the OPERATOR field. Like the other listing control pseudo-ops, .CREF and .XCREF support no ARGUMENTs.

Suppress Cross References

.XCREF

> .XCREF turns of the .CREF (default) pseudo-op. .XCREF remains in effect until MACRO-80 encounters .CREF. Use .XCREF to suppress the creation of cross references in selected portions of the file. Because neither .CREF nor .XCREF takes effect until the /C switch is set in the MACRO-80 command line, there is no need to use .XCREF if you want the usual List file (one without cross references); simply omit /C from the ALDS assembler command line.

List Cross References

.CREF

> .CREF is the default condition. Use .CREF to restart the creation of a cross reference file after using the .XCREF pseudo-op. .CREF remains in effect until MACRO-80 encounters .XCREF. Note, however, that .CREF has no effect until the /C switch is set in the MACRO-80 command line.

4.2 MACRO FACILITY

The macro facility allows you to write blocks of code which can be repeated without recoding. The blocks of code begin with either the macro definition pseudo-op or one of the repetition pseudo-ops and end with the ENDM pseudo-op. All of the macro pseudo-ops may be used inside a macro block. In fact, nesting of macros is limited only by memory.

The macro facility of the MACRO-80 macro assembler includes pseudo-ops for:

        macro definition:
            MACRO

        repetitions:
            REPT   (repeat)
            IRP   (indefinite repeat)
            IRPC   (indefinite repeat character)

        termination:
            ENDM
            EXITM

        unique symbols within macro blocks:
            LOCAL

The macro   facility also supports some   special   macro operators:
            &
            ;;
            !
            %

Macro Definition

```
<name> MACRO <dummy>[,<dummy>...]
            .
            .
            .
        ENDM
```

The block of statements from the MACRO statement line to the ENDM statement line comprises the body of the macro, or the macro's definition.

<name> is like a LABEL and conforms to the rules for forming symbols. Note that <name> may be any length, but only the first 16 characters are passed to the linker. After the macro has been defined, <name> is used to invoke the macro.

A <dummy> is a place holder that is replaced by a parameter in a one-for-one text substitution when the MACRO block is used. Each <dummy> may be up to 32 characters long. The number of dummys is limited only by the length of a line. If you specify more than one dummy, they must be separated by commas. MACRO-80 interprets all characters between commas as a single dummy.

NOTE

A dummy is always recognized exclusively as a dummy. Even if a register name (such as A or B) is used as a dummy, it will be replaced by a parameter during expansion.

A macro block is not assembled when it is encountered. Rather, when you call a macro, the assembler "expands" the macro call statement by bringing in and assembling the appropriate macro block.

If you want to use the TITLE, SUBTTL, or NAME pseudo-ops for the portion of your program where a macro block appears, you should be careful about the form of the statement. If, for example, you enter SUBTTL MACRO DEFINITIONS, MACRO-80 will assemble the statement as a macro definition with SUBTTL as the macro name and DEFINITIONS as the dummy. To avoid this problem, alter the word MACRO in some way; e.g., - MACRO, MACROS, and so on.

Calling a Macro

To use a macro, enter a macro call statement:

<name> <parameter>[,<parameter>...]

<name> is the <name> of the MACRO block. A <parameter> replaces a <dummy> on a one-for-one basis. The number of parameters is limited only by the length of a line. If you enter more than one parameter, they must be separated by commas. If you place angle brackets around parameters separated by commas, the assembler will pass all the items inside the angle brackets as a single parameter. For example:

FOO 1,2,3,4,5

passes five parameters to the macro, but:

FOO <1,2,3,4,5>

passes only one.

The number of parameters in the macro call statement need not be the same as the number of dummys in the MACRO definition. If there are more parameters than dummys, the extras are ignored. If there are fewer, the extra dummys will be made null. The assembled code will include the macro block after each macro call statement.

EXAMPLE:

```
EXCHNG      MACRO       X,Y
            PUSH        X
            PUSH        Y
            POP         X
            POP         Y
            ENDM
```

If you then enter as part of a program some code and a macro call statement:

```
            LDA         2FH
            MOV         HL,A
            LDA         3FH
            MOV         DE,A
            EXCHNG      HL,DE
```

assembly generates the code:

```
0000'   3A 002F                    LDA      2FH
0003'   67                         MOV      HL,A
0004'   3A 003F                    LDA      3FH
0007'   57                         MOV      DE,A
                                   EXCHNG   HL,DE
0008'   E5            +            PUSH     HL
0009'   D5            +            PUSH     DE
000A'   E1            +            POP      HL
000B'   D1            +            POP      DE
```

Repeat Pseudo-ops

The pseudo-ops in this group allow the operations in a block
of code to be repeated for the number of times you specify.
The major differences between the Repeat pseudo-ops and
MACRO pseudo-op are:

1.   MACRO gives the block a name by which to call in
     the code wherever and whenever needed; the macro
     block can be used in many different programs by
     simply entering a macro call statement.

2.   MACRO allows parameters to be passed to the MACRO
     block when a MACRO is called; hence, parameters
     can be changed.

Repeat pseudo-op parameters must be assigned as a part of
the code block. If the parameters are known in advance and
will not change, and if the repetition is to be performed
for every program execution, then Repeat pseudo-ops are
convenient. With the MACRO pseudo-op, you must call in the
MACRO each time it is needed.

Note that each Repeat pseudo-op must be matched with the
ENDM pseudo-op to terminate the repeat block.

Repeat

REPT <exp>
   .
   .
   .
ENDM

Repeat block of statements between REPT and ENDM
<exp> times. <exp> is evaluated as a 16-bit
unsigned number. If <exp> contains an External
symbol or undefined operands, an error is
generated.

EXAMPLE:

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

assembles as:

```
0000        X    SET    0
                 REPT   10    ;generates DB 1 - DB 10
            X    SET    X+1
                 DB     X
                 ENDM
0000'  01   +    DB     X
0001'  02   +    DB     X
0002'  03   +    DB     X
0003'  04   +    DB     X
0004'  05   +    DB     X
0005'  06   +    DB     X
0006'  07   +    DB     X
0007'  08   +    DB     X
0008'  09   +    DB     X
0009'  0A   +    DB     X
                 END
```

Indefinite Repeat

```
IRP <dummy>,<parameters inside angle brackets>
  .
  .
  .
ENDM
```

Parameters must be enclosed in angle brackets.
Parameters may be any legal symbol, string,
numeric, or character constant. The block of
statements is repeated for each parameter. Each
repetition substitutes the next parameter for every
occurrence of <dummy> in the block. If a parameter
is null (i.e., <>), the block is processed once
with a null parameter.

EXAMPLE:

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

This example generates the same bytes (DB 1 - DB
10) as the REPT example.

When IRP is used inside a MACRO definition block,
angle brackets around parameters in the macro call
statement are removed before the parameters are
passed to the macro block. An example, which
generates the same code as above, illustrates the
removal of one level of brackets from the
parameters:

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

When the macro call statement

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

is assembled, the macro expansion becomes:

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

The angle brackets around the parameters are
removed, and all items are passed as a single
parameter.

Indefinite Repeat Character

```
IRPC <dummy>,<string>
   .
   .
   .
ENDM
```

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.

EXAMPLE:

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

This example generates the same code (DB 1 - DB 10) as the two previous examples.

Termination

End Macro

ENDM

> ENDM tells the assembler that the MACRO or Repeat block is ended.
>
> Every MACRO, REPT, IRP, and IRPC 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.
>
> If you wish to be able to exit from a MACRO or repeat block before expansion is completed, use EXITM.

Exit Macro

EXITM

> The EXITM pseudo-op is used inside a MACRO or Repeat block to terminate an expansion when some condition makes the remaining expansion unnecessary or undesirable. Usually EXITM is used in conjunction with a conditional pseudo-op.
>
> When an EXITM is assembled, the expansion is exited immediately. 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.
>
> EXAMPLE:
>
> ```
>         FOO     MACRO   X
>         Y       SET     0
>                 REPT    X
>         Y       SET     Y+1
>                 IFE     Y-0FFH  ;test Y
>                 EXITM           ;if true, exit REPT
>                 ENDIF
>                 DB      Y
>                 ENDM
>                 ENDM
> ```

Macro Symbol

LOCAL <dummy>[,<dummy>...]

The LOCAL pseudo-op is allowed only inside a MACRO
definition block. When LOCAL is executed, the
assembler creates a unique symbol for each <dummy>
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 multiple-defined labels on
successive expansions of the macro. The symbols
created by the assembler range from ..0001 to
..FFFF. Users should avoid the form ..nnnn for
their own symbols. A LOCAL statement must precede
all other types of statements in the macro
definition.

EXAMPLE:

```
                    FOO         MACRO   NUM,Y
                                LOCAL   A,B,C,D,E
                    A:          DB      7
                    B:          DB      8
                    C:          DB      Y
                    D:          DB      Y+1
                    E:          DW      NUM+1
                                JMP     A
                                ENDM
                                FOO     0C00H,0BEH
                                END
```

generates the following code (notice that MACRO-80
has substituted LABEL names in the form ..nnnn for
the instances of the dummy symbols):

```
                    FOO         MACRO   NUM,Y
                                LOCAL   A,B,C,D,E
                    A:          DB      7
                    B:          DB      8
                    C:          DB      Y
                    D:          DB      Y+1
                    E:          DW      NUM+1
                                JMP     A
                                ENDM
                                FOO     0C00H,0BEH
0000'   07          +..0000:    DB      7
0001'   08          +..0001:    DB      8
0002'   BE          +..0002:    DB      0BEH
0003'   BF          +..0003:    DB      0BEH+1
0004'   0C01        +..0004:    DW      0C00H+1
0006'   C3 0000'    +           JMP     ..0000
                                END
```

Special Macro Operators

Several special operators can be used in a macro block to select additional assembly functions.

    &   Ampersand concatenates text or symbols. (The & may not be used in a macro call statement.) A dummy parameter in a quoted string will not be substituted in expansion unless preceded immediately 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
```

The call ERRGEN A will then generate:

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

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

   !   An exclamation point may be entered in an argument to indicate that the next character is to be taken literally. Therefore, !; is equivalent to <;>.

   %   The percent sign is used only in a macro argument to convert the expression that follows it (usually a symbol) to a number in the current radix (set by the .RADIX pseudo-op). During macro expansion, the number derived from converting the expression is substituted for the dummy. Using the % special operator allows a macro call by value. (Usually, a macro call is a call by reference with the text of the macro argument substituting exactly for the dummy.)

The expression following the % must conform to the
same rules as expressions for the DS (Define Space)
pseudo-op.   That is,   a valid expression   that
evaluates to an absolute (non-relocatable) constant
is required.

'EXAMPLE:

```
PRINTE      MACRO      MSG,N
            .PRINTX    * MSG,N *
            ENDM
SYM1        EQU        100
SYM2        EQU        200
            PRINTE <SYM1 + SYM2 = >,%(SYM1 + SYM2)
```

Normally, the macro call statement would cause the
string (SYM1 + SYM2) to be substituted for the
dummy N.   The result would be:

```
    .PRINTX    * SYM1 + SYM2 = (SYM1 + SYM2)
```

When the % is placed in front of the parameter, the
assembler generates:

```
    .PRINTX    * SYM1 + SYM2 = 300 *
```

4.3 CONDITIONAL ASSEMBLY FACILITY

Conditional pseudo-ops allow users to design blocks of code
which test for specific conditions then proceed accordingly.

All conditionals follow the format:

```
IFxxxx [argument]        COND [argument]
   .                        .
   .                        .
   .                        .
[ELSE                    [ELSE
   .                        .
   .                        .
   . ]                      .]
ENDIF                    ENDC
```

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

The assembler evaluates the conditional statement to TRUE
(which equals FFFFH, or -1, or any non-zero value), or to
FALSE (which equals 0000H).   The code in the conditional
block is assembled if the evaluation matches the condition
defined in the conditional statement.   If the evaluation
does not match, the assembler either ignores the conditional
block completely or, if the conditional block contains the
optional ELSE statement, assembles only the ELSE portion.

Conditionals may be nested up to 255 levels.   Any argument
to a conditional must be known on pass 1 to avoid V errors
and incorrect evaluation. For IF/IFT/COND and IFF/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.

Each conditional block may include the optional ELSE
pseudo-op, which allows alternate code to be generated when
the opposite condition exists. Only one ELSE is permitted
for a given IFxxxx/COND.   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.

Conditional Pseudo-ops

        IF <exp>
        IFT <exp>
    *   COND <exp>

            If <exp> evaluates to not-0, the statements within
            the conditional block are assembled.

IFE <exp>
IFF <exp>

            If <exp> evaluates to 0, the statements in the
            conditional block are assembled.

IF1   Pass 1 Conditional

            If the assembler is in pass 1, the statements in
            the conditional block are assembled.

IF2   Pass 2 Conditional

            If the assembler is in pass 2, the statements in
            the conditional block are assembled.

IFDEF <symbol>

            If the <symbol> is defined or has been declared
            External, the statements in the conditional block
            are assembled.

IFNDEF <symbol>

            If the <symbol> is not defined or not declared
            External, the statements in the conditional block
            are assembled.

IFB <arg>

            The angle brackets around <arg> are required.

            If the <arg> is blank (none given) or null (two
            angle brackets with nothing in between, <>), the
            statements in the conditional block are assembled.

IFNB <arg>

      The angle brackets around <arg> are required.

      If <arg> is not blank, the statements in the conditional block are assembled. Used for testing for dummy parameters.

IFIDN <arg1>,<arg2>

      The angle brackets around <arg1> and <arg2> are required.

      If the string <arg1> is identical to the string <arg2>, the statements in the conditional block are assembled.

IFDIF <arg1>,<arg2>

      The angle brackets around <arg1> and <arg2> are required.

      If the string <arg1> is different from the string <arg2>, the statements in the conditional block are assembled.

ELSE

      The ELSE pseudo-op allows you to generate alternate code when the opposite condition exists. May be used with any of the conditional pseudo-ops.

    ENDIF
*   ENDC

      These pseudo-ops terminate conditional blocks. A terminate pseudo-op must be given for every conditional pseudo-op used. ENDIF must be matched with an IFxxxx pseudo-op. ENDC must be matched with the COND pseudo-op.

Contents

Chapter 5   Running MACRO-80

CHAPTER 5

RUNNING MACRO-80

When you have completed creating the assembly language source file, you are ready to assemble it. MACRO-80 assembles the source file statements, including expanding macros and repeat pseudo-ops. The result of assembly will be relocatable object code which is ready to link and load with LINK-80. The relocatable object code can be saved in a disk file, which the assembler gives the filename extension .REL. The assembled (REL) file is not an executable file. The file will be executable only after it is processed through LINK-80.

MACRO-80 resides in approximately 19K of memory and has an assembly rate of over 1000 lines per minute. MACRO-80 runs under the CP/M operating system.

MACRO-80 assembles your source file in two passes. During pass 1, MACRO-80 evaluates the program statements, calculates how much code it will generate, builds a symbol table where all symbols are assigned values, and expands macro call statements. During pass 2, MACRO-80 fills in the symbol and expression values from the symbol table, again expands macro call statements, and emits the relocatable code. MACRO-80 checks the values of symbols, expressions, and macros during both passes. If a value during pass 2 is different from the value during pass 1, MACRO-80 returns a phase error code.

Before MACRO-80 can be run, the diskette which contains MACRO-80 must be inserted in the appropriate disk drive. The diskette on which you created the source file must also be in a disk drive.

5.1   INVOKING MACRO-80

To invoke MACRO-80, enter:

        M80

The program file M80.COM will be loaded. MACRO-80 will
display an asterisk (*) to indicate that the assembler is
ready to accept a command line.

5.2   MACRO-80 COMMAND LINE

The command line for MACRO-80 consists of four fields,
labeled:

        Object,List=Source/Switch

The command line may be entered on its own line, or it may
be entered at the same time as the M80 command. (If M80 and
the command line are entered on one line, MACRO-80 will not
return the asterisk prompt.) Entering the command line on
its own line allows single drive configurations to use
MACRO-80. In addition, by entering M80 and the command line
separately, you are able to perform another assembly without
reinvoking MACRO-80. When assembly is finished, MACRO-80
will return the asterisk (*) prompt and wait for another
command line. To exit MACRO-80 when you have entered M80
and the command line separately, type <CTRL-C>.

If you are performing only one assembly, entering the
command line on the same line as M80 is convenient; it
requires less typing and allows the assembly operation to be
part of a SUBMIT command. When you enter M80 and the
command line together, MACRO-80 exits automatically to the
operating system.

                            NOTE

                    If you enter M80 and the
                    command line separately, you
                    must enter the command line in
                    upper case only. If you do
                    not, MACRO-80 will return a
                    ?Command Error message. If
                    you enter M80 and the command
                    line on one line, the entries
                    may be in either upper or
                    lower case (or mixed) because
                    CP/M converts all entries to
                    upper case before passing the
                    entries.

Source (=filename)

To assemble your source program, you must enter at least an
equal sign (=) and the source filename.

The =filename indicates which source file you want to
assemble. If the source file disk is not in the currently
logged drive, you must include the drive designation as part
of the filename. If the source filename is entered without
an extension, MACRO-80 assumes that the extension is .MAC.
If the extension is not .MAC, you must include the extension
as part of the filename. For other possibilities for
drive/device designations and filename extensions, see the
Additional Command Line Entries section, below.)

The Source entry is the only entry required besides M80.

The simplest command is:

        M80 =Source

This command directs MACRO-80 to assemble the source file
and save the result in a relocatable object file (called a
REL file) with the same name as the source file. If the
source file is NEIL.MAC, the command line:

        M80 =NEIL

generates an assembled file named NEIL.REL.

An additional option is to enter only a comma (,) to the
left of the equal sign. When MACRO-80 sees a comma as the
first entry after the M80 entry, it suppresses all output
files (Object and List). The command line

        M80 ,=NEIL

causes MACRO-80 to assemble the file NEIL.MAC, but no output
files are created. Programmers use this command line to
check syntax in the source program before saving the
assembled program. Because no files are generated, the
assembly is completed faster and errors are known sooner.

Object (filename)

An Object entry is always optional. However, certain
circumstances will compel you to make some entry for the
Object.

The Object file saves the assembled program in a disk file.
LINK-80 uses the Object file to create an executable
program. If both Object and List entries are omitted from a
command line (as in =Source), MACRO-80 will generate an
Object file with the same filename as the Source, but with
the default extension .REL.

If you want your Object file to have a name different from
the source file, you must enter a filename in the Object
field. MACRO-80 will still append the filename extension
.REL, unless you also enter an extension.

Also, if you want both a List file and a REL file generated,
you must enter a filename for the Object, even if you want
the REL file named after the source file. If you enter a
filename for the List but omit the Object, no REL file will
be generated. Programmers do use this feature for checking
the program for errors before final assembly. The program
listing aids debugging.

The name for the Object file may be the same as the source
filename or any other legal filename you choose. Since it
is practical to have all files which relate to a program
carry some mutual indication of their relationship, most
often you will want to give your object file the same name
as your source file.

List (,filename)

A List entry is always optional. The comma is required in
front of all List entries. If you want a List file, enter a
,filename for the List. (There is an alternative to this
rule.   See the Switches section below for discussion of the
/L switch.)

MACRO-80 appends the default extension .PRN to the List file
unless you specify a different extension in the List entry.

The command line:

        M80 ,NEIL=NEIL

assembles the file NEIL.MAC (source file) and creates the
List file NEIL.PRN. An Object (REL) file is not created.

The name may be the same as the source filename or any other
legal filename you choose. Since it is practical to have
all files which relate to a program carry some mutual
indication of their relationship, most often you will want
to give your listing file the same name as your source file.

Avoid entering only a comma for the List after entering a
filename for the Object. For example:

        M80 NEIL,=NEIL

MACRO-80 will probably ignore the comma and assemble the
source file into a REL file. It is possible that MACRO-80
might return a COMMAND ERROR message.

If you enter only a comma for the List and nothing for the
Object, MACRO-80 will assemble the source file, but will
generate no output files. This command

        M80 ,=Source

allows you to check the source program for syntax errors
before saving the assembled program in a disk file. While
MACRO-80 always checks for errors, this command   form
provides much faster assembly because the output files do
not have to be created.

At the end of assembly, MACRO-80 will print the message:

        [xx][No] Fatal errors [,xx warnings]

This message reports the number of fatal errors and warning
errors encountered in the program. The message is listed at
the end of every assembly on the terminal screen and in the
listing file.   When the message appears, the assembler has
finished. When the message No Fatal Errors appears, the
assembly is complete and successful.

Switches (/Switch)

You can command MACRO-80 to perform some additional functions besides assembling and creating object and listing files. These additional commands are given to MACRO-80 as entries at the end of the command line. A Switch entry directs MACRO-80 to "switch on" some additional or alternate function; hence, these entries are called switches. Switches are letters preceded by slash marks (/). Any number of switches may be entered, but each switch must be preceded by a slash. For example:

        M80 ,=NEIL/L/R

The available switches for MACRO-80 are:

Switch   Action

   /O   Octal listing. MACRO-80 generates List file
            addresses in octal radix.

   /H   Hexadecimal listing. MACRO-80 generates List file
            addresses in hexadecimal. This is the default.

   /R   Force generation of an Object file with the same
            name as the source file. May be used instead of
            giving a filename in the Object field of the
            command line.

            This switch is convenient when you want a REL file
            but forgot to enter a filename in the Object field
            and entered a comma and filename or a comma only
            in the List field. (Remember: if no filenames or
            comma is entered before the equal sign, a REL file
            is generated.) Thus, if you enter

                M80 ,NEIL=NEIL
             or M80 ,=NEIL

            then decide, before pressing <ENTER>, that you
            want a REL file, simply add /R. The command line
            would then be:

                M80 ,NEIL=NEIL/R
             or M80 ,=NEIL/R

/L   Force generation of a listing file with   the   same
            name   as   the source file. May be used instead of
            giving a filename in the List field of the command
            line.

            This switch is convenient when you   want   a   List
            file   but   forgot   to enter a filename in the List
            field.   If you enter the command line:

                M80 =NEIL
             or M80 ,=NEIL
             or M80 NEIL=NEIL

            then decide, before pressing <ENTER>, that you   do
            want   a   List   file,   simply   add /L. The command
            would then be:

                M80 =NEIL/L
             or M80 ,=NEIL/L
             or M80 NEIL=NEIL/L

/C   Causes MACRO-80 to generate a   special   List   file
            (with   the   same   name as the Source file) for use
            with CREF-80 Cross   Reference   Facility.   If   you
            want   to   use CREF-80, you must assemble your file
            with this switch set.   See   Chapter   8,   CREF-80
            Cross Reference Facility, for further details.

/Z   Directs MACRO-80 to assemble Z80 opcodes. If your
            source file contains Z80 opcodes and if you do not
            include the .Z80 pseudo-op in   your   source   file,
            then   you   must use the /Z switch at assembly time
            so that MACRO-80 will assemble the Z80 opcodes.

/I   Directs MACRO-80 to   assemble   8080   opcodes.   If
            your   source file contains 8080 opcodes and if you
            do not include the .8080 pseudo-op in your   source
            file,   then you must use the /I switch at assembly
            time so   that   MACRO-80   will   assemble   the   8080
            opcodes.   (Default)

/P   Each /P allocates an   extra   256   bytes   of   stack
            space   for   use   during assembly. Use /P if stack
            overflow errors occur during assembly. Otherwise,
            /P is not needed.

/M   The /M switch initializes Block data areas. If
            you want the area that is defined by the DS
            (Define Space) pseudo-op initialized to zeros,
            then you should use the /M switch in the command
            line. Otherwise, the space is not guaranteed to
            contain zeros. That is, DS does not automatically
            initialize the space to zeros, in which case you
            may not know what is stored in the DS space or how
            the program will be affected.

/X   The /X switch sets the default and current setting
            to suppress the listing of false conditionals.
            Absence of /X in the command line sets the default
            and current setting to list conditional blocks
            which evaluate false. /X is often used in
            conjunction with the conditional listing pseudo-op
            .TFCOND. Refer to the Listing Pseudo-ops section
            in Chapter 4 for details.

Additional Command Line Entries

Each command line field supports two additional types of entries--filename extensions and device designations. These two types of entries are actually part of a "file specification." A file specification includes the device where a file is located, the name of the file, and the filename extension.

Usually, filename extensions and device designations are handled by defaults--the MACRO-80 program "inserts" these entries if their positions are left blank in a command line. The default assignments in no way prevent you from entering either filename extensions or device designations, including entries that match the default entries. The programmer may enter or omit these additional entries in any combination.

The format for a file specification under MACRO-80 is:

        dev:filename.ext

where: dev: is a 1-3 letter device designation followed by
         a (required) colon.

         filename is a 1-8 letter filename.

         .ext is a 1-3 character filename extension preceded
         by a (required) period.

Filename Extensions (.ext)

To distinguish between Source file, Object file, and List
file, MACRO-80 appends an extension to each filename.
Filename extensions are three-letter mnemonics appended to
the filename with a period (.) between the filename and the
extension. The extension which MACRO-80 appends reflects
the type of file. Since the extensions are supplied by
MACRO-80, they are called default extensions. The default
extensions which MACRO-80 supplies are:

> .REL   Relocatable object file
> .PRN   Listing file
> .COM   Absolute (executable object) file

Also, MACRO-80 assumes that, if no filename extension is
entered, a source file carries the filename extension .MAC.

You may supply your own extensions, if you find this
necessary or desirable. The disadvantage is that whenever
you call the file, you must always remember to include your
extension. Also, you must remember what type of file it
is--relocatable, source, absolute, etc. The advantage of
allowing MACRO-80 to assign default extensions is that you
always have a mnemonic indication of the type of file, and
you can call the filename without appending the extension,
in most cases.

Device Designations (dev:)

Each of the fields in a command line (except Invocation) also may include a device designation.

When a device designation is specified in the Source field, the designation tells MACRO-80 where to find the source file. When a device designation is specified in the Object or List fields, the designation tells MACRO-80 where to output the object or list file. If the device designation is omitted from any of these fields, MACRO-80 assumes (defaults to) the currently logged drive. Thus, any time the device designation is the currently logged drive or device, the device designation need not be specified.

It is important to include device designations if several devices or drives will be used during an assembly. For example, if your ALDS diskette is in drive A and your program diskette is in drive B, and you want your REL file output to drive B, you need to give the command line:

        M80 =B:NEIL

When the REL file is output, the currently logged drive is drive B. (However, when MACRO-80 is finished, drive A will be the currently logged drive again.) In contrast, if you saved your source program on the MACRO-80 diskette in drive A and want the REL file output to a diskette in drive B, then you need to enter the command line:

        M80 B:=A:NEIL

As a rule of thumb, if you are not sure if you need to include the device designation (especially the drive designation), enter a designation; it is the one sure way to get the right files in the right places.

The available device designations for MACRO-80 are:

        A:, B:, C:,...   Disk drives
        LST: Line Printer
        TTY: Terminal Screen or Keyboard
        HSR: High Speed Reader

Device Designations as Filenames

As an option, you may enter a device designation only in the
command line fields in place of a filename. The use of this
option gives various results depending on which device is
specified and in which field the device is specified. For
example:

        M80 ,TTY:=TTY:

allows you to assemble a line immediately on input to check
for syntax or other errors. A modification of this command
(that is, M80 ,LST:=TTY:), provides the same result but
printed on a line printer instead of the terminal screen.

If you use either of these commands (,TTY:=TTY: or
,LST:=TTY:), your first entry should be an END statement.
You need to put the assembler into pass 2 before it will
emit the code. If you simply start entering statement lines
without first entering END, you will receive no response
until an END statement is entered. Then you will have to
reenter all your statements before you see any code
generated.

The following table illustrates the results of the various
choices. The table is meant to indicate the possibilities
rather than provide an exhaustive list of the combinations.

dev:	Object	,List	=Source
A:, B:, C:, D:	write file to drive specified	write file to drive specified	N/A (a filename must be specified)
HSR:	N/A (input only)	N/A (input only)	reads source program from high-speed reader
LST:	N/A (unreadable file format)	writes listing to line printer	N/A (output only)
TTY:	N/A (unreadable file format)	"writes" listing to screen	"reads" source program from keyboard

Figure 5.1: Effects of Device Designations without Filenames

5.3   MACRO-80 LISTING FILE FORMATS

A listing of a MACRO-80 file displays the two parts of the
file in two different formats. One format displays the file
lines. The second format displays symbol table listings.

File Format

Each page of a MACRO-80 listing prints header data in the
first two lines.   If no header data were commanded in the
source file (neither the TITLE nor SUBTTL pseudo-op was
given), those portions of the header lines are left blank.

The format is:

           [TITLE text]   M80 z.zz   PAGE x
           [SUBTTL text]

where: TITLE text is the text supplied with the .TITLE
         pseudo-op, if .TITLE was included in the source
         file. If no .TITLE pseudo-op was given in the
         source file, this space is left blank.

         z.zz is the version number of your MACRO-80 program.

         x is the page number, which is shown and incremented
         only when a .PAGE pseudo-op is encountered in the
         source file, or whenever the current page size has
         been filled.

         SUBTTL text is the text supplied with the .SUBTTL
         pseudo-op, if .SUBTTL was included in the source
         file. If no .SUBTTL was given in the source file,
         this space is left blank.

A blank line follows the header data.   The text of the
listing file begins on the next line.

The format of a listing line is:

           [error] ####m xx xxxxm[w]   text

where: error represents a one-letter error code. An error
         code is printed only if the line contains an error.
         Otherwise, the space is left blank.

         #### represents the location counter. The number is
         a 4-digit hexadecimal number or a 6-digit octal
         number. The radix of the location counter number is
         determined by the use of the /O or /H switch in the
         MACRO-80 command line Switch field.   If no radix
         switch was given, the default radix is hexadecimal
         (4-digit).

m represents the PC mode indicator character. The
possible symbols are:

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

xx and xxxx represent the assembled code. xx
represents a one-byte value. One-byte values are
always followed immediately by a space. xxxx
represents a two-byte value, with the high-order
byte printed first (this is the opposite of the
order in which they are stored). Two-byte values
are followed by one of the mode indicators discussed
above (indicated by the second m).

[w] represents a line in the MACRO-80 file that came
from another file through an INCLUDE pseudo-op; or
a line that is part of an expansion (MACRO, REPT,
IRP, IRPC). For lines from an INCLUDE statement, a
C is printed following the assembled code; for
lines in an expansion, a plus sign (+) is printed
following the assembled code. Otherwise, this space
is blank.

text represents the rest of the line, including
labels, operations, arguments, and comments.

Symbol Table Format

The symbol table listing page follows the same header data
format as the file line pages. However, instead of a page
number, the symbol table page shows PAGE S.

Then, in a symbol table listing, all macro names in a
program are listed alphabetically. Next, all symbols are
listed, also alphabetically. A tab follows each symbol,
then the value of the symbol is printed. Each symbol value
is followed by one of the following characters:

I   PUBLIC symbol

U   Undefined symbol

C   COMMON block name. The value shown for the
         COMMON block name is its length in bytes in
         hexadecimal or octal radix.

*   External symbol

<space> Absolute value

    '   Program relative value

    "   Data relative value

    !   COMMON relative value

5.4   ERROR CODES AND MESSAGES

Errors encountered during assembly cause MACRO-80 to return
either an error code or an error message. Error codes are
one-character flags printed in column one of the listing
file.   If a listing file is not being printed on the
terminal screen,   the   lines   containing   errors   will
nevertheless be printed on the terminal screen. Error
messages are printed at the end of the listing file, or, if
the listing file is not being displayed on the terminal
screen, any error messages will be displayed at the end of
the error code lines.

ERROR
CODE   MEANING

  A   Argument error.
        The argument to a pseudo-op is not in correct
        format or is out of range.

  C   Conditional nesting error.
        ELSE without IF, ENDIF without IF, two ELSEs for
        one IF, ENDC without COND.

  D   Double defined symbol.
        Reference to a symbol which has more than one
        definition.

  E   External error.
        Use of an External is illegal in the flagged
        context.   For   example, FOO SET NAME or LXI
        B,2-NAME.

  M   Multiply defined symbol.
        The definition is for a symbol that already has a
        definition.

  N   Number error.
        An error in a number, usually a bad digit.   For
        example, 8Q.

O   Bad opcode or objectionable syntax.
ENDM, LOCAL outside a block; SET, EQU, or MACRO without a name; bad syntax in an opcode; or bad syntax in an expression (for example, mismatched parentheses, quotes, consecutive operators).

P   Phase error.
The value of a label or EQU name is different during pass 2 from its value during pass 1.

Q   Questionable.
Usually, a line is not terminated properly. For example, MOV AX,BX,. This is a warning error.

R   Relocation.
Illegal use of relocation in an 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 some pseudo-ops, a V error is printed for pass 1 then a U error for pass 2. Compare with V error code definition below.

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

ERROR MESSAGES

%No END statement

No END statement: either it is missing or it is not parsed because it is 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.

Symbol table full

As MACRO-80 was building the symbol table, the
memory available was exhausted. The most usual
cause is a large number of macro blocks which also
contain statements for many of the statement lines.
Macro blocks are stored in the symbol table
verbatim, including the comments appended to the
lines inside the macro block. You should check all
macro blocks in the source program. To exclude
comments inside macro blocks from the symbol table,
precede these comments by double semicolons (;;).
This method should free enough space to assemble
your program.

[xx][No] Fatal errors [,xx warnings]

The number of fatal errors and warning errors
encountered in the program. The message is listed
at the end of every assembly on the terminal screen
and in the listing file. When the message appears,
the assembler has finished. When the message No
Fatal Errors appears, the assembly is complete and
successful.

Contents

CHAPTER 6

LINK-80 LINKING LOADER

The .REL files which MACRO-80 creates are not executable.
To make a REL file executable, you need to load and link the
REL file with the LINK-80 linking loader. The result is an
executable object file.

Loading means physically placing the file in memory and
assigning absolute addresses to the code and data in place
of the relative addresses assigned by the assembler. This
is one of the required steps for converting a relocatable
(REL) file into an executable (COM) file.

Linking means that each loaded file (or module) that directs
program flow outside itself (by a CALL, an EXTERNAL symbol,
or an Include) will be "linked" to the module that contains
the corresponding code.

LINK-80 can also save the assembled-and-linked program as an
executable object program on disk in a file with the
extension .COM. Consequently, any time you wish to run your
program, you need only insert the disk which contains your
COM file into an appropriate disk drive and "call" your
program -- a simple process of typing in the filename you
used to save the program, followed by a carriage return.

6.1 INVOKING LINK-80

To invoke LINK-80, enter:

        L80

The program file L80.COM will be loaded. LINK-80 will
display an asterisk (*) to indicate that the linking loader
is ready to accept a command. The REL file(s) you want
link-loaded must be available in a disk drive. If you have
only one drive, you will need to swap diskettes in the drive
at each step of the link-loading process.

6.2 LINK-80 COMMANDS

LINK-80 commands are filenames and switches.

You can enter your commands to LINK-80 one at a time;   or,
you can enter all of your commands (including L80) on one
line.

A command line has a flexible format, allowing you a number
of options for loading and linking files and for performing
other operations. The basic rule for LINK-80 commands is
that files are loaded in the order they are named, beginning
at the (default) address 103H under CP/M. Even though the
files will be loaded in the order entered, you do not have
to enter the files in the order of execution.   LINK-80
places a jump instruction at address 100H-102H which jumps
to the start address of the first instruction to be
executed, regardless of its location in memory.

LINK-80 can perform about eleven different tasks.   Even
though you could use them all, you will rarely use more than
three or four at a time.

When you enter a command to LINK-80, LINK-80 returns an
asterisk (*) prompt that tells you to enter another command.
For example:

        A>L80<RETURN>
        */switch<RETURN>
        *filename<RETURN>
        */switch<RETURN>
        *filename/switch<RETURN>
        */E<RETURN>   (to exit LINK-80)

Note that all of the above lines may be entered as one line.
For example:

        L80 /switch,filename/switch,filename/switch/E<RETURN>

This shows further the flexibility of the LINK-80 command
line.

Although entering each command on a separate line is slow
and tedious, the advantage is, especially if you are new to
a linking loader, that you know at all times what function
LINK-80 is performing.

6.2.1 Filenames

Files processed by LINK-80 are REL files. A filename
commands LINK-80 to load the named file (also called a
module). If any file has been loaded already, a filename
also commands LINK-80 to link the loaded files as required.

Normally each linking session requires <u>at</u> <u>least</u> <u>two</u>
filenames. One filename directs LINK-80 which REL file to
load and link; the other commands LINK-80 to save the
executable code in a file with the specified name.

If you enter only one filename during the link session,
either the COM file will not be saved (in which case you may
have wasted your time), or LINK-80 will return the error
message

                ?NOTHING LOADED

Note, however, that if you enter only one filename followed
by the /G switch, the COM file will not be saved, but the
program will execute as soon as LINK-80 is finished loading
and linking. (Refer to the description of the switches in
the next section.)

You may enter as many filenames as will fit on one line.
The files named may be REL files in different languages
(BASIC, COBOL, FORTRAN, or assembly) or runtime library REL
files for any of the high-level programming languages. (For
exact procedures for high-level language REL files, see the
product manual included with the high-level language
compiler.)

When LINK-80 is finished, the results are saved in the file
named by the programmer in the command line (the filename
followed by a /N -- see below, Section 6.2.2, Switches).
LINK-80 gives this filename the extension .COM.

A filename command in LINK-80 actually means a file
specification. A file specification includes a device
designation, a filename, and a filename extension. The
format of a file specification is:

                dev:filename.ext

LINK-80 defaults the dev: to the default or currently
logged disk drive. LINK-80 defaults the input filename
extension to .REL and the output filename extension to .COM.
You can alter the device designation to any applicable
output device supported by MACRO-80 and/or the filename
extension to any three characters by specifying a device or
a filename extension when you enter a filename command.

6.2.2 Switches

Switches command LINK-80 to perform functions besides
loading and linking. Switches are letters preceded by slash
marks (/). You can place as many switches as you need in a
single command line, but each switch letter must be preceded
by a slash mark (/). For example, if you want to link and
load a program named NEIL, save an image of it on diskette,
then execute the program, you need two filenames and two
switches, so you would enter the commands:

               NEIL,NEIL/N/G<RETURN>

LINK-80 saves a memory image on diskette (the /N switch),
then runs the NEIL program (the /G switch).

Some switches can be entered by themselves (/E, /G, /R, /P,
/D, /U, /M, /O, /H). Some switches must be appended to the
filename they affect (/N, /S). Some switches work only if
other switches are also entered during the LINK-80 session
(/X, /Y). Some switches must precede any filenames you want
affected (/P, /D). Some switches command actions that are
deferred until the end of the LINK-80 session (/N, /X, /Y).
Some switches command actions that take place when entered
(/S, /R -- a filename entered without a switch appended acts
this way, too). These "rules of behavior" should be kept in
mind when entering LINK-80 commands. See the descriptions
for each switch for full details of its action.

The chart below summarizes the switches by function.   Full
descriptions of the switches by function follow the chart.

BE CAREFUL: Do not confuse the LINK-80 switches with the
MACRO-80 switches.

FUNCTION	SWITCH	ACTION
Execute	/G /G:Name	Execute .COM file then exit to operating system. Set .COM file start address equal to value of Name, execute .COM file, then exit to operating system.
Exit	/E /E:Name	Exit to operating system. Set .COM file start address equal to value of Name, then exit to operating system.
Save	/N /N:P	Save all previously loaded programs and subroutines using filename immediately preceding /N. Alternate form of /N; save only program area.
Address Setting	/P /D /R	Set start address for programs and data. If used with /D, /P sets only the program start. Set start address for data area only. Reset LINK-80.
Library Search	/S	Search the library named immediately preceding /S.
Global Listing	/U /M	List undefined globals. List complete global reference map.
Radix Setting	/O /H	Octal radix. Hexadecimal radix (default).
Special Code	/X /Y	Save "COM" file in Intel ASCII Hex format. Requires /N switch. Gives "COM" file the extension .HEX. Creates a special file for use with SID/ZSID debugger. Requires /N and /E switches. Gives special file the extension .SYM.

Figure 6.1: Table of LINK-80 Switches

At least two switches will probably be used in every linking
session. These switches belong to the first three functions
-- Execute, Exit, and Save.

EXECUTE

Switch   Action

/G   The /G switch causes LINK-80   to   load   the
            filename(s) entered in the command line, to link
            the program(s) together, then to execute the
            link-loaded program. After the program run, your
            computer returns to operating system   command
            level. For example,

                 L80 NEIL,NEIL/N/G

            links NEIL.REL, saves the result in a disk file
            named NEIL.COM, then exits to the operating
            system.

            Execution takes place as soon as the command line
            has been interpreted. Just before execution
            begins, LINK-80 prints three numbers and a BEGIN
            EXECUTION message. These three numbers can be
            very useful to you in developing future assembly
            language programs. The first number is the start
            address of the program. The second number is the
            address of the next available byte; that is, the
            end address plus one byte. The third number is
            the number of 256-byte pages taken up by the
            program (the difference between the start   address
            and the end address converted to 256-byte pages).

            If you do not want to save the .COM file, use the
            /G switch and enter only one filename on the
            command line. For example:

                 L80 NEIL/G

            But Remember: No COM file is created (since you
            did not include /N). To run the program again,
            you will have to run LINK-80 again.

/G:<name> The /G:<name> switch performs exactly like the
           plain /G switch but with one additional feature.
           <name> is a global symbol which was defined
           previously in one of the modules which is being
           linked and loaded. When LINK-80 sees <name>, it
           uses <name> as the start of the program and loads
           the address of the line with <name> as its LABEL
           into the jump instruction at 100H-102H.

           The value of this switch (and of /E:<name> below)
           is the ability to tell LINK-80 where to start
           execution when the assembled modules do not make
           this clear. Usually this is no problem because
           you link in a high-level language program (which
           LINK-80 takes as the main program by default), or
           you link only assembly language modules and only
           one has an END <name> statement to signal LINK-80
           which assembly language program to execute first.
           But if two or more assembly language modules
           contain an END <name> statement, or if none of the
           assembly language modules contain an END <name>
           statement, then /G:<name> tells LINK-80 to use
           this module as the starting point for execution.

           Programmers who want to execute an assembly
           language module before a high-level language
           program should use a CALL or INCLUDE statement at
           the beginning of the high-level language program
           to cause execution of the assembly language module
           before execution of the high-level language
           program.

EXIT

Switch   Action

/E   Use /E to link and load a program and perform some
            other functions on the files (for example, save it
            on a diskette) when you do not want to run the
            program at this time. When LINK-80 has finished
            the tasks, it will exit to the operating system.

            (The /G switch is the only other switch which
            exits LINK-80.)

            When linking is finished, LINK-80 outputs three
            numbers: start address, next available byte,
            number of 256-byte pages.

/E:<name>   The /E:<name> switch performs exactly like the
            plain /E switch but with one additional feature.
            <name> is a global symbol which was defined
            previously in one of the modules which is being
            linked and loaded. When LINK-80 sees <name>, it
            uses <name> as the start of the program and loads
            the address of the line with <name> as the LABEL
            into the jump instruction at 100H-102H.

            The value of this switch (and of /G:<name> above)
            is the ability to tell LINK-80 where to start
            execution when the assembled modules do not make
            this clear. Usually this is no problem because
            you link in a high-level language program (which
            LINK-80 takes as the main program by default), or
            you link only assembly language modules and only
            one has an END <name> statement to signal LINK-80
            which assembly language program to execute first.
            But if two or more assembly language modules
            contain an END <name> statement, or if none of the
            assembly language modules contain an END <name>
            statement, then /E:<name> tells LINK-80 to use
            this module as the starting point for execution.

            Programmers who want to execute an assembly
            language module before a high-level language
            program should use a CALL or INCLUDE statement at
            the beginning of the high-level language program
            to cause this order of execution.

SAVE

Switch   Action

/N   The /N switch causes the assembled-linked program
            to be saved in a disk file. It is important that
            a filename always be specified for the /N switch.
            If you do not specify an extension, the default
            extension for the saved file is .COM.   The COM
            filename will be the name the programmer appends
            the /N switch to. The /N switch must immediately
            follow the filename under which you wish to save
            the results of the link-load session.

            The /N switch does not take effect unless a /E or
            /G switch follows it.

            The most common error programmers make with the /N
            switch is to forget that they must specify at
            least two filenames; one as the file to be linked
            and another one as the name for the file to be
            saved. Therefore, at a minimum the command line
            should include:

                L80 NEIL,NEIL/N/G

            The first filename NEIL is the file to be loaded
            and linked; the second filename NEIL is the name
            for the COM file that will save the result of the
            link-loading session.

            It is, of course, possible to specify filenames in
            any order. For example:

                L80 NEIL/N,ASMSUB1,ASMSUB2,BASPROG/G

            Here LINK-80 will load and link the files BASPROG,
            ASMSUB1, and ASMSUB2; then save the result in the
            file named NEIL.

            From these two examples, it is possible to see
            that the filename followed by the /N save switch
            is not loaded; it is only a specification for an
            output file;   you must also always name at least
            one input file, too.

            You will use this switch almost every time you
            link a REL file because there is no other way to
            save the result of a link-load session and because
            not saving the result means you would have to link
            load again to run your program.

            Once saved on disk, you need only type the COM
            filename at operating system command level to run
            the program.

/N:P   By default, LINK-80 saves both the program and
            data   areas   in the COM file. If you wish to save
            only the program area to make your disk files
            smaller, use the /N switch in the form /N:P. With
            this switch set, only the program code will be
            saved.

Two of these switches (/N plus either a /G or a /E type) are
all the switches required for most LINK-80 operations. Some
additional functions are available through the use of other
switches which allow programmers to manipulate the LINK-80
processes in more detail. The switches which turn on these
additional functions are arranged in categories according to
type of function. The function of each category is defined
by the category name.

ADDRESS SETTING

Switch   Action

/P   The /P switch is used to set both the program and
            data origin.   If you do not enter the /P switch,
            LINK-80 performs this task automatically, using a
            default address for both program and data. (103H
            for CP/M)

            The format of the /P switch is:

                /P:<address>,

            The address value must be expressed in the current
            radix.   The default radix is hexadecimal.

            The /P switch is designed to allow you to place
            program (or code) segments at addresses other than
            the default. The default value for the /P switch
            is 103H.

            REMEMBER: The /P switch takes effect as soon as
            it is seen, but it does not affect files already
            loaded. So be sure to place the /P switch before
            any files you want to load starting at the
            specified address. The /P switch and /D switch,
            when used, must be separated from the REL filename
            by a comma. For example,

                L80 /P:103,NEIL,NEIL/N/E

            The /P switch affects primarily the CSEG code in
            your assembly language program. If /P is given
            but not /D, both data and program (CSEG and DSEG)
            areas will be loaded starting at the /P:<address>.
            DSEG (and any COMMON areas) will be loaded first.
            If both /P and /D switches are given, /P sets the
            start of the CSEG area only.   Normally, unless
            your programs are all CSEG, you will use /P and /D
            together.

            Note especially that ASEG areas are not affected
            by the /P switch.   So be careful to set the /P
            address outside any ASEG areas unless you want the
            program or data areas to write over the ASEG
            areas.

            You may enter more than one /P switch during a
            single link session to place different program
            (code) segments at addresses which are not end to
            end.   LINK-80 will automatically place one program
            segment (CSEG) after the next.   You can cause
            space to be left between modules. However, some

restrictions one the placement of modules apply:

1. Be sure that program areas do not overlay one
    another. LINK-80 returns a warning error
    message if they do.

2. Be sure that the program areas are not split
    by data or COMMON areas; that is, a CSEG at
    200H, a DSEG at 300H, and another CSEG at 400H
    is illegal. LINK-80 returns a fatal error in
    this case.

When the loading session is all done, LINK-80
wants to see a segment of memory loaded with data
and COMMON and another segment loaded with program
code. The code segments may have gaps between the
modules as long as a data segment is not loaded
between the start of the first code segment module
and the end of the last code segment module, and
vice versa. So, placing DSEG modules at
103H-115H, 150H-165H, 170H-175H, and CSEG modules
at 200H-250H, 300H-350H, 400H-450H is acceptable.
LINK and 80 will show Data between 103H and 175H
and Program between 200H and 450H.

Note that any gaps you leave may contain data or
program code from a previous program. LINK-80
does not initialize gaps to zero or null. This
could cause unpredictable results.

/D   The /D switch sets the origin for DSEG and COMMON
            areas. If you do not enter the /D switch, LINK-80
            performs this task automatically, using a default
            address for both data and program. (103H for
            CP/M)

            The format for the /D switch is:

                /D:<address>,

            The address for the /D switch must be in the
            current radix. The default radix is hexadecimal.

            The /D switch is designed to allow you to place
            data and COMMON segments at addresses other than
            the default. The default value for the /D switch
            is 103H. The /D switch must be separated from the
            REL filenames by a comma. For example,

                L80 /D:103,NEIL,NEIL/N/E

            When the /P switch is used with the /D switch,
            data and common areas load starting at the address
            given with the /D switch. (The program will be

loaded beginning at the program origin given with
the /P switch.) This is the <u>only</u> occasion when the
address given in /P: is the start address for the
actual program code.

<u>REMEMBER:</u> The /D switch takes effect as soon as
LINK-80 "sees" the switch, so the /D switch has no
effect on programs or data already loaded.
Therefore, it is important to place the /D switch
(as well as the /P switch) <u>before</u> the files you
want to load starting at the address specified.

You may enter more than one /D switch during a
single link session to place different program
(code) segments at addresses which are not end to
end.   LINK-80 will automatically place one data
segment (DSEG) after the next.   You can cause
space to be left between modules. However, some
restrictions on the placement of modules apply:

1.   Be sure that data areas do not overlay one
     another.   LINK-80 returns a warning error
     message if they do.

2.   Be sure that the data areas are not split by
     program areas;   that is, a DSEG at 200H, a
     CSEG at 300H, and another DSEG at 400H is
     illegal.   LINK-80 returns a fatal error in
     this case.

When the loading session is all done, LINK-80
wants to see a segment of memory loaded with data
and COMMON and another segment loaded with program
code. The data segments may have gaps between the
modules as long as a program segment is not loaded
between the start of the first data segment module
and the end of the last data segment module, and
vice versa.   So, placing DSEG modules at
103H-115H, 150H-165H, 170H-175H, and CSEG modules
at 200H-250H, 300H-350H, 400H-450H is acceptable.
LINK and 80 will show Data between 103H and 175H
and Program between 200H and 450H.

Note that any gaps you leave may contain data or
program code from a previous program. LINK-80
does not initialize gaps to zero or null.   This
could cause unpredictable results.

```
ADDITIONAL NOTE FOR /P AND /D SWITCHES

If your program is too large for the  loader,  you
will   sometimes  be  able  to load it anyway if you
use /D and /P together.  This way you will be able
to  load  programs  and  data of a larger combined
total.  While LINK-80 is loading and  linking,  it
builds  a  table consisting of five bytes for each
program relative reference.  By  setting  both  /D
and  /P,  you  eliminate  the  need for LINK-80 to
build this  table,  thus  giving  you  some  extra
memory to work with.

To set the two switches, look to the  end  of  the
List  file.   Take the address you decided for the
/D switch (where  you  want  the  DSEG  to  start
loading),  add  the  number for the total of data,
add that number to 103H, add another  100H+1,  and
the  result  should  be  the  /P:  address for the
start of the program area.  The /D  switch  should
be set at 103H or higher (D:103).
```

/R   The /R switch "resets" LINK-80 to its initialized
          condition.   LINK-80 scans the command line before
          it begins the functions commanded.   As soon as
          LINK-80 sees the /R switch, all files loaded are
          ignored, LINK-80 resets itself, and the asterisk
          (*) prompt is returned showing that LINK-80 is
          running and waiting for you to enter a command
          line.

LIBRARY SEARCH

Switch   Action

/S   The /S switch causes LINK-80 to search the file
            named immediately prior to the switch for
            routines, subroutines, definitions for globals,
            and so on. In a command line, the filename with
            the /S switch appended must be separated from the
            rest of the command line by commas. For example:

            L80 NEIL/N,MYLIB/S,NEIL/G

            The /S switch is used to search library files
            only, including a library you constructed, using
            the LIB-80 Library Manager (see Chapter 8).

GLOBAL LISTING

Switch   Action

/U   The /U switch tells LINK-80 to list all undefined
             globals. The /U works only in command lines that
             do not include either a /G or a /E switch. Note
             that if your program contains any undefined
             globals, LINK-80 lists them automatically, unless
             the command line also contains a /S (library
             search) switch. In these cases, enter only the /U
             switch, and the list of undefined globals will be
             listed. Use CTRL-S to suspend the listing if you
             want to study a portion of the list that would
             scroll off the screen. Use CTRL-Q to restart the
             listing.

             The various runtime libraries provide definitions
             for the globals you need to run your high-level
             language programs.

             In addition to listing undefined globals, the /U
             switch directs LINK-80 to list the origin, end,
             and size of the program and data areas.   These
             areas are listed as one lump area unless both the
             /P and /D switches are set. If both /P and /D are
             set, the start, end, and size of both areas are
             listed separately.

/M   The /M switch directs LINK-80 to list all globals,
             both defined and undefined, on the screen. The
             listing cannot be sent to a printer.   In the
             listing, defined globals are followed by their
             values, and undefined globals are followed by an
             asterisk (*).

             In addition to listing all globals, the /M switch
             directs LINK-80 to list the origin, end, and size
             of the program and data areas. These areas are
             listed as one lump area unless both the /P and /D
             switches are set. If both /P and /D are set, the
             start, end, and size of both areas are listed
             separately.

RADIX SETTING

Switch   Action

   /O   The /O switch sets the current radix to Octal. If
             you have   a   reason   to   use octal values in your
             program, give the /O switch in the   command   line.
             If you can think of no reason to switch to octal
             radix, then there is no reason to use this switch.

   /H   The /H switch resets the current   radix   to
             Hexadecimal.   Hexadecimal is the default radix.
             You do not need to give this switch in the command
             line unless you previously gave the /O switch and
             now want to return to hexadecimal.

SPECIAL CODE

Switch   Action

/X   The /X switch saves the "COM" file in Intel ASCII
            HEX format. The /X switch requires the /N switch
            appended to the same filename as the /X. For
            example:

                L80 NEIL,NEIL/X/N/E

            The file that is saved with the /X switch set is
            given the filename extension .HEX.

            The primary use of the /X switch is to prepare
            programs to be burned into PROMs. The hex format
            was originally developed to facilitate the
            movement of programs from one machine to another.
            The hex format provides more code checking than
            object code does. Also, a HEX file can be edited
            with some sophisticated line editors.

/Y   The /Y switch saves a file in a special format for
            use with Digital Research's Symbolic Debuggers,
            SID and ZSID. The /Y switch requires the /N and
            the /E (not /G) switches be given in the command
            line. For example:

                L80 NEIL,NEIL/Y/N/E

            The file that is saved with the /Y switch set is
            given the filename extension .SYM. A COM file
            will also be saved. So the sample command line
            above creates both NEIL.COM and NEIL.SYM.

            The SYM file contains the names and addresses of
            all globals, which allows you to use Digital
            Research's Symbolic Debuggers SID and ZSID with
            the SYM file.

6.3   ERROR MESSAGES

Errors encountered during the running of LINK-80 will return
messages, most preceded by either the symbol ? or the
symbol %.   No error codes are returned, so once you
understand the meaning of the message, error recognition
should be easy.

?No Start Address

>   The /G switch was issued, but no main program has
>   been loaded.

?Loading Error

>   The last file given for input was not a properly
>   formatted LINK-80 object file.

?Out of Memory

>   Not enough memory to load the module.

?Command Error

>   Unrecognizable LINK-80 command.

?<filename> Not Found

>   <filename>, as given in the command string, did not
>   exist.

?Start Symbol - <name> - Undefined

>   The /E:Name or /G:Name switch was given, but the
>   Name specified was not defined.

?Nothing Loaded

> A <filename>/S or /E or /G was given, but no object
> file was loaded. That is, an attempt was made to
> search a library, to exit LINK-80, or to execute a
> program, when in fact nothing had been loaded. For
> example:
>
>   TEST/N/E
>
> Results in "?Nothing Loaded" because TEST/N names
> TEST.COM, but does not load TEST.REL.
>
> To load a file, enter the filename. To save a
> file, enter a filename followed by the /N switch
> and either a /E or a /G switch. For example, any
> of the following sets of commands should work:
>
>   L80 NEIL,NEIL/N/E
>
>   or
>
>   L80
>   *NEIL
>   *NEIL/N/E
>
>   or
>
>   L80 NEIL/N,NEIL/E

?Can't Save Object File

> A disk error occurred when the file was being
> saved. Usually, this means that the disk is full
> or that it is write-protected.

%2nd COMMON larger /XXXXXX/

> When loading modules which include COMMON blocks,
> LINK-80 takes the size of the first COMMON block
> loaded to set the amount of memory needed before
> program code is loaded. If a subsequent module
> contains a COMMON block larger than the first one
> loaded, LINK-80 returns this error message. It
> means that the first definition of the COMMON block
> /XXXXXX/ encountered in the modules loaded was not
> the largest block defined with that name. Reorder
> module loading sequence or change COMMON block
> definitions so that all blocks are the same size.

%Mult. Def. Global YYYYYY

> You have one global (PUBLIC) symbol name YYYYYY
> with more than one definition. Usually, two or
> more of the modules being loaded have declared the
> same symbol name as PUBLIC.

%Overlaying Program Area   ,Start   = xxxx
                            ,Public   = <symbol name> (xxxx)
                            ,External = <symbol name> (xxxx)

> Usually this occurs when either /D or /P is set to
> an address inside the area taken by LINK-80. You
> should reset the switch address above 102H. It may
> also occur if you set addresses for programs loaded
> after some initial programs were loaded and the
> addresses were not set high enough. For example,
> if MYPROG is larger than 147 bytes and you enter
> the commands:

>   MYPROG,/P:150,SUBR1,FUNNY/N/E

> you will receive the %Overlaying Program Area error
> message.

%Overlaying Data Area   ,Start   = xxxx
                         ,Public   = <symbol name> (xxxx)
                         ,External = <symbol name> (xxxx)

> The /D and /P switches were set too close together.
> For example, if /D was given a higher address than
> /P but not high enough to be beyond the end of the
> program area, when the program is loaded, the top
> end will be laid over the data area. Or, if /D is
> lower than /P, /P was not high enough to prevent
> the beginning of the program from starting in the
> area already loaded with data.

?Intersecting Program Area
  <u>or</u>
?Intersecting Data Area

> The program and data areas 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.

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

> This message will appear only after either the /E
> or the /G switch command was given to LINK-80. If
> LINK-80 has not enough memory to load a module but
> a /E or /G has not been entered, you will receive
> the ?Out of Memory message.
>
> LINK-80 can load modules only between its first
> address in memory and the top of available memory.
> If the program is too large for this space or if
> you set a /D and/or /P switch too high for the size
> of your program, LINK-80 runs out of memory and
> returns the Origin Above Loader Memory message.
>
> If you set a /D and/or /P switch below the first
> address of LINK-80 (100H for CP/M), LINK-80 returns
> the Origin Below Loader Memory message. This
> prevents you from loading your program into memory
> designated for the operating system.
>
> 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 the /N switch was given,
> the image will already have been saved.

Contents

Chapter 7   CREF-80 Cross Reference Facility

CHAPTER 7

CREF-80 CROSS REFERENCE FACILITY

A cross reference facility processes a specially assembled
listing file to list the locations of all intermodule
references and the locations of their definitions. The
result is a cross reference listing. This cross reference
listing can be used to aid debugging your program.

The CREF-80 Cross Reference Facility allows a programmer to
process the cross reference file generated by MACRO-80.
This cross reference file contains embedded control
characters, set up during MACRO-80 assembly. CREF-80
interprets the control characters and generates a file that
lists cross references among variables.

CREF-80 produces a listing, resembling the PRN listing of
MACRO-80, with two additional features:

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

    2. At the end of the listing, variable names appear in
        alphabetic order. Each name is followed by the
        line number where the variable is defined (flagged
        with #) followed by the numbers of other lines
        where the variable is referenced.

The CREF listing file replaces the MACRO-80 PRN List file
and receives the filename extension .LST instead of .PRN.

7.1 CREATING A CREF LISTING

Creating a CREF listing involves two steps: (1) creating a
cross reference file (.CRF), and (2) generating a cross
reference listing (.LST). The first step occurs in the
MACRO-80 macro assembler; the second in the CREF-80 Cross
Reference Facility.

Creating a Cross Reference File

To create a cross reference file, set the /C switch in the MACRO-80 command line. For example:

        M80 =NEIL/C

This command line assembles the file NEIL.MAC, generating the output files NEIL.REL (object file) and NEIL.CRF (cross reference file).

Generating a Cross Reference Listing

The cross reference listing is generated by running the .CRF file through CREF-80.

To invoke the cross reference facility, enter:

        CREF80

CREF-80 will return an asterisk (*) prompt.

To create the cross reference listing file, enter:

        =filename

where filename is the name of your .CRF file. For example:

        CREF80 =NEIL

will generate a .LST file (NEIL.LST) containing the cross reference information.

This .LST file can be printed or sent to the terminal screen using operating system commands. Additionally, CREF-80 supports the same output device designations as MACRO-80. Simply enter the device designation in front of the filename. For example:

        CREF80 LST:=NEIL

sends the .LST listing to the printer only (no disk file is generated).

        CREF80 TTY:=NEIL

sends the .LST listing to the CRT only (no disk file is generated).

You will need to give a drive designation if you want the
.LST file saved elsewhere than the currently logged drive
(where the .CRF file resides). For example:

        CREF80 B:=A:NEIL

saves NEIL.LST on drive B.

When finished, CREF-80 prompts with an asterisk. You may
enter another =filename, or exit from CREF-80 to the
operating system.

To exit CREF-80, enter:

        CTRL-C

If you want the .LST file named differently from the default
(.CRF filename and extension .LST), enter the name in front
of the equal sign. For example:

        CREF80 NEIL.CRL=NEIL
     or CREF80 NEILCREF=NEIL

The former command line generates a cross reference list
file named NEIL.CRL; the latter generates a file named
NEILCREF.LST.

Look at the filename extensions to distinguish a cross
reference listing file from the listing file MACRO-80
normally generates. The listing file MACRO-80 normally
generates (without the /C switch set in the command line)
receives the default filename extension .PRN. The cross
reference listing file generated by CREF-80 receives the
default filename extension .LST.

7.2 <u>CREF</u> <u>LISTING</u> <u>CONTROL</u> <u>PSEUDO-OPS</u>

You may want the option of generating a cross reference
listing for part of a program but not all of it. To control
the listing or suppressing of cross references, use the
cross reference listing control pseudo-ops, .CREF and
.XCREF, in the source file for MACRO-80. These two
pseudo-ops may be entered at any point in the program in the
OPERATOR field. Like the other listing control pseudo-ops,
.CREF and .XCREF support no ARGUMENTs.

Pseudo-op   Definition

.CREF   Create cross references.
                .CREF is the default condition. Use .CREF to
                restart the creation of a cross reference file
                after using the .XCREF pseudo-op. .CREF
                remains in effect until MACRO-80 encounters
                .XCREF. Note, however, that .CREF has no
                effect until the /C switch is set in the
                MACRO-80 command line.

.XCREF   Suppress cross references.
                .XCREF turns off the .CREF (default) pseudo-op.
                .XCREF remains in effect until MACRO-80
                encounters .CREF. Use .XCREF to suppress the
                creation of cross references in selected
                portions of the file. Because neither .CREF
                nor .XCREF takes effect until the /C switch is
                set in the MACRO-80 command line, there is no
                need to use .XCREF if you want the usual List
                file (one without cross references); simply
                omit /C from the MACRO-80 command line.

CHAPTER 8

LIB-80 LIBRARY MANAGER

WARNING

```
┌─────────────────────────────┐
│ Read  this  chapter  carefully │
│ and make  a  back-up  copy  of │
│ your  libraries  before  using │
│ LIB-80.    LIB-80   is    very │
│ powerful  and thus can be very │
│ destructive.  It  is  easy  to │
│ destroy a library with LIB-80. │
└─────────────────────────────┘
```

LIB-80 is designed as a runtime library manager for CP/M versions of Microsoft FORTRAN-80 and COBOL-80. LIB-80 may also be used to create your own library of assembly language subroutines.

LIB-80 creates runtime libraries from assembly language programs that are subroutines to COBOL, FORTRAN, and other assembly language programs. The programs collected by LIB-80 may be special modules created by the programmer or modules from an existing library (FORLIB, for example). With LIB-80, you can build specialized runtime libraries for whatever execution requirements you design.

The value of building a library is that all the routines needed to execute a program can be linked with it into an executable object (COM) file by entering the library name followed by /S in a LINK-80 command line. For example:

        L80 MAIN,NEWLIB/S,NEIL/N/G

This is much more convenient than entering the necessary subroutines individually, especially if there are many modules. With a library file you can be sure all the necessary modules will be linked into the COM file, plus there is no danger of running out of space on the LINK-80

command line. Additionally, the library makes this special
collection of subroutines available for easy linking into
any program.

8.1 SAMPLE LIB-80 SESSION

The two most common uses you will have for LIB-80 are
building a library and listing a library. The following
sample sessions illustrate the basic commands for these two
uses.

BUILDING A LIBRARY:

```
A>LIB
*TRANLIB=SIN,COS,TAN,ATAN,ACOG
*EXP
*/E
A>
```

In this sample session, LIB invokes LIB-80, which
returns an asterisk (*) prompt. TRANLIB is the
name of the library being created.
SIN,COS,TAN,ATAN,ACOG are filenames to be
concatenated into TRANLIB. EXP is another filename
to be concatenated into TRANLIB. (EXP could be
listed on the previous command line; this example
shows files entered singly and multiply.) /E causes
LIB-80 to rename TRANLIB.LIB to TRANLIB.REL then to
exit to CP/M.

LISTING A LIBRARY:

```
A>LIB
*TRANLIB.LIB/U
*TRANLIB.LIB/L
      .
      .
      .

(List of symbols in TRANLIB.LIB)

      .
      .
      .

*CTRL-C
A>
```

In this sample session, LIB invokes LIB-80.
TRANLIB.LIB/U tells LIB-80 to search TRANLIB.LIB
for any intermodule references that would not be
defined during a single pass through the library

(that is, any "backward" referencing symbols). TRANLIB.LIB/L directs LIB-80 to list the modules in TRANLIB.LIB and the symbol definitions the modules contain. CTRL-C exits to CP/M without destroying any files.

WARNING

/E will destroy your current library if there is no new library under construction. This is a special danger to your FORTRAN runtime library FORLIB.REL. IF YOU ARE ONLY LISTING THE LIBRARY AND NOT REVISING IT, EXIT LIB-80 USING CTRL-C.

8.2 LIB-80 COMMANDS

Invoking LIB-80

To invoke LIB-80, enter:

        LIB

LIB-80 will return an asterisk (*) prompt, indicating ready to accept commands. Each command in LIB-80 adds modules to the library under construction.

Commands to LIB-80 consist of an optional Destination field, a Source field, and an optional Switch field.

The format of a LIB-80 command is:

        Destination=Source/Switch

Each field is described below. The general format for each field is shown in parentheses after the field name.

Destination field (filename=)

This field is optional. The equal sign is required if any
entry is made in this field.

Enter in this field the filename (and extension, if you
choose) for the library file you want to create.

If this field is omitted, LIB-80 defaults to the filename
FORLIB. The default filename extension is .REL.

WARNING

Do not confuse this default
filename FORLIB.LIB with
FORLIB.REL, the runtime library
supplied with FORTRAN-80. These
two libraries will not be the
same unless you command LIB-80
to copy all the files from the
FORTRAN runtime library to the
new library. Furthermore, when
you exit LIB-80, the default
library name will be given the
filename extension .REL, which
means that it replaces the
FORLIB.REL supplied with
FORTRAN-80. For this reason,
unless you want your FORTRAN-80
runtime library destroyed, we
recommend emphatically that you
always specify a Destination
filename when creating a new
library.

Source field (filename<module>)

Some entry is required in this field. All Source files must
be REL files.

Source field entries tell LIB-80 which files or parts of
files (modules) you want added to the destination library
file. You have two choices for entries:

   1. Filename(s) only

   2. Any combination of filename(s) and module name(s)

The following syntax rules apply:

   1. If a command consists of filenames only, the
      entries are separated by commas only. For example:

        FILE1,FILE2,FILE3

   2. If a command consists of filenames and module
      names, the module names must be enclosed in angle
      brackets (<>). Modules follow the filename where
      they are found.   Each filename<module name>
      combination is separated from other command line
      entries by commas. For example:

        FILE1,FILE2<MODZ>,FILE3<MODR>,FILE4

   3. If more than one module is named from the same
      file, the module names, enclosed in angle brackets
      (<>), must be separated from each other by commas.
      For example:

        FILE1,FILE2<MODZ,MODR>,FILE3

      See Additional Details about Source Modules, option
      2, below.

Files and modules are typically FORTRAN or COBOL subprograms
or main programs, or ALDS assembly language programs that
contain ENTRY, GLOBAL, or PUBLIC   statements.   (These
statements are called entry points.) LIB-80 recognizes a
module by its program name, which may be a filename, or a
name given by either the .TITLE or the NAME pseudo-op in
MACRO-80. All Source files must be REL files.

LIB-80 concatenates REL files and modules of REL files;
that is, LIB-80 strings one file or module after the other.

So there is no difference between the command under syntax
rule 2 above and

        FILE1
        FILE2<MODZ>
        FILE3<MODR>
        FILE4

Also, because the library file is built by concatenation, it
is important to order the modules so that all intermodule
references are "forward." That is, the module containing the
external reference should physically appear ahead of the
module containing the ENTRY point (the definition).
Otherwise, when you direct LINK-80 to search the library,
LINK-80 may not satisfy all references on a single pass
through the library.

Additional Details about Source Modules

To extract modules from previous libraries and other REL
files, LIB-80 uses a powerful syntax to specify ranges of
modules within a REL file.

These ranges may be from one module to the entire file (in
which case no module specification is given).

The basic principle of specifying a range of modules is,
generally, that any module named in a command will be
included. (There is an exception, when specifying a
relative offset range--item 6, below.)

The options for specifying modules are:

    1. One module only
            Enter the module name. For example:

                FILE1<MODZ>

            includes only module MODZ of FILE1.

    2. Several discontiguous modules from one file
            Enter the module names separated by commas.
            For example:

                FILE1<MODZ,MODR,MODK>

                includes modules MODZ, MODR, and MODK. Note
                that these modules may be given in any order
                you need them concantenated for a proper
                one-pass search, regardless of their order in
                the original file.

3. From the first module through the named module
        Enter two periods (..) and the name of the last
        module to be included. For example:

            FILE1<..MODK>

        includes all modules from the first module in
        FILE1 through module MODK.

4. From a named module through the last module
        Enter the name of the module that starts the
        range followed by two periods (..).   For
        example:

            FILE1<MODR..>

        includes all the modules, beginning with module
        MODR, through the last module in FILE1.

5. From one named module through another named module
        Enter the name of the module that starts the
        range followed by two periods (..) followed by
        the name of the module that ends the range.
        For example:

            FILE1<MODZ..MODK>

        includes all modules, beginning with module
        MODZ, through module MODK.

6. Relative offset range
        Enter the module name followed by a + or - and
        the number of modules to be included. + means
        following the named module. - means preceding
        the named module. The named module is not
        included in the library. The offset number
        must be an integer in the range 1 to 255. For
        example:

            FILE1<MODZ+2>

        includes the two modules immediately following
        module MODZ. While

            FILE1<MODK-3>

        includes the   three   modules   immediately
        preceding module MODK.

Additionally, ranges and offsets may be used together. For example:

       FILE1<MODR+1..MODK-1>

includes all the modules between module MODR and module MODK (but neither MODR nor MODK is included).

7. All modules in a file
    Enter the filename only. For example:

       FILE1

    includes the entire file (all modules in FILE1).

Switch field (/switch)

An entry in the Switch field commands LIB-80 to perform additional functions. A Switch field entry is a letter preceded by a slash mark (/).

                          WARNING

       ┌─────────────────────────────────┐
       │ /E will destroy your current   │
       │ library if there is no new   │
       │ library under construction.   │
       │ This is a special danger to   │
       │ your FORTRAN runtime library   │
       │ FORLIB.REL because FORLIB is   │
       │ the default filename used if   │
       │ you do not specify a   │
       │ destination filename.   │
       │ Therefore, unless you want to   │
       │ delete your complete FORTRAN   │
       │ runtime library, give LIB-80 a │
       │ destination filename for the   │
       │ new library. If you are only   │
       │ listing the library and not   │
       │ revising it, exit LIB-80 using │
       │ CTRL-C.   │
       └─────────────────────────────────┘

Switch   Action

/E   Exit to CP/M. <u>If you are not creating a new
            library or revising an existing library, use
            CTRL-C instead of /E.</u>

            The library under construction (.LIB) is renamed
            to .REL and any previous copy of the library file
            is deleted. This is why /E is so dangerous and
            not to be used unless you are constructing a new
            library. Again, we recommend emphatically that
            you <u>always</u> enter a filename in the Destination
            field of the LIB-80 command line.

/R   Rename the library currently being built (.LIB) to
            .REL. <u>The same warnings and cautions apply to /R
            as apply to /E.</u>

            The previous copy of the library is deleted. Use
            /R only if you are building a new library. /R
            performs the same functions as /E, but does not
            exit to CP/M on completion. Use /R instead of /E
            when you want to exit the current library but want
            to continue using LIB-80 for other library
            managing.

/L   List the modules in the file specified and the
            symbol definitions the modules contain. The
            contents of a file are listed in cross reference
            format.

            Listings are currently always sent to the
            terminal; use CTRL-P before running LIB-80 to
            send the listing to the printer.

/U   Use /U to list the symbols which could be
            undefined in a single pass through a library. If
            a symbol in a library module refers "backward" (to
            a preceding module), /U will list that symbol.

/C   Use /C to clear commands from LIB-80 without
            exiting the LIB-80 program. The library under
            construction is deleted and the LIB-80 session
            starts over. The asterisk (*) prompt will appear.

            Use /C if you specified the wrong module(s) or the
            wrong order and want to start over with new LIB-80
            commands.

/O   Use /O to set typeout mode to Octal radix.   /O
            will be given together with the /L switch, which
            commands LIB-80 to list. REMEMBER: When switches
            are given together, a slash must precede each
            switch. For example:

                NEWLIB/L/O

/H   Use /H to set typeout mode to Hexadecimal radix.
            Hexadecimal is the default radix.

APPENDIX A

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 word

        $EJECT

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

The control word

        $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 invoked, 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

APPENDIX B

The Utility Software Package with TEKDOS

The command formats for MACRO-80, LINK-80, and CREF-80 differ slightly under the TEKDOS operating system.

B.1 TEKDOS COMMAND FILES

The files M80, L80, and C80 are actually TEKDOS command files for the assembler, loader, and cross reference programs, respectively. These command files set the emulation mode to 0 and select the Z-80 assembler processor (see TEKDOS documentation), then execute the appropriate program file. You will note that all of these command files are set up to execute the Microsoft programs from drive #1. LINK-80 will also look for the library on drive #1. If you wish to execute any of this software from drive #0, the command file must be edited. Then, LINK-80 should be given an explicit library search directive, such as MYLIB-S. See the Switches section in Chapter 6, LINK-80 Linking Loader.

Filenames under TEKDOS do not use the Utility Software Package default filename extensions.

B.2 MACRO-80

The MACRO-80 assembler accepts command lines only (the invoke command, M80, and all filenames and switches must be on one line). No prompt is displayed, and the interactive commands (,TTY:=TTY: and ,LPT:=TTY:) are not accepted. Commands have the same format as TEKDOS assembler commands; that is, up to three filenames or device names plus optional switches.

M80 [object] [list] source [switch [switch [...]]]

The object and list file entries are optional. These files will not be created if the parameters are omitted. Any

error messages will still be displayed on the console. The available switches are described in Chapter 5 of this manual. All command line entries may be delimited by commas or spaces. If you do not want to request an object file, you must enter a <space comma space> between the M80 entry and the name of the list file. For example:

        M80 , LIST SOURCE

B.3 CREF-80

The form of commands to CREF-80 is:

        C80 list source

Both filenames are required. The source file is always the name of a CREF-80 file created during assembly by the C switch.

Example:

To create a CREF-80 file from the source TSTMAC using MACRO-80, enter:

        M80 , TSTCRF TSTMAC C

To create a cross reference listing from the CREF-80 file TSTCRF, enter:

        C80 TSTLST TSTCRF

B.4 LINK-80

With TEKDOS, the LINK-80 loader accepts interactive commands only. Command lines are not supported.

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 or spaces 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 Chapter 6) are supported, except -G and -N, which are not implemented at this time.

EXAMPLE:

1. Assemble a MACRO-80 program named XTEST, creating
   an object file called XREL and a listing file
   called XLST:

   >M80 XREL XLST XTEST

2. Load XTEST and save the loaded module:

   ```
   >L80
   *XREL-E
   [04AD  22B8]
   *DOS*ERROR 46
   L80 TERMINATED
   >M XMOD 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 TEKDOS 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.

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, which allows execution to begin at 0.
The memory locations between 0003 and 0400H are reserved for
SRB's and I/O buffers at runtime.

APPENDIX C

ASCII CHARACTER CODES

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	043	2BH	+	086	56H	V
001	01H	SOH	044	2CH	,	087	57H	W
002	02H	STX	045	2DH	−	088	58H	X
003	03H	ETX	046	2EH	.	089	59H	Y
004	04H	EOT	047	2FH	/	090	5AH	Z
005	05H	ENQ	048	30H	0	091	5BH	[
006	06H	ACK	049	31H	1	092	5CH	\
007	07H	BEL	050	32H	2	093	5DH	]
008	08H	BS	051	33H	3	094	5EH	^
009	09H	HT	052	34H	4	095	5FH	_
010	0AH	LF	053	35H	5	096	60H	`
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH	<	103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH	>	105	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	"	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	$	079	4FH	O	122	7AH	z
037	25H	%	080	50H	P	123	7BH	{
038	26H	&	081	51H	Q	124	7CH		
039	27H	'	082	52H	R	125	7DH	}
040	28H	(	083	53H	S	126	7EH	~
041	29H	)	084	54H	T	127	7FH	DEL
042	2AH	*	085	55H	U			

Dec=decimal, Hex=hexadecimal (H), CHR=character.
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

APPENDIX D

FORMAT OF LINK COMPATIBLE OBJECT FILES

This appendix contains reference material for users who wish to know the load format of LINK-80 relocatable object files. None of this material is necessary to the operation of ALDS. There is nothing in the format material presented here which can be manipulated by the user. The material is highly technical, and it is not presented in any tutorial manner.

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 (read one-zero-zero) 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 described 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 nn    zzz + characters of symbol name
          \_____/    _____/
          A field              B field
```

where: xxxx is four-bit control field (0-15 below)
        yy   is two-bit address type field
        nn   is sixteen-bit value
        zzz   is 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   Extension LINK items (see below)

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)

The following special LINK items have an A field only:

> 8   External - offset. Used for JMP and CALL to externals
>
> 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

An Extension LINK item follows the general format of a B-field-only special LINK item, but the contents of the B-field are not a symbol name. Instead, the symbol area contains one character to identify the type of extension LINK item, followed by from 1 to 7 characters of additional information.

Thus, every extension LINK item has the format:

> 1 00 0100 111 s bbbbbb

where: 111   is 3 bits containing the length of the field bbbbbb (0 implys 1 since F80 emits entry length of 0 for Blank Common),

> s   is an eight bit extension LINK item sub-type identifier, and

> bbbbbb   are 1 to 6 bytes for additional information. If used as B field for name, bbbbbb may be only 6 characters.

The present extension LINK item sub-types are:

> 5   X'35' COBOL overlay segment sentinel
>
> A   X'41' Arithmetic Fixup (Arithmetic Operator)
>
> B   X'42' Arithmetic Fixup (External Reference)
>
> C   X'43' Arithmetic Fixup (Area Base + Offset)

Descriptions of Sub-types

Sub-type 5

When the overlay segment sentinel is encountered by
LINK-80, 111 receives the value 010 (binary), and the
current overlay segment number is set to the value b+49.
If the previously existing segment number was non-zero
and the /N switch is in effect, the data area is written
to disk in a file whose name is the current program name
and whose extension is Vnn, where nn are the two
hexadecimal digits representing the number b+49
(decimal).

Sub-types A,B,C

Sub-types A, B, and C allow the processing of Polish
Arithmetic text. Items must be read as Reverse Polish
Expression. One or more Value items (sub-type B or C)
are followed by one or more Arithmetic Operators
(sub-type A) and end with a Store-Result Arithmetic
Operator (B.STBT or B.STWD).

All Items are put in the Fixup Table afer any offset
entries have been converted to final addresses. The
Polish expression is executed out of the Fixup Table at
the end of link. The result is stored at the PC given
when the Items were read.

Table of MACRO-80 Pseudo-ops

Notation: * means Z80 pseudo-op
            no stars means 8080 pseudo-op

SINGLE-FUNCTION PSEUDO-OPS

Instruction Set Selection

        .Z80
        .8080

Data Definition and Symbol Definition

        <name> ASET <exp>
        BYTE EXT <symbol>
        BYTE EXTRN <symbol>
        BYTE EXTERNAL <symbol>
        DB <exp>[,<exp>...]
        DB <string>[<string>...]
        DC <string>
        DDB <exp>[,<exp>...]
    *   DEFB <exp>[,<exp>...]
    *   <name> DEFL <exp>
    *   DEFM <string>[,<string>...]
    *   DEFS <exp>[,<val>]
    *   DEFW <exp>[,<exp>...]
        DS   <exp>[,<val>]
        DW <exp>[,<exp>...]
        ENTRY <name>[,<name>...]
        <name> EQU <exp>
        EXT <name>[,<name>...]
        EXTRN <name>[,<name>...]
    *   EXTERNAL <name>[,<name>...]
        GLOBAL <name>[,<name>...]
        PUBLIC <name>[,<name>...]
        <name> SET <exp> (not in .Z80 mode)

PC Mode Pseudo-ops

        ASEG
        CSEG
        DSEG
        COMMON /<block name>/
        ORG <exp>
        .PHASE <exp>/.DEPHASE

File Related Pseudo-ops

        .COMMENT <delim><text><delim>
        END [<exp>]
        INCLUDE <filename>
        $INCLUDE <filename>
        MACLIB <filename>
        .RADIX <exp>
        .REQUEST <filename>[,<filename>...]

Listing Pseudo-ops

Format Control Pseudo-ops

    *   *EJECT [<exp>] (one star is part of *EJECT)
        PAGE <exp>
        SUBTTL <text>
        TITLE <text>
        $TITLE

General Listing Control Pseudo-ops

        .LIST
        .XLIST
        .PRINTX <delim><text><delim>

Conditional Listing Control Pseudo-ops

        .SFCOND
        .LFCOND
        .TFCOND

Expansion Listing Control Pseudo-ops

        .LALL
        .SALL
        .XALL

Cross-Reference Listing Control Pseudo-ops

        .XCREF
        .CREF

MACRO FACILITY PSEUDO-OPS

Macro Pseudo-ops

        <name> MACRO <parameter>[,<parameter>...]
        ENDM
        EXITM
        LOCAL <parameter>[,<parameter>...]

Repeat Pseudo-ops

        REPT <exp>
        IRP <dummy>,<parameters in angle brackets>
        IRPC <dummy>,string

Conditional Assembly Facility

    *   COND <exp>
        ELSE
    *   ENDC
        ENDIF
        IF <exp>
        IFB <arg>
        IFDEF <symbol>
        IFDIF <arg1>,<arg2>
        IFE <exp>
        IFF <exp>
        IFIDN <arg1>,<arg2>
        IFNB <arg>
        IFNDEF <symbol>
        IFT <exp>
        IF1
        IF2

APPENDIX F

Table of Opcodes

The opcodes are listed alphabetically by instruction set.
For details, refer to the reference books listed in Chapter
1.

F.1 Z80 OPCODES

Opcode		Function
ADC	A	Add with Carry to Accumulator
ADC	HL,rp	Add Register Pair with Carry to HL
ADD		Add
AND		Logical AND
BIT		Test Bit
CALL	addr	Call Subroutine
CALL	cond,addr	Call Conditional
CCF		Complement Carry Flag
CP		Compare
CPD		Compare, Decrement
CPDR		Compare, Decrement, Repeat
CPI		Compare, Increment
CPIR		Compare, Increment, Repeat
CPL		Complement Accumulator
DAA		Decimal Adjust Accumulator
DEC		Decrement
DI		Disable Interrupts
DJNZ		Decrement and Jump if Not Zero
EI		Enable Interrupts
EX		Exchange
EXX		Exchange Register Pairs and Alternatives
HALT		Halt
IM	x	Set Interrupt Mode
IN		Input
INC		Increment
IND		Input, Decrement
INDR		Input, Decrement, Repeat
INI		Input, Increment
INIR		Input, Increment, Repeat
JP	addr	Jump
JP	cond,addr	Jump Conditional
JR		Jump Relative

```
JR     cond,addr       Jump Relative Conditional
LD     A,(addr)        Load Accumulator Direct
LD   A,(BC) or (DE)    Load Accumulator Secondary
LD     A,I             Load Accumulator from Interrupt Vector Register
LD     A,R             Load Accumulator from Refresh Register
LD   HL,(addr)         Load HL Direct
LD     data            Load Immediate
LD   xy,(addr)         Load Index Register Direct
LD     reg,(HL)        Load Register
LD   reg,(xy+disp)     Load Register Indexed
LD   rp,(addr)         Load Register Pair Direct
LD     SP,HL           Move HL to Stack Pointer
LD     SP,xy           Move Index Register to Stack Pointer
LD     dst,scr         Move Register-to-Register
LD     (addr),A        Store Accumulator Direct
LD   (BC) or (DE),A    Store Accumulator Secondary
LD     I,A             Store Accumulator to Interrupt Vector Register
LD     R,A             Store Accumulator to Refresh Register
LD   (addr),HL         Store HL Direct
LD   (HL),data         Store Immediate to Memory
LD   (xy+disp),data    Store Immediate to Memory Indexed
LD   (addr),xy         Store Index Register Direct
LD     (HL),reg        Store Register
LD   (xy+disp),reg     Store Register Indexed
LD   (addr),rp         Store Register Pair Direct
LDD                    Load, Decrement
LDDR                   Load, Decrement, Repeat
LDI                    Load, Increment
LDIR                   Load, Increment, Repeat
NEG                    Negate (Two's Complement) Accumulator
NOP                    No Operation
OR                     Logical OR
OUT                    Output
OUTD                   Output, Decrement
OTDR                   Output, Decrement, Repeat
OUTI                   Output, Increment
OTIR                   Output, Increment, Repeat
POP                    Pop from Stack
PUSH                   Push to Stack
RES                    Reset Bit
RET                    Return from Subroutine
RET    cond            Return Conditional
RETI                   Return from Interrupt
RETN                   Return from Non-Maskable Interrupt
RL                     Rotate Left Through Carry
RLA                    Rotate Accumulator Left Through Carry
RLC                    Rotate Left Circular
RLCA                   Rotate Accumulator Left Circular
RLD                    Rotate Accumulator and Memory Left Decimal
RR                     Rotate Right Through Carry
RRA                    Rotate Accumulator Right Through Carry
RRC                    Rotate Right Circular
RRCA                   Rotate Accumulator Right Circular
RRD                    Rotate Accumulator and Memory Right Decimal
RST                    Restart
```

```
SET              Set Bit
SBC              Subtract with Carry (Borrow)
SCF              Set Carry Flag
SLA              Shift Left Arithmetic
SRA              Shift Right Arithmetic
SRL              Shift Right Logical
SUB              Subtract
XOR              Logical Exclusive OR
```

F.2 8080 OPCODES

```
Opcode           Function

ADC,ACI          Add with Carry
ADD,ADI          Add
ANA,ANI          Logical AND
CALL             Call Subroutine
CC               Call on Carry
CM               Call on Minus
CMA              Complement Accumulator
CMC              Complement Carry
CMP,CPI          Compare
CNC              Call on No Carry
CNZ              Call on Not Zero
CP               Call on Positive
CPE              Call on Parity Even
CPO              Call on Parity Odd
CZ               Call on Zero
DAA              Decimal Adjust
DAD              16-bit Add
DCR              Decrement
DCX              16-bit Decrement
DI               Disable Interrupts
EI               Enable Interrupts
HLT              Halt
IN               Input
INR              Increment
INX              Increment 16 bits
JC               Jump on Carry
JM               Jump on Minus
JMP              Jump
JNC              Jump on Not Carry
JNZ              Jump on Not Zero
JP               Jump on Positive
JPE              Jump on Parity Even
JPO              Jump on Parity Odd
JZ               Jump on Zero
LDA              Load Accumulator
LDAX             Load Accumulator Indirect
LHLD             Load HL Direct
LXI              Load 16 bits
```

MOV	Move
MVI	Move Immediate
NOP	No Operation
ORA,ORI	Logical OR
OUT	Output
PCHL	HL to Program Counter
POP	Pop from Stack
PUSH	Push to Stack
RAL	Rotate with Carry Left
RAR	Rotate with Carry Right
RC	Return on Carry
RET	Return from Subroutine
RLC	Rotate Left
RM	Return on Minus
RNC	Return on No Carry
RNZ	Return on Not Zero
RP	Return on Positive
RPE	Return on Parity Even
RPO	Return on Parity Odd
RRC	Rotate Right
RST	Restart
RZ	Return on Zero
SBB,SBI	Subtract with Borrow
SHLD	Store HL Direct
SPHL	HL to Stack Pointer
STA	Store Accumulator
STAX	Store Accumulator Indirect
STC	Set Carry
SUB,SUI	Subtract
XCHG	Exchange D and E, H and L
XRA,XRI	Logical Exclusive OR
XTHL	Exchange Top of Stack, HL

INDEX

Microsoft
Software Problem Report

Use this form to report errors or problems in: ☐ FORTRAN-80

☐ COBOL-80

☐ MACRO-80

☐ LINK-80

Release (or version) number: _____

Date _____

Report only one problem per form.

Describe your hardware and operating system: _____

_____

Please supply a concise description of the problem and the circumstances surrounding its occurrence. If possible, reduce the problem to a simple test case. Otherwise, include all programs and data in machine readable form (preferably on a diskette). If a patch or interim solution is being used, please describe it.

This form may also be used to describe suggested enhancements to Microsoft software.

Problem Description:

Did you find errors in the documentation supplied with the software? If so, please include page numbers and describe:

Fill in the following information before returning this form:

Name_____ Phone_____

Organization_____

Address_____ City_____ State____ Zip_____

Return form to: Microsoft, Inc.
                       10700 Northup Way
                       Bellevue, WA 98004