

MICROSOFT

# BASIC COMPILER (CP/M)

HEATH

ZENITH  
data  
systems



三  
二  
一

三  
二  
一



一



# **Microsoft** **BASIC COMPILER**

**Microsoft BASIC Version 5.0  
and  
CP/M® Version 2.2**



<sup>®</sup>Registered Trademark, Digital Research.

**Copyright © 1981**  
Heath Company  
*All Rights Reserved*

595-2558-02

Printed in the United  
States of America

Technical consultation is available for any problems you encounter in verifying the proper operation of these products. Sorry, but we are not able to evaluate or assist in the debugging of any programs you may develop. For technical assistance, call:

(616)-982-3884 Application Software  
(616)-982-3860 Operating System/Language Software

Consultation is available from 8:00 AM to 12:00 NOON and from 1:00 PM to 4:30 PM (Eastern Time Zone) on normal business days.

Zenith Data Systems  
Software Consultation  
Hilltop Road  
St. Joseph, Michigan 49085

*Portions of this Manual have been adapted from Microsoft publications or documents.*

COPYRIGHT © by Microsoft, 1979, all rights reserved.

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

ZENITH DATA SYSTEMS  
SAINT JOSEPH, MICHIGAN 49085

# Table of Contents

## Chapter Zero — Installation Guide

Contents of the Diskettes .....	0-4
Sample Output of PI.BAS .....	0-6
Diskette Use .....	0-7
Diskette Loading .....	0-7
Diskette Handling .....	0-8
Write Protection .....	0-8
Preparing Working Diskettes .....	0-10

## Chapter One — Introduction

Unimplemented Direct Mode Statements .....	1-2
Language Differences .....	1-2
CALL .....	1-2
COMMON .....	1-3
CHAIN and RUN .....	1-3
DEFINT .....	
DEFSNG .....	
DEFDBL .....	
DEFSTR .....	1-3
DIM and ERASE .....	1-4
END .....	1-5
FOR/NEXT and WHILE/WEND .....	1-5
%INCLUDE .....	1-5
ON ERROR GOTO .....	1-6
REM .....	1-6
STOP .....	1-6
TRON and TROFF .....	1-6
USRn Functions .....	1-6
Arithmetic Evaluation Differences .....	1-6
Expression Evaluation .....	1-7
Integer Variables .....	1-8
Other Differences .....	1-8

---

## Chapter Two — BASIC Compiler Command Format

Source Program File X:SRCEPROG.EXT .....	2-2
Object Code File X:OBJCODE.EXT .....	2-2
Listing File X:LISTFIL.EXT .....	2-4
Command String Examples .....	2-6
Compiler Switches .....	2-7
/E .....	2-7
/X .....	2-7
/N .....	2-7
/D .....	2-8
/Z .....	2-9
/S .....	2-9
/4 .....	2-9
/C .....	2-9
Switch Examples .....	2-10

## Chapter Three — Compiling and Executing Programs

Compiling Programs .....	3-2
Executing and Saving Compiled Programs .....	3-2
/N .....	3-3
/E .....	3-4
/G .....	3-5
LINK-80 Command Examples .....	3-6

## Appendix A — Compiler Error Messages

Compiler Fatal Errors .....	A-1
Compiler Warning Errors .....	A-4

## Appendix B — BASIC Runtime Error Messages

BASIC Runtime Error Messages .....	B-1
------------------------------------	-----

## Appendix C — LINK-80 Error Messages

LINK-80 Error Messages .....	C-1
INDEX .....	I-1

Within this manual, we have used the following symbols:

- △ stands for a required space, which you produce by depressing the space bar.
- © stands for a carriage return, which you produce by depressing the RETURN key.
- CTRL stands for the CONTROL key on the terminal keyboard.



## Chapter Zero

# Installation Guide

*for the Microsoft BASIC-80 Interpreter  
and BASIC Compiler*

This "Installation Guide" describes system requirements and the duplicating procedure for the Microsoft BASIC package(s) that you have purchased. Since it covers several different models of software and hardware, not all of the information given here will apply to your particular system.

The applicable software model numbers, package contents, and hardware requirements are listed below:

### **Model: HMS-817-1 Microsoft BASIC-80 Interpreter**

Contents: Microsoft BASIC-80 Interpreter Distribution Disk (one 5.25-inch diskette).  
Manual in three-ring binder.

Requirements: 1. Heath H-8 with H-88-7 option installed.  
or  
Heath/Zenith H/Z-89-FA.  
or  
Heath/Zenith H/Z-89 with Z-89-7 option installed.  
2. 48K usable memory.  
3. At least one 5.25-inch disk drive.  
4. Heath/Zenith CP/M Operating System (Model HOS-817-2).

## **Model: HMS-847-1 Microsoft BASIC-80 Interpreter**

Contents: Microsoft BASIC-80 Interpreter Distribution Disk (one 8-inch diskette).  
Manual in three-ring binder.

Requirements: 1. Heath H-8 with H-88-7 option installed.  
or  
Heath/Zenith H/Z-89-FA.  
or  
Heath/Zenith H/Z-89 with Z-89-7 option installed.  
2. 48K usable memory.  
3. At least one 8-inch disk drive (Model H/Z-47).  
4. Heath/Zenith CP/M Operating System (Model HOS-847-2).

## **Model: HMS-817-4 Microsoft BASIC Compiler**

Contents: Microsoft BASIC Compiler Distribution Disk I.  
Microsoft BASIC Compiler Distribution Disk II (two 5.25-inch diskettes).  
Manual in three-ring binder.

Requirements: 1. Heath H-8 with H-88-7 option installed.  
or  
Heath/Zenith H/Z-89-FA.  
or  
Heath/Zenith H/Z-89 with Z-89-7 option installed.  
2. 48K usable memory.  
3. Two 5.25-inch disk drives.  
4. Heath/Zenith CP/M Operating System (Model HOS-817-2).

## Model: HMS-847-4 Microsoft BASIC Compiler

Contents: Microsoft BASIC Compiler Disk (one 8-inch diskette).  
Manual in three-ring binder.

Requirements:

1. Heath H-8 with H-88-7 option installed.  
or  
Heath/Zenith H/Z-89-FA.  
or  
Heath/Zenith H/Z-89 with Z-89-7 option installed.
2. 48K usable memory.
3. Two 8-inch disk drives (Model H/Z-47).
4. Heath/Zenith CP/M Operating System (Model HOS-847-2).

## CONTENTS OF THE DISKETTES

The diskettes you have received contain the following files:

### **Microsoft BASIC-80 INTERPRETER DISKETTE**

MBASIC.COM  
PI.BAS

MBASIC.COM is the BASIC Interpreter. Its commands and functions are discussed in the accompanying Microsoft BASIC-80 Software Reference Manual. PI.BAS is a sample program written in BASIC which calculates the value of pi. PI.BAS is provided to help familiarize you with the workings of the interpreter.

### **Microsoft BASIC COMPILER DISTRIBUTION DISK I**

BASCOM.COM  
BASLIB.REL

The commands and functions of the BASIC Compiler, is named BASCOM.COM, are documented in the BASIC Compiler Software User's Manual. BASLIB.REL is the BASIC Compiler System Library. You may modify this file using the Library Manager (LIB.COM, on Compiler Distribution Disk II).

## Microsoft BASIC COMPILER DISTRIBUTION DISK II

L80.COM  
M80.COM  
CREF.COM  
LIB.COM  
PI.BAS  
PI.REL

Section 2 of the Microsoft Utility Manual defines the use and operation of the MACRO-80 Assembler (M80.COM). CREF.COM, the Cross-Reference Facility, is described in Section 3 of the Utility Manual; L80, the Linking Loader, is discussed in Section 4; and LIB.COM, the Library Manager, is discussed in Section 5.

PI.BAS is a sample program designed to calculate the value of pi. It is provided to help you learn how to compile, link, and execute a program. PI.REL is the relocatable object file generated by the Compiler from PI.BAS.

Note that the contents of the 5.25-inch BASIC Compiler Distribution Disks I and II are provided on one 8-inch disk.

## SAMPLE OUTPUT OF PI.BAS

The listings provided below are sample outputs of the PI.BAS program. Note that the results generated by the Interpreter and Compiler may differ due to the different algorithms used to manipulate data.

### BOUNDS ON PI – DOUBLE PRECISION BINOMIAL THEOREM VERSION

SIDES	SIDE LENGTH	PI-LOWER BOUND	PI-UPPER BOUND
3	8	0.76536691188812	3.06146764755249
4	16	0.390180644737320	3.12144517898560
5	32	0.19603428244591	3.13654851913452
6	64	0.09813534468412	3.14033102989197
7	128	0.04908246546984	3.14127779006958
8	256	0.02454307302833	3.14151334762573
9	512	0.01227176748216	3.14157247543335
10	1,024	0.00613591633737	3.14158916473389
11	2,048	0.00306796119548	3.14159226417542
12	4,096	0.00153398059774	3.14159226417542
13	8,192	0.00076699029887	3.14159226417542
14	16,384	0.00038349514944	3.13159226417542
15	32,768	0.00019174757472	3.14159226417542
16	65,536	0.00009587385284	3.14159440994263
17	131,072	0.00004793689368	3.14159226417542
18	262,144	0.00002396846321	3.14159440994263
19	524,288	0.00001198423161	3.14159440994263
20	1,048,576	0.00000599211580	3.14159440994263

### BOUNDS ON PI – DOUBLE PRECISION BINOMIAL THEOREM VERSION

## INTERPRETER RESULTS

SIDES	SIDE LENGTH	PI-LOWER BOUND	PI-UPPER BOUND
3	8	0.76536686473018	3.06146745892072
4	16	0.39018064403226	3.12144515225805
5	32	0.19603428065912	3.13654849054594
6	64	0.09813534865484	3.14033115695475
7	128	0.04908245704582	3.14127725093277
8	256	0.02454307657144	3.14151380114430
9	512	0.01227176929831	3.14157294036709
10	1,024	0.00613591352593	3.14158772527716
11	2,048	0.00306796037257	3.14159142151120
12	4,096	0.00153398063749	3.14159234557012
13	8,192	0.00076699037514	3.14159257658487
14	16,384	0.00038349519462	3.14159263433856
15	32,768	0.00019174759819	3.14159264877699
16	65,536	0.00009587379921	3.14159265238659
17	131,072	0.00004793689962	3.14159265328899
18	262,144	0.00002396844981	3.14159265351459
19	524,288	0.00001198422491	3.14169265357099
20	1,048,576	0.00000599211245	3.14159265358509

## COMPILED RESULTS

## DISKETTE USE

### DISKETTE LOADING

Refer to Figure 1-A or 1-B, open the disk drive door, and insert the diskette(s) so that the diskette label faces the open door. Then carefully close the drive door.

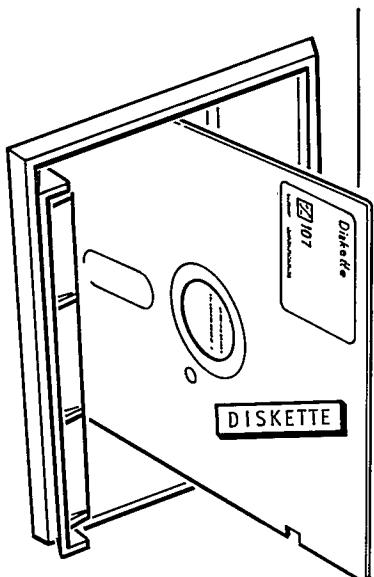


Figure 1-A  
Inserting 5.25-inch diskettes.

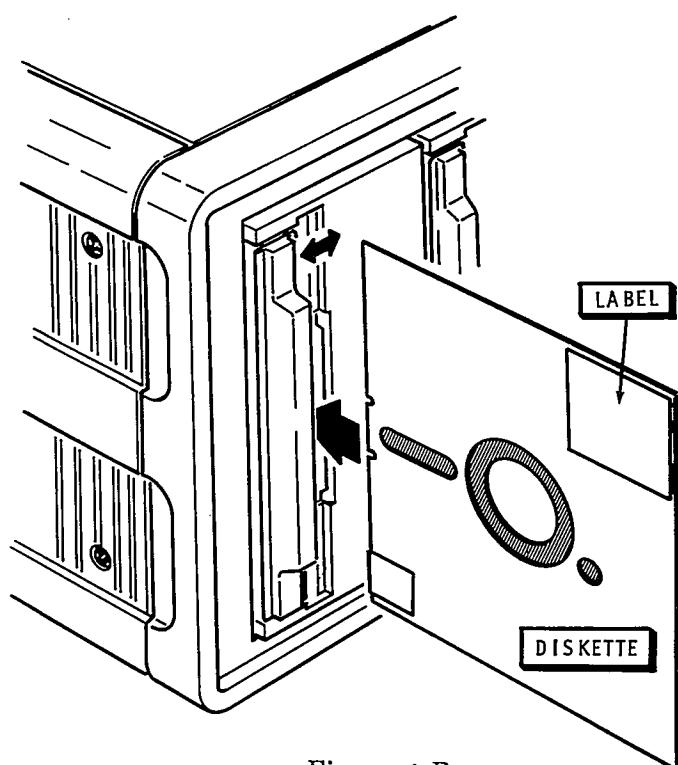


Figure 1-B  
Inserting 8-inch diskettes.

## DISKETTE HANDLING

Diskettes are easily damaged. Observe the following precautions when you are handling diskettes:

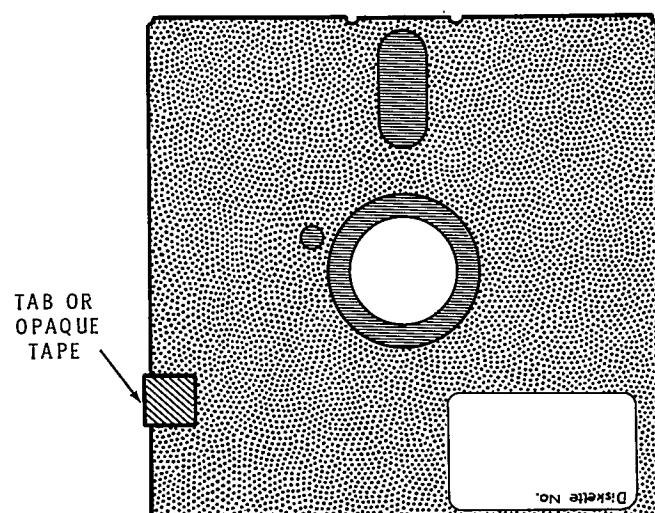
1. Keep the diskette in its storage envelope whenever it is not in use.
2. Keep the diskette away from magnetic fields, including magnetic paper clip holders, magnetized scissors or screwdrivers, and heavy electrical equipment. Magnetic fields can distort the data recorded on the diskette.
3. Replace damaged or excessively worn storage envelopes.
4. Write only on the diskette label, and then only with a felt-tip pen. Do not use a pencil or ball-point pen, as these may damage the recording surface.
5. Keep the diskette away from hot or contaminating material.
6. Do not expose the diskette to sunlight, liquids, or smoke.
7. Do not touch the diskette surface. Abrasions can alter stored data.

## WRITE-PROTECTION

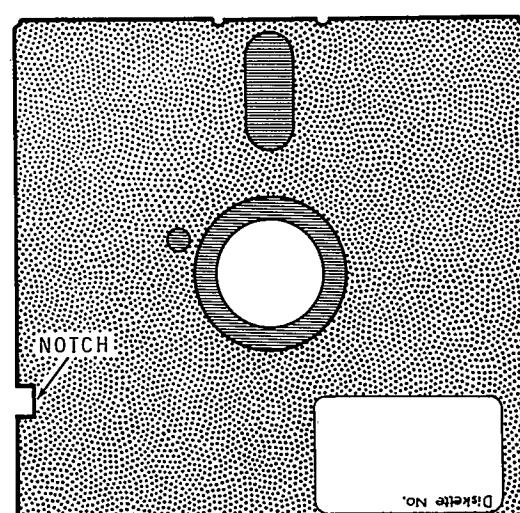
The diskette can be write-protected so that data cannot be written to it. (All distribution diskettes are shipped write-protected). The method of write-protection depends on the size of the diskette.

A 5.25-inch diskette has a write-protect notch on the side. When this notch is covered with a tab or opaque tape, no data can be written on the diskette. Figure 2-A on Page 0-9 illustrates a write-protected 5.25-inch diskette; Figure 2-B (also on Page 0-9) depicts a write-enabled 5.25-inch diskette.

An 8-inch diskette has a write-enable notch on its side. If this write-enable notch is exposed, no data can be written to the diskette. To write-enable an 8-inch diskette, cover the write-enable notch with a tab or opaque tape. Figure 3-A on Page 0-9 shows a write-protected 8-inch diskette; Figure 3-B (also on Page 0-9) shows a write-enabled 8-inch diskette.



WRITE-PROTECTED



WRITE-ENABLED

Figure 2-A

Figure 2-B

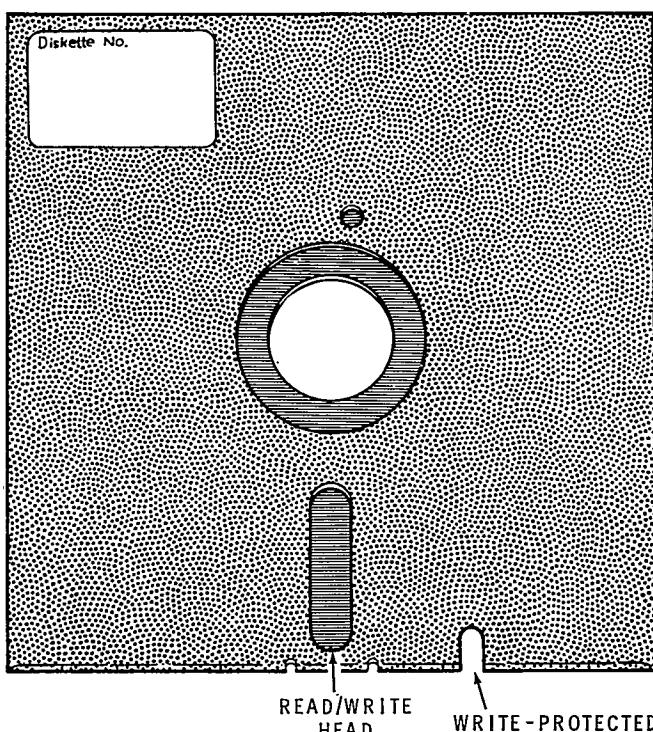


Figure 3-A

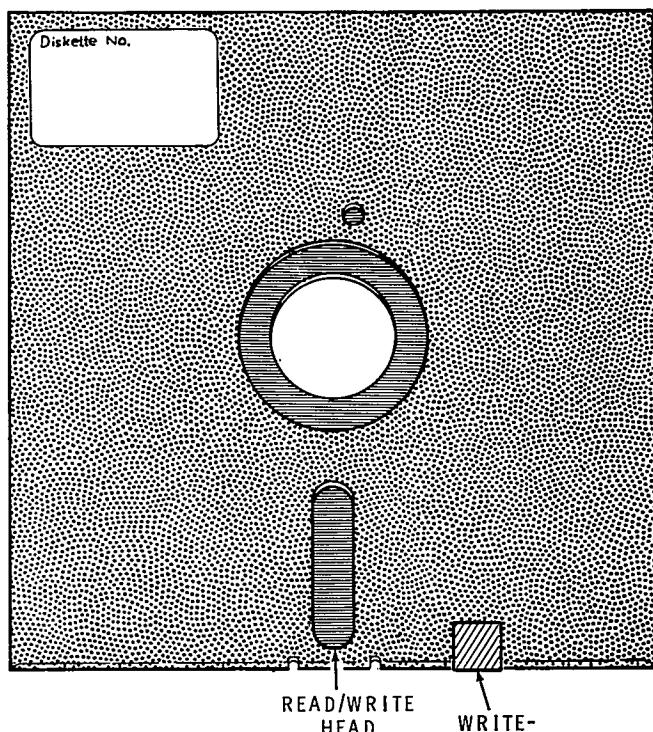


Figure 3-B

## PREPARING WORKING DISKETTES

Using the procedure outlined in your CP/M manual, power-up your computer and boot-up CP/M from CP/M Distribution Disk I.

If you have two or more drives of the same size, duplicate your distribution diskette(s) using DUP.COM. If you do not have two or more drives of the same size:

1. Initialize the blank diskette(s) to which you will copy using FORMAT.COM.
2. Duplicate the distribution disk(s) using PIP.COM.

NOTE: All distribution diskettes are write-protected to ensure that you always have an accurate copy of the software. Therefore, duplicate the distribution diskettes and then store them in a safe place. Use your copies for day-to-day use of the programs.

## Chapter One

# Introduction

The BASIC Compiler (BASCOM.COM) is a utility designed to create relocatable machine code files from source programs written in Microsoft BASIC. In effect, the Compiler converts each line of the BASIC program into assembly language statements; and then, in turn, converts the assembly language statements into relocatable machine code. You can then load and link this relocatable machine code with the LINK-80 Linking Loader (L80.COM) in order to execute the relocatable file and/or store it as a CP/M command (.COM) file.

The action of the BASIC Compiler on a source program is different from the action of the BASIC Interpreter. The primary difference lies in the manner in which programs are converted to machine code and executed.

When you type RUN under the BASIC-80 Interpreter, it proceeds sequentially through the source program, supplying machine code to perform the function(s) of each program line. The program lines are converted one line at a time, and the conversion takes place in consideration of the current environment (i.e., current DIMs, current DEFxxx statements, etc.). Because of the ongoing nature of the conversion of program lines, the Interpreter is capable of reconverting statements to reflect a change in the current environment. Thus, arrays may be redimensioned, variable types redefined, and so on.

The Compiler, on the other hand, evaluates and converts all the statements in the source program only once, at the time of compilation. The Compiler performs the evaluation and conversion by scanning the entire source program, observing the interrelationships between program lines, and then generating relocatable code to perform the functions of the various BASIC-80 statements. The Compiler does not execute or interpret the statements it encounters; it merely translates them into machine language. The relocatable code must be loaded into memory using LINK-80 before it can be executed. Since the relocatable code generated by the Compiler is based upon all the statements in the source program, the code is, for all practical purposes, unalterable. The only way to alter any part of a relocatable program is to recompile it.

Since the BASIC Compiler translates the program as a unit, it handles some statements differently than does the Interpreter. These differences are the subject of this Chapter.

## UNIMPLEMENTED DIRECT MODE STATEMENTS

Unlike the BASIC-80 Interpreter, the BASIC Compiler interacts with the console only to read Compiler commands and to determine which files to compile. The Compiler does not accept commands in the direct or immediate mode. Thus, most commands which are normally issued in the direct mode under the BASIC-80 Interpreter are not implemented in the Compiler, and will generate an error message. The Interpreter direct mode statements and commands not implemented in the Compiler include:

```
AUTO  CLEAR  COMMON  CONT    DELETE  EDIT  
LIST   LLIST   LOAD     MERGE   NEW    RENUM  SAVE
```

Since there is no direct mode in the Compiler which would enable you to enter and edit programs, use either a text editor or the BASIC-80 Interpreter for creating and editing programs. If you use the Interpreter to create programs, be sure to **SAVE** the program using the A (ASCII format) option. Programs which have been **SAVED** in compressed text will generate Compiler fatal syntax errors.

## LANGUAGE DIFFERENCES

Most programs that run under the Microsoft BASIC-80 Interpreter will run under the BASIC Compiler with little or no change. There are, however, differences in the use of some program mode statements.

### CALL

The <variable name> field in the CALL statement must contain an external symbol (i.e., a symbol which LINK-80 recognizes as a global symbol). The syntax is:

```
CALL SUBRNAME
```

For example:

```
CALL TEST
```

The CALLed subroutine must have been assembled or stored in a relocatable format. It can be supplied by the user as an assembly language subroutine, or CALLed from the BASIC or FORTRAN-80 library. You can also generate your own library or add to or modify the BASIC and FORTRAN libraries using the LIB-80 Library Manager (LIB.COM). For more information about LIB-80, consult the LIB-80 Utility Software Manual.

If you use a CALL statement within a program, be sure to specify the external symbol (i.e., the subroutine module name) within the L80 command line during loading. Refer to the LINK-80 Utility Software Manual for the command line syntax.

## COMMON

The COMMON statement has not been implemented in the Compiler, and will generate a fatal error.

## CHAIN and RUN

The CHAIN and RUN statements have been implemented only in the forms CHAIN "FILENAME" and RUN "FILENAME". The default extension is .COM. For example:

```
1020 CHAIN "TEST"
```

The machine code generated for this program line will cause CP/M to load and execute the machine code program TEST.COM.

Compiled programs can CHAIN to any .COM file, but command line arguments (as in PIP A:=B:\*.\*) are not automatically passed. To pass command line information to the chained program, POKE the appropriate information into the command line area (80H).

**DEFINT  
DEFSNG  
DEFDBL  
DEFSTR**

The Compiler does not execute statements of the form DEFxxx. Rather, it simply converts such statements. Since the Compiler translates but does not execute each line of the source program, the conversion takes place without regard for any previous program line, regardless of whether or not a previous line may affect the evaluation of a variable or expression. Thus, where a variable type is defined twice, only the last definition of variable type in the source program affects the variable type in the compiled program. To illustrate this, consider the following BASIC-80 program:

```
10 DEFINT A
20 GOTO 40
30 DEFSNG A
40 A=4/3
50 PRINT A
60 END
```

Under the BASIC-80 Interpreter, this program returns a value of 1 for A, since the GOTO in line 20 causes MBASIC to skip over line 30. But since the Compiler converts each line sequentially, and since the last definition of variable type for A in the numeric sequence of the program is single precision at line 30, the compiled version of this program returns a value of 1.33333 for A.

We urge that you define all variables and variable types at the beginning of the program in order to avoid unintentional duplication.

**DIM and ERASE**

The Compiler handling of the DIM statement is similar to the handling of the DEFxxx statement, in that the Compiler scans DIMs instead of executing them. That is, during compilation, DIM takes effect when its line is encountered within the numeric sequence of program lines. If the default dimension (10) has already been established for an array variable, and that variable is later encountered in a DIM statement, an "Array Already Dimensioned" error results.

Also note that the values of the subscripts in a DIM statement must be integer constants; they may not be variables, arithmetic expressions, or floating point values. For example,

```
DIM BL(I)
DIM BL(1+1)
```

are both illegal.

There is no ERASE statement in the Compiler, so arrays cannot be erased and then redimensioned. An ERASE statement will produce a fatal error.

## END

During the execution of a compiled program, the END statement closes files and returns control to the operating system. Omitting the END statement does not affect program termination, since the Compiler assumes an END statement after the last line of the program.

## FOR/NEXT and WHILE/WEND

FOR/NEXT and WHILE/WEND statements must be statically nested. That is, FOR/NEXT and WHILE/WEND routines must not contain GOTOS, or GOSUBS, etc., which might cause execution of the loop to be terminated early. The following example is invalid because it contains a conditional GOTO:

```
10 FOR X=1 TO 10
20 INPUT Y
30 IF Y<X THEN 50
40 NEXT
50 END
```

## %INCLUDE

The %INCLUDE statement enables you to instruct the Compiler to compile a source program from an alternate file and include that source code as part of the current source program. The Compiler supplies a default extension of .BAS. %INCLUDE must be the last statement on a line. The general format of the %INCLUDE statement is:

```
%INCLUDE FILENAME
```

For example:

```
90 %INCLUDE TEST
90 FOR X=1 TO 20:PRINT:NEXT:%INCLUDE TEST
```

**ON ERROR GOTO <line number>RESUME <line number>**

If a program contains ON ERROR GOTO and/or RESUME <line number> statements, use the /E compilation switch. If a program contains the RESUME, RESUME 0, or RESUME NEXT statements, use the /X switch instead of the /E switch. See Chapter 2 for an explanation of the /E and /X switches.

**REM**

REM statements and remarks which are delimited with an apostrophe do not take up time or space during execution. Thus, you may use them freely.

**STOP**

The STOP statement is identical to the END statement. Open files are closed and control returns to the operating system.

**TRON and TROFF**

In order to use TRON and TROFF within a program, use the /D compilation switch. Otherwise, TRON and TROFF are ignored and the Compiler issues a warning message.

**USRn Functions**

Under the BASIC Compiler, the argument to the USRn function is ignored and an integer result is returned in the HL registers. We recommend that you use CALL instead of the USRn function.

**ARITHMETIC EVALUATION DIFFERENCES**

The Compiler handles some arithmetic operations differently than does the Interpreter. The two primary areas of difference are expression evaluation and the handling of integer variables.

## Expression Evaluation

During expression evaluation, the Compiler converts the operands of each operator to the type of the most precise operand. For example, the expression:

```
QR=J%+A!+Q#
```

causes J% to be converted to single precision and added to A!. This result (J%+A!) is then converted to double precision and added to Q#.

The Compiler is more limited than the Interpreter in its handling of numeric overflow. For example, when run under the Interpreter, the program

```
I%=20000  
J%=20000  
K%=-30000  
M%=I%+J%-K%
```

yields 10000 for M%. That is, it adds I% to J%, and, because the number is too large, it converts the result into a floating point number. K% is then converted to floating point and subtracted. The result, 10000, is converted back to integer and saved as M%.

The Compiler, however, must make type conversion decisions during compilation — it cannot defer the decision until the actual values of the variables are known. Thus, the Compiler would generate code to perform the entire operation in integer mode. If the /D switch had been set before compilation, the error would be detected. Otherwise, the Compiler would generate an incorrect answer. Refer to Chapter 2 for a discussion of the /D switch.

In order to produce optimum efficiency in the compiled program, the Compiler may perform any number of valid algebraic transformations before generating the code. For example, the program

```
I%=20000  
J%=-18000  
K%=20000  
M%=I%+J%+K%
```

might produce an incorrect result when run under the Compiler. If the Compiler performs the arithmetic in the order shown, no overflow occurs.

## Integer Variables

In order to produce the fastest and most compact object code possible, make maximum use of integer variables. For example, the program

```
10 FOR I=1 TO 10
20 A(I)=0
30 NEXT I
```

can execute approximately 30 times faster if you substitute the variable "I%" for "I". It is an especially good practice to use integer variables to compute array subscripts. The generated code is significantly faster and more compact.

When you use variables with logical operators, be sure to declare the variable as type integer. Otherwise, the Compiler may produce erroneous results. For example, the program

```
10 A=7
20 B=-5
30 PRINT A AND B
```

yields the correct result of 3 when run under the Interpreter, but the compiled version of the program returns 4. You can obtain the correct result under both Interpreter and Compiler by simply declaring the variables as type integer.

## OTHER DIFFERENCES

The Compiler cannot accept a physical line longer than 127 characters in length. A logical statement, however, may contain as many physical lines as you prefer. Use LINE FEED to start a new physical line within a logical statement.

To reduce the size of the compiled program, the Compiler does not include line numbers in the object code unless the /D, /E, or /X switch is set in the Compiler command. Therefore, error messages contain the address where the error occurred rather than a line number. The Compiler listing and the map generated by LINK-80 are used to identify the line containing the error. We recommend that you debug programs using the BASIC-80 Interpreter before attempting to compile them.

## Chapter Two

# BASIC Compiler Command Format

To invoke the BASIC Compiler, type:

```
A>BASCOM @
```

The Compiler will return the prompt \*, which indicates that it is ready to accept commands. To inform the BASIC Compiler what to compile and which options to use, it is necessary to enter a "command string" which is read by the Compiler command scanner. The general format of a BASIC Compiler command string is:

```
*X:OBJCODE.EXT,X:LISTFIL.EXT=X:SRCEPROG.EXT @
```

where X:OBJCODE.EXT is the object code file generated by the Compiler; X:LISTFIL.EXT is the listing file; and X:SRCEPROG.EXT is the file which contains the source program.

You can also invoke the Compiler by typing:

```
A>BASCOM A:X:OBJCODE.EXT,X:LISTFIL.EXT=X:SRCEPROG.EXT @
```

This command causes CP/M to load the Compiler and then immediately execute the command string. The effect of the preceding command is the same as:

```
A>BASCOM @  
*X:OBJCODE.EXT,X:LISTFIL.EXT=X:SRCEPROG.EXT @
```

No matter how you invoke the Compiler, it returns control to CP/M after the program has been compiled. If for some reason you need to exit before compilation is complete, type CTRL-C.

## Source Program File X:SRCEPROG.EXT

The X:SRCEPROG.EXT portion of the command string is the name of the source program file, including device specification (X:), which will be compiled. The X:SRCEPROG.EXT portion of the command string format is the only portion which cannot be omitted.

You may omit the drive specification of the X:SRCEPROG.EXT portion of the command string if the program you will compile is on the disk in the default drive. Thus, for example, the command:

```
*X:OBJCODE.EXT,X:LISTFIL.EXT=TEST.BAS
```

causes the Compiler to compile the program called TEST.BAS on the default drive.

You can also omit the extension, in which case the Compiler will supply a default extension of .BAS. The source program must have been stored in ASCII by means of the BASIC-80 SAVE “FILENAME”, A format. In the following example, the Compiler would compile a source program called TEST.BAS on the default drive:

```
*X:OBJCODE.EXT,X:LISTFIL.EXT=TEST
```

Note that both this and the previous example have exactly the same effect.

## Object Code File X:OBJCODE.EXT

The X:OBJCODE.EXT portion of the command string is the name of the file, including device specification (X:) where the binary object code will be written. The X:OBJCODE.EXT portion of the command string may be omitted entirely.

You may omit the drive specification of the X:OBJCODE.EXT portion of the command string if you want the object code to be stored on the disk in the default drive. For example, the command string:

```
*TEST.OBJ,X:LISTFIL.EXT=TEST
```

instructs the Compiler to put the object code generated from the source program TEST.BAS into a file called TEST.OBJ on the default drive.

The .EXT portion of the X:OBJCODE.EXT format may also be omitted. In such a case, the Compiler supplies the default extension, .REL. For example, the command line:

```
TEST,X:LISTFIL.EXT=TEST
```

causes the Compiler to write the object code to a file named TEST.REL on the default drive.

You can also omit the X:OBJCODE.EXT portion of the command string entirely, in which case the Compiler will place the object code in a file whose FILENAME is that of the source program, and whose extension is .REL. The general format for this command is:

```
*=X:SRCEPROG.EXT
```

For example:

```
*=TEST
```

The effect of this command is to compile the file named TEST.BAS on the default drive and create an object code file called TEST.REL on the default drive. No listing file will be generated.

Note that when you do not specify a relocatable file name using the =X:SRCEPROG.EXT format, the drive specification of the generated relocatable file is the same as the drive specification of the source file. In the following example, the Compiler would compile the file named TEST.BAS on drive D: and place the relocatable file TEST.REL on drive D:

```
*=D:TEST
```

If you do not want to create an object file at all, you can explicitly instruct the Compiler not to generate an object file. To do this, substitute a comma for the object code FILENAME, as in the general format:

```
,X:LISTFIL.EXT=X:SRCEPROG.EXT
```

## Listing File X:LISTFIL.EXT

The X:LISTFIL.EXT portion of the command string is the name of the file, including device specification (X:) to which the program listing will be written. You may replace the X:LISTFIL.EXT portion of the command string with a listing device name; or you may omit it entirely, in which case the Compiler will generate no listing file. The program listing is an assembly language translation of the source program which utilizes Intel\* mnemonics. As an example, for the BASIC program

```
10 FOR X=1 TO 10
20 PRINT X
30 NEXT
40 END
```

the Compiler will produce the following listing file:

```
BASCOM 5.23 - Copyright 1979, 80 (C) by MICROSOFT - 07597 Bytes Free
0014 0007      10 FOR X=1 TO 10
    ** 0014'      CALL    $5.0
    ** 0017'L00010: CALL    $LFMA
    ** 001A'      DW      <const>
    ** 001C'      JMP    I00000
    ** 001F'I00001:
001F 0007      20 PRINT X
    ** 001F'L00020: CALL    $PROA
    ** 0022'      LXI    H,X!
    ** 0025'      CALL    $PV2A
0028 000B      30 NEXT
    ** 0028'L00030: CALL    $FADA
    ** 002B'      DW      X!
    ** 002D'      DW      <const>
    ** 002F'I00000:
    ** 002F'      CALL    $FAS0
    ** 0032'      DW      X!
    ** 0034'      CALL    $LEJA
    ** 0037'      DW      X!
    ** 0039'      DW      <const>
    ** 003B'      DW      I00001
003D 000B      40 END
    ** 003D'L00040: CALL    $END
0040 000B
    ** 0040'      CALL    $END
004D 0015

00000 Fatal Error(s)
07334 Bytes Free
```

Note in the preceding listing how the Compiler interprets each line of the BASIC program into an assembly language subroutine. The Compiler lists each line of the BASIC source program along with the assembly language subroutine which it generated for that line. For example, below line 10 of the BASIC program is a corresponding assembly language subroutine labeled "L00010:". Subroutine L00010, and all other subroutines in this listing, contain the labels for entry points into the BASIC library, such as \$LFMA at address 17 of subroutine L00010. The dollar sign preceding some labels indicates to LINK-80 that the label refers to a routine external to this relocatable file.

You can specify the X:LISTFIL.EXT portion of the command string without a device specification if you want the listing file to be written to the default drive. In the following example, the operator instructed the Compiler to send the program listing to a file called TEST.LST on the default drive:

```
*TEST, TEST.LST=TEST
```

The .EXT portion of the X:LISTFIL.EXT format may also be omitted. If the extension has been omitted, the Compiler supplies a default extension of .PRN. The following example will create a listing file called TEST.PRN on the default drive:

```
*TEST, TEST=TEST
```

Instead of having the listing file stored in a disk file, you may substitute the name of the logical listing device (LST:) or printer/terminal (TTY:) for the X:LISTFIL.EXT file name format. In the following example, the listing file would be printed on the printer/terminal instead of being written to a file:

```
*TEST, TTY:=TEST
```

You can also omit the X:LISTFIL.EXT portion of the command string altogether, in which case the Compiler will not generate a listing file. Use the general formats:

```
*X:OBJPROG.EXT=X:SRCEPROG.EXT
```

or

```
*=X:SRCEPROG.EXT
```

If you want to instruct the Compiler to omit both object code and listing file, replace both portions of the command string with a comma, as in the following general format:

\* ,=X:SRCEPROG.EXT

## Command String Examples

The following examples summarize the conventions for and restrictions upon Compiler command strings. Each sample command is listed along with a description of the effect of the command. The \* at the beginning of each sample command string is the Compiler prompt, so you need not enter it.

\*B:TEST.OBJ,B:TEST.LST=TEST

Compile the program TEST.BAS; place the object code in B:TEST.OBJ, and the program listing in B:TEST.LST.

\*TEST,B:TEST.LST=TEST

Compile the program TEST.BAS; place the object code in TEST.REL, and the program listing in B:TEST.LST.

\*TEST,TEST=TEST

Compile the program TEST.BAS; put the object code in TEST.REL, and the program listing in TEST.PRN.

\* ,TTY:=TEST

Compile the program TEST.BAS; create no object code, and send the program listing to the printer/terminal.

\* ,TEST=TEST

Compile the program TEST.BAS; create no object code, and place the program listing in TEST.PRN.

\* ,=TEST

Compile the program TEST.BAS; create no object code or program listing.

\*=B:TEST

Compile the program B:TEST.BAS; put the object code in B:TEST.REL; generate no program listing.

## Compiler Switches

A switch appended to a Compiler command string specifies a special parameter to be used during compilation. Switches are always preceded by a slash (/). Multiple switches may be used in the same command.

### /E

The /E switch informs the Compiler that the program to be compiled contains the ON ERROR GOTO statement. If you have used any RESUME statement other than RESUME <line number> with the ON ERROR GOTO statement, use the /X switch instead of /E (see below). In order to handle ON ERROR GOTO properly in a compiled environment, the Compiler must generate some extra code for the ON ERROR GOTO and RESUME statements. Therefore, to conserve memory, do not use the /E switch unless your program contains the ON ERROR GOTO statement.

The /E switch causes line numbers to be included in the binary file, so runtime error messages will return the number of the line rather than the address at which the error occurred.

### /X

The /X switch informs the Compiler that the program to be compiled contains one or more RESUME, RESUME NEXT, or RESUME 0 statements. The Compiler assumes /E when /X has been specified. Since the Compiler must relinquish certain optimizations when compiling RESUME statements, do not use the /X switch unless your program contains a RESUME statement other than RESUME <line number>.

Like the /E switch, /X also produces line numbers in the binary file, so runtime error messages will return the number of the line instead of the address at which the error occurred.

### /N

The /N switch prevents the listing of the generated code in symbolic notation. If this switch is **not** set, the program listing produced by the Compiler will contain the labels (entry points) of subroutines as well as the Intel mnemonics generated for each statement.

The following listing was generated from a program line compiled normally (i.e., without /N). Note how the listing includes the line numbers of the original BASIC source program, labels (entry points), and Intel mnemonics, as well as addresses:

```
BASCOM 5.23 - Copyright 1979, 80 (C) by MICROSOFT - 07597 Bytes Free
0014 0007      10 PRINT "THIS LINE WAS COMPILED WITHOUT /N"
    ** 0014'      CALL    $5.0
    ** 0017'L00010: CALL    $PROA
    ** 001A'      LXI    H,<const>
    ** 001D'      CALL    $PV2D
0020 0007
    ** 0020'      CALL    $END
0049 0011

00000 Fatal Error(s)
07479 Bytes Free
```

The following listing was generated from a program line compiled using /N. Note how the listing includes only the lines of the original source program and addresses:

```
BASCOM 5.23 - Copyright 1979, 80 (C) by MICROSOFT - 07597 Bytes Free
0014 0007      10 PRINT "THIS LINE WAS COMPILED USING /N"
0020 0007
0047 0011

00000 Fatal Error(s)
07481 Bytes Free
```

#### /D

The /D switch causes the Compiler to generate debugging code at runtime. This switch must be set if you want to use TRON/TROFF. The Compiler generates somewhat larger and slower code in order to perform the following checks:

1. Arithmetic overflow. All integer and floating point arithmetic operations are checked for overflow and underflow.
2. Array bounds. The Compiler checks all array references to determine whether the subscripts are within the bounds specified in the DIM statement.
3. RETURN is checked for a prior GOSUB.

The Compiler includes line numbers in the generated binary code so that any runtime errors can be isolated to the statement which contains the error.

**/Z**

The /Z switch instructs the Compiler to use Z-80 opcodes wherever possible. The program listing reflects all instances where Z-80 opcodes have been used.

**/S**

The /S switch instructs the Compiler to write quoted strings longer than four characters to the binary file as they are encountered. This enables the Compiler to use less memory in compiling large programs with many quoted strings. There are two main disadvantages: memory space is wasted if the program contains identical long quoted strings, and code generated by means of the /S switch cannot be placed in ROM.

**/4**

The /4 switch instructs the Compiler to use the lexical conventions of the Microsoft 4.51 BASIC Interpreter. According to 4.51 conventions, spaces are insignificant, variables with embedded reserved words are illegal, variable names are restricted to two significant characters, and so on. This feature is useful if you want to compile a program which contains lines such as:

```
FORI=ATOBSTEPC
```

If the /4 switch has not been set, the Compiler will assign the variable ATOBSTEPC to the variable FORI. If the /4 switch has been set, the Compiler will recognize this as a FOR statement.

We strongly urge that you upgrade your programs to the 5.0 lexical standards instead of using the /4 switch. The use of spaces to delimit statements causes no increase in the generated code, and furthermore improves readability.

**/C**

The /C switch tells the Compiler to relax line numbering restraints. When /C has been specified, line numbers may appear in any order, or may be eliminated entirely. Lines are compiled normally but, of course, they cannot be the targets of GOTOS, GOSUBs, etc.

Note that /C and /4 cannot be used in the same command string.

### SWITCH EXAMPLES

The following sample command strings illustrate valid uses and combinations of Compiler switches. The \* is the Compiler prompt—you do not need to enter it.

- |                 |  |
|-----------------|--|
| * , TTY:=TREK/N | Compile TREK.BAS and send the listing to the printer/terminal, but do not include Intel mnemonics in the listing file.   |
| *=TEST/E        | Compile TEST.BAS, which contains ON ERROR GOTO and RESUME <line number> statements. Put the object code in TEST.REL.   |
| *=TEST/X        | Compile TEST.BAS, which contains ON ERROR GOTO and RESUME, RESUME 0, and RESUME NEXT statements. Put the object code in TEST.REL.                                  |
| *=BIGFILE/D     | Compile BIGFILE.BAS, and check it for overflow and out-of-bound array subscripts. Include line numbers in the object code file, and put the object in BIGFILE.REL. |
| *EDIT=EDIT/Z    | Compile EDIT.BAS. Use Z-80 opcodes wherever possible. Put the object code in EDIT.REL.   |

## Chapter Three

# Compiling and Executing Programs

Within the limitations discussed in Chapter One, you can compile any Microsoft BASIC source program using the BASIC Compiler. To create BASIC source programs, use the BASIC-80 Interpreter or a text editor. FILENAMES should be eight characters or less, with three-character extensions. BASIC source programs should have the extension .BAS. As we noted earlier, the Compiler supplies a default extension of .BAS to any FILENAME in the command string which does not explicitly supply an extension. All source programs must have been stored in ASCII using the BASIC-80 SAVE "FILENAME",A format.

Before you attempt to compile the source program, we recommend that you perform a simple syntax check by running the program under the BASIC-80 Interpreter. This will help to eliminate the need to recompile due to syntax or other easily corrected errors.

You can also check for errors by compiling without producing an object or listing file. For example, if your BASIC source file was called TEST.BAS, you would type:

```
A>BASCOM ,=TEST
```

This command instructs the Compiler to compile the BASIC program TEST.BAS without producing an object or listing file.

If necessary, return to the Editor or Interpreter and correct any errors.

## Compiling Programs

To compile the edited program and produce an object and listing file, use the general format:

A>BASCOM△OBJCODE,LISTFILE=SRCEPROG/switch1/switch2 ↵

or

A>BASCOM ↵

\*OBJCODE,LISTFILE=SRCEPROG/switch1/switch2 ↵

Either form of the command will cause the Compiler to create a relocatable (.REL) object code file and a program listing (.PRN) file.

## Executing and Saving Compiled Programs

To execute a compiled program, use the LINK-80 Loader. To run the loader, type:

A>L80 ↵

LINK-80 will return the prompt \* , which means that it awaits a command. The syntax for LINK-80 commands is:

\*RELFNAME/switch1/switch2 ↵

where RELFNAME is the FILENAME of the relocatable file (.REL file) which you want to execute, and /switchN is a LINK-80 switch. You can also append a command line to the CP/M run command, as in the form:

A>L80△RELFNAME/switch1/switch2 ↵

LINK-80 supplies a default .EXT of .REL to any FILENAME which does not explicitly supply an extension.

Because the relocatable file generated by the Compiler from your source program utilizes many of the subroutines in the BASIC library (BASLIB.REL), before executing LINK-80, be sure that the current default disk contains the file BASLIB.REL. You can alternatively specify BASLIB/S within the LINK-80 command line, as in the following general format:

```
A>L80△RELFNAME/switch1/switch2,X:BASLIB/S @@
```

where X: is the letter of the drive which contains the disk on which BASLIB.REL resides. The L80 /S switch instructs L80 to search the BASIC library for any subroutines which are referred to but not contained in the specified RELFNAME.

You can also specify the name of any user-defined library which you want L80 to search for subroutines not contained in the specified RELFNAME. This is especially useful if you have used the CALL statement within your BASIC source program to pass control to a subroutine which you have written. Use the syntax:

```
A>L80△RELFNAME/switch1/switch2,X:USRLIB/S @@
```

Normally, LINK-80 exits after executing your command. If something goes wrong, or if for some reason you need to exit before execution is complete, type CTRL-C.

In the following section, we will briefly discuss those LINK-80 switches which directly pertain to executing and saving compiled programs. For a more thorough and comprehensive discussion of LINK-80 switches, refer to the "LINK-80 Utility Software Manual".

/N

Use the /N switch to create an executable .COM file from the relocatable file. The syntax is:

```
A>L80△RELFNAME, COMFNAME/N @@
```

Where RELFNAME is the FILENAME of the relocatable object code file and COMFNAME is the FILENAME you want to be assigned to the absolute machine code (.COM) file. For example:

```
A>L80△TEST, TEST/N @@
```

In this example, LINK-80 would link and load the relocatable file TEST.REL and then create a file called TEST.COM on the default drive.

/E

The /E switch causes LINK-80 to exit to CP/M. Before exiting, LINK-80 creates a memory image of the object code, which can then be saved for later execution. This switch is useful if you want to create a .COM file from the memory image. Use the syntax:

A>L80 $\Delta$ RELFILE $\Delta$ E  $\oslash$

For example:

A>L80 $\Delta$ TEST $\Delta$ E  $\oslash$

This example causes LINK-80 to create a memory image of TEST.REL and then exit. When LINK-80 exits, it will print three numbers: the first is the starting address for execution of the program; the second is the end address of the program; and the third is the number of 256-byte memory pages used. For example:

[210C 301A 48]

In this example, the program begins execution at address 210C, ends at address 301A, and occupies 48 256-byte pages of memory.

To save the memory image as a .COM file after LINK-80 exits, use the CP/M SAVE command (for a discussion of the CP/M SAVE command, refer to the "CP/M Applications Programmer's Manual"). The number of pages of memory used (the third number printed by LINK-80 before exiting) is the argument to the SAVE command. Use the general format:

A>SAVE $\Delta$ nn $\Delta$ FILENAME.COM  $\oslash$

where nn is the number of pages used, and FILENAME is the name you want to have assigned to the relocatable file. For example, if the output from LINK-80 were

[210C 301A 48]

you would type:

A>SAVE $\Delta$ 48 $\Delta$ TEST.COM  $\oslash$

To execute the new .COM file from the CP/M command mode, use the general format:

A>FILENAME @@

For example:

A>TEST @@

/G

The /G switch causes LINK-80 to load the specified program into memory and then execute it. Use the syntax:

A>L80△RELNAME/G

For example:

A>L80△TEST/G

This command would cause LINK-80 to load and execute TEST.REL. As with the E command, you can create a .COM file after execution by using the CP/M SAVE command syntax:

A>SAVE△nn△FILENAME.COM @@

where nn is the number of 256-byte pages of memory used.

### LINK-80 COMMAND EXAMPLES

The following sample commands constitute valid input to LINK-80. Note that, except where noted, these sample commands presuppose that BASLIB.REL resides on the current default disk.



A>L80<sub>Δ</sub>TEST, B:TEST/N/E

Load and link the file TEST.REL from the default drive, and create TEST.COM on drive B: before exiting.

A>L80<sub>Δ</sub>TEST/G

Load and execute TEST.REL from the default drive.

A>L80<sub>Δ</sub>C:TEST, TEST/N/G

Load and link TEST.REL from drive C:; create TEST.COM on the default drive; execute TEST.REL.

A>L80<sub>Δ</sub>TEST/E

Load and create a memory image of TEST.REL, which resides on the default drive.

A>L80  
\*D:TEST, TEST/N

Execute LINK-80.  
Load TEST.REL from drive D:; Create TEST.COM on the default drive.

A>L80<sub>Δ</sub>D:TEST/G, C:BASLIB.REL/S

Load and execute TEST.REL from drive D:; search BASLIB.REL on drive C: in order to resolve any unresolved references to subroutines.

A>L80<sub>Δ</sub>TEST, TEST/N/G, E:MYLIB.REL/S

Load TEST.REL from the default drive; create TEST.COM on the default drive; execute TEST.REL; search MYLIB.REL on drive E: in order to resolve any unresolved references to subroutines.

## Appendix A

# Compiler Error Messages

The following errors may occur while a program is compiling. When the Compiler encounters an error, it prints a two-character code for the error, along with an arrow which indicates where in the line the error occurred. In the event that the Compiler has read beyond an error before discovering it, the arrow may point a few characters beyond the error, or at the end of a line.

The Compiler error codes fall into two general categories: fatal errors prevent the successful compilation of a program; warning errors indicate that the Compiler ignored the indicated line.

All of the following errors are produced by the Compiler during compilation. For a summary of BASIC Runtime and LINK-80 errors, refer to Appendices B and C.

## Compiler Fatal Errors

Compiler fatal errors indicate that the Compiler encountered a line which it could not compile. Fatal errors prevent the successful compilation of a program. Therefore, if you receive a fatal error, correct the source program and then recompile.

BS            Bad subscript

Causes:

Illegal dimension value.  
Wrong number of subscripts.

DD            Array already dimensioned.

FD            Function already defined.

FN FOR/NEXT error

Causes:

FOR loop index variable already in use.  
FOR without NEXT.  
NEXT without FOR.

IN Invalid use of %INCLUDE statement

LL Line too long

OM Out of memory

Causes:

Array too large.  
Data memory overflow.  
Program memory overflow.  
Too many statement numbers.

OV Math overflow

SN Syntax error

Causes:

Expected GOTO or GOSUB  
Formal parameters must be unique  
Illegal argument name  
Illegal assignment target  
Illegal constant format  
Illegal debug request  
Illegal DEFxxx character specification  
Illegal expression syntax  
Illegal FOR loop index variable  
Illegal format for statement number  
Illegal function argument list  
Illegal function name  
Illegal function normal parameter  
Illegal separator  
Illegal subroutine name  
Illegal subroutine syntax  
Illegal syntax

Invalid character  
Missing AS  
Missing BASE  
Missing comma  
Missing equal sign  
Missing GOTO or GOSUB  
Missing INPUT  
Missing left parenthesis  
Missing line number  
Missing minus sign  
Missing operand in expression  
Missing right parenthesis  
Missing semicolon  
Missing THEN  
Missing TO  
Name too long  
Single variable only allowed  
String assignment required  
String expression required  
String variable required here  
Variable required here  
Wrong number of arguments

**SQ Sequence error**

Causes:

Duplicate statement number.  
Statement out of sequence.

**TC Too complex**

Causes:

Expression too complex.  
Too many arguments in function call.  
Too many dimensions.  
Too many variables for INPUT.  
Too many variables for LINE INPUT.

TM            Type mismatch

**Causes:**

Data type conflict.  
Variables must be of same type.

UC            Unrecognizable command

UF            Function not defined

WE            WHILE/WEND error

**Causes:**

WHILE without WEND.  
WEND without WHILE.

/0            Division by zero

/E            Missing /E switch

/X            Missing /X switch

## **Compiler Warning Errors**

The Compiler issues warning errors in order to caution you that it ignored, and therefore generated no code for, the line(s) which generated the error.

ND            Array not dimensioned

SI            Statement ignored

**Causes:**

Statement ignored.  
Unimplemented command.

## Appendix B

# BASIC Runtime Error Messages

The following errors may occur while a compiled program is executing. The error numbers listed here match those issued by the BASIC Interpreter, and thus can be used with the Interpreter error variable (ERR) function.

The Compiler runtime system prints long error messages, followed by the address at which the error occurred, unless the /D, /E, or /X switch has been set before compilation. If /D, /E, or /X has been set, the error message is followed by the number of the line in which the error occurred.

<u>Number</u>	<u>Message</u>
---------------	----------------

2	Syntax error
---	--------------

Cause:

A line which contains an incorrect sequence of characters in a DATA statement.

3	RETURN without GOSUB
---	----------------------

Cause:

A RETURN statement for which there is no previous, unattached GOSUB statement.

4	Out of data
---	-------------

Cause:

A READ statement for which there are no DATA statements which have unread DATA.

Number    Message

5            Illegal function call

## Cause:

A parameter that is out of range is passed to a math or string function. A function call error may also result from:

A negative or unreasonably large subscript.

A negative or zero argument with LOG.

A negative argument to SQR.

A negative mantissa with a non-integer exponent.

A call to a USR function for which the starting address has not yet been given.

An improper argument to ASC, CHR\$, MID\$, LEFT\$, RIGHT\$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING\$, SPACE\$, INSTR, or ON. . GOTO.

A string concatenation that is longer than 255 characters.

---

**Number      Message**

6            Floating overflow or integer overflow

Cause:

The result of a calculation is too large to be represented in BASIC-80's number format. If underflow occurs, the result is zero and execution continues without an error.

9            Subscript out of range

Cause:

A reference to an array element which is outside the dimensions of the array.

11          Division by zero

Cause:

Division by zero within an expression, or the evaluation of an expression causing zero to be raised to a negative power. Machine infinity (1.7014E<sup>38</sup>) with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the exponentiation, and execution continues.

14          Out of string space

Cause:

String variables exceed the allocated amount of string space.

20          RESUME without error

Cause:

A resume statement is encountered before an error trapping routine has been entered.

Number    Message

21              Unprintable error

Cause:

There is no error message for this error condition. This is usually caused by an error with an undefined error code.

50              Field overflow

Cause:

A FIELD statement is attempting to allocate more bytes than the amount that was specified as the record length of the random file.

51              Internal error

Cause:

An internal malfunction within BASIC-80. Report to Microsoft the conditions under which this error occurred.

52              Bad file number

Causes:

A statement or command makes reference to a file using a file number for which there is no OPEN file, or the file number is beyond the range specified at the beginning of the program.

53              File not found

Cause:

A LOAD, KILL, or OPEN statement makes reference to a file which does not exist on the current default disk.

Number    Message

54              Bad file mode

Causes:

Using PUT, GET, or LOF with a sequential file; attempting to LOAD a random file; attempting to execute an OPEN using a file mode other than I, O, or R.

55              File already open

Causes:

Issuing a sequential output mode OPEN to a file that is already open; issuing a KILL for a file which is open.

57              Disk I/O error

Cause:

An error during a disk I/O operation. This is a fatal error since there is no way for the operating system to recover from it.

58              File already exists.

Cause:

The FILENAME specified in the NAME statement is identical to a FILENAME already in use on the disk.

61              Disk full

Cause:

All disk storage space is in use.

**Number**    **Message**

62              Input past end

**Causes:**

An INPUT statement is executed after all the data in a file has been INPUT, or an INPUT statement attempts to read an empty file. To avoid this error, use the EOF function to detect end of file.

63              Bad record number

**Cause:**

The record number in a PUT or GET statement is either greater than the maximum allowable (i.e., > 32767), or equal to zero.

64              Bad file name

**Cause:**

Using a FILENAME longer than eight characters with LOAD, SAVE, KILL, or OPEN.

67              Too many files

**Cause:**

Attempting to create a new file using SAVE or OPEN when all 255 directory entries are full.

## Appendix C

# LINK-80 Error Messages

The error messages listed below are the LINK-80 error messages you are most likely to encounter.

?<file> Not Found

<file>, as specified in the command line, does not exist. You will receive this message in the form “?BASLIB Not Found” if BASLIB.REL is not on the default disk and you have not specified BASLIB in the command line.

?Can't Save Object File

A disk error occurred while the file was being saved.

?Command Error

LINK-80 could not recognize your command.

?Loading Error

The last file given for input was not a properly formatted LINK-80 object file.

%Mult. Def. Global YYYYYYY

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

?No Start Address

A /G switch was issued, but no main program had been loaded.

?Nothing loaded

A FILENAME/E or FILENAME/G was specified in the command line, but no object file was loaded. That is, an attempt was made to exit the Linker or execute a program, but nothing had been loaded. For example, the command line:

TEST/N/E

results in “?Nothing loaded”, since TEST/N names TEST.COM, but does not load TEST.REL. To load TEST.REL and name TEST.COM, use the syntax:

TEST,TEST/N/E

?Out of Memory

There is not enough memory to load the program.

# Index

- ASCII format, 1-2
- Address
  - program starting, 3-3
  - program ending, 3-3
- Arguments, command line, 1-3
- Arithmetic evaluation differences, 1-6
- Arithmetic overflow, 1-7
- Array bounds, 2-8
  
- BASIC library (see BASLIB.REL)
- BASIC runtime errors, B-1
- BASIC-80
  - ASCII format option, 1-2
  - direct mode statements, unimplemented, 1-2
  - error codes, B-1
  - 4.51 conventions, 2-9
  - program mode statements, unimplemented, 1-2
  - source programs, creating, 1-2
- BASLIB.REL, 3-3
  
- /C switch, compiler, 2-9
- CALL, 1-2
- CHAIN, 1-3
- .COM files, 1-1
  - creating, 3-3
  - executing, 3-4
- Command examples
  - compiler command string, 2-6
  - compiler switch, 2-9
  - LINK-80, 3-6
- Command file, 3-4
- Command line arguments, 1-3
- Command string, compiler, 2-1
  
- COMMON, 1-3
- Compiler
  - and source program line length, 1-9
  - command format, 2-1
  - default extensions, 2-2, 2-5
  - exiting, 2-1
  - expression evaluation, 1-7
  - integer variables and, 1-8
  - invoking, 2-1
  - line numbers, 1-9
  - switches (see compiler switches)
- Compiler switch examples, 2-9
- Compiler switches
  - /C, 2-11
  - /D, 2-8
  - /E, 2-7
  - /4, 2-9
  - /N, 2-7
  - /S, 2-9
  - /X, 2-7
  - /Z, 2-9
- Conventions, notation, V
- CP/M SAVE command, 3-4
- CP/M command file, 1-1
- Creating BASIC source programs, 1-1
- Creating .COM files, 3-3
  
- /D switch (compiler), 2-8
- Debugging code, 2-8
- Default extension
  - compiler, 2-2, 2-5
  - LINK-80, 3-2, 3-3
  - with CHAIN and RUN, 1-3
  - with %INCLUDE, 1-5

DEFDBL, 1-3  
DEFINT, 1-3  
DEFSNG, 1-3  
DEFSTR, 1-3  
DIM, 1-4  
Direct mode statements, BASIC-80, 1-2

/E switch,  
  compiler, 2-7  
  LINK-80, 3-2

END, 1-5  
End address, compiled program, 3-4  
Entry points, 2-5  
ERASE, 1-4  
ERR function, BASIC-80, B-1  
Executing .COM files, 3-3  
Executing compiled programs, 3-2  
Executing relocatable files, 3-3  
Exiting  
  compiler, 2-1  
  LINK-80, 3-3

Expression evaluation, 1-7  
Extension, default  
  compiler, 2-2, 2-5  
  LINK-80, 3-2  
  with CHAIN and RUN, 1-3  
  with %INCLUDE, 1-5

Fatal errors, compiler, A-1  
FOR/NEXT, 1-5  
/4 switch, Compiler, 2-9

/G switch, LINK-80, 3-5  
GOSUB, 1-5  
GOTO, 1-5

%INCLUDE, 1-5  
Integer variables, 1-8  
Interpreter (see BASIC-80)

Labels, 2-4  
Language differences, 1-2  
Libraries, user-defined, 3-3  
  searching, 3-3  
Line length, source program, 1-9  
Line numbers, 1-9  
LINK-80  
  command examples, 3-6  
  command format, 3-2  
  default extension, 3-2  
  exiting from, 3-3  
  invoking, 3-2  
LINK-80 errors, C-1  
LINK-80 switches  
  /E, 3-4  
  /G, 3-5  
  /N, 3-3  
Listing file, 2-4  
  and Compiler /N switch, 2-7  
Logical operators, 1-8  
Logical statements, 1-8

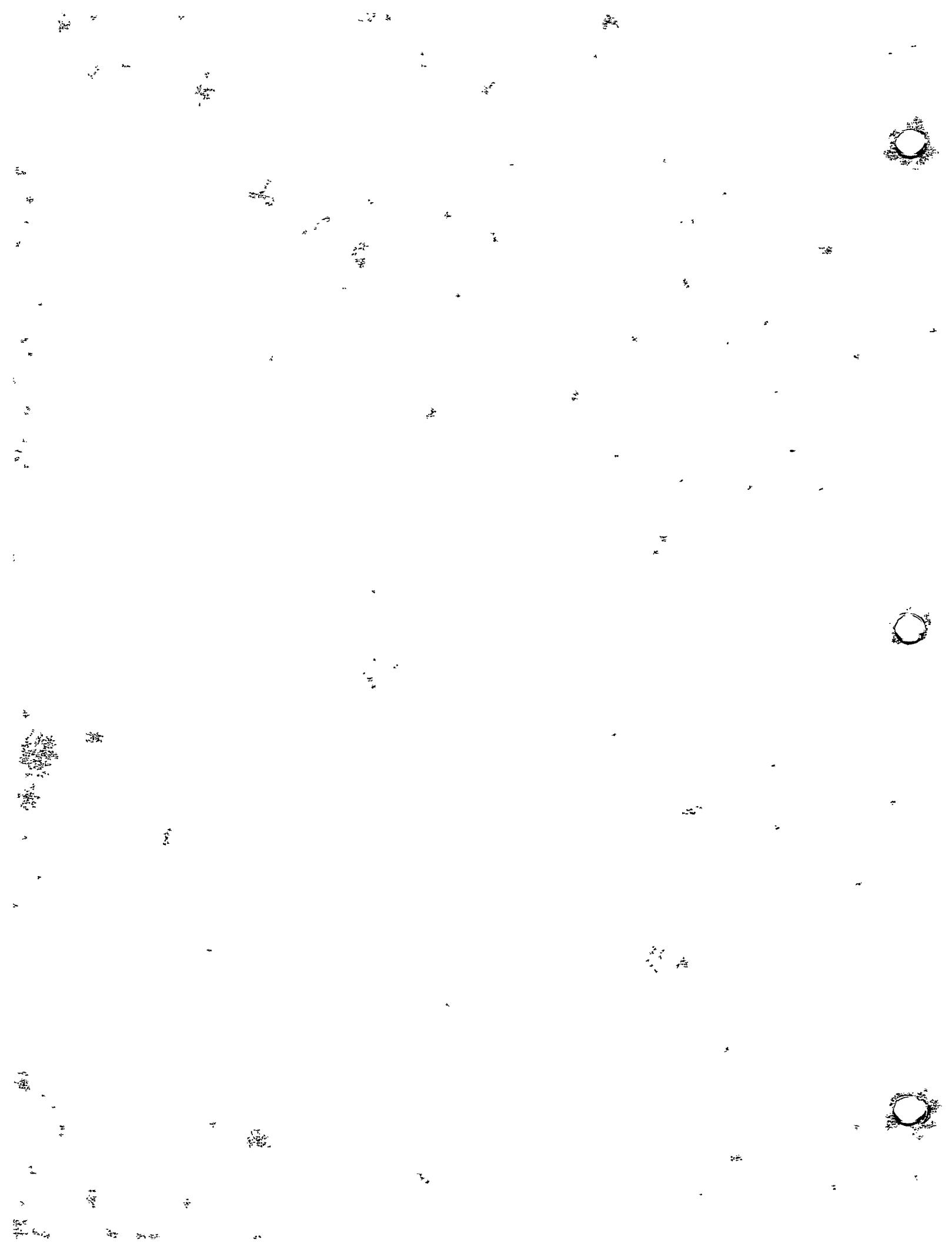
Memory image, saving, 3-4  
Memory pages, 3-4  
Mnemonics, 2-4

Notation conventions, V  
Numeric overflow, 1-7

ON ERROR GOTO, 1-6  
Object code file, 2-2  
Operator precedence, 1-7

Pages, memory, 3-4  
Printer, copying listing file to, 2-5  
Program listing, 2-4

Relocatable file, 2-2, 3-3  
Relocatable machine code, 1-1  
REM, 1-6  
RESUME, 1-6  
RESUME 0, 1-6  
RESUME <line number>, 1-6  
RESUME NEXT, 1-6  
RUN, 1-3  
  
/S switch, LINK-80, 3-3  
SAVE command  
  BASIC-80, 2-2  
  CP/M, 3-4  
Saving memory image, 3-4  
Searching libraries, 3-3  
Source program  
  format, 1-1  
  line length, 1-8  
  quoted strings in, 2-10  
Starting address, compiled program, 3-3  
Statements, unimplemented, 1-1  
STOP, 1-5  
Strings, quoted, 2-9  
Switches, LINK-80  
  /E, 3-4  
  /G, 3-5  
  /N, 3-3  
Switches, compiler  
  /C, 2-9  
  /D, 2-8  
  /E, 2-7  
  /4, 2-9  
  /N, 2-7  
  /S, 2-9  
  /X, 2-7  
  /Z, 2-9  
Symbol conventions, V  
TROFF, 1-6  
TRON, 1-6  
Unimplemented direct mode statements, 1-2  
User-defined libraries, 3-3  
USRn functions, 1-6  
  
Variables,  
  expression evaluation, 1-7  
  type conversion, 1-7  
  and logical operators, 1-8  
  
Warning errors, A-4  
WHILE/WEND, 1-5  
  
Z80 opcodes, 2-9



MICROSOFT  
**MACRO-80**

HEATH | ZENITH  
data  
systems



# **Microsoft MACRO-80 ASSEMBLER**

**CP/M® Version**

## **Software Reference Manual**

**for HEATH/ZENITH 8-bit digital computer systems**

 Copyright © 1981  
Heath Company  
All Rights Reserved

**HEATH COMPANY**  
BENTON HARBOR, MICHIGAN 49022  
CP/M is a registered trademark of Digital Research

595-2666-02  
Printed in the  
United States of America

Technical consultation is available for any problems you may encounter in verifying proper operation of this product. We are sorry, but we are not able to evaluate or assist in the debugging of any programs you may develop with this product. For technical assistance, call:

(616) 982-3884 Application Software  
(616) 982-3860 Operating Systems/Language Software

Consultation is available between 8:00 am and 4:30 pm (Eastern), on normal business days.

Zenith Data Systems  
Software Consultation  
Hilltop Road  
St. Joseph, MI 49085

Portions of this Manual have been adapted from Microsoft publications or documents.

COPYRIGHT © by Microsoft, 1979, all rights reserved.

# Table of Contents

## Chapter One — Using the MACRO-80 Assembler

Overview .....	1-1
Format of MACRO-80 Source Files .....	1-2
Statements .....	1-3
Symbols .....	1-4
Numeric Constants .....	1-4
Strings .....	1-4
Format of Commands .....	1-5
MACRO-80 Switches .....	1-7
Symbol Table Listing .....	1-10
MACRO-80 Errors .....	1-11
Error Codes .....	1-11
Error Messages .....	1-12

## Chapter Two — Expression Evaluation

Overview .....	2-1
Arithmetic and Logical Operators .....	2-2
Modes .....	2-3
Externals .....	2-4
Opcodes as Operands .....	2-4

## Chapter Three — Pseudo-Opcodes/Assembler Directives

Overview .....	3-1
Pseudo-Opcodes .....	3-2
Conditional Pseudo-Operations .....	3-9
Listing Control Pseudo-Operations .....	3-10
Relocatable Pseudo-Operations .....	3-13
ORG Pseudo-Op .....	3-14
Relocation Before Loading .....	3-15

## Chapter Four — Macros and Block Pseudo-Operations

Overview .....	4-1
Macros and Block Pseudo-Operations .....	4-2
Terms .....	4-2
Special Macro Operators and Forms .....	4-7
Using Z80 Pseudo-Ops .....	4-10

## Chapter Five — Index

IV





## Chapter One

# Using the MACRO-80 Assembler

## OVERVIEW

The MACRO-80 Assembler is an 8080/Z80 Assembler with complete facilities for macro development.

In order to use the Assembler, a source program must first be written using an editor, such as ED. A MACRO-80 source program is composed of a series of statements. The format of each statement must follow a predefined format.

After the source program has been written, it must be assembled using the Macro Assembler. The result of this process will be a relocatable module. This module must then be linked using the Linking Loader. (See Section 3, "LINK-80", for information on the Linking Loader.) After the relocatable module has been linked, it can be executed.

In order to provide the Assembler with the information it needs to successfully assemble a source program, a command string must be input. This command string tells the Assembler where to find the source program, where to put the relocatable module and where to write the listing.

There are also several switches which can be set in the command string. Some of these switches are used to control the format of the listing file. A switch can also be set to allow the Assembler to assemble Z80 mnemonics.



## FORMAT OF MACRO-80 SOURCE FILES

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

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

## Statements

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

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

It is not necessary that statements begin in column one. Multiple blanks or tabs may be used to improve readability.

If a label is present, it is the first item in the statement and is immediately followed by a colon (:). If it is followed by two colons, it is declared as PUBLIC. Therefore:

FOO:: RET

is equivalent to:

```
PUBLIC FOO  
FOO: RET
```

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

The evaluation order is as follows:

1. Macro call
2. Opcode/Pseudo-operation
3. Expression

Instead of flagging an expression as an error, the Assembler treats it as if it were a DB statement. The arguments following the operator will, of course, vary in form according to the operator.

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

## Symbols

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

A-Z      0-9      \$      .      ?      @

The underline character is also legal in a symbol. A symbol may not start with a numeric digit. Lower case symbols are translated to upper case. If a symbol reference is followed by ## it is declared external.

## Numeric Constants

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

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

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

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

## Strings

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

## FORMAT OF COMMANDS

To run MACRO-80, type M80 followed by a carriage return. MACRO-80 will return the prompt “\*”, indicating it is ready to accept commands. The format of a MACRO-80 command string is:

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

Where:

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

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

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

**filename.ext** The file name and file name extension of the object program file, the listing file, and the source file. If the file name extensions are omitted, the operating system will insert the default extensions.

The default file name extensions are:

source file	.MAC
relocatable object file	.REL
listing file	.PRN
cross reference file	.CRF

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

Examples:

(NOTE: The asterisk represents the prompt from the Assembler.)

\*EXP.REL, EXP.PRN=EXP.MAC

Assemble the program EXP.MAC and place the object file in EXP.REL and the list file in EXP.PRN.

\*=EXP

Assemble the program EXP.MAC and place the object file in EXP.REL.

\* , LST:=EXP

Assemble the program EXP.MAC, place the object file in EXP.REL and list on the device LST:.

\*SMALL, TTY:=TEST

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



## MACRO-80 Switches

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

<u>Switch</u>	<u>Action</u>
O	Print all listing addresses, etc. in octal.
H	Print all listing addresses, etc. in hexadecimal. (Default)
R	Force generation of an object file.
L	Force generation of a listing file.
C	Force generation of a cross reference file. (See Page 1-11, "Cross-Reference Facility".)
Z	Assemble Z80 (Zilog format) mnemonics.
I	Assemble 8080 mnemonics. (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, it is not needed.

**Switch**    **Action**

/M      Initialize Block Data Areas.

If the programmer wants the area that is defined by the DS (Define Space) pseudo-op initialized to zeros, then the programmer 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.

/X      The presence or absence of /X in the command line sets the initial current mode and the initial value of the default for listing or suppressing lines in false conditional blocks. /X sets the current mode and initial value of default to not-to-list. No /X sets current mode and initial value of default to list. Current mode determines whether false conditionals will be listed or suppressed.

The initial value of the default is used with the .TFCOND pseudo-op so that .TFCOND is independent of .SFCOND and .LFCOND. If the program contains .SFCOND or .LFCOND, /X has no effect after .SFCOND or .LFCOND is encountered until a .TFCOND is encountered in the file. So /X has an effect only when used with a file that contains no conditional listing pseudo-ops or when used with .TFCOND.

- The following chart illustrates the effects of the three pseudo-ops when encountered under /X and under no /X.

PSEUDO-OP	NO /X	/X
(none)	ON	OFF
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	OFF
.	.	.
.	.	.
.	.	.
.LFCOND	ON	ON
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON
.	.	.
.	.	.
.	.	.
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON

## Examples:

(NOTE: The asterisk represents the prompt from the Assembler.)

**\*=TEST/L**      Compile TEST.MAC with object file TEST.REL and listing file TEST.PRN

\*LAST, LAST/C=MOD1    Compile MOD1.MAC with object file LAST.REL and cross reference file LAST.CRF for use with CREF-80

## Symbol Table Listing

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

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

## MACRO-80 ERROR MESSAGES

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

### Error Codes

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

**R Relocation —**

Illegal use of relocation in expression, such as abs-rel. Data, code and COMMON areas are relocatable.

**U Undefined symbol —**

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

**V Value error —**

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

## Error Messages:

'No end statement encountered on input file'

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

'Unterminated conditional'

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

'Unterminated REPT/IRP/IRPC/MACRO'

At least one block is unterminated.

[xx] [No] Fatal error(s) [,xx warnings]

The number of fatal errors and warnings. The message is listed on the console and in the list file.

## ○ Chapter Two

# Expression Evaluation

## OVERVIEW

In most cases, the operand field of a given opcode may be coded as an operand expression. Such expression is a string of integers, symbols and characters. This character string is combined using certain operators.

The symbols used in the expression can be expressed in several modes. A symbol can also be classified as either external or not external.

Additionally, 8080 opcodes can be used as valid one-byte operands.



## ARITHMETIC AND LOGICAL OPERATORS

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

**NUL**

**LOW, HIGH**

**\*, /, MOD, SHR, SHL**

**Unary Minus**

**+,**

**EQ, NE, LT, LE, GT, GE**

**NOT**

**AND**

**OR, XOR**

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

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

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

## MODES

All symbols used as operands in expressions are in one of the following modes:

**Absolute**  
**Data Relative**  
**Program (Code) Relative**  
**COMMON**

Symbols assembled under the ASEG, CSEG (default), or DSEG pseudo-ops are in Absolute, Code Relative or Data Relative mode respectively.

The number of COMMON modes in a program is determined by the number of COMMON blocks that have been named with the COMMON pseudo-op. Two COMMON symbols are not in the same mode unless they are in the same COMMON block.

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

If the operation is addition, the following rules apply:

1. At least one of the operands must be Absolute.
2. Absolute + <mode> = <mode>

If the operation is subtraction, the following rules apply:

1. <mode> - Absolute = <mode>
2. <mode> - <mode> = Absolute

where the two <mode>s are the same.

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

FOO + BAZ - ZAZ

will generate an R error because the first step (FOO + BAZ) adds two relocatable values. (One of the values must be Absolute.)

This problem can always be fixed by inserting parentheses.

FOO + (BAZ - ZAZ)

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

## Externals

Aside from its classification by mode, a symbol is either External or not External. An External value must be assembled into a two-byte field. (Single-byte Externals are not supported.)

The following rules apply to the use of Externals in expressions:

1. Externals are legal only in addition and subtraction.
2. If an External symbol is used in an expression, the result of the expression is always External.
3. When the operation is addition, either operand (but not both) may be External.
4. When the operation is subtraction, only the first operand may be External.

## Opcodes as Operands

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

For example:

```
MVI    A, (JMP)
MVI    B, (RNZ)
MVI    C, MOV A,B
```

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

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

NOTE: Opcodes are not valid operands in Z80 mode.

## ○ Chapter Three

# Pseudo-Opcodes/Assembler Directives

## Overview

Within the Macro-80 Assembler there exists a set of instructions known as pseudo-opcodes or assembler directives. These instructions represent commands to the Assembler. They are called pseudo because although they are coded into the source program, they are not translated as instructions.

The following chapter explains the form and usage of the available pseudo-opcodes.

## PSEUDO-OPCODES

### **ASEG**

#### **ASEG**

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

### **COMMON**

COMMON            /<block name>/

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

### **CSEG**

#### **CSEG**

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



## Define Byte

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

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

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

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

Example:

```
0000' 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.

## Define Space

```
DS  <exp>
```

DS reserves an area of memory. The value of <exp> gives the number of bytes to be allocated. All names used in <exp> must be previously defined (i.e., all names known at that point on pass 1).



Otherwise, a V error is generated during pass 1 and a U error may be generated during pass 2. If a U error is not generated during pass 2, a phase error will probably be generated because the DS generated no code on pass 1.

**DSEG****DSEG**

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

**Define Word**

DW      <exp>[ ,<exp>. . . ]

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

**END**

END      [ <exp> ]

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

**ENTRY/PUBLIC**

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

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

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

**EQU**

<name> EQU <exp>

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



## EXT/EXTRN

EXT <name>[ ,<name>. . . ]

EXTRN <name>[ ,<name>. . . ]

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

## INCLUDE

INCLUDE <filename>

The INCLUDE pseudo-op assembles source statements from an alternate source file into the current source file. Use of INCLUDE eliminates the need to repeat an often-used sequence of statements in the current source file. The pseudo-ops INCLUDE, \$INCLUDE and MACLIB are synonymous.



<filename> is any valid specification, as determined by the operating system. Defaults for filename extensions and device names are the same as those in a MACRO-80 command line.

The INCLUDE file is opened and assembled into the current source file immediately following the INCLUDE statement. When end-of-file is reached, assembly resumes with the statement following INCLUDE.

On a MACRO-80 listing, a plus sign is printed between the assembled code and the source line on each line assembled from an INCLUDE file.

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 given, and the INCLUDE is ignored.



**NAME**

NAME ('modname')

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

**Define Origin**

ORG <exp>

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

**PAGE**

PAGE [<exp>]

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

**SET**

<name> SET <exp>

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

**SUBTTL**

SUBTTL <text>

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

## TITLE

**TITLE <text>**

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

## .COMMENT

**.COMMENT <delim><text><delim>**

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

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

## PRINTX

**.PRINTX <delim><text><delim>**

The first non-blank character encountered after .PRINTX is the delimiter. The following text is listed on the terminal during assembly until another occurrence of the delimiter is encountered.

.PRINTX is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches.

For example:

```
IF CP/M
  .PRINTX /CP/M version
ENDIF
```

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

**.RADIX**

.RADIX <exp>

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

For example:

```
LXI H, OFFH  
.RADIX 16  
LXI H, OFF
```

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

**.REQUEST**

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

.REQUEST sends a request to the LINK-80 Loader to search the file names in the list for undefined globals before searching the FORTRAN library. The file names in the list should be in the form of legal MACRO-80 symbols. They should not include file name extensions or disk specifications. The LINK-80 loader will supply the default extension .REL and will assume the default drive.

**.Z80**

.Z80 enables the Assembler to accept Z80 opcodes. Z80 mode may also be set by appending the /Z switch to the MACRO-80 command string.

**.8080**

.8080 enables the Assembler to accept 8080 opcodes. This is the default condition. 8080 mode may also be set by appending the /I switch to the MACRO-80 command string.

## CONDITIONAL PSEUDO-OPERATIONS

The conditional pseudo-operations are:

<b>IF/IFT &lt;exp&gt;</b>	True if <exp> is not 0.
<b>IFE/IFF &lt;exp&gt;</b>	True if <exp> is 0.
<b>IF1</b>	True if pass 1.
<b>IF2</b>	True if pass 2.
<b>IFDEF &lt;symbol&gt;</b>	True if <symbol> is defined or has been declared External.
<b>IFNDEF &lt;symbol&gt;</b>	True if <symbol> is undefined or not declared External.
<b>IFB &lt;arg&gt;</b>	True if <arg> is blank. The angle brackets around <arg> are required.
<b>IFNB &lt;arg&gt;</b>	True if <arg> is not blank. Used for testing when dummy parameters are supplied. The angle brackets around <arg> are required.
<b>IFIDN &lt;arg1&gt;,&lt;arg2&gt;</b>	True if the string <arg1> is IDeNtical to the string <arg2>. The angle brackets around <arg1> and <arg2> are required.
<b>IFDIF &lt;arg1&gt;,&lt;arg2&gt;</b>	True if the string <arg1> is DIFFerent from the string <arg2>. The angle brackets around <arg1> and <arg2> are required.

All conditionals use the following format:

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

Conditionals may be nested to any level.

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

#### ELSE

Each conditional pseudo-operation may optionally be used with the ELSE pseudo-opcode which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a given IF, and an ELSE is always bound to the most recently opened IF. A conditional with more than one ELSE or an ELSE without a conditional will cause a C error.

#### ENDIF

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

### **Listing Control Pseudo-Operations**

There are five listing control pseudo-ops. Output to the listing file can be controlled by the following pseudo-ops:

**.LIST, .XLIST, .SFCOND, .LFCOND, .TFCOND**

If a listing is not being made, these pseudo-ops have no effect.

.LIST is the default condition. When a .XLIST is encountered, source and object code will not be listed until a .LIST is encountered.

The latter three pseudo-ops control the listing of conditional pseudo-op blocks which evaluate as false. These pseudo-ops give the programmer control over four cases.

1. **Normally list false conditionals** — For this case, the programmer simply allows the default mode to control the listing. The default mode is list false conditionals. If the programmer decides to suppress false conditionals, the /X switch can be issued in the command line instead of editing the source file.

2. **Normally suppress false conditionals** — For this case, the programmer issues the .TFCOND pseudo-op in the program file. .TFCOND reverses (toggles) the default, causing false conditionals to be suppressed. If the programmer decides to list false conditionals, the /X switch can be issued in the command line instead of editing the source file.
3. **Always suppress/list false conditionals** — For these cases, the programmer issues either the .SFCOND pseudo-op to suppress false conditionals, or the .LFCOND pseudo-op to list all false conditionals.
4. **Suppress/list some false conditionals** — For this case, the programmer has decided for most false conditionals whether to list or suppress; but for some false conditionals, the programmer has not yet decided. For the false conditionals decided about, use .SFCOND or .LFCOND. For those not yet decided, use .TFCOND. .TFCOND sets the current and default settings to the opposite of the default. Initially, the default is set by giving /X or no /X in the command line. Two subcases exist:
  - A. The programmer wants some false conditionals not to list unless /X is given. The programmer uses the .SFCOND and .LFCOND pseudo-ops to control which areas always suppress or list false conditionals. To selectively suppress some false conditionals, the programmer issues .TFCOND at the beginning of the conditional block and again at the end of the conditional block. (NOTE: The second .TFCOND should be so that the default setting will be the same as the initial setting. Leaving the default equal to the initial setting makes it easier to keep track of the default mode if there are many such areas.) If the conditional block evaluates as false, the lines will be suppressed. In this subcase, issuing the /X switch in the command line causes the conditional block affected by .TFCOND to list even if it evaluates as false.
  - B. The programmer wants some false conditionals to list unless /X is given of the file. Two consecutive .TFCONDS places the conditional listing setting in an initial state which is determined by the presence or absence of the /X switch (the first .TFCOND sets the default to not initial; the second to initial). The selected conditional block then responds to the /X switch: if a /X switch is issued in the command line, the conditional block is suppressed if false; if no /X switch is issued in the command line, the conditional block is listed even if false.

The programmer then must reissue the .SFCOND or .LFCOND conditional listing pseudo-op to restore the suppress or list mode. Simply issuing another .TFCOND will not restore the prior mode, but will toggle the default setting. Since in this subcase, the next area of code is supposed to list or suppress false conditionals always, the programmer must issue .SFCOND or .LFCOND.

The three conditional listing pseudo-ops are summarized below.

<u>PSEUDO-OP</u>	<u>DEFINITION</u>
------------------	-------------------

<b>.SFCOND</b>	Suppresses the listing of conditional blocks that evaluate as false.
<b>.LFCOND</b>	Restores the listing of conditional blocks that evaluate as false.
<b>.TFCOND</b>	Toggles the current setting which controls the listing of false conditionals. .TFCOND sets the current and default setting to not default. If a /X switch is given in the MACRO-80 run command line for a file which contains .TFCOND, /X reverses the effect of .TFCOND.

The output of MACRO/REPT/IRP/IRPC expansions is controlled by three pseudo-ops:

**.LALL, .SALL, and .XALL.**

Where:

**.LALL** lists the complete macro text for all expansions.

**.SALL** lists only the object code produced by a macro and not its text.

**.XALL** is the default condition; it is similar to .SALL, except a source line is listed only if it generates object code.

## RELOCATION PSEUDO-OPERATIONS

The ability to create relocatable modules is one of the major features of MACRO-80. Relocatable modules offer the advantages of easier coding and faster testing, debugging and modifying. In addition, it is possible to specify segments of assembled code that will later be loaded into RAM (the Data Relative segment) and ROM/PROM (the Code Relative segment).

The pseudo-operations that select relocatable areas are CSEG and DSEG. The ASEG pseudo-op is used to generate non-relocatable (absolute) code. The COMMON pseudo-op creates a common data area for every COMMON block that is named in the program.

The default mode for the Assembler is Code Relative. That is, assembly begins with a CSEG automatically executed and the location counter in the Code Relative mode, pointing to location 0 in the Code Relative segment of memory. All subsequent instructions will be assembled into the Code Relative segment of memory until an ASEG or DSEG or COMMON pseudo-op is executed.

For example, the first DSEG encountered sets the location counter to location zero in the Data Relative segment of memory. The following code is assembled in the Data Relative mode, where, it is assigned to the Data Relative segment of memory. If a subsequent CSEG is encountered, the location counter will return to the next free location in the Code Relative segment and so on.

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

## ORG Pseudo-Op

At any time, the value of the location counter may be changed by use of the the ORG pseudo-op.

The form of the ORG statement is:

**ORG <exp>**

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

For example, the statements

```
DSEG  
ORG 50
```

set the Data Relative location counter to 50, relative to the start of the Data Relative segment of memory.

## LINK-80

The LINK-80 Linking Loader (see Section C) combines the segments and creates each relocatable module in memory when the program is loaded. The origins of the relocatable segments are not fixed until the program is loaded and the origins are assigned by LINK-80. The command to LINK-80 may contain user-specified origins through the use of the /P (for Code Relative) and /D (for Data and COMMON segments) switches.

For example, a program that begins with the statements:

```
ASEG  
ORG 800H
```

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

## ○ Relocation Before Loading

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

For example:

```
0100    CD 0106      FOO:    .PHASE 100H
        C3 0007'          CALL    BAZ
0103    C9              BAZ:    JMP     Z00
        0106    C9          Z00:    RET
                                .DEPHASE
        0007'    C3 0005      JMP     5
```

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



## ○ Chapter Four

# Macros and Block Pseudo-Operations

## OVERVIEW

The Macro-80 Assembler provides complete facilities for constructing macros within the source program. Three repeat pseudo-operations as well as the macro definition operation are included. The following chapter explains the construction and use of the macro facilities.



## MACROS AND BLOCK PSEUDO OPERATIONS

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

### Terms

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

1. <dummy> is used to represent a dummy parameter. All dummy parameters are legal symbols that appear in the body of a macro expansion.
2. <dummylist> is a list of <dummy>s separated by commas.
3. <arglist> is a list of arguments separated by commas. <arglist> must be delimited by angle brackets. Two angle brackets with no intervening characters (< >) or two commas with no intervening characters (,,,) enter a null argument in the list. Otherwise an argument is a character or series of characters terminated by a comma or >.

With angle brackets that are nested inside an <arglist>, one level of brackets is removed each time the bracketed argument is used in an <arglist>.

A “quoted string” is an acceptable argument and is passed as such. Unless enclosed in <brackets> or a “quoted string”, leading and trailing spaces are deleted from arguments.

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

## ○ Block Pseudo Op-Codes

### REPT-ENDM

```
REPT <exp>
```

```
.
```

```
.
```

```
.
```

```
ENDM
```

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

Example:

```
X SET 0
    REPT 10 ;generates bytes 01-0A
    X SET X+1
    DB X
    ENDM
```

### IRP-ENDM

```
IRP <dummy>,<arglist>
```

```
.
```

```
.
```

```
.
```

```
ENDM
```

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

For example:

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

generates the same bytes as the REPT example.

**IRPC-ENDM**

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

```
.
```

```
.
```

```
.
```

```
ENDM
```

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

For example:

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

generates the same code as the two previous examples.

**MACRO**

Often it is convenient to be able to generate a given sequence of statements from various places in a program, even though different parameters may be required each time the sequence is used.

This capability is provided by the MACRO statement.

```
<name> MACRO <dummylist>
```

```
.
```

```
.
```

```
.
```

```
ENDM
```

where <name> conforms to the rules for forming symbols. <name> is the name that will be used to invoke the macro. The <dummy>s in <dummylist> are the parameters that will be changed (replaced) each time the MACRO is invoked. The statements before the ENDM comprise the body of the macro.

During assembly, the macro is expanded everytime it is invoked but, unlike REPT/IRP/IRPC, the macro is not expanded when it is encountered.

In the listing, the expansion of the macro will be marked with a plus (+).



The form of a macro call is:

**<name> <paramlist>**

where **<name>** is the name supplied in the MACRO definition, and the parameters in **<paramlist>** will replace the **<dummy>**s in the MACRO **<dummylist>** on a one-to-one basis. The number of items in **<dummylist>** and **<paramlist>** is limited only by the length of a line.

The number of parameters used when the macro is called need not be the same as the number of **<dummy>**s in **<dummylist>**. If there are more parameters than **<dummy>**s, the extras are ignored. If there are fewer, the extra **<dummy>**s will be made null. The assembled code will contain the macro expansion code after each macro call.

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

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

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

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

FOO 10

is executed.



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

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

When the call

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

is made, the macro expansion looks like this:

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

**ENDM**

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

#### **EXITM**

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

#### **LOCAL**

```
LOCAL <dummylist>
```

The LOCAL pseudo-op is allowed only inside a MACRO definition.

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

## Special Macro Operators and Forms

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

For example:

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

In this example, the call ERRGEN A will generate:

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

- ;; In a block operation, a comment preceded by two semicolons is not saved as part of the expansion (i.e., it will not appear on the listing even under .LALL). A comment preceded by one semicolon, however, will be preserved and appear in the expansion.
- ! When an exclamation point is used in an argument, the next character is entered literally (i.e., !; and <;> are equivalent).
- NUL NUL is an operator that returns true if its argument (a parameter) is null. The remainder of a line after NUL is considered to be the argument to NUL.

The conditional:

IF NUL argument

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

%      The percent sign is used only in a macro argument. % converts the expression that follows it (usually a symbol) to a number in the current radix. 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 the DS (Define Space) pseudo-op. A valid expression returning a nonrelocatable constant is required.

For Example:

Normally, LB, the argument to MAKLAB, would be substituted for Y, the argument to MACRO, as a string. The % causes LB to be converted to a nonrelocatable constant which is then substituted for Y. Without the % special operator, the result of assembly would be 'Error LB' rather than 'Error 1', etc.

MAKLAB	MACRO	Y
ERR&Y:	DB	'Error &Y',0
	ENDM	
MAKERR	MACRO	X
LB	SET	0
	REPT	X
LB	SET	LB+1
	MAKLAB	%LB
	ENDM	
	ENDM	

When called by MAKERR 3, the assembler will generate:

ERR&1:	DB	'Error 1',0
ERR&2:	DB	'Error 2',0
ERR&3:	DB	'Error 3',0

- **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.

## Using Z80 Pseudo-Ops

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

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

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

## MACRO-80 Reference Manual Index

- Absolute mode, 2-3
- Arithmetic operators, 2-2
- ASEG, 3-2
- Assembler directives, 3-1
- Block Psuedo-Opcodes, 4-3
- .COMMENT, 3-6
- COMMON, 3-2
- COMMON mode, 2-3
- Conditional pseudo-opcodes, 3-10
- Constants, 1-4
- Cross reference facility, 1-10
- CSEG, 3-2
- Data relative mode, 2-3
- Data storage, 3-8
- DB, 3-3
- Default filename extensions, 1-5
- Directives, assembler, 3-1
- DS, 3-3
- DSEG, 3-4
- DW, 3-4
- ELSE, 3-10
- END, 3-4
- ENDIF, 3-10
- ENDM, 4-6
- ENTRY, 3-4
- EQU, 3-4
- Error messages, 1-11, 1-12
- Examples, 1-6, 1-9
- EXITM, 4-6
- Expression evaluation, 2-1
- EXT, 3-5
- Extensions, filename, 1-5
- Externals, 2-4
- EXTRN, 3-5
- Format of Commands, 1-5
- Format of MACRO-80 source files, 1-2
- IRP-ENDM, 4-3
- IRPG-ENDM, 4-4
- .LALL, 3-12
- LINK-80, 3-12
- Listing control pseudo opcodes, 3-10
- Listing, symbol table, 1-8
- Loading, relocation before, 3-13
- LOCAL, 4-6
- Logical operators, 2-2
- Macros, 4-1, 4-5
- MACRO-80 switches, 1-7
- Macro operators, special, 4-7
- Messages, error, 1-9, 1-10
- Modes, 2-3
- Name, 3-5
- Numeric Constants, 1-4
- Opcodes as operands, 2-4
- Opcodes, pseudo-, 3-1
- Operators, special Macro, 4-7
- ORG, 3-5, 3-12
- PAGE, 3-5
- .PRINTX, 3-7
- Program relative mode, 2-3
- Pseudo-opcodes, 3-1
  - Block, 4-2
  - Conditional, 3-9
  - Listing control, 3-10
  - Relocation, 3-11
  - Z80, 4-8
- Public, 3-4

.RADIX, 3-7  
Reference, cross, 1-11  
Relocation before loading, 3-13  
Relocation, pseudo opcodes, 3-11  
REPT-ENDM, 4-3  
REQUEST, 3-8  
.SALL, 3-12  
SET, 3-6  
Special operators, 4-7  
Statements, 1-4  
Strings, 1-4  
SUBTTL, 3-6  
Switches, MACRO-80, 1-6  
Symbols, 1-4

Terms, Macro, 4-2  
TITLE, 3-6  
.XALL, 3-12  
.Z80, 3-8  
Z80 pseudo-opcodes, 4-8

# **Microsoft LINK-80 LOADER**

**CP/M® Version**

## **Software Reference Manual**

**for HEATH/ZENITH 8-bit digital computer systems**

II | \_\_\_\_\_



## Table of Contents

### **LINK-80, Linking Loader**

Overview .....	1-1
LINK-80 Command Strings .....	1-2
LINK-80 Switches .....	1-3
LINK-80 Error Messages .....	1-5
Format of LINK-80 Compatible Object Files .....	1-7



# LINK-80 Linking Loader

## OVERVIEW

The following Section contains reference information about the LINK-80 Linking Loader. The Linking Loader is used to load the relocatable modules produced by the FORTRAN-80, COBOL-80 or BASIC compiler and the Macro-80 Assembler. The Linking Loader also links these modules to any internal routines that may be needed for execution of the relocatable module.

For example, to perform formatted random I/O several routines are referenced in the FORTRAN-80 library. These routines contain the actual machine language code needed in order to access the disk drive. The linker is used to link the main program to these routines.

The linker can also be used to create an absolute file that can be executed under CP/M. This file has the default extension .COM and is completely compatible with CP/M.

NOTE: Be sure to use only 8080 op-codes if the absolute file is intended to run on an H8 without the HA-8-6 Z80 CPU board.

## LINK-80 COMMAND STRINGS

To run LINK-80, type L80 followed by a carriage return. LINK-80 will return the prompt “\*”. Each command to LINK-80 consists of a string of file names and switches separated by commas:

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

If the input device for a file is omitted, the default drive is used. If the extension of an input file is omitted, the default is .REL. After each line is typed, LINK-80 will load or search (see /S below) the specified files. After LINK-80 finishes this process, it will list all symbols that remained undefined followed by an asterisk.

Before execution begins, LINK-80 will always search the system library to satisfy any unresolved external references. The system library must reside on the default drive..

If the user wishes to first search non-standard libraries, the file names that are followed by /S should be appended to the end of the loader command string.

The following examples illustrate a typical use of the Linking Loader.

\*TEST

This will load the file TEST.REL.

\*TEST/N/E

This command string tells the linker to output the results of the linking and loading process in a file called TEST.COM. The /E will cause the linker to first search the system library to clear up any unresolved references, then exit to CP/M.



## LINK-80 Switches

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

These switches are:

**/R**

Reset. Put loader back in its initial state. Use /R if the wrong file is accessed and it is necessary to re-start. /R takes effect as soon as it is encountered in a command string.

**/E or /E:Name**

Exit LINK-80 and return to CP/M. The system library will be searched on the default drive to satisfy any existing undefined globals.

The optional form /E:Name (where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program.



**/G or /G:Name**

Start execution of the program as soon as the current command line has been interpreted. The system library will be searched on the default disk to satisfy any existing undefined globals if they exist.

Before execution actually begins, LINK-80 prints two numbers and a BEGIN EXECUTION message. The two numbers are the start address, and the address of the next available byte.

The optional form /G:Name (where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program.

**/N**

If a <filename>/N is specified, the program will be saved on disk under the selected name (with a default extension of .COM when a /E or /G is done. A jump to the start of the program is inserted so the program will run properly.



**/P and /D**

/P and /D allow the origin(s) to be set for the next program loaded. /P and /D take effect when seen and they have no effect on programs already loaded. The form is /P:<address> or /D:<address>, where <address> is the desired origin in the current radix. (Default radix is hex. /O sets radix to octal; /H to hex.)

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

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

**/U**

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

**/M**

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

**/S**

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

**/X**

If a filename/N was specified, /X will cause the file to be saved in Intel ASCII HEX format with an extension of HEX.

EXAMPLE: FOO/N/X/E will create an Intel ASCII HEX formatted load module named FOO.HEX.

**/Y**

If a filename/N was specified, /Y will create a filename.SYM file when /E is entered. This file contains the names and addresses of all Globals for use with Digital Research's Symbolic Debugger, SID and ZSID.

EXAMPLE: FOO/N/Y/E Creates FOO.COM and FOO.SYM.  
MYPROG/N/X/Y/E creates MYPROG.HEX and MYPROG.SYM.

## LINK-80 Error Messages

LINK-80 has the following error messages:

### ?No Start Address

A /G switch was issued, but no main program had been loaded.

### ?Loading Error

The last file given for input was not a properly formatted LINK-80 object file.

### ?Out of Memory

Not enough memory to load program. (A minimum of 40K RAM is required.)

### ?Command Error

Unrecognizable LINK-80 command string.

### ?<file> Not Found

<file>, as given in the command string, did not exist.

### %2nd COMMON Larger /XXXXXX/

The first definition of COMMON block /XXXXXX/ was not the largest definition. Re-order module loading sequence or change COMMON block definitions. (See the FORTRAN Reference Manual for more information on the COMMON statement.)

### %Mult. Def. Global YYYYYYY

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

### %Overlaying Program Area

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

**?Intersecting Program Area  
Data**

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

**?Start Symbol - <name> - Undefined**

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

**Origin Above Loader Memory, Move Anyway (Y or N)?  
Below**

After a /E or /G was given, either the data or program area has an origin or top which lies outside loader memory. If a Y <cr> is given, LINK-80 will move the area and continue. If anything else is given, LINK-80 will exit.

In either case, if a /N was given, the image will already have been saved.

**?Can't Save Object File**

A disk error occurred when the file was being saved. Usually this occurs when there is no more room left on the disk.

**?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, exit the Linker, or 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.

## FORMAT OF LINK-80 COMPATIBLE OBJECT FILES

The following information is reference material for users who wish to know the load format of LINK-80 relocatable object files.

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

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

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

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

A four-bit control field.

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

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

A general representation of a special LINK item is:

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

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

The following special types have a B-field only:

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

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

- 5 Define COMMON size
- 6 Chain external (A is head of address chain, B is name of external symbol)
- 7 Define entry point (A is address, B is name)
- 8 External-offset. Used for JMP and CALL to externals.

The following special LINK items have an A field only:

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

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

- 15 End File

## LINK-80 Linking Loader Index

- Bit stream, 1-7
- Can't Save Object File, 1-6
- Command Error, 1-5
- Command Strings, 1-2
- COMMON Larger, 1-5
- Exit LINK-80, 1-2
- Format of object file, 1-7, 1-8
- Intersecting Data Area, 1-6
- Intersecting Program Area, 1-6
- Linking loader, 1-1
- LINK-80, 1-1
  - Command string, 1-2
  - Error messages, 1-5
  - Switches, 1-3, 1-4
- Loading Error, 1-5
- Multiple Defined Globals, 1-5
- No Start Address, 1-5
- Object files, format of, 1-7, 1-8
- Origin Above Loader Area, 1-6
- Origin Below Loader Area, 1-6
- Out of Memory, 1-5
- Overlaying Data Area, 1-5
- Overlaying Program Area, 1-5
- Reset, 1-3
- Save file on disk, 1-3
- Start execution of program, 1-3
- Start Symbol Undefined, 1-6



# **Microsoft**

# **LIB-80**

# **Library Manager**

**CP/M® Version**

## **Software Reference Manual**

**for HEATH/ZENITH 8-bit digital computer systems**

II [

---

Portions of this Manual have been adapted from Microsoft publications or documents.

COPYRIGHT © by Microsoft, 1979, all rights reserved.

## Table of Contents

### LIB-80, Library Manager

Overview .....	1-1
LIB-80 Command Strings .....	1-2
Modules .....	1-3
LIB-80 Switches .....	1-4
LIB-80 Listings .....	1-5
Sample LIB Session .....	1-6



# LIB-80 LIBRARY MANAGER

## OVERVIEW

LIB-80 is the object time library manager for the CP/M version of FORTRAN-80, COBOL-80 and the BASIC Compiler. It is used to create and modify libraries which are then linked with compiled programs.

### WARNING

Read this document carefully and make a back-up copy of your libraries before using LIB. It is not difficult to destroy a library with LIB-80.

## LIB-80 COMMAND STRINGS

To run LIB-80, type LIB and press RETURN. The LIB-80 library manager will be loaded into memory and executed. LIB-80 will return the prompt "\*" indicating it is ready to accept commands. Each command in LIB-80 either lists information about a library or adds new modules to the library under construction.

Commands to LIB-80 consist of an optional designation file name which sets the name of the library being created, followed by an equal sign, followed by module names separated by commas. The default designation file name is FORLIB.REL.

Examples:

```
*NEWLIB=FILE1<MOD2>,FILE3, TEST  
*SIN,COS,TAN,ATAN
```

Any command specifying a set of modules concatenates the modules selected onto the end of the last destination file name given. Therefore,

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

is equivalent to:

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

## MODULES

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

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

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

SIN

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

SIN+1

specifies the module after SIN and:

SIN-1

specifies the one before it.

You may also specify ranges of modules by using two dots:

SIN.. means all modules from and including SIN to the end of the file.  
SIN..COS means SIN and COS and all modules in between.

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

SIN+1..COS-1

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

Examples:

```
FORLIB<SIN..COS>
MYLIB.REL<TEST>
BIGLIB.REL<FIRST,MIDDLE,LAST>
```

If no modules are selected from a file, then all the modules in the file are selected.

## LIB-80 SWITCHES

### NOTE

/E will destroy the current library if there is no new library under construction. Exit LIB-80 using Control-C if the library is not being revised.

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

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

## LIB-80 LISTINGS

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

\*FORLIB/L

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

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

Example:

SYSLIB/U

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

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

## SAMPLE LIB SESSION

Building a library:

```
A>LIB  
*TRANLIB=SIN,COS,TAN,ATAN, ALOG  
*EXP  
*/E  
A>
```

Listing a library:

```
A>LIB  
*TRANLIB.LIB/U  
*TRANLIB.LIB/L
```

(list of symbols in TRANLIB.LIB)

```
*Control-C  
A>
```

# **Microsoft CREF-80 CROSS REFERENCE FACILITY**

**CP/M® Version**

## **Software Reference Manual**

**for HEATH/ZENITH 8-bit digital computer systems**

II

---

Portions of this Manual have been adapted from Microsoft Publications or Documents.

COPYRIGHT © by Microsoft, 1979, all rights reserved.

## Table of Contents

### CREF-80, Cross Reference Facility

Overview .....	1-1
Using the Cross Reference Facility .....	1-2
Example .....	1-3



# CREF-80 CROSS REFERENCE FACILITY

## OVERVIEW

The following section contains reference information about the CREF-80 Cross Reference Facility. The cross reference facility will generate a special listing that can be an important diagnostic tool. Assume, for example, that a program uses a field called FIELD1, and that program testing reveals an error in the manipulating of this field. The cross reference listing can be used to check every instruction that references this field.

## USING THE CROSS REFERENCE FACILITY

The Cross Reference Facility is invoked by typing CREF80. To generate a cross reference listing, the Assembler must output a special listing file with embedded control characters. The MACRO-80 command string tells the assembler to output this special listing file. /C is the cross reference switch. When the /C switch is encountered in a MACRO-80 command string, the Assembler opens a .CRF file instead of a .PRN file.

Example:

(NOTE: The asterisk represents the prompt from the Assembler.)

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

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

When the Assembler is finished, exit to CP/M with CTRL-C. Then call the Cross Reference Facility by typing CREF.

The command string is:

\*listing file=source file

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

Possible command strings are:

**\*=TEST** Examine file TEST.CRF and generate a cross reference listing file TEST.PRN.

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

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

1. Each source statement is numbered with a Cross Reference number.
  2. At the end of the listing, variable names appear in alphabetic order along with the numbers of the lines on which they are referenced or defined. Line numbers on which the symbol is defined are flagged with '#'.

- The following example uses the macro assembler, M80, with the cross reference switch. A printout of the cross reference listing is also shown.

```
A>TYPE TEST.MAC
      ORG    100H
ABLE   EQU    10
BAKER  EQU    20
      XRA    A
      LXI    H,STORE
      MVI    B,5
LOOP: ADD    M
      CPI    ABLE
      JNC    LOOP
      HLT
STORE: DB     1,2,3,4,5
      END
```

A>M80

\*=TEST/C

No Fatal error(s)

\*↑C
A>CREF

\*=TEST

\*↑C

A>TYPE TEST.PRN

	MACRO-80 3.4	26-Nov-80	PAGE	1
1			ORG	100H
2	000A		ABLE	EQU 10
3	0014		BAKER	EQU 20
4	0100' AF			XRA A
5	0101' 21 010D'			LXI H,STORE
6	0104' 06 05			MVI B,5
7	0106' 86		LOOP:	ADD M
8	0107' FE 0A			CPI ABLE
9	0109' D2 0106'			JNC LOOP
10	010C' 76			HLT
11	010D' 01 02 03 04		STORE:	DB 1,2,3,4,5
12	0111' 05			
13				END

MACRO-80 3.4      26-Nov-80      PAGE      S

Macros:

Symbols:

ABLE      000A      BAKER      0014      LOOP      0106'      STORE      010D'

No Fatal error(s)

ABLE	2#	8
BAKER	3#	
LOOP	7#	9
STORE	5	11#



## Appendix A

# 8080 Op-Codes

## INSTRUCTION SET

<u>Mnemonic</u>	<u>Description</u>	<u>Mnemonic</u>	<u>Description</u>
ACI	Add immediate to A with carry	DAA	Decimal adjust A
ADC M	Add memory to A with carry	DAD B	Add B & C to H & L
ADC r	Add register to A with carry	DAD D	Add D & E to H & L
ADD M	Add memory to A	DAD H	Add H & L to H & L
ADD r	Add register to A	DAD SP	Add stack pointer to H & L
ADI	Add immediate to A	DCR M	Decrement memory
ANA M	And memory with A	DCR r	Decrement register
ANA r	And register with A	DCX B	Decrement B & C
ANI	And immediate with A	DCX D	Decrement D & E
CALL	Call unconditional	DCX H	Decrement H & L
CC	Call on carry	DCX SP	Decrement stack pointer
CM	Call on minus	DI	Disable Interrupt
CMA	Complement A	EI	Enable Interrupts
CMC	Complement carry	HLT	Halt
CMP M	Compare memory with A	IN	Input
CMP r	Compare register with A	INR M	Increment memory
CNC	Call on no carry	INR r	Increment register
CNZ	Call on no zero	INX B	Increment B & C registers
CP	Call on positive	INX D	Increment D & E registers
CPE	Call on parity even	INX H	Increment H & L registers
CPI	Compare immediate with A	INX SP	Increment stack pointer
CPO	Call on parity odd	JC	Jump on carry
CZ	Call on zero	JM	Jump on minus

<u>Mnemonic</u>	<u>Description</u>	<u>Mnemonic</u>	<u>Description</u>
JMP	Jump unconditional	RAL	Rotate A left through carry
JNC	Jump on no carry	RAR	Rotate A right through carry
JNZ	Jump on no zero	RC	Return on carry
JP	Jump on positive	RET	Return
JPE	Jump on parity even	RLC	Rotate A left
JPO	Jump on parity odd	RM	Return on minus
JZ	Jump on zero	RNC	Return on no carry
LDA	Load A direct	RNZ	Return on no zero
LDAX B	Load A indirect	RP	Return on positive
LDAX D	Load A indirect	RPE	Return on parity even
LHLD	Load H & L direct	RPO	Return on parity odd
LXI B	Load immediate register Pair B & C	RRC	Rotate A right
LXI D	Load immediate register Pair D & E	RST	Restart
LXI H	Load immediate register Pair H & L	RZ	Return on zero
LXI SP	Load immediate stack pointer	SBB M	Subtract memory from A with borrow
MVI M	Move immediate memory	SBB r	Subtract register from A with borrow
MVI r	Move immediate register	SBI	Subtract immediate from A with borrow
MOV M, r	Move register to memory	SHLD	Store H & L direct
MOV r,M	Move memory to register	SPHL	H & L to stack pointer
MOV r1, r2	Move register to register	STA	Store A direct
NOP	No-operation	STAX B	Store A indirect
ORA M	Or memory with A	STAX D	Store A indirect
ORA r	Or register with A	STC	Set carry
ORI	Or immediate with A	SUB M	Subtract memory from A
OUT	Output	SUB r	Subtract register from A
PCHL	H & L to program counter	SUI	Subtract immediate from A
POP B	Pop register pair B & C off stack	XCHG	Exchange D & E, H & L Registers
POP D	Pop register pair D & E off stack	XRA M	Exclusive Or memory with A
POP H	Pop register pair H & L off stack	XRA r	Exclusive Or register with A
POP PSW	Pop A and Flags off stack	XRI	Exclusive Or immediate with A
PUSH B	Push register B & C on stack	XTHL	Exchange top of stack, H & L
PUSH D	Push register pair D & E on stack		
PUSH H	Push register pair H & L on stack		
PUSH PSW	Push A and Flags on stack		

## Appendix B

# Z80 Op-Codes

## INSTRUCTION SET

<u>Mnemonic</u>	<u>Description</u>	<u>Mnemonic</u>	<u>Description</u>
ADC HL, ss	Add with Carry Reg. pair ss to HL	CPI	Compare location (HL) and Acc. increment HL and decrement BC
ADC A, s	Add with carry operand s to Acc.	CPIR	Compare location (HL) and Acc. increment HL, decrement BC repeat until BC=0
ADD A, n	Add value n to Acc.	CPL	Complement Acc. (1's comp)
ADD A, r	Add Reg. r to Acc.	DAA	Decimal adjust Acc.
ADD A, (HL)	Add location (HL) to Acc.	DEC m	Decrement operand m
ADD A, (IX+d)	Add location (IX+d) to Acc.	DEC IX	Decrement IX
ADD A, (IY+d)	Add location (IY+d) to Acc.	DEC IY	Decrement IY
ADD HL, ss	Add Reg. pair ss to HL	DEC ss	Decrement Reg. pair ss
ADD IX, pp	Add Reg. pair pp to IX	DI	Disable interrupts
ADD IY, rr	Add Reg. pair rr to IY	DJNZ e	Decrement B and Jump relative if B=0
AND s	Logical 'AND' of operand s and Acc.	EI	Enable interrupts
BIT b, (HL)	Test BIT b of location (HL)	EX (SP), HL	Exchange the location (SP) and HL
BIT b, (IX+d)	Test BIT b of location (IX+d)	EX (SP), IX	Exchange the location (SP) and IX
BIT b, (IY+d)	Test BIT b of location (IY+d)	EX (SP) IY	Exchange the location (SP) and IY
BIT b, r	Test BIT b of Reg. r	EX AF, AF	Exchange the contents of AF and AF'
CALL cc, nn	Call subroutine at location nn if condition cc is true	EX DE, HL	Exchange the contents of DE and HL
CALL nn	Unconditional call subroutine at location nn	EXX	Exchange the contents of BC, DE, HL with contents of BC', DE', HL' respectively
CCF	Complement carry flag	HALT	HALT (wait for interrupt or reset)
CP s	Compare operand s with Acc.		
CPD	Compare location (HL) and Acc. decrement HL and BC		
CPDR	Compare location (HL) and Acc. decrement HL and BC, repeat until BC=0		

<u>Mnemonic</u>	<u>Description</u>	<u>Mnemonic</u>	<u>Description</u>
IM 0	Set interrupt mode 0	LD A, (BC)	Load Acc. with location (BC)
IM 1	Set interrupt mode 1	LD A, (DE)	Load Acc. with location (DE)
IM 2	Set interrupt mode 2	LD A, I	Load Acc. with I
IN A, (n)	Load the Acc. with input from device n	LD A, (nn)	Load Acc. with location nn
IN r, (C)	Load the Reg. r with input from device (C)	LD A, R	Load Acc. with Reg. R
INC (HL)	Increment location (HL)	LD (BC), A	Load location (BC) with Acc.
INC IX	Increment IX	LD (HL), n	Load location (HL) with value n
INC (IX+d)	Increment location (IX+d)	LD dd, nn	Load Reg. pair dd with value nn
INC IY	Increment IY	LD HL, (nn)	Load HL with location (nn)
INC (IY+d)	Increment location (IY+d)	LD (HL), r	Load location (HL) with Reg. r
INC r	Increment Reg. r	LD I, A	Load I with Acc.
INC ss	Increment Reg. pair ss	LF IX, nn	Load IX with value nn
IND	Load location (HL) with input from port (C), decrement HL and B	LD IX, (nn)	Load IX with location (nn)
INDR	Load location (HL) with input from port (C), decrement HL and B, repeat until B=0	LD (IX+d), n	Load location (IX+d) with value n
INI	Load location (HL) with input from port (C), and increment HL and decrement B.	LD (IX+d), r	Load location (IX+d) with Reg. r
INIR	Load location (HL) with input from port (C), increment HL and decrement B, repeat until B=0	LD IY, nn	Load IY with value nn
JP (HL)	Unconditional Jump to (HL)	LD IY, (nn)	Load IY with location (nn)
JP (IX)	Unconditional Jump to (IX)	LD (IY+d), n	Load location (IY+d) with value n
JP (IY)	Unconditional Jump to (IY)	LD (IY+d), r	Load location (IY+d) with Reg. r
JP cc, nn	Jump to location nn if condition cc is true	LD (nn), A	Load location (nn) with Acc.
JP nn	Unconditional jump to location nn	LD (nn), dd	Load location (nn) with Reg. pair dd
JP C, e	Jump relative to PC+e if carry=1	LD (nn), HL	Load location (nn) with HL
JR e	Unconditional Jump relative to PC+e	LD (nn), IX	Load location (nn) with IX
JP NC, e	Jump relative to PC+e if carry=0	LD (nn), IY	Load location (nn) with IY
JR NZ, e	Jump relative to PC+e if non zero (Z=0)		
JR Z, e	Jump relative to PC+e if zero (Z=1)		

<u>Mnemonic</u>	<u>Description</u>	<u>Mnemonic</u>	<u>Description</u>
LDD	Load location (DE) with location (HL), decrement DE, HL and BC	RET cc	Return from subroutine if condition cc is true
LDDR	Load location (DE) with location (HL), decrement DE, HL and BC, repeat until BC=0	RETI	Return from interrupt
LDI	Load location (DE) with location (HL), increment DE, HL, decrement BC	RETN	Return from non maskable interrupt
LDIR	Load location (DE) with location (HL), increment DE, HL, decrement BC and repeat until BC=0	RL m	Rotate left through carry operand m
NEG	Negate Acc. (2's complement)	RLA	Rotate left Acc. through carry
NOP	No operation	RLC (HL)	Rotate location (HL) left circular
OR s	Logical 'OR' or operand s and Acc.	RLC (IX+d)	Rotate location (IX+d) left circular
OTDR	Load output port (C) with location (HL) decrement HL and B, repeat until B=0	RLC (IY+d)	Rotate location (IY+d) left circular
OTIR	Load output port (C) with location (HL), increment HL, decrement B, repeat until B=0	RLC r	Rotate Reg. r left circular
OUT (C), r	Load output port (C) with Reg. r	RLCA	Rotate left circular Acc.
OUT (n), A	Load output port (n) with Acc.	RLD	Rotate digit left and right between Acc. and location (HL)
OUTD	Load output port (C) with location (HL), decrement HL and B	RR m	Rotate right through carry operand m
OUTI	Load output port (C) with location (HL), increment HL and decrement B	RRA	Rotate right Acc. through carry
POP IX	Load IX with top of stack	RRC m	Rotate operand m right circular
POP IY	Load IY with top of stack	RRCA	Rotate right circular Acc.
POP qq	Load Reg. pair qq with top of stack	RRD	Rotate digit right and left between Acc. and location (HL)
PUSH IX	Load IX onto stack	RST p	Restart to location p
PUSH IY	Load IY onto stack	SBC A, s	Subtract operand s from Acc. with carry
PUSH qq	Load Reg. pair qq onto stack	SBC HL, ss	Subtract Reg. pair ss from HL with carry
RES b, m	Reset Bit b of operand m	SCF	Set carry flag (C=1)
RET	Return from subroutine	SET b, (HL)	Set Bit b of location (HL)
		SET b, (IX+d)	Set Bit b of location (IX+d)
		SET b, (IY+d)	Set Bit b of location (IY+d)
		SET b, r	Set Bit b of Reg. r
		SLA m	Shift operand m left arithmetic
		SRA m	Shift operand m right arithmetic
		SRL m	Shift operand m right logical
		SUB s	Subtract operand s from Acc.
		XOR s	Exclusive 'OR' operand s and Acc.



## Appendix C

# ASCII Codes

### DECIMAL TO OCTAL TO HEX

### TO ASCII CONVERSION

I

II

III

IV

DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII
0 .	000 .	00 .	NUL	32 .	040 .	20 .	SPACE	64 .	100 .	40 .	@	96 .	140 .	60 .	'
1 .	001 .	01 .	SOH	33 .	041 .	21 .	!	65 .	101 .	41 .	A	97 .	141 .	61 .	a
2 .	002 .	02 .	STX	34 .	042 .	22 .	"	66 .	102 .	42 .	B	98 .	142 .	62 .	b
3 .	003 .	03 .	ETX	35 .	043 .	23 .	#	67 .	103 .	43 .	C	99 .	143 .	63 .	c
4 .	004 .	04 .	EOT	36 .	044 .	24 .	\$	68 .	104 .	44 .	D	100 .	144 .	64 .	d
5 .	005 .	05 .	ENQ	37 .	045 .	25 .	%	69 .	105 .	45 .	E	101 .	145 .	65 .	e
6 .	006 .	06 .	ACK	38 .	046 .	26 .	&	70 .	106 .	46 .	F	102 .	146 .	66 .	f
7 .	007 .	07 .	BEL	39 .	047 .	27 .	'	71 .	107 .	47 .	G	103 .	147 .	67 .	g
8 .	010 .	08 .	BS	40 .	050 .	28 .	(	72 .	110 .	48 .	H	104 .	150 .	68 .	h
9 .	011 .	09 .	HT	41 .	051 .	29 .	)	73 .	111 .	49 .	I	105 .	151 .	69 .	i
10 .	012 .	0A .	LF	42 .	052 .	2A .	*	74 .	112 .	4A .	J	106 .	152 .	6A .	j
11 .	013 .	0B .	VT	43 .	053 .	2B .	+	75 .	113 .	4B .	K	107 .	153 .	6B .	k
12 .	014 .	0C .	FF	44 .	054 .	2C .	,	76 .	114 .	4C .	L	108 .	154 .	6C .	l
13 .	015 .	0D .	CR	45 .	055 .	2D .	-	77 .	115 .	4D .	M	109 .	155 .	6D .	m
14 .	016 .	0E .	S0	46 .	056 .	2E .	PERIOD	78 .	116 .	4E .	N	110 .	156 .	6E .	n
15 .	017 .	0F .	SI	47 .	057 .	2F .	/	79 .	117 .	4F .	O	111 .	157 .	6F .	o
16 .	020 .	10 .	DLE	48 .	060 .	30 .	Ø	80 .	120 .	50 .	P	112 .	160 .	70 .	p
17 .	021 .	11 .	DC1	49 .	061 .	31 .	1	81 .	121 .	51 .	Q	113 .	161 .	71 .	q
18 .	022 .	12 .	DC2	50 .	062 .	32 .	2	82 .	122 .	52 .	R	114 .	162 .	72 .	r
19 .	023 .	13 .	DC3	51 .	063 .	33 .	3	83 .	123 .	53 .	S	115 .	163 .	73 .	s
20 .	024 .	14 .	DC4	52 .	064 .	34 .	4	84 .	124 .	54 .	T	116 .	164 .	74 .	t
21 .	025 .	15 .	NAK	53 .	065 .	35 .	5	85 .	125 .	55 .	U	117 .	165 .	75 .	u
22 .	026 .	16 .	SYN	54 .	066 .	36 .	6	86 .	126 .	56 .	V	118 .	166 .	76 .	v
23 .	027 .	17 .	ETB	55 .	067 .	37 .	7	87 .	127 .	57 .	W	119 .	167 .	77 .	w
24 .	030 .	18 .	CAN	56 .	070 .	38 .	8	88 .	130 .	58 .	X	120 .	170 .	78 .	x
25 .	031 .	19 .	EM	57 .	071 .	39 .	9	89 .	131 .	59 .	Y	121 .	171 .	79 .	y
26 .	032 .	1A .	SUB	58 .	072 .	3A .	:	90 .	132 .	5A .	Z	122 .	172 .	7A .	z
27 .	033 .	1B .	ESC	59 .	073 .	3B .	;	91 .	133 .	5B .	[	123 .	173 .	7B .	{
28 .	034 .	1C .	FS	60 .	074 .	3C .	<	92 .	134 .	5C .	\	124 .	174 .	7C .	
29 .	035 .	1D .	GS	61 .	075 .	3D .	=	93 .	135 .	5D .	]	125 .	175 .	7D .	}
30 .	036 .	1E .	RS	62 .	076 .	3E .	>	94 .	136 .	5E .	^	126 .	176 .	7E .	~
31 .	037 .	1F .	US	63 .	077 .	3F .	?	95 .	137 .	5F .	_	127 .	177 .	7F .	DELETE

## CONTROL CHARACTER DEFINITIONS

NUL	Null: Tape feed,
SOH	Start of Heading; Start of Message
STX	Start of Text; End of Address
ETX	End of Text; End of Message
EOT	End of Transmission; Shuts off TWX machines
ENQ	Enquiry; WRU
ACK	Acknowledge; RU
BEL	Rings Bell
BS	Backspace
HT	Horizontal TAB
LF	Line Feed or Space (New Line)
VT	Vertical TAB
FF	Form Feed (PAGE)
CR	Carriage Return
SO	Shift Out
SI	Shift In
DLE	Data Link Escape
DC1	Device Control 1; Reader on
DC2	Device Control 2; Punch on
DC3	Device Control 3; Reader off
DC4	Device Control 4; Punch off
NAK	Negative Acknowledge; Error
SYN	Synchronous Idle (SYNC)
ETB	End of Transmission Block; Logical End of Medium
CAN	Cancel (CANCL)
EM	End of Medium
SUB	Substitute
ESC	Escape
FS	File Separator
GS	Group Separator
RS	Record Separator
US	Unit Separator

Refer to the chart on page C-1. Note that any print control character defined above and listed in column I of the chart can be produced from the combination of CTRL and the alphabetical character in column III or IV which is on the same line and to the right of the print control character. That is, DLE is CTRL-P or ↑P, BEL is CTRL-G or ↑G, and so on.

## ○ Appendix D

# Error Messages

## MACRO-80 ERROR MESSAGES

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

### Error Codes

- A Argument error —  
Argument to pseudo-op is not in correct format or is out of range (.PAGE 1; .RADIX 1; PUBLIC 1; STAX H; MOV M,N; INX C).
- C Conditional nesting error —  
ELSE without IF, ENDIF without IF, two ELSEs on one IF.
- D Double Defined symbol —  
Reference to a symbol which is multiply defined.
- E External error —  
Use of an external illegal in context (e.g., FOO SET NAME ; MVI A,2-NAME ).
- M Multiply-Defined symbol —  
Definition of a symbol which is multiply-defined.
- N Number error —  
Error in a number, usually a bad digit (e.g., 8Q).

- O    Bad opcode or objectionable syntax —  
      ENDM, LOCAL outside a block; SET, EQU or MACRO without a name; bad syntax in an opcode (MOV A:); or bad syntax in an expression (mismatched parenthesis, quotes, consecutive operators, etc.).
- P    Phase error —  
      Value of a label or EQU name is different on pass 2.
- Q    Questionable —  
      Usually means a line is not terminated properly. This is a warning error (e.g., MOV A,B.).
- R    Relocation —  
      Illegal use of relocation in expression, such as abs-rel. Data, code and COMMON areas are relocatable.
- U    Undefined symbol —  
      A symbol referenced in an expression is not defined. (For certain pseudo-ops, a V error is printed on pass 1 and a U on pass 2.)
- V    Value error —  
      On pass 1 a pseudo-op which must have its value known on pass 1 (e.g., .RADIX, .PAGE, DS, IF, IFE, etc.), has a value which is undefined later in the program, a U error will not appear on the pass 2 listing.

#### Error Messages:

'No end statement encountered on input file'

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

'Unterminated conditional'

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

'Unterminated REPT/IRP/IRPC/MACRO'

At least one block is unterminated.

[xx] [No] Fatal error(s) [,xx warnings]

The number of fatal errors and warnings. The message is listed on the console and in the list file.

## LINK-80 ERROR MESSAGES

### ?No Start Address

A /G switch was issued, but no main program had been loaded.

### ?Loading Error

The last file given for input was not a properly formatted LINK-80 object file.

### ?Out of Memory

Not enough memory to load program.

(A minimum of 40K RAM is required.)

### ?Command Error

Unrecognizable LINK-80 command string.

### ?<file> Not Found

<file>, as given in the command string, did not exist.

### %2nd COMMON Larger /XXXXXX/

The first definition of COMMON block /XXXXXX/ was not the largest definition. Re-order module loading sequence or change COMMON block definitions. (See Chapter 9 in the FORTRAN Reference Manual for more information on the COMMON statement.)

### %Mult. Def. Global YYYYYYY

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

### %Overlaying Program Area

    Data

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

### ?Intersecting Program Area

    Data

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

?Start Symbol — <name> — Undefined

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

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

After a /E or /G was given, either the data or program area has an origin or top which lies outside loader memory. If a Y <cr> is given, LINK-80 will move the area and continue. If anything else is given, LINK-80 will exit.

In either case, if a /N was given, the image will already have been saved.

?Can't Save Object File

A disk error occurred when the file was being saved. Usually this occurs when there is no more room left on the disk.

?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, exit the Linker, or execute a program, when in fact nothing had been loaded.









MICROSOFT  
**BASIC-80(CP/M®)**

HEATH

**ZENITH**  
data  
systems



# **Microsoft**

# **BASIC-80**

## **Software Reference Manual**

for HEATH/ZENITH 8-bit digital computer systems

595-2538-03

Printed in the United  
States of America

Copyright © 1981  
Heath Company  
*All Rights Reserved*

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

II

---

Portions of this Manual have been adapted from Microsoft publications or documents.

COPYRIGHT © by Microsoft, 1979, all rights reserved.

# Table of Contents

## Chapter One — System Introduction and General Information

Overview .....	1-1
Installation Guide .....	1-2
Contents of the Diskettes .....	1-3
Sample Output of PI.BAS .....	1-4
Diskette Use .....	1-5
Preparing Working Diskettes .....	1-8
System Introduction .....	1-9
Manual Scope .....	1-9
Hardware Requirements .....	1-10
System Software Requirements .....	1-10
Preparing the Diskette .....	1-11
Initialization of BASIC-80 .....	1-11
General Information .....	1-12
Modes of Operation .....	1-12
Line Format .....	1-12
Line Numbers .....	1-12
Character Set .....	1-13
Control Characters .....	1-14
BASIC-80 Programming .....	1-15
Loading the BASIC-80 Interpreter .....	1-15
Writing a BASIC-80 Program .....	1-17
Running a BASIC-80 Program .....	1-19
Debugging a BASIC-80 Program .....	1-20
Saving a BASIC-80 Program .....	1-22
Loading a BASIC-80 Program .....	1-23
Listing a BASIC-80 Program on a Hard Copy Device .....	1-24

## Chapter Two — Expression

Overview .....	2-1
Constants .....	2-2
String Constants .....	2-2
Numeric Constants .....	2-2
Integer Constants .....	2-2
Fixed Point Constants .....	2-2
Floating Point Constants .....	2-2
Hex Constants .....	2-2
Octal Constants .....	2-3
Single and Declaration Characters .....	2-3
Variables .....	2-4
Variable Names and Declaration Characters .....	2-4
Examples of BASIC-80 Variable names .....	2-5
Array Variables .....	2-5
Type Conversions .....	2-6

Expressions and Operators .....	2-8
Arithmetic Operators .....	2-8
Integer Division and Modulus Arithmetic .....	2-9
Overflow and Division by Zero .....	2-9
Relational Operators .....	2-10
Logical Operators .....	2-11
Logical Operators in Relational Expressions .....	2-14
Functional Operators .....	2-14

### **Chapter Three — Command Mode Statements**

Overview .....	3-1
Command Mode Statements .....	3-2
AUTO .....	3-2
CLEAR .....	3-3
CONT .....	3-4
DELETE .....	3-4
EDIT .....	3-5
FILES .....	3-6
LIST .....	3-7
LLIST .....	3-7
LOAD .....	3-8
MERGE .....	3-9
NEW .....	3-9
RENUM .....	3-10
RESET .....	3-11
RUN .....	3-12
SAVE .....	3-13
SYSTEM .....	3-13

### **Chapter Four — Program Statements**

Overview .....	4-1
Data Type Definition .....	4-2
DEFINT .....	4-2
DEFSNG .....	4-2
DEFDBL .....	4-3
DEFSTR .....	4-3
Assignment and Allocation Statements .....	4-4
DIM .....	4-4
OPTION BASE .....	4-4
ERASE .....	4-5
LET .....	4-5
REM .....	4-6
SWAP .....	4-6

---

Control Statements .....	4-7
Sequence of Execution .....	4-7
END .....	4-7
FOR/NEXT .....	4-8
Examples .....	4-9
Nested Loops .....	4-10
GOSUB/RETURN .....	4-11
GOTO .....	4-12
ON/GOTO and ON/GOSUB .....	4-13
STOP .....	4-14
Conditional Execution .....	4-14
IF/THEN/ELSE .....	4-15
Additional Considerations .....	4-16
Nesting of IF Statements .....	4-16
WHILE/WEND .....	4-17
I/O Statements (Non-Disk) .....	4-18
DATA .....	4-18
INPUT .....	4-19
LINE INPUT .....	4-20
LPRINT .....	4-21
PRINT .....	4-21
Print Positions .....	4-21
Examples .....	4-22
READ .....	4-23
RESTORE .....	4-24
WRITE .....	4-25

## Chapter Five — Strings

Overview .....	5-1
String Input/Output .....	5-2
String Operations .....	5-3
String Functions .....	5-4
ASC .....	5-5
CHR\$ .....	5-5
HEX\$ .....	5-6
INKEY\$ .....	5-6
INPUT\$ .....	5-7
INSTR .....	5-8
LEFT\$ .....	5-8
LEN .....	5-9
MID\$ .....	5-9
MID\$ .....	5-10
OCT\$ .....	5-10
RIGHT\$ .....	5-11
SPACE\$ .....	5-11
STR\$ .....	5-12
STRING\$ .....	5-12
VAL .....	5-13

## Chapter Six — Arrays

Overview .....	6-1
Arrays .....	6-2
Array Declarator .....	6-2
Array Subscript .....	6-3
OPTION BASE Statement .....	6-3
Vertical Arrays .....	6-4
Multi-Dimensional Arrays .....	6-5
Matrix Manipulation .....	6-6
Matrix Input Subroutine .....	6-6
Scalar Multiplication .....	6-7
Transposition of a Matrix .....	6-7
Matrix Addition .....	6-8
Matrix Multiplication .....	6-8

## Chapter Seven — Functions

Overview .....	7-1
Arithmetic Functions .....	7-2
ABS .....	7-3
ATN .....	7-3
CDBL .....	7-4
CINT .....	7-4
COS .....	7-5
CSNG .....	7-5
EXP .....	7-6
FIX .....	7-6
INT .....	7-7
LOG .....	7-7
RND .....	7-8
RANDOMIZE .....	7-8
SGN .....	7-9
SIN .....	7-10
SQR .....	7-10
TAN .....	7-10
Mathematical Functions .....	7-11
Special Functions .....	7-12
FRE .....	7-13
INP .....	7-13
LPOS .....	7-14
OUT .....	7-14
PEEK .....	7-15
POKE .....	7-15
POS .....	7-16
SPC .....	7-16
TAB .....	7-17
VARPTR .....	7-18
WAIT .....	7-21
WIDTH .....	7-22
User-Defined Functions .....	7-23
DEF FN .....	7-23
Assembly Language Programs .....	7-24
DEF USR .....	7-24
USR .....	7-25
CALL .....	7-26

## Chapter Eight — Special Features

Overview .....	8-1
Error Trapping .....	8-2
ON ERROR GOTO .....	8-2
RESUME .....	8-3
Error Trap Example .....	8-3
ERROR .....	8-4
ERR and ERL Variables .....	8-5
Error Codes .....	8-6
Formatted Output .....	8-8
PRINT USING .....	8-8
String Fields .....	8-8
Numeric Fields .....	8-9
Trace Flag .....	8-14
TRON/TROFF .....	8-14
Overlay Management .....	8-15
CHAIN .....	8-15
COMMON .....	8-16

## Chapter Nine — Editing

Overview .....	9-1
Moving the Cursor .....	9-3
Inserting Text .....	9-4
Deleting Text .....	9-6
Finding Text .....	9-7
Replacing Text .....	9-8
Ending and Restarting Edit Mode .....	9-9
Other Edit Mode Features .....	9-11

## Chapter Ten — BASIC-80 Disk File Operations

Overview .....	10-1
File Manipulation Commands .....	10-2
FILES .....	10-2
KILL .....	10-2
LOAD .....	10-2
MERGE .....	10-2
NAME .....	10-2
RESET .....	10-3
RUN .....	10-3
SAVE .....	10-3
Protected Files .....	10-3

File Management Statements .....	10-4
OPEN .....	10-5
CLOSE .....	10-8
EOF .....	10-9
LOF .....	10-9
LOC .....	10-10
BASIC-80 Sequential I/O .....	10-11
Sequential Access Statements .....	10-11
INPUT# .....	10-11
Numeric Input .....	10-12
String Input .....	10-14
LINE INPUT# .....	10-16
PRINT# and PRINT# USING .....	10-17
WRITE# .....	10-19
Sequential Access Techniques .....	10-21
Creating and Accessing a Sequential File .....	10-21
Adding Data to a Sequential File .....	10-23
BASIC-80 Random I/O .....	10-25
Random Access Statements .....	10-26
FIELD .....	10-27
LSET/RSET .....	10-29
GET .....	10-30
PUT .....	10-31
MKI\$, MKS\$, MKD\$ .....	10-32
CVI, CVS, CVD .....	10-33
Random Access Techniques .....	10-34
Creating a Random Access File .....	10-34
Accessing a Random Access File .....	10-36
Additional Features .....	10-37

## Chapter Eleven — Microsoft BASIC-80 Summary

Overview .....	11-1
Abbreviations .....	11-2
Data Type Declaration Characters .....	11-2
Arithmetic Operators .....	11-3
String Operator .....	11-3
Relational Operators .....	11-3
Logical Operators .....	11-4
Commands .....	11-5
Edit Mode Subcommands and Functions .....	11-9
Print Using Format Field Specifiers .....	11-10
NumericSpecifier .....	11-10
StringSpecifier .....	11-10

---

Program Statements .....	11-11
Data Type Definition .....	11-11
Assignment and Allocation .....	11-11
Sequence of Execution .....	11-12
Conditional Execution .....	11-13
Non-Disk I/O Statements .....	11-14
String Functions .....	11-16
Arithmetic Functions .....	11-18
Special Functions .....	11-19
Special Features .....	11-20
Error Trapping .....	11-20
Trace Flag .....	11-20
Overlay Management .....	11-21
Disk Input/Output Statements .....	11-22
Disk Input/Output Functions .....	11-24

#### **Appendix A — Error Messages**

General Errors .....	A-1
Disk Related Errors .....	A-6
Reserved Words .....	A-8

#### **Appendix B — ASCII Codes**

Decimal to Octal to Hex to ASCII Conversion .....	B-1
Control Character Definitions .....	B-2

#### **Appendix C — New Features in BASIC-80**

New Features in BASIC-80 .....	G-1
--------------------------------	-----

#### **Appendix D — Programming Hints**

Conserving Memory Space .....	D-1
Saving Execution Time .....	D-3

## Appendix E — Assembly Language Subroutines

Memory Allocation .....	E-2
User Function Calls .....	E-3
Numeric Storage Format .....	E-5
Integer Storage Format .....	E-5
Single-Precision Storage Format .....	E-5
Double-Precision Storage Format .....	E-5
String Storage Format .....	E-6
Data Type Conversions .....	E-6
CALL Statement .....	E-7
Interrupts .....	E-9

## Appendix F — Random And Sequential I/O Programming Examples F-1

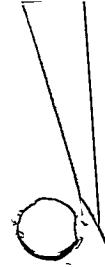
### Index

Index .....	I-1
-------------	-----

## Tables

### Table

2-1	Arithmetic Operators .....	2-8
2-2	Relational Operators .....	2-10
2-3	Logical Operators .....	2-11
2-4	Truth Table for Logical Operators .....	2-12
5-1	String Functions .....	5-4
6-1	Array Storage Allocation .....	6-4
6-2	Multi-Dimensional Array Storage Allocation .....	6-5
7-1	Arithmetic Functions .....	7-2
7-2	Mathematical Functions .....	7-11
7-3	Special Functions .....	7-12
8-1	Error Codes .....	8-6
10-1	File Management Statements .....	10-4
10-2	Sequential Access Statements .....	10-11
10-3	Random Access Statements .....	10-26
E-1	Register Values Used to Specify Data Types .....	E-4





## Chapter One

# System Introduction and General Information

## OVERVIEW

This Chapter contains an "Installation Guide" and general reference information pertaining to the BASIC-80 Programming Language. BASIC-80 is one of the most extensive implementations of BASIC available for the 8080 and Z80 micro-processors.



The hardware and systems software requirements for BASIC-80 are presented in this Chapter.

This Chapter also contains a user-oriented explanation of the operating environment of BASIC-80.



## INSTALLATION GUIDE

*for the Microsoft BASIC-80  
Interpreter and BASIC Compiler*

*Technical consultation is available for  
any problems you encounter in verifying  
the proper operation of these products.  
We are not able to evaluate or assist in the  
debugging of any programs you may de-  
velop. For technical assistance, call:*

*(616) 982-3860*

*Consultation is available between 8:00  
AM and 4:30 PM on normal business  
days.*

## Contents of the Diskettes

### **Microsoft BASIC-80 Interpreter**

There is one diskette distributed with the Interpreter package. It contains the following files:

MBASIC.COM  
PI.BAS

MBASIC.COM is the BASIC Interpreter. Its commands and functions are discussed in this Reference Manual. PI.BAS is a sample program written in BASIC which calculates the value of pi. PI.BAS is provided to help familiarize you with the workings of the interpreter.

### **Microsoft BASIC Compiler**

There may be two diskettes distributed with the Compiler package: Distribution Disk I and Distribution Disk II. Distribution Disk I contains the following files:

BASCOM.COM  
BASLIB.REL

The commands and functions of the BASIC Compiler, which is stored in the file BASCOM.COM, are documented in the "BASIC Compiler User's Manual." BASLIB.REL is the BASIC Compiler System Library. You may modify this file by using the Library Manager (LIB.COM, on Compiler Distribution Disk II).

Distribution Disk II contains the following files:

L80.COM	LIB.COM
M80.COM	PI.BAS
CREF.COM	PI.REL

Section 2 of the "Microsoft Utility Manual" defines the use and operation of the MACRO-80 Assembler (M80.COM). CREF.COM, the Cross-Reference Facility, is described in Section 3 of the Utility Manual; L80, the Linking Loader, is discussed in Section 4; and LIB.COM, the Library Manager, is discussed in Section 5.

PI.BAS is a sample program designed to calculate the value of pi. It is provided to assist you in learning how to compile, link, and execute a program. PI.REL is the relocatable object file generated by the Compiler from PI.BAS.

Based on the type of distribution media you received, the files mentioned above may be recorded on one or more disks.

## Sample Output of PI.BAS

The listings provided below are sample outputs of the PI.BAS program. Note that the results generated by the Interpreter and Compiler differ due to the different algorithms used to manipulate data.

### BOUNDS ON PI — DOUBLE PRECISION BINOMIAL THEOREM VERSION

N	SIDES	SIDE LENGTH	PI-LOWER BOUND	PI-UPPER BOUND
3	8	0.76536691188812	3.06146764755249	4.95931573036713
4	16	0.39018064737320	3.12144517898560	3.87800677621650
5	32	0.19603428244591	3.13654851913452	3.47739260077205
6	64	0.09813534468412	3.14033102989197	3.30237067197655
7	128	0.04908246546984	3.14127779006958	3.22030812114884
8	256	0.02454307302833	3.14151334762573	3.18054350336212
9	512	0.01227176748216	3.14157247543335	3.16096780640274
10	1,024	0.00613591633737	3.14158916473389	3.15125708966375
11	2,048	0.00306796119548	3.14159226417542	3.14641880958168
12	4,096	0.00153398059774	3.14159226417542	3.14400368450104
13	8,192	0.00076699029887	3.14159226417542	3.14279751177684
14	16,384	0.00038349514944	3.14159226417542	3.14219477240231
15	32,768	0.00019174757472	3.14159226417542	3.14189348940372
16	65,536	0.00009587385284	3.14159440994263	3.14174501554227
17	131,072	0.00004793689368	3.14159226417542	3.14166756506744
18	262,144	0.00002396846321	3.14159440994263	3.14163205998885
19	524,288	0.00001198423161	3.14159440994263	3.14161323485294
20	1,048,576	0.00000599211580	3.14159440994263	3.14160382236958

### Interpreter Results

### BOUNDS ON PI — DOUBLE PRECISION BINOMIAL THEOREM VERSION

N	SIDES	SIDE LENGTH	PI-LOWER BOUND	PI-UPPER BOUND
3	8	0.76536686473018	3.06146745892072	4.95931523537420
4	16	0.39018064403226	3.12144515225805	3.87800673496263
5	32	0.19603428065912	3.13654849054594	3.47739256563251
6	64	0.09813534865484	3.14033115695475	3.30237081249040
7	128	0.04908245704582	3.14127725093277	3.22030755454287
8	256	0.02454307657144	3.14151380114430	3.18054396821973
9	512	0.01227176929831	3.14157294036709	3.16096827709498
10	1,024	0.00613591352593	3.14158772527716	3.15125564133382
11	2,048	0.00306796037257	3.14159142151120	3.14641796432625
12	4,096	0.00153398063749	3.14159234557012	3.14400376602075
13	8,192	0.00076699037514	3.14159257658487	3.14279782442605
14	16,384	0.00038349519462	3.14159263433856	3.14219514270746
15	32,768	0.00019174759819	3.14159264877699	3.14189387407905
16	65,536	0.00009587379921	3.14159265238659	3.14174325781772
17	131,072	0.00004793689962	3.14159265328899	3.14166795419967
18	262,144	0.00002396844981	3.14159265351459	3.14163030351872
19	524,288	0.00001198422491	3.14159265357099	3.14161147846025
20	1,048,576	0.00000599211245	3.14159265358509	3.14160206600152

### Compiler Results

## Diskette Use

### DISKETTE LOADING

Refer to Figure 1-1A or 1-1B, open the disk drive door, and insert the diskette(s) so the diskette label faces the open door. Then carefully close the drive door.

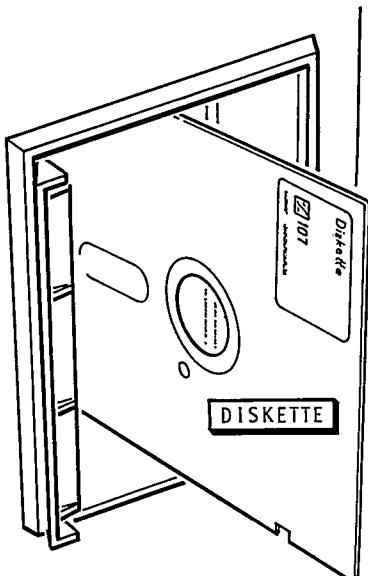


Figure 1-1A

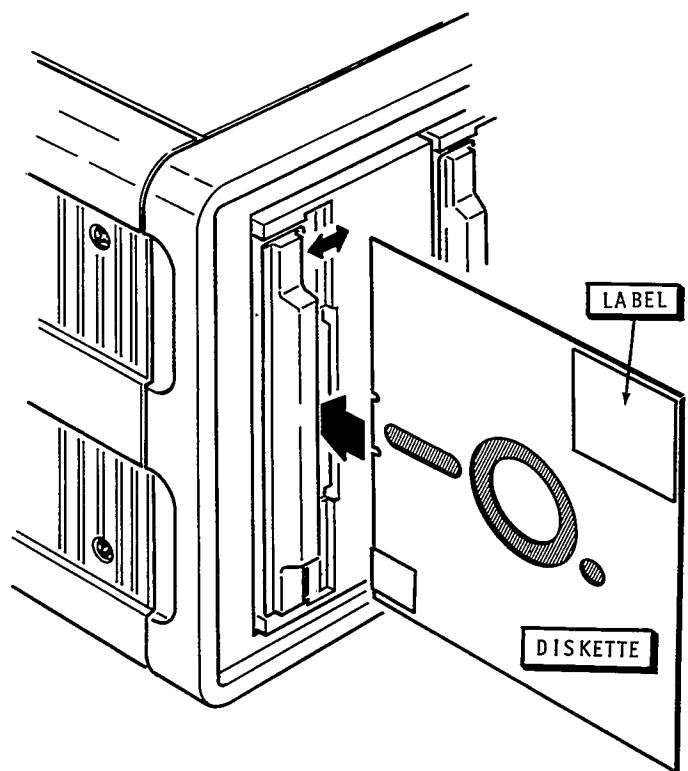


Figure 1-1B

## DISKETTE HANDLING

Diskettes are easily damaged. Observe the following precautions when handling diskettes:

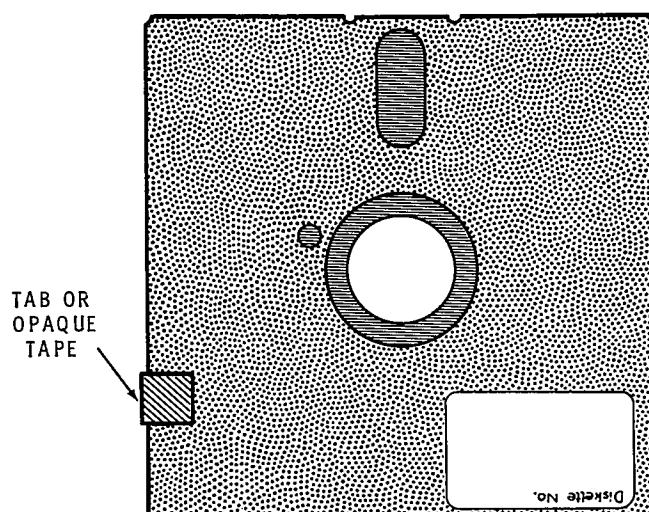
1. Keep the diskette in its storage envelope whenever it is not in use.
2. Keep the diskette away from magnetic fields, including magnetic paper clip holders, magnetized scissors or screwdrivers, and heavy electrical equipment. Magnetic fields can distort the data recorded on the diskette.
3. Replace damaged or excessively worn storage envelopes.
4. Write only on the diskette label, and then only with a felt-tip pen. Do not use a pencil or ball-point pen, as these may damage the recording surface.
5. Keep the diskettes away from hot or contaminating material.
6. Do not expose the diskette to sunlight, liquids, or smoke.
7. Do not touch the diskette surface. Abrasions can alter stored data.

## WRITE-PROTECTION

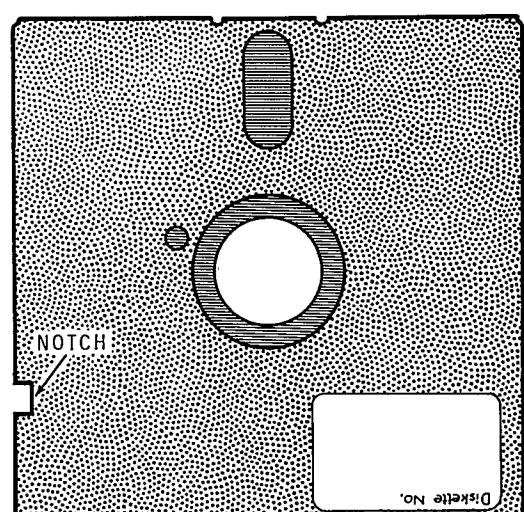
The diskette can be write-protected so that data cannot be written to it. (All distribution diskettes are shipped write-protected). How a disk is write-protected depends on the size of the diskette.

A 5.25-inch diskette has a **write-protect** notch on the side. When this notch is covered with a tab or opaque tape, no data can be written on the diskette. Figure 1-2A illustrates a write-protected 5.25-inch diskette. Figure 1-2B depicts a write-enabled 5.25-inch diskette.

An 8-inch diskette has a **write-enable** notch on its side. If this write-enable notch is exposed, no data can be written to the diskette. To write-enable an 8-inch diskette, cover the write-enable notch with a tab or opaque tape. Figure 1-3A shows a write-protected 8-inch diskette. Figure 1-3B shows a write-enabled 8-inch diskette.



WRITE-PROTECTED



WRITE-ENABLED

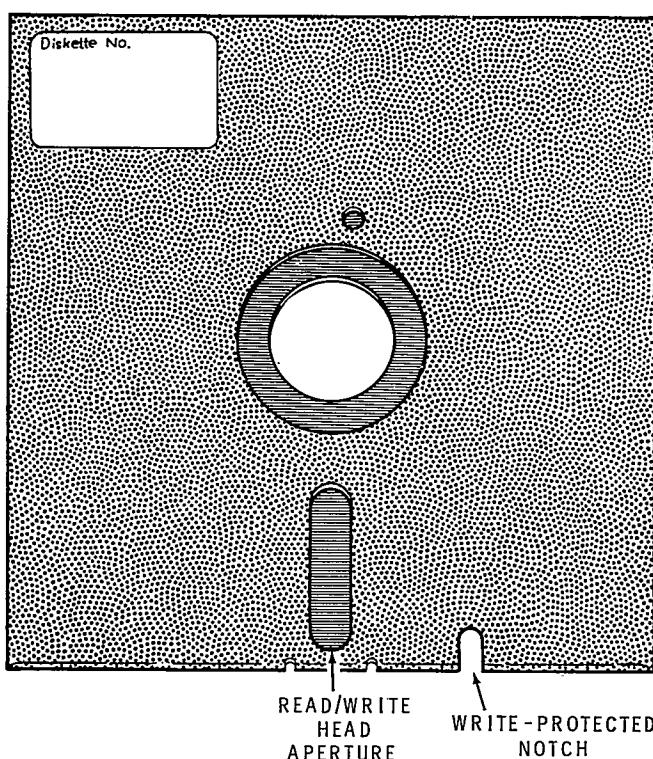


Figure 1-3A

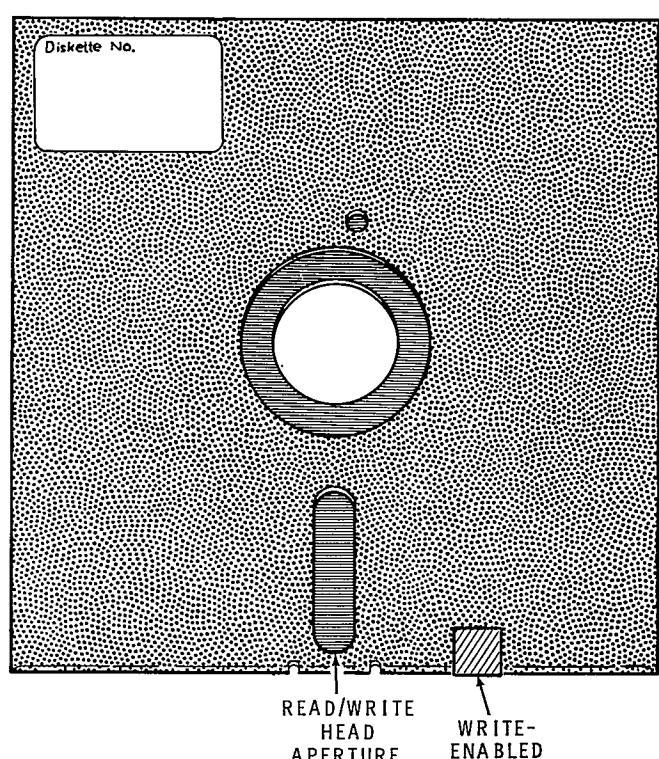


Figure 1-3B

## Preparing Working Diskettes

Using the procedure outlined in your CP/M manual, power-up your computer and boot-up CP/M from CP/M Distribution Disk I.

If you have two or more drives of the same size, duplicate your MBASIC distribution diskette(s) using DUP.COM. If you do not have two or more drives of the same size:

1. Initialize the blank diskette(s) to which you will copy using FORMAT.COM.
2. Duplicate the MBASIC distribution disk(s) using PIP.COM.

NOTE: All distribution diskettes are write-protected to ensure that you always have an accurate copy of the software. Therefore, duplicate the distribution diskettes and then store them in a safe place. Use your copies for day-to-day use of the programs.

## SYSTEM INTRODUCTION

### Manual Scope

This BASIC-80 Reference Manual is your reference source for the BASIC-80 language. Its Chapters are organized in a functional manner. If, for example, you need information about strings, simply refer to Chapter Five, Strings.

Also included with the BASIC-80 package are an Installation Guide and a Reference Card. The Guide contains the information you needed to create a working copy of the BASIC-80 Interpreter. Keep the Reference Card handy, as it contains often needed information.

## Hardware Requirements

The hardware required to run the BASIC-80 Interpreter is:

1. 8080 or Z80 microcomputer
2. 48K of RAM.
3. One floppy disk drive.
4. Terminal device.
5. Optionally — a hard copy device

This is the minimum hardware configuration. We recommend that you have more than one disk drive. If you plan to develop large programs, you will no doubt need a hard copy device.

## System Software Requirements

The BASIC-80 Interpreter is designed to run under CP/M version 2.0 and later.

## Preparing the Diskette

The BASIC-80 Interpreter is distributed on either a 5.25" mini-floppy or an 8" floppy. The Installation Guide furnished with this product contains the information you will need when you create your working diskette.

Never use your distribution copy of BASIC-80 except to make copies for your own use. Keep your distribution copy in a safe place. The Installation Guide contains more information about disk handling procedures.

## Initialization of BASIC-80

BASIC-80 is distributed in an absolute binary format. BASIC-80 is stored on the disk with the file name MBASIC.COM. BASIC-80 can be directly loaded into memory and used. To load BASIC-80, type the following in response to the CP/M prompt:

MBASIC

This command will load MBASIC into memory. After MBASIC has been loaded into memory, a sign-on message will be displayed. The message should look similiar to this:

```
BASIC-80 Rev. 5.2
[CP/M Version]
Copyright 1977, 78, 79, 80 (C) by Microsoft
Created: 11-Aug-80
15430 Bytes free
```

Note that the revision number, the creation date, and the number of free bytes might be different with your system.

A BASIC-80 program can be automatically executed when the file name is appended to the command string. For example, if you want to load the interpreter and run the program SAMPLE.BAS, you could use the following command string:

MBASIC $\Delta$ SAMPLE

The space between MBASIC and SAMPLE is required. (Throughout this manual, we will use the symbol  $\Delta$  to indicate a required space.) The default extension .BAS will be assumed. If the file name specified can not be found, the message "File not found" will be displayed, and you will be returned to the CP/M Command Mode.

## GENERAL INFORMATION

### Modes of Operation

After you have loaded the interpreter, BASIC-80 will type "Ok". This prompt signifies that BASIC-80 is in the Command Mode.

In the Command Mode, the BASIC-80 Interpreter will execute your instruction as soon as you terminate the entry with a RETURN. The commands and statements entered in Command Mode should not be preceded by line numbers. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC-80 as a "calculator" for quick computations that do not require a complete program.

If you begin a program line with a line number, BASIC-80 assumes that you wish to store this program line for execution at a later date. This is called the Intermediate or Program Mode. The program stored in memory will be executed if you enter the RUN command.

### Line Format

Program lines in a BASIC-80 program have the following format (square brackets indicate optional):

nnnnn BASIC-80 statement [:BASIC-80 statement... ]

At the programmer's option, more than one BASIC-80 statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A BASIC-80 program line always begins with a line number, ends with a carriage return, and may contain a maximum of 255 characters.

It is possible to extend a logical line over more than one physical line by use of the terminal's LINE FEED key. LINE FEED lets you continue typing a logical line on the next physical line without entering a RETURN.

### Line Numbers

Every BASIC-80 program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references for branching and editing. Line numbers must be in the range 0 to 65529. A period (.) may be used in EDIT, LIST, AUTO and DELETE commands to refer to the current line.

## Character Set

The BASIC-80 character set is comprised of alphabetic characters, numeric characters and special characters. The alphabetic characters are the upper case and lower case letters of the alphabet. The numeric characters are the digits 0 through 9.

BASIC-80 also recognizes the following special characters and terminal keys:

<u>Character</u>	<u>Name</u>
	Blank
;	Semicolon
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
↑	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number (or pound) sign
\$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
DELETE	Deletes last character typed.
ESC	Escapes Edit Mode subcommands.
TAB	Moves print position to next tab stop. Tab stops are every eight columns.
LINE FEED	Moves to next physical line.
RETURN	Terminates input of a line.

## Control Characters

The following control characters are in BASIC-80:

- CTRL-A Enters Edit Mode on the line being typed.
- CTRL-C Interrupts program execution and returns to BASIC-80 command level.
- CTRL-G Rings the bell at the terminal.
- CTRL-H Backspace. Deletes the last character typed.
- CTRL-I Tab. Tab stops are every eight columns.
- CTRL-O Halts program output while execution continues. A second Control-O restarts output.
- CTRL-R Retypes the line that is currently being typed.
- CTRL-S Suspends program execution.
- CTRL-Q Resumes program execution after a Control-S.
- CTRL-U Deletes the line that is currently being typed.

To execute any of these control characters, hold down the CTRL key while simultaneously typing the letter. Thus, to execute CTRL-G, hold down the CTRL key while simultaneously typing the letter G.

## BASIC-80 PROGRAMMING

This section will tell you how to write a BASIC-80 program and explain the unique features of the BASIC-80 programming environment. No attempt will be made to teach the subject of BASIC programming, but enough information will be provided so that you should be able to successfully use the BASIC-80 Interpreter.

### Loading the BASIC-80 Interpreter

The BASIC-80 Interpreter, which must be loaded into your computers' memory before you can use it, is an absolute binary file. This means that it is in a form which can be directly executed by your computer. Before you can perform the procedures listed below, you must "boot-up" your computer. If you are unsure how to do this, refer to the appropriate operating system manual.

The CP/M file name used to reference the interpreter is: MBASIC.COM. So, to load the BASIC-80 Interpreter into memory, type the following response to the prompt from CP/M:

A>MBASIC

(Do not type the A>, as this represents the prompt from CP/M; and remember to terminate the line by pressing the RETURN key.)

This assumes that the file MBASIC.COM resides on the current default disk. If the file does not reside on the current default disk, type the drive name and then the file name. For example, if A: is the current default disk, and the BASIC-80 file resides on drive B:, you would use the following command to load BASIC-80:

A>B: MBASIC

After BASIC-80 is loaded into memory, a sign-on message will be displayed on your screen. The amount of free memory, as well as the BASIC-80 version number, will also be displayed. Take note of the amount of free memory, as this will no doubt be a crucial issue if you wish to write large, complex programs.

When BASIC-80 is loaded in the manner described above, it will make certain assumptions about the operating environment. BASIC-80 assumes that:

No more than 3 disk files will be open,  
All available memory will be used,  
Random record size is 128 bytes.

You can change these assumptions by using certain switches.

The number of disk files that can be open can range from 0-15. The /F: switch is used to specify the maximum number of files. BASIC-80 will establish a file buffer in memory for each file specified with the /F: switch. This will decrease the amount of free memory that you have to work with. For example, to set up five file buffers, you could use the following command:

A>MBASIC△/F:5

Note the space that is required between MBASIC and the /F:5. If you do not type this space, CP/M will assume that the switch is part of the file name.

You can also specify the highest memory location BASIC-80 will use with the /M: switch. In some cases it is desirable to set the amount of memory well below the CP/M BDOS to reserve space for assembly language subroutines. In all cases, the highest memory location should be below the start of BDOS (whose address is contained in locations 6 and 7). If the /M: switch is omitted, all memory up to the start of BDOS is used.

NOTE: The number of files and the highest memory location numbers can be either decimal, octal (preceded by a&O), or hexadecimal (preceded by&H).

You can also change the record size of a random file by using the /S: switch. The default record size is 128 bytes, and the maximum record size is 256 bytes. For example, to set the maximum record size to 200 bytes, you could use the following command:

A>MBASIC△/S:200

Any combination of these three switches can be used in a command line. For example:

A>MBASIC△PAYROLL.BAS

Use all memory and 3 files, load and execute PAYROLL.BAS

A>MBASIC△INVENT/F:6

Use all memory and 6 files, load and execute INVENT.BAS

A>MBASIC△M:32768

Use first 32K of memory and 3 files.

After the BASIC-80 interpreter has been loaded into memory, a program may be written.

## Writing a BASIC-80 Program

A BASIC-80 program is composed of lines of statements containing instructions to BASIC-80. Each of these program lines begins with a line number, followed by one or more BASIC-80 program statements. These line numbers indicate the sequence of statement execution, although this sequence may be changed by certain statements.

The format of a BASIC-80 program line is:

<u>line number</u>	<u>statement keyword</u>	<u>statement text</u>	<u>line terminator</u>
100	LET	X = X+1	<RETURN>

Every program line in a BASIC-80 program must begin with a line number, which must be a positive integer within the range 0 - 65529. This BASIC-80 line number is a label that distinguishes one line from another within a program. Thus, each line number in the program must be unique.

Each program line in a BASIC-80 program is terminated with a carriage return, which you can generate by pressing the RETURN key on your console device.

You could use consecutive line numbers like 1,2,3,4. For example:

```
1 X = 1  
2 Y = 2  
3 Z = X+Y  
4 END
```

However, a useful practice is to write line numbers in increments of 10. This method will allow you to insert additional statements later between existing program lines.

```
10 X = 1  
20 Y = 2  
30 Z = X+Y  
40 END
```

Another useful practice is to let BASIC-80 automatically generate line numbers for you. This is accomplished with the AUTO statement. The AUTO statement tells BASIC-80 to automatically generate line numbers. For example, if you type AUTO 100,10, then BASIC-80 will generate line numbers beginning with line number 100 and incrementing each line by 10. Then all you need to do is type the BASIC-80 program line after the generated line number.

## Running a BASIC-80 Program

After a BASIC-80 program has been written, it is usually desirable to execute the program. The task can be accomplished by the RUN command. The following statement would tell BASIC-80 to execute the program currently in memory:

RUN

Execution would begin at the lowest number line and continue with the next lowest number line (unless the sequence of execution was altered with a statement like the GOTO statement). The RUN command can also specify the first line number to be executed. For example, the following command would cause execution to begin with line number 100:

RUNΔ100

The RUN command can also be used to execute a BASIC-80 program that is currently residing on a disk file. For example, assume the file ALBUM.BAS resides on the current default disk. The following statement would be used to execute ALBUM.BAS:

RUN "ALBUM"

Note that no drive specification or file name extension was included in the file name string. In this case, the current default drive and the extension .BAS are assumed.

Also make sure that you always use only upper-case letters in the file name string. BASIC-80 must rely on CP/M to manipulate files for it, and most CP/M utilities cannot recognize any file whose name is stored in lower-case letters. Thus, storing a file under a lower-case file name can be very unpleasant, since CP/M cannot recognize the lower-case file name, and therefore cannot ERAse or RENAME the file. Files whose names are stored in lower-case letters can be deleted only from within BASIC-80. This practice of using only upper-case letters in a file name applies to all BASIC-80 statements which require a file name to be specified.

This is not to say that there is anything intrinsically wrong in using lower-case letters in a file name; it is just that assigning lower-case file names may produce an undesirable result. You may want to use a lower-case file name to record a file in such a way that it cannot be easily renamed or erased. Thus, using lower-case file names can provide an extra level of protection for important programs.

## Debugging a BASIC-80 Program

In some cases, a BASIC-80 program will not execute as you expected. This is usually a result of either a syntax error or a logic error. A syntax error is much easier to detect, as BASIC-80 will not only detect these syntax errors for you, but also it will point out the offending program line and invoke the Edit Mode. A logic error is much harder to detect, but several statements have been provided to make this a much more pleasant task.

When BASIC-80 detects a syntax error, it will automatically enter the Edit Mode at the line that caused the error. At this point, you may wish to press the L key in order to list this line. ( L is a command to the BASIC-80 Editor, for more information about the Editor, see Chapter Nine, "Editing".)

Syntax errors are usually a result of a misspelled keyword or an incorrectly structured program line. Remember that BASIC-80 requires all keywords to be delimited by a space. The easiest way to correct a syntax error is to rely heavily on the Reference Manual.

Anytime you have a syntax error, you should refer to the appropriate page in the Reference Manual. Use the Index to find the appropriate page. After you discover and correct your error, remember what you did wrong so you can avoid making the same mistake again.

Because of the interactive nature of BASIC-80, it is very convenient to debug a BASIC-80 program. Several statements have been provided to help you debug a BASIC-80 program. But your first step is to find out the nature of the "bug".

A program "bug" may cause the wrong values to be output. Or maybe a program is branching to the wrong statement. The results of a calculation may be wrong, or the results of a calculation may be incomprehensible. A program "bug" might cause an error condition to be flagged. So you must discover what the program is doing before you can discover why the program is doing it.

Also keep in mind that, in most cases (99.99%), it is a bug in your program that is causing a problem. It is highly unlikely that the BASIC-80 Interpreter is at fault. This Interpreter represents one of the most comprehensive implementations of BASIC available for the 8080/Z80, and as such is very stable. So, it is best to always assume that a problem is caused by a user program bug.

Once you have decided what the program is doing, you can take steps to discover why it is not executing correctly. For example, assume that a program is branching to a line number different than where you want it to branch. The trace flag has been provided to trace the flow of a program. To enable the trace, the TRON statement is used, and to disable the trace, the TROF statement is used.

The trace flag will print each line number as it is being executed. The line number will be enclosed in square brackets ([]). It is best to generate a hard copy listing of the program first so you can follow this listing while the trace is running.

Another important technique you can use is to set breakpoints in a program. You can use the STOP statement to temporarily terminate program execution, and then enter commands to print the values of various variables. You can also assign new values to these variables. Then you can continue program execution with a CONT command or a Command Mode GOTO.

Although you can print and change the values assigned to variables, you must not change the BASIC-80 program after you interrupted execution with a STOP statement. If you do change the program, all the previously stored variable values will be lost, and all open files will be closed.

## Saving a BASIC-80 Program

When you have completed a BASIC-80 programming session, you will no doubt want to save a copy of your most current program on the disk. This is accomplished with the SAVE command. The general form of the SAVE command is:

```
SAVE "<filename>"
```

The <file name> must be a valid CP/M file name. If no device specification is given, the current default drive will be assumed. If no file name extension is given, the default extension of .BAS will be assumed. For example, if you wish to save a program called GAME.BAS, you could use the following statement:

```
SAVE "C:GAME.BAS"
```

Note that this file will be written on drive C:. The file name extension of .BAS could have been omitted and then it would have been supplied as the default. Make sure you always use upper case letters when specifying a file name. BASIC-80 will usually save files in a compressed binary format. A program can optionally be saved in ASCII format, but it will take more disk space to store it this way. To save a program in ASCII format, append an A to the end of the file name string. For example:

```
SAVE "C:GAME",A
```

This will save the file on drive C: in ASCII format with a file name of GAME.BAS. You can also save a program in a protected format so it can not be listed or edited. Just append a P to the end of the file name string. For example:

```
SAVE "C:GAME",P
```

This file will be saved in an encoded binary format. When this protected file is later RUN or (LOADed), any attempt to LIST or EDIT this program will fail.

## ○ Loading a BASIC-80 Program

When you begin a BASIC-80 programming session, you may want to load a program from the disk into memory. This is accomplished with the LOAD command. The general form of the LOAD command is:

```
LOAD "<filename>"
```

For example, if you wanted to load the program PAYROL.BAS, you could use the command:

```
LOAD "PAYROL"
```

Note that the file name extension was omitted. BASIC-80 will assume a file name extension of .BAS. Also note that the drive specification was omitted. In this case, the current default drive will be assumed.

You must specify the file name using only upper case letters. This applies to all string constants or variables that contain file names.

It is also possible to execute a program with the LOAD command. In this case, an R is appended to the end of the file name string. For example:

```
LOAD "PAYROL",R
```

This form of the LOAD command will load a program into memory and execute it as if a RUN command had been typed. All currently open files will remain open for use by the program.

## **Listing a BASIC-80 Program to a Hard Copy Device**

At some point during your programming effort, you may want a hard copy listing of a BASIC-80 program. A BASIC-80 program is listed to a hard copy device in much the same manner as it is listed to a console device. Use the LLIST command.

The general form of the LLIST command is :

LLIST

This will list the current program on the hard copy device. It is also possible to specify the range of line numbers to be listed. For example in order to list a single line, you can use the command:

LLIST 100

This will list only the line number 100. A range of line numbers can also be specified:

LLIST 100-500

This will list line numbers 100 through 500, inclusive.

The LLIST command will direct the output to the CP/M LST: device. This logical device can be assigned to several different physical devices. Refer to your CP/M manual for information about this process.





## *Chapter Two*

# **Expressions**

## **OVERVIEW**

An expression is a group of symbols to be evaluated by BASIC-80. Expressions are composed of numeric or string variables, numeric or string constants, and functions references. These operands can be alone, or they can be combined by arithmetic, logical, or relational operators. This Chapter explains the various rules for constructing and evaluating expressions.

## CONSTANTS

Constants are the actual values BASIC-80 uses during execution. There are two types of constants: string and numeric.

### String Constants

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

```
"HELLO"  
"25,000.00"  
"Number of Employees"
```

### Numeric Constants

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

#### INTEGER CONSTANTS

Integer constants are whole numbers between -32768 and +32767. Integer constants can not have decimal points.

#### FIXED POINT CONSTANTS

Fixed point constants are positive or negative real numbers, i.e., numbers that contain decimal points.

#### FLOATING POINT CONSTANTS

Floating point constants are positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is  $10^{-38}$  to  $10^{+38}$ .

Examples:

```
235.988E-7 = .0000235988  
2359E6 = 2359000000
```

(Double-precision floating point constants use the letter D instead of E.)

## HEX CONSTANTS

Hexadecimal constants are hexadecimal numbers with the prefix &H.

Examples:

&H76  
&H32F

## OCTAL CONSTANTS

Octal constants are octal numbers with the prefix &O or &.

Examples:

&0347  
&1234

## SINGLE AND DOUBLE-PRECISION NUMERIC CONSTANTS

Fixed and floating point numeric constants may be either single-precision or double-precision numbers. With double-precision, the numbers are stored with 16 digits of precision, and printed with up to 16 digits.

A single-precision constant is any numeric constant that has:

1. Seven or fewer digits, or,
2. Exponential form using E, or,
3. A trailing exclamation point (!).

A double-precision constant is any numeric constant that has:

1. Eight or more digits, or,
2. Exponential form using D, or,
3. A trailing number sign (#).

Examples:

### Single-Precision Constants

46.8  
-7.09E-06  
3489.0  
22.5!

### Double-Precision Constants

345692811  
-1.09432D-06  
3489.0#  
7654321.1234

## VARIABLES

Variables are names which represent values that are used in a BASIC-80 program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

### Variable Names and Declaration Characters

BASIC-80 variable names may be any length. However, only the first 40 characters are significant. The characters allowed in a variable name are letters and numbers, and the decimal point is also allowed in a variable name. The first character must be a letter.

A variable name may not be a reserved word. BASIC-80 will allow embedded reserved words to be part of a variable name. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all BASIC-80 commands, statements, function names, and operator names.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign (\$) as the last character. For example: A\$ = "SALES REPORT". The dollar sign is a variable type declaration character; that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer, single-precision, or double-precision values. The type declaration characters for these variable names are as follows:

%	Integer variable
!	Single-precision variable
#	Double-precision variable

The default type for a numeric variable name is single-precision.

## Examples of BASIC-80 Variable Names:

PI#	Declares a double-precision value.
MINIMUM!	Declares a single-precision value.
LIMIT%	Declares an integer value.

There is a second method by which variable types may be declared. The BASIC-80 statements DEFINT, DEFSTR, DEFSNG and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Chapter Four, "Program Statements."

## Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with integers or integer expressions. An array variable name has as many subscripts as there are dimensions in the array.

For example, V(10) would reference a value in a one-dimensional array, T(1,4) would reference a value in a two-dimensional array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767. See Chapter Six, "Arrays," for more information.

## TYPE CONVERSIONS

When necessary, BASIC-80 will convert a numeric constant from one type to another. The following rules and examples illustrate these type conversions.

---

If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)

Example:

```
10 A% = 23.42
20 PRINT A%
RUN
23
```

---

During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision; i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

Example:

```
10 D# = 6#/7
20 PRINT D#
RUN
.8571428571428571 ~
```

In the above example, the arithmetic was performed in double-precision and the result was returned in D# as a double-precision value.

---

```
10 D = 6#/7
20 PRINT D
RUN
.857143
```

In this example, the arithmetic was performed in double-precision and the result was returned to D (a single-precision variable); thus rounded and printed as a single-precision value.

---

When a fixed point value is converted to an integer, the fractional portion is rounded.

Example:

```
10 C% = 55.88
20 PRINT C%
RUN
56
```

---

If a double-precision variable is assigned a single-precision value, only the first seven digits, rounded, of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single-precision value.

The absolute value of the difference between the printed double-precision number and the original single-precision value will be less than 6.3E-8 times the original single-precision value.

Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
2.04    2.039999961853027
```

---

## EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC-80 may be divided into four categories:

1. Arithmetic.
2. Relational.
3. Logical.
4. Functional.

### Arithmetic Operators

The arithmetic operators, in order of precedence, are:

<u>Operator</u>	<u>Operation</u>	<u>Sample Expression</u>
$^$	Exponentiation	$X^Y$
$-$	Negation	$-X$
$*, /$	Multiplication, Floating Point Division	$X*Y$ $X/Y$
$+,-$	Addition, Subtraction	$X+Y$

**Table 2-1**  
Arithmetic Operators.

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Thus, the expressions:

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

is evaluated in the following sequence:

```

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

## INTEGER DIVISION AND MODULUS ARITHMETIC

Two additional arithmetic operators are available in BASIC-80, integer division and modulus arithmetic.

Integer division is denoted by the backslash (\). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer. For example:

$$\begin{aligned}10\backslash 4 &= 2 \\25.68\backslash 6.99 &= 3\end{aligned}$$

The precedence of integer division is just after multiplication and floating point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

$$\begin{aligned}10.4 \text{ MOD } 4 &= 2 \text{ (10}\backslash 4\text{=2 with a remainder 2)} \\25.67 \text{ MOD } 6.99 &= 5 \text{ (26}\backslash 7\text{=3 with a remainder 5)}\end{aligned}$$

The precedence of modulus arithmetic is just after integer division.

## OVERFLOW AND DIVISION BY ZERO

If, during the evaluation of an arithmetic expression, a division by zero is encountered, the “Division by zero” error message is displayed, machine infinity (i.e., 1.70141E +38) with the sign of the numerator is supplied as the result of the division, and execution continues.

If the evaluation of an exponentiation results in zero being raised to a negative power, the “Division by zero” error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the “Overflow” error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

## Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then be used to make a decision regarding program flow.

<u>Operator</u>	<u>Relation Tested</u>	<u>Expression</u>
=	Equality	X=Y
<>	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

**Table 2-2**  
Relational Operators.

(The equal sign is also used to assign a value to a variable.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

$$X+Y < (T-1)/Z$$

is true if the value of X plus Y is less than the value of T-1 divided by Z.

Examples:

```
IF SIN (X)<0 GOTO 1000
IF I MOD J <> 0 THEN K=L+1
```

## Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either “true” (not zero) or “false” (zero). In an expression, logical operations are performed after arithmetic and relational operations. Logical operators convert their operands to integers and return an integer result. Operands must be in the range  $-32768$  to  $32767^*$  or an “Overflow” error occurs.

The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

<u>OPERATOR</u>	<u>EXAMPLE</u>	<u>EXPLANATION</u>
NOT	NOT A	The logical negative of A. If A is true, NOT A is false.
AND	A AND B	The logical product of A and B. A AND B has the value true only if A and B are both true. A AND B has the value false if either A or B is false.
OR	A OR B	The logical sum of A and B. A OR B has the value true if either A or B or both is true. A OR B has the value false only if both A and B are false.
XOR	A XOR B	The logical exclusive OR of A and B. A XOR B is true if either A or B (but not both) is true. Otherwise, A XOR B is false.
IMP	A IMP B	The logical implication of A and B. A IMP B is false if and only if A is true and B is false; otherwise the value is true.
EQV	A EQV B	A is logically equivalent to B. A EQV B is true if A and B are both true or both false. Otherwise, A EQV B is false.

**Table 2-3**  
Logical Operators

\*When you use variables with any of the logical operators, declare the variable as type integer by using either the “%” type declaration character or the DEFINT statement (See Page 4-2 for a discussion of DEFINT).

NOTXNOT X

1

0

0

1

ANDXYX AND Y

1

1

1

1

0

0

0

1

0

0

0

0

ORXYX OR Y

1

1

1

1

0

1

0

1

1

0

0

0

XORXYX XOR Y

1

1

0

1

0

1

0

1

1

0

0

0

IMPXYX IMP YXYX EQV Y

1

1

1

1

1

1

1

0

0

1

0

0

0

1

1

0

1

0

0

0

1

0

0

1

Table 2-4

Truth Table for Logical Operators.

Logical operators work by converting their operands to sixteen bit, signed, two's-complement integers in the range +32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion; i.e., each bit of the result is determined by the corresponding bits in the two operands. In binary arguments, bit 15 is the most significant bit and bit 0 is the least significant bit.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator maybe used to “mask” all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to “merge” two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work. (In all of the examples below, leading zeros on binary numbers are not shown.)

Examples:

$$63 \text{ AND } 16 = 16$$

63 = binary 111111 and 16 = binary 10000, so 63 and 16 = 16

$$15 \text{ AND } 14 = 14$$

15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 binary 1110)

$$-1 \text{ AND } 8 = 8$$

-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8

$$4 \text{ OR } 2 = 6$$

4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)

$$10 \text{ OR } 10 = 10$$

10 = binary 1010, so 1010 OR 1010 = 1010 (10)

$$-1 \text{ OR } -2 = -1$$

-1 = binary 1111111111111111 and -2 = binary 1111111111111110,

so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.

$$\text{NOT } X = -(X+1)$$

The two's complement of any integer is the bit complement plus one.

$$6 \text{ IMP } 2 = -5$$

6 = binary 110 and 2 = binary 10, so 6 IMP 2 = -5

$$3 \text{ EQV } 4 = -8$$

3 = binary 11 and 4 = binary 100, so 3 EQV 4 = binary -8.

### LOGICAL OPERATORS IN RELATIONAL EXPRESSIONS

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision.

Examples:

```
IF D<200 AND F<4 THEN 80  
IF I>10 OR K>0 THEN 50  
IF NOT P THEN 100
```

The result of evaluating the relational expression will be either true (-1) or false (0). This result will then be used as the operand for the logical operator.

## Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. BASIC-80 has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC-80's intrinsic functions are described in Chapter Three, "Functions."

BASIC-80 also allows "user-defined" functions that are written by the programmer. The proper format for constructing and referencing user-defined functions is described in Chapter Seven, "Functions."

## Chapter Three

# Command Mode Statements

## OVERVIEW

Whenever the “Ok” prompt is displayed on the console, BASIC-80 is in the Command Mode. In this Mode, BASIC-80 will respond to a command as soon as it is entered.

Several commands are useful in Command Mode. These are:

AUTO	EDIT	LOAD	RESET
CLEAR	FILES	MERGE	RUN
CONT	LIST	NEW	SAVE
DELETE	LLIST	RENUM	SYSTEM

All of the commands (except CONT) may also be used within a program.

## COMMAND MODE STATEMENTS

### AUTO (enable automatic line numbering)

Form:            AUTO $\Delta$ <line number>,<increment>

The AUTO command will turn on the automatic line numbering function. The AUTO command allows you to enter only the actual program text, as the line numbers will automatically be generated.

AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. If no line number or increment is specified, the default value of 10 is supplied. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing CTRL-C. The line in which CTRL-C is typed is not saved. After CTRL-C is typed, BASIC-80 returns to the Command Mode.

Examples:

AUTO 100,50            Generates line numbers 100,150,200 ...

AUTO                    Generates line numbers 10,20,30,40 ...

AUTO 500               Generates line numbers 500,510,520 ...

## CLEAR (initialize variables)

Form:      CLEAR,<expression1>,<expression2>.

The CLEAR command will set all numeric variables to zero and all string variables to null. The CLEAR command can optionally be used to set the high memory limit and the amount of stack space that is available to BASIC-80.

<expression1> is a memory location (expressed in decimal) which, if specified, sets the highest memory location available for use by BASIC-80.

<expression2> sets aside stack space for use by BASIC-80. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

NOTE: In previous versions of Microsoft BASIC, <expression1> specified the amount of memory to be used for string storage and <expression2> set the end of memory. BASIC-80 release 5.0 allocates string space dynamically, so there is no need to specify the amount of memory for string storage. An "Out of string space" error occurs only if there is no free memory left for use by BASIC-80.

Examples:

CLEAR

Sets all numeric variables to zero and all strings to null.

CLEAR ,32768

Sets 32768 as the highest memory location for use by BASIC-80.

CLEAR ,2000

Allocates 2000 bytes for stack space.

CLEAR,32768,2000

Sets 32768 as the highest memory location for use by BASIC-80 and allocates 2000 bytes for stack space.

**CONT (continue program execution)**

Form:           **CONT**

The CONTinue statement is used to resume execution of a program after a CTRL-C has been typed, or a STOP or END statement has been executed. The CONTinue statement can also be used to resume execution after an error.

Execution will resume at the line after the break. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (?) or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, variable values may be examined and changed using Command Mode statements. Execution may be resumed with CONT or a Command Mode GOTO, which resumes execution at a specified line number.

CONT is invalid if the program has been edited during the break. CONT is also invalid if any changes were made to the program during the break. If any changes are made to the program during the break, the error message "Can't continue" will appear on your screen.

**DELETE (delete program lines)**

Form:           **DELETE**Δ<line number>-<line number>

The DELETE statement is used to delete program lines from memory.

BASIC-80 will always return to Command Mode after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

Examples:

**DELETE 40**            deletes line 40

**DELETE 40-100**      deletes lines 40-100, inclusive

**DELETE -40**          deletes all lines up to and including line 40

### EDIT (enter Edit Mode)

Form: EDITΔ<line number>

The EDIT statement will enter the Edit Mode at the specified line number.

In Edit Mode, it is possible to edit portions of a line without retyping the entire line. Upon entering Edit Mode, BASIC-80 types the line number of the line to be edited. Then it types a space and waits for an Edit Mode subcommand.

The Edit Mode subcommands may be categorized according to the following functions:

1. Moving the cursor.
2. Inserting text.
3. Deleting text.
4. Finding text.
5. Replacing text.
6. Ending and restarting Edit Mode.

The Edit Mode subcommands are not displayed on the terminal device. Some of the Edit Mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When a preceding integer is not specified, it is assumed to be one.

The Edit Mode subcommands are explained in Chapter Nine, "Editing."

**FILES (list names of files)**

Form:            FILES "<filename>"

The FILES command is used to list the names of files residing on the disk.

"<filename>" must follow the normal CP/M naming conventions. If <filename> is omitted, all the files on the current default drive will be listed. "<filename>" is a string which may contain question marks (?) to match any character in the file name or extension. An asterisk (\*) can be used to match any file name or extension.

Examples:

FILES            list all file names on current default disk

FILES "\* .BAS"    list all file names with extension .BAS

FILES "B: \*.\*"    list all file names on drive B:

Note that, in the last example, the drive specification is given in upper case. All references to disk drives from within MBASIC must be given in upper case. Specifying a drive name in lower case will generate a "Bad File Name" error.

**LIST (list program on terminal)**

Form: LIST $\Delta$ <line number>-<line number>

The LIST command is used to list all or part of the program currently in memory. The listing will be displayed on the terminal device.

BASIC-80 will always return to Command Mode after a LIST is executed.

If the line numbers are omitted, the entire program is listed beginning at the lowest line number. The listing is terminated by either typing CTRL-C or by reaching the end of the program.

If one line number is specified, then only this line will be displayed on the terminal device.

Examples:

LIST	List the entire program.
LIST 500	List line number 500.
LIST 150-	List all lines from 150 to the end of the program.
LIST -100	List all lines from the lowest number through 100.
LIST 150-400	List lines 150 through 400, inclusive.

**LLIST (list program on line printer)**

Form: LLIST $\Delta$ <line number>-<line number>

The LLIST command will list all or part of the program currently in memory. The listing will be printed on the line printer. The options for LLIST are the same as LIST. BASIC-80 will always return to the Command Mode after an LLIST is executed.

LLIST will assume a 132-character wide printer.

Examples: See the examples for LIST

**LOAD (load program file from disk)**

Form:            LOAD "<filename>",R

The LOAD command is used to load a file from the disk into memory.

"<filename>" is the CP/M file name associated with the program file. The default extension .BAS will be supplied.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program.

The R option can be used to RUN the program after it has been LOADED. If the R option is used, all open files will be left open.

The R option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using temporary disk data files.

Example:

```
LOAD "STARTRK",R  
LOAD "B:GAME1.BAS"
```

NOTE: BASIC-80 will not map a file name to upper case. Thus, all of the statements which specify a CP/M file name should have the file name expressed in upper case letters. If a lower case file name is created in the directory, it can then only be accessed with BASIC-80.

### MERGE (merge program)

Form: MERGE "<filename>"

The MERGE command will merge a disk program file into the program currently in memory.

"<filename>" is the CP/M file name associated with the disk program file. The default file name extension .BAS will be supplied. The file must have been saved in ASCII format. If the file is not in ASCII format, a "Bad file mode" error occurs.

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on the disk will replace the corresponding lines in memory. Merging may be thought of as "inserting" the program lines on the disk into the program in memory.

BASIC-80 will always return to the Command Mode after executing a MERGE command.

Examples:

MERGE "PROG1" Insert PROGR1.BAS

MERGE "B:TEST.BAS" Insert B:TEST.BAS

### NEW (delete current program)

Form: NEW

The NEW command is used to delete the program currently in memory and clear all variables. After a NEW command has been executed, all numeric variables are set to zero and all string variables to null.

BASIC-80 will always return to Command Mode after a NEW is executed.

**RENUM (renumber program lines)**

Form:           RENUMΔ<new number>,<old number>,<increment>

The RENUM command will renumber program lines.

<new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default increment is 10.

The RENUM command will also change all line number references following GOTO, THEN, ON/GOTO, ON/GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyy may be changed.

RENUM can not be used to change the order of program lines or to create line numbers greater than 65529. In these cases, an "Illegal function call" error will result.

Examples:

RENUM

Renumber the entire program. The first new line number will be 10. The line numbers will be incremented by 10.

RENUM 300,,50

Renumber the entire program. The first new line number will be 300. Lines will increment by 50.

RENUM 1000,900,20

Renumber the lines from 900 up so they start with line number 1000 and increment by 20.

**RESET (change diskette)**

Form:           RESET

The RESET command enables you to exchange a new disk for the disk in the current default drive. RESET cannot be used with a drive name argument. Any attempt to supply a drive name argument will generate a "Syntax error".

The RESET command should be issued only after you replace the old default disk with the new default disk. If you issue a RESET command before switching disks, BASIC-80 will read the directory information off of the old disk.

The only effect of the RESET command is to read the directory information off of the new disk and into memory. RESET does not close open files.

Example:

RESET

**RUN (execute program)**

Form 1: RUNΔ<line number>

Form 1 of the RUN command is used to execute a program currently in memory.

If <line number> is specified, execution begins on that line. A RUN command without the <line number> will start execution at the lowest line number. BASIC-80 will always return to Command Mode after a RUN is executed.

Example:

RUN 10              Executes the program currently in memory.  
                        Execution starts at line number 10.

RUN              Executes the program currently in memory.  
                        Execution starts at the lowest numbered line.

Form 2: RUN "<filename>",R

Form 2 of the RUN command is used to load a BASIC-80 program from disk into memory and run it. The R is optional and if used will leave all data files open.

"<filename>" is the name of the file on the disk. The default extension is .BAS.  
<filename> must be a valid CP/M file name enclosed in quotation marks.

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the R option, all data files will remain open.

Example:

RUN "PROG1"              Loads and executes PROG1.BAS

RUN "B:GAME",R              Loads and executes B:GAME.BAS leaving all data files open.

### SAVE (write program to disk)

Form:      SAVE "<filename>",A

              SAVE "<filename>",P

              SAVE "<filename>"

The SAVE command will write to a disk file the program that is currently in memory.

"<filename>" is a string enclosed in quotes that conforms to the CP/M requirements for file name construction. The default extension .BAS is supplied. If <filename> already exists, the file will be written over.

The A option will save the file in ASCII format. Otherwise, BASIC-80 will assume the compressed binary format. ASCII format takes more space on the disk, but some disk commands require that the files be in ASCII format. For example, the MERGE command requires an ASCII format file.

The P option will protect the file by saving it in an encoded binary format. When a protected file is later RUN or (LOADed), any attempt to list or edit it will fail.

#### Examples:

SAVE"COM2",A

SAVE"PROG",P

### SYSTEM (perform CP/M warm start)

Form:      SYSTEM

The SYSTEM command will close all files and then perform a CP/M warm start. Because CTRL-C will always return to BASIC-80 Command Mode, the SYSTEM command must be used to return to CP/M.

#### Example:

```
SYSTEM
A> [prompt from CP/M]
      (assuming A: is the current default disk)
```



## Chapter Four

# Program Statements

## OVERVIEW

The program statements available to the BASIC-80 programmer can be divided into four functional groups: Data type definition, Assignment and allocation, Control, and I/O (Non-disk). This Chapter will explain the various program statements in these four groups.

Note: These program statements can also be used as Command Mode statements.

## DATA TYPE DEFINITION

A DEF statement declares that the variable name beginning with a certain range of letters is of the specified data type. However, a type declaration character always takes precedence over a DEF statement.

If no data type declaration statements are encountered, BASIC-80 assumes all variables without declaration characters are single precision variables.

### **DEFINT (declare variable as integer)**

Form:            **DEFINT $\Delta$ <letter range>**

The DEFINT statement is used to declare a range of variable names as integer data types.

An integer data type will take up less memory than a single-precision or double-precision data type. However, a variable declared as an integer data type can only be assigned values in the range -32768 and +32767 inclusive.

Example:

**DEFINT I-N**       All variables beginning with the letters I,J,K,L,M,N will be integer variables.

### **DEFSNG (declare variable as single-precision)**

Form:            **DEFSNG $\Delta$ <letter range>**

The DEFSNG statement is used to declare a range of variable names as single-precision data types.

Single-precision variables are stored with seven digits of precision and they are printed with six digits of precision.

Example:

**DEFSNG A-D**       All variables beginning with the letters A,B,C, and D will be single-precision variables.

### DEFDBL (declare variable as double-precision)

Form: DEFDBL△<letter range>

The DEFDBL statement is used to declare a range of variable names as double-precision data types.

Double-precision variables are stored with 17 digits of precision and they are printed with 16 digits of precision.

Examples:

DEFDBL X-Z, A      All variables beginning with the letters X, Y, Z and A will be double precision variables.

### DEFSTR (declare variable as string)

Form: DEFSTR△<letter range>

The DEFSTR statement is used to declare a range of variable names as string data types.

A string is a sequence of characters — letters, blanks, numbers, and special characters — up to 255 characters long.

Example:

DEFSTR S      All variables beginning with the letter S will be string variables.

## ASSIGNMENT AND ALLOCATION STATEMENTS

### DIM (set-up array)

Form:            DIM <list of subscripted variables>

The DIMension statement is used to set up the maximum values for array variable subscripts and allocate storage accordingly.

If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a “Subscript out of range” error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement.

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example:

```
10 DIM A(20)
20 FOR I = 0 TO 20
30 A(I) = I+1
40 NEXT I
```

### OPTION BASE (set minimum value for array subscript)

Form:            OPTION<sub>n</sub>BASE<sub>n</sub>

The OPTION BASE statement is used to declare the minimum value for array subscripts. The default base is 0. This may be changed to 1. The OPTION BASE statement must be executed before the DIM statement is executed. If an OPTION BASE statement appears after an array has been DIMensioned, a “Duplicate definition” error will result.

Example:

```
OPTION BASE 1
```

For more information on array storage allocation, see Chapter Six, “Arrays.”

### ERASE (remove array from program)

Form: ERASE $\Delta$ <list of array names>

The ERASE statement is used to remove an array from a program. Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes.

If an attempt is made to redimension an array without first ERASEing it, a "Duplicate Definition" error occurs. If an attempt is made to ERASE an array that has not been defined in a DIM statement, an "Illegal function call" error will result.

Example:

```
10 DIM A(40)
20 ERASE A
30 DIM A(50)
```

### LET (assign value to a variable)

Form: LET $\Delta$ <variable> = <expression>

The LET statement is used to assign the value of an expression to a variable.

Note that the word LET is optional, as the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
10 LET D = 12
20 LET SUM = X + Y + Z
```

or

```
10 D = 12
20 SUM = X + Y + Z
```

**REM (insert remark)**

Form:            REM <remark>

The REM statement allows explanatory remarks to be inserted in a program.

REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

Remarks may also be added to a line by preceding the remark with a single quotation mark.

Example:

```
10 REM THIS IS A REMARK
20 ' THIS IS ALSO A REMARK
```

**SWAP (exchange variable values)**

Form:            SWAP△<variable>,<variable>

The SWAP statement is used to exchange the values of two variables.

Any type variable may be swapped (integer,single-precision, double-precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example:

```
10 A$=" ONE ":B$="FOR":C$="ALL"
20 PRINT A$:B$:C$
30 SWAP A$,C$
40 PRINT A$:B$:C$
RUN
ONE FOR ALL
ALL FOR ONE
Ok
```

## CONTROL STATEMENTS

Two types of control statements are available to the BASIC-80 programmer. One type affects the sequence of execution, and the other type is used for conditional execution.

### Sequence of Execution

The sequence of execution statements are used to alter the sequence in which the lines of a program are executed. Normally, execution begins with the lowest numbered line and continues, sequentially, until the highest numbered line is reached.

The sequence of execution statements allow the BASIC-80 programmer to execute the lines in any sequence the program logic dictates.

#### END (terminate program execution)

Form:            END

The END statement will terminate program execution, close all files, and return to Command Mode.

END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be displayed. An END statement at the end of a program is optional. BASIC-80 will always return to Command Mode after an END is executed.

Example:

```
520 IF K>1000 THEN END
```

**FOR/NEXT (repetitive execution loop)**

Form:      FOR <variable> = X TO Y [STEP Z].

.

.

NEXT [<variable>]

where X,Y and Z are constants, variables, or numeric expressions.

The FOR/NEXT statement will allow a series of instructions to be performed in a loop a given number of times.

<variable> is used as the loop counter. The first numeric expression (X) is the initial value of the counter. The second numeric expression (Y) is the terminal value of the counter. The third numeric expression (Z) is the incremental value for the loop counter.

Before the FOR/NEXT loop is executed, these three numeric values are evaluated. First, the terminal value is evaluated. Then the initial value is evaluated. The loop counter is then set equal to the initial value.

Any attempt to change these three values during the execution of the loop will have no effect. However, the loop counter must not be changed or the loop will not operate as expected.

After the numeric values are evaluated, a check is performed to see if the initial value of the loop exceeds the terminal value. If the initial value of the loop exceeds the terminal value, the loop will not be executed. (If the STEP value is negative, the initial value must be greater than the terminal value or the loop will not be executed.)

The program lines following the FOR are executed until the NEXT statement is encountered. Then the loop counter is incremented by the amount specified by STEP. A check is performed to see if the value of the loop counter is now greater than the terminal value.

If it is not greater, BASIC-80 branches back to the statement after the FOR statement and the process is repeated. If the value of the loop counter is greater than the terminal value, execution continues with the statement following the NEXT statement. The statements between the FOR and the NEXT statements constitute the range of the FOR/NEXT loop.

If STEP is not specified, the incremental value is assumed to be one. If STEP is a negative value, the loop counter is decremented each time through the loop. The loop is executed until the loop counter is less than the final value.

**Examples:**

---

```
10 FOR J = 5 TO 1 STEP -1
20 PRINT J;
30 NEXT J
RUN
5 4 3 2 1
Ok
```

The statement in the range of this loop will be executed five times. In this example, 5 is the initial value, 1 is the terminal value, and  $-1$  is the incremental value. Note that the initial value is greater than the terminal value. This is valid because the incremental value is negative. Also note that the variable J could have been omitted from the NEXT statement in line 30.

---

```
10 FOR J = 5 TO 1
20 PRINT J;
30 NEXT J
RUN
Ok
```

In this example, the statement in the range of the loop will not be executed because the initial value is greater than the terminal value. The STEP value has been omitted, so it is assumed to be 1.

---

```
10 I = 5
20 FOR I = 1 TO I+5
30 PRINT I;
40 NEXT
RUN
1 2 3 4 5 6 7 8 9 10
Ok
```

In this example, the loop executes 10 times. The terminal value for the loop is evaluated first. The terminal value ( $I+5$ ) is 10. Next, the initial value is evaluated. The initial value is 1. The loop counter is then set equal to the initial value. Because the STEP value has been omitted, the incremental value is assumed to be 1.

---

### Nested Loops

FOR/NEXT loops may be nested. That is, a FOR/NEXT loop may be placed within the range of another FOR/NEXT loop.

When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before the NEXT for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable in a NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

#### Valid Nesting

```
FOR J = 1 TO 10
  FOR I = 1 TO 5
    NEXT I
  NEXT J
```

#### Invalid Nesting

```
FOR J = 1 TO 10
  FOR I = 1 TO 5
    NEXT J
  NEXT I
```

Note that with the valid nesting, the range of the inner loop is completely contained within the range of the outer loop.

### GOSUB/RETURN (branch to subroutine)

Form: GOSUB <line number>

·  
·  
·  
RETURN

The GOSUB/RETURN statement is used to branch to and return from a subroutine.

<line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement in a subroutine causes BASIC-80 to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement.

Subroutines may appear anywhere in the program, but it is good programming practice to separate the subroutine from the main program. To prevent inadvertent entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

Example:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
35 REM
40 REM THIS IS THE SUBROUTINE
45 REM
50 PRINT "SUBROUTINE";
60 PRINT " IN ";
70 PRINT "PROGRESS"
80 RETURN
RUN
SUBROUTINE IN PROGRESS.
BACK FROM SUBROUTINE
0k
```

**GOTO (unconditional branch)**

Form:            GOTO <line number>

The GOTO statement will branch unconditionally out of the normal program sequence and continue execution at the specified line number.

If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

If <line number> has not been previously defined, an "Undefined line number" error will be displayed.

Example:

```
10 GOTO 30
20 PRINT "LINE 20"
30 PRINT "LINE 30"
40 END
RUN
LINE 30
Ok
```

### ON/GOTO and ON/GOSUB (evaluate and branch)

Forms:      **ON <expression> GOTO <list of line numbers>**

**ON <expression> GOSUB <list of line numbers>**

The ON/GOTO and the ON/GOSUB statements are used to branch to one of several specified line numbers, depending on the value returned when an expression is evaluated. The result of evaluating <expression> must be positive and less than 255. If the value of <expression> is non-integer, the fractional portion is rounded.

The value of <expression> determines which line number in the list will be used for branching. For example, if the value of the expression is three, the third line number in the list will be the destination of the branch.

If the value of <expression> is zero or greater than the number of line numbers in the list, BASIC-80 will continue with the next executable statement. If the value is negative or greater than 255, an "Illegal function call" error occurs.

In the ON/GOSUB statement, each line number in the list must be the first line number of a subroutine.

#### Example:

```
10 L=4
20 ON L GOTO 50,60,70,80
30 END
50 PRINT "LINE 50":GOTO 90
60 PRINT "LINE 60":GOTO 90
70 PRINT "LINE 70":GOTO 90
80 PRINT "LINE 80":GOTO 90
90 STOP
RUN
LINE 80
Ok
```

In this example, L=4, thus causing a branch to the fourth line number in the list. The fourth line number in the list is 80. If L > 4 or if L = 0, then the program would have branched to line number 30.

**STOP (suspend execution)**

Form: STOP

The STOP statement is used to terminate program execution and return BASIC-80 Command Mode.

STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

Break in line nnnnn

Unlike the END statement, the STOP statement does not close any files.

BASIC-80 will always return to the Command Mode after a STOP is executed. Execution can be resumed by issuing a CONT command.

Example:

```
10 PRINT "LINE 10"
20 STOP
30 PRINT "LINE 30"
40 END
RUN
LINE 10
BREAK IN 20
Ok
CONT
LINE 30
Ok
```

## Conditional Execution

The conditional execution statements are used to optionally execute a statement or series of statements. The statement or series of statements will be executed if a certain condition is met.

## IF/THEN/ELSE (conditional execution)

Form:

IF <expression> THEN <statement(s)> ELSE <statement(s)>

IF <expression> GOTO <line number> ELSE <statement(s)>

The IF/THEN/ELSE statement is used to make a decision regarding program flow based on the result returned by an expression.

If the result of <expression> is true (i.e. not zero), the THEN clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. If multiple statements are to be executed, they must be separated by colons (:).

If the result of <expression> is false (i.e. zero), the THEN clause is ignored and the ELSE clause, if present, is executed. ELSE may be followed by either a line number for branching or one or more statements to be executed. If multiple statements are to be executed, they must be separated by colons (:).

The keyword THEN can optionally be replaced with a GOTO statement. In this case, if the result of the expression is true, the program will branch to the statement number specified in the GOTO statement.

Examples:

IF I THEN PRINT "I IS NOT ZERO" ELSE PRINT "I IS ZERO"

This statement will print "I IS NOT ZERO" if the value of I is not zero. If the value of I is zero, the message "I IS ZERO" will be printed.

IF X=A GOTO 100 ELSE PRINT "NOT EQUAL"

This statement will branch to line number 100 if X = A. If X is not equal to A, the message "NOT EQUAL" will be printed.

IF IOFLAG THEN PRINT A\$ ELSE LPRINT A\$

This statement causes printed output to go either to the terminal or the line printer depending upon the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer. If IOFLAG is not zero, output goes to the terminal.

### Additional Considerations

When an IF/THEN statement is followed by a line number in the Command Mode, an “Undefined line number” error results unless a statement with the specified line number had previously been entered in the Indirect Mode.

When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exactly the same as the printed value. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test the single-precision variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-06 THEN . . .
```

This test returns TRUE if the value of A is 1.0 with a relative error of less than 1.0E-6.

### Nesting of IF Statements

IF/THEN/ELSE statements may be nested, but make sure that the same number of IF's and ELSE's are used. Each ELSE will be matched with the closest unmatched THEN. In the following example, the operator was able to include the ELSE statements in line 20 by using line feeds.

Example:

```
10 INPUT A
20 IF A=C THEN IF A=B THEN PRINT "A=B A=C"
<operator-typed LINE FEED>
    ELSE PRINT "A NOT = B"
<operator-typed LINE FEED >
    ELSE PRINT "A NOT = C"
30 PRINT A
```

This nested IF will first test to see if A=C. If A does not equal C, the second ELSE will be executed. If A does not equal C, the message “A NOT = C” will be printed and execution will be continued at line 30.

If A=C, the first THEN will be executed. This will result in another test. This time, A will be compared to B. If A does not equal B, the first ELSE will be executed. So, if A does not equal B, the message “A NOT = B” will be printed and execution will continue with line 30.

If A=B, the second THEN will be executed, resulting in the message “A=B A=C” being printed on the terminal. After printing this message, execution will be continued at line 30.

### WHILE/WEND (conditional execution)

Form:

```
 WHILE <expression>
       .<loop statements>
       .
       WEND
```

The WHILE...WEND statement is used to execute a series of statements in a loop as long as a given condition is true.

If <expression> is not zero (i.e.,true), <loop statements> are executed until the WEND statement is encountered. BASIC-80 then returns to the WHILE statement and checks <expression>. If it is still not zero (true), the process is repeated. If the value of the expression is zero (false), execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example:

```
10 I = 1
20 WHILE I
30 PRINT "WHILE/WEND LOOP"
40 I = 0
50 WEND
60 END
RUN
WHILE/WEND LOOP
Ok
```

## I/O Statements (Non-Disk)

### DATA (store constants)

Form: DATA <list of constants>

The DATA statement is used to store numeric and string constants. These constants are assigned to variables by using the READ statement.

DATA statements are nonexecutable and they may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a logical line. Any number of DATA statements may be used in a program.

The READ statement will access the DATA statement in line number sequence and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.)

String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement.

Example:

```
10 DATA 12.3, HELLO, "GOOD,BYE", 34
20 DATA 1,2,3,4,5
```

### INPUT (input from terminal)

Form:      INPUT [ <"prompt string"> ]; <list of variables>

The INPUT statement is used to input data from the terminal during program execution.

When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data.

If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal. (The question mark can be suppressed by putting a comma instead of a semicolon between the prompt string and the list of variables.)

If the keyword INPUT is immediately followed by a semicolon, then the carriage return typed by the user does not echo a carriage return/line feed sequence.

The data that is entered is assigned to the variable(s) given in the variable list. The number of data items supplied must be the same as the number of variables in the list. The data items input must be separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. Strings input to an INPUT statement need not be surrounded by quotation marks.

Responding to INPUT with too many or too few items, or with the wrong type of data (numeric instead of string, etc.) causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

NOTE: Previous versions of Microsoft BASIC handled illegal INPUT in a somewhat different manner.

Example:

```
10 INPUT"ENTER VALUE";X
20 PRINT X
30 END
RUN
ENTER VALUE? [you type] 5
5
0k
```

**LINE INPUT (input entire line)**

Form:            LINE INPUT [ <;> <"prompt string">; ] <string variable>

The LINE INPUT statement is used to input an entire line (up to 255 characters) to a string variable, without the use of delimiters.

The <"prompt string"> is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt string to the carriage return is assigned to <string variable>.

If the key words LINE INPUT are immediately followed by a semicolon, then the RETURN typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT may be escaped by typing CTRL-C. BASIC-80 will return to the Command Mode and type "Ok". A CONT command will resume execution at the LINE INPUT.

Example:

```
10 LINE INPUT"NAME?--";J$  
20 PRINT J$  
30 STOP  
RUN  
NAME?--[you type] JONES,JACK L.  
JONES,JACK L.  
Ok
```

### LPRINT (output data to line printer)

Form: LPRINT <list of expressions>

The LPRINT statement is used to print data on the line printer.

The LPRINT statement is the same as the PRINT statement, except output goes to the line printer.

LPRINT defaults to a 132-character wide printer.

### PRINT (output data at terminal)

Form: PRINT <list of expressions>

The PRINT statement is used to output data to the terminal. (A question mark may be used in place of the keyword PRINT in a PRINT statement.)

If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. String constants must be enclosed in quotation marks.

### Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC-80 divides the line into print zones of 14 spaces each.

In the list of expressions, a comma (,) causes the next value to be printed at the beginning of the next zone. A semicolon (;) causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is inserted at the end of the line. If the printed line is longer than the terminal width, BASIC-80 goes to the next physical line and continues printing.

Printed numeric values are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign.

Single-precision numbers that can be accurately represented with 6 or fewer digits in the unscaled format are output using the unscaled format. For example,  $10^{-6}$  is output as .000001 and  $10^{-7}$  is output as 1E-7.

Double-precision numbers that can be accurately represented with 16 or fewer digits in the unscaled format are output using the unscaled format. For example, 1D-16 is output as .0000000000000001 and 1D-17 is output as 1D-17.

---

**Examples:**

```
10 X=5
20 PRINT X+5,X-5,X*(-5),X^5
30 END
RUN
      10          0          -25          3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

---

```
10 FOR X = 1 TO 5
20 J = J +5
30 K=K+10
40 ?J;K;
50 NEXT X
RUN
      5  10  10  20  15  30  20  40  25  50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

---

### READ (read values from DATA statement)

Form:      READ <list of variables>

The READ statement is used to read values from a DATA statement and assign them to variables.

A READ statement must always be used in conjunction with a DATA statement. READ statements assign the constant values contained in a DATA statement to the variables contained in the READ statement.

The assignment of values is on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If data types do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement.

If the number of variables in <list of variables> exceeds the number of data constants in the DATA statement, an "Out of data" error will result.

If the number of variables specified is fewer than the number of elements in the DATA statement, subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement.

Example:

```
10 FOR I = 1 TO 10
20 READ A(I)
30 NEXT I
40 DATA 3,4,5,6,7,8,9,10,11,12
```

This program segment READs the values from the DATA statement into the array A. After execution, the value of A(1) will be 3, and so on.

**RESTORE (reset data pointer)**

Form: RESTORE <line number>

The RESTORE statement is used to reset the data pointer in a DATA statement so that the data may be reread.

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement will access the first item in the specified DATA statement.

Example:

```
10 READ A,B,C  
20 RESTORE  
30 READ D,E,F  
40 DATA 57,68,79
```

This program segment will assign the constants 57,68,79 to the variables A,B,C. The RESTORE statement in line 200 will reset the DATA pointer so that the READ statement in line 30 will assign the constants 57,68,79 to the variables D,E,F.

## WRITE (output data to terminal)

Form:            WRITE <list of expressions>

The WRITE statement is used to output data to the terminal.

If <list of expressions> is omitted, a blank line will be output. If <list of expressions> is included, the values of the expressions are output to the terminal. The expressions in the list may be numeric and/or string expressions, and they must be separated by commas.

When the printed items are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, BASIC-80 will insert a carriage return/line feed.

The WRITE statement outputs numeric values using the same format as the PRINT statement.

Example:

```
10 A=80:B=90:C$="BASIC-80"
20 WRITE A,B,C$
RUN
 80,90,"BASIC-80"
0k
```



## Chapter Five

# Strings

## OVERVIEW

A string is a sequence of characters — letters, blanks, numbers, and special characters — up to 255 characters long. A string constant is constructed by enclosing these characters in a set of double quotation marks. A string variable can be declared by simply adding the string declaration character, \$ , to the variable name. A variable can also declare a variable a string variable by using the DEFSTR statement.

Microsoft BASIC-80 provides complete facilities for manipulating strings. A string can be compared, PRINTed, concatenated with other strings , etc. Several functions for manipulating strings are also available to the BASIC-80 programmer.

This Chapter will cover the following subjects:

“String Input/Output”

“String Operations”

“String Functions”

## STRING INPUT/OUTPUT

String constants can be input to a program in the same manner as numeric constants. The INPUT statement can be used. The string can be usually typed without quotes.

```
10 INPUT "YOUR NAME";J$  
20 PRINT "HELLO ";J$  
RUN  
YOUR NAME? [you type] JOHN  
HELLO JOHN  
Ok
```

However, if you wish to input a string constant which contains commas, colons, or leading or trailing blanks, the string must be enclosed in quotes. (When the INPUT statement is used.)

```
10 INPUT "YOUR NAME";J$  
20 PRINT J$  
RUN  
YOUR NAME? [you type] "JONES, JOHN"  
JONES, JOHN  
Ok
```

The LINE INPUT statement can be used to input strings containing commas, colons, and leading or trailing blanks. The string does not have to be enclosed in quotes with the LINE INPUT statement.

```
10 LINE INPUT "YOUR NAME";J$  
20 PRINT J$  
RUN  
YOUR NAME [you type] JONES, JOHN  
JONES, JOHN  
Ok
```

## STRING OPERATIONS

Strings may be concatenated using the + . For example:

```
10 X$="FIRST"
20 Y$=" AND "
30 Z$="LAST"
40 PRINT X$+Y$+Z$
RUN
FIRST AND LAST
Ok
```

Strings may be compared using the same relational operators that are used with numbers:

=      <>      <      >      <=      >=

The strings are compared character-for-character from left to right. The ASCII codes for the character are compared, and the character with the lower ASCII value is considered to precede the other character.

For example, the string "Z\$" precedes the string "Z\*" because "\$" (ASCII code - decimal 36) has a lower value than does "\*" (ASCII code - decimal 42).

When strings of different lengths are compared, the shorter string is considered to precede the longer string. Every character, including blanks and any non-printing character is significant in a string comparision. For example, the string "AB" will precede the string "AB " because of the trailing blank in the string "AB".

A string constant must also be enclosed in double quotes whenever it is used in an assignment statement or in a comparison expression.

Example:

```
Z$="STRING CONSTANT"
IF Z$="NUMERIC CONSTANT" THENV GH.N Z$
```

## STRING FUNCTIONS

The string functions available to the BASIC-80 programmer are:

<u>Function</u>	<u>Definition</u>
ASC(X\$)	string to ASCII value conversion
CHR\$(I)	ASCII value to string conversion
HEX\$(X)	decimal to hexadecimal conversion
INKEY\$	read one character from terminal
INPUT\$(X,Y)	read characters
INSTR(I,X\$,Y\$)	search for substring
LEFT\$(X\$,I)	return leftmost characters
LEN(X\$)	length of string
MID\$(X\$,I,J)	return substring
MID\$(X\$,I,J)=Y\$	replace portion of string
OCT\$(X)	convert decimal to octal
RIGHT\$(X\$,I)	return rightmost characters
SPACE\$(X)	return string of spaces
STR\$(X)	return string representation
STRING\$(I,J)	build string
STRING\$(I,X\$)	
VAL(X\$)	return numerical representation of the string

Table 5-1  
String Functions

### ASC (convert string to ASCII value)

Form:      ASC(X\$)

The ASC function will return a numerical value that is the ASCII decimal code of the first character of the string X\$. If X\$ is a null string, an “Illegal function call” error is returned.

Example:

```
10 X$="TEST"
20 PRINT ASC(X$)
RUN
84
Ok
```

In the above example, the first letter of the string X\$ is a T. The ASCII code for T is 84.

### CHR\$ (convert ASCII value to string)

Form:      CHR\$(I)

The CHR\$ function will return a string whose one element has ASCII decimal code I. (ASCII codes are listed in “Appendix B.”) CHR\$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent; PRINT CHR\$(7).

Example:

```
PRINT CHR$(66)
B
Ok
```

**HEX\$ (convert decimal to hexadecimal)**

Form:        **HEX\$(X)**

The HEX\$ function will return a string which represents the hexadecimal value of the decimal argument. X is rounded to an integer before HEX\$(X) is evaluated.

Example:

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X;" DECIMAL IS ";A$;" HEXADECIMAL"
RUN
? 32
32 DECIMAL IS 20 HEXADECIMAL
```

**INKEY\$ (read one character from keyboard)**

Form:        **INKEY\$**

The INKEY\$ function will return either a one-character string containing a character read from the terminal or a null string if no character is pending at the terminal. No character is echoed and all characters are passed through the program except for CTRL-C which terminates the program and returns BASIC-80 to the Command Mode.

Example:

```
10 X$ = INKEY$
20 IF X$=CHR$(32) THEN STOP
30 GO TO 10
```

This example would read from the keyboard until a space (ASCII decimal-32) was typed.

### INPUT\$ (read characters)

Form: INPUT\$(X,Y)

The INPUT\$ function will return a string of X characters, read from the terminal or from file number Y. If the terminal is used for input, no characters will be echoed and all control characters are passed through except CTRL-C, which is used to interrupt the execution of the INPUT\$ function.

Example:

```
10 OPEN "I",1,"DATA.DAT"
20 IF EOF(1) THEN 50
30 PRINT INPUT$(1, 1)
40 GOTO 20
50 END
```

The above example will print all the characters in the file DATA.DAT

```
10 X$=INPUT$(1)
20 IF X$="P" THEN 500
30 IF X$="S" THEN 700 ELSE 10
```

This example would read one character from the keyboard. If the character is a P, program control would be transferred to line number 500. If the character is an S, control would be transferred to line number 700. If the character is not an S or P, control would be transferred back to line number 10.

**INSTR (search for substring)**

Form: INSTR(I,X\$,Y\$)

The INSTR function will search for the first occurrence of string Y\$ in X\$ and return the position at which the match is found. Optionally, the offset I sets the position for starting the search. I must be in the range 1-255. If I>LEN(X\$) or if X\$ is null or if Y\$ can not be found, INSTR will return 0. If Y\$ is null, INSTR returns I or 1.

X\$ and Y\$ may be string variables, string expressions or string literals.

Example:

```
10 X$ = "ABCDEB"  
20 Y$ = "B"  
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)  
RUN  
2 6  
Ok
```

**LEFT\$ (return leftmost characters)**

Form: LEFT\$(X\$,I)

The LEFT\$ function will return a string comprised of the leftmost characters of X\$. I must be in the range 0 to 255. If I is greater than the length of X\$, the entire string (X\$) will be returned. If I equals 0, the null string (length zero) is returned.

Example:

```
10 A$ = "BASIC-80"  
20 B$ = LEFT$(A$,5)  
30 PRINT B$  
RUN  
BASIC  
Ok
```

### LEN (return length of a string)

Form: LEN(X\$)

The LEN function will return the number of characters in X\$. Non-printing characters and blanks are counted.

Example:

```
10 X$ = "ABC DEF"  
20 PRINT LEN(X$)  
RUN  
7  
Ok
```

### MID\$ (return substring)

Form: MID\$(X\$,I,J)

The MID\$ function will return a string of length J characters from X\$ beginning with the Ith character. I and J must be in the range 0 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all right-most characters beginning with the Ith character are returned. If I is greater than the length of string X\$, MID\$ will return a null string.

Example:

```
10 A$="GOOD"  
20 B$="MORNING EVENING AFTERNOON"  
30 PRINT A$;MID$(B$,8,8)  
RUN  
GOOD EVENING  
Ok
```

**MID\$ (replace portion of string)**

Form:      **MID\$(X\$,I,J)=Y\$**

This form of the MID\$ function will replace a portion of one string with another string.

The characters in string X\$, beginning at position I, are replaced by the characters in string Y\$. The value, which is optional, refers to the number of characters from string Y\$ that will be used in the replacement.

However, regardless of whether J is omitted or included, the replacement of characters never goes beyond the original length of X\$.

Examples:

A\$="1234567" at the beginning of each example

<u>Statement</u>	<u>Resultant A\$</u>
MID\$(A\$,3,4)="ABCDE"	12ABCDE7
MID\$(A\$,5)="ABCDE"	1234ABC
MID\$(A\$,1,2)="A"	A234567

**OCT\$ (convert decimal to octal)**

Form:      **OCT\$(X)**

The OCT\$ function will return a string which represents the octal value of the decimal argument. X is rounded to an integer before OCT\$(X) is evaluated.

Example:

```
PRINT OCT$(24)
30
Ok
```

**RIGHT\$ (return rightmost characters)**

Form:      **RIGHT\$(X\$,I)**

The RIGHT\$ function will return the right-most I characters of string X\$. If I equals the length of the string X\$, the function will return the entire string. If I equals 0, the null string (length zero) will be returned.

Example:

```
10 A$="DISK BASIC-80"
20 PRINT RIGHT$(A$,8)
RUN
BASIC-80
Ok
```

**SPACE\$ (return string of spaces)**

Form:      **SPACE\$(X)**

The SPACE\$ function will return a string of spaces of length X. The expression X is rounded to an integer and must be in the range 0-255.

Example:

```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
1
2
3
4
5
Ok
```

**STR\$ (return string representation)**

Form:           STR\$(X)

The STR\$ function will return the string representation of X. For example, if X = 45.3, then STR\$(X) equals the string " 45.3". A leading blank will be inserted before "45.3" to allow for the sign of X. Arithmetic operations may be performed on X, but not on the string STR\$(X).

Examples:

```
PRINT STR$(100)
      100
```

```
PRINT STR$(-100)
      -100
```

**STRING\$ (build string)**

Form:           STRING\$(I,J)

STRING\$(I,X\$)

The STRING\$ function will return a string of length I composed of the ASCII code J or the first character of X\$. I and J must be expressed in decimal and their values must be in the range 0-255.

Examples:

```
PRINT STRING$(10,"*")
*****
```

```
PRINT STRING$(15,65)
AAAAAAAAAAAAAAA
```

## VAL (return numerical representation)

Form:      **VAL(X\$)**

The VAL function will return the numerical representation of the string X\$. The VAL function will strip all leading blanks, tabs, and line feeds from the argument string.

If the first valid character of X\$ is not +, -, &, or a digit, then  $\text{VAL}(X\$) = 0$ . The & is used to specify an octal value. The VAL function will convert this octal value to decimal when  $\text{VAL}(X\$)$  is evaluated. If the string X\$ contains both numeric and alphanumeric characters, only the leading numeric characters will be used in evaluating X\$.

Examples:

```
PRINT VAL("100 FEET")
100
```

```
PRINT VAL("FEET 100")
0
```

```
PRINT VAL("&100")
64
```

```
PRINT VAL(" -3")
-3
```



## Chapter Six

# Arrays

### OVERVIEW

This Chapter explains the methods used to create and reference an array, which is simply an ordered list of data items. This list of data items can be a one-dimensional vertical array, or it can be a table of data items consisting of rows and columns.

These data items may be either string or numeric. Each one is referred to as an “element”. To help illustrate the concept of arrays, an example is included in this Chapter.

This Chapter also contains several sample routines which can be used to manipulate arrays. These sample routines can be used to add, multiply, transpose and perform other useful operations on numeric arrays.

## ARRAYS

### Array Declarator

Before an array is referenced, it should be "declared" by use of an array declarator. The DIM statement is used to establish the maximum number of elements in an array. The general form of the DIM statement is:

DIM <name>(<integer expression>)

where:

<name> is a valid BASIC-80 symbolic name

<integer expression> is any valid integer expression which when evaluated, will be rounded to a positive integer value. This positive integer value will then become the maximum number of elements associated with that specific array name. The maximum number of dimensions is 255. The maximum number of elements per dimension is 32767.

Examples:

```
DIM A(3),D$(2,2,2)
DIM Q1(R+T)
DIM Z#(100)
```

An array can also be declared without the use of the array declarator. When BASIC-80 encounters a subscripted variable that has not been defined with a DIM statement, it will assume a maximum subscript of 10. Thus, an array can be established without the use of the DIM statement.

## Array Subscript

Each element of an array can be uniquely referenced by having an array subscript appended to end of the array name. This array subscript is an integer expression which references a unique element of the array.

Examples:

```
A(1), D$(I,J,K)  
Q1(2)  
Z#(55)
```

Any attempt to reference an array element with a subscript that is negative will result in an “Illegal Function Call” error. References to subscripts which are larger than the maximum value established by a DIM statement and references which contain too many or too few subscripts will generate a “Subscript Out of Range” error.

## OPTION Base Statement

The minimum subscript for an array element is assumed to be 0. The array declarator A(10) actually establishes an 11-element array, A(0) - A(10). The OPTION BASE statement can be used to change this default minimum array subscript to 1. The following example illustrates the use of the OPTION BASE statement.

Example:

```
OPTION BASE 1  
DIM A(10)
```

This program segment will establish a 10 element array, A(1) - A(10). The OPTION BASE statement must appear before any DIM statement or before any subscripted variable is referenced. An attempt to use the OPTION BASE statement after an array has already been established will result in a “Duplicate Definition” error.

## Vertical Arrays

A vertical array is a 1-dimensional array. This type of array is established if the DIM statement is used, or by letting BASIC-80 establish the default array size. Assuming that the default array size of 11 elements has been established for the array A, BASIC-80 would allocate storage as follows:

<u>Array element</u>	<u>Subscripted variable</u>
Element #1	A(0)
Element #2	A(1)
Element #3	A(2)
Element #4	A(3)
Element #5	A(4)
Element #6	A(5)
Element #7	A(6)
Element #8	A(7)
Element #9	A(8)
Element #10	A(9)
Element #11	A(10)

**Table 6-1**  
Array Storage Allocation.

The variable A(9) would reference the tenth element of this vertical array. (Although, the OPTION BASE statement could be used to set the minimum subscript to 1, then A(9) would reference the ninth element of the array.)

## Multi-Dimensional Arrays

A multi-dimension array is declared in the same manner as a vertical array, except that both row and column size are declared. For example, to declare a  $3 \times 3$  array, the following sequence of statements could be used:

```
OPTION BASE 1  
DIM A(3,3)
```

After this program segment is executed, BASIC-80 would reserve nine storage locations for the array. (Note that the minimum subscript value was set to 1 with the OPTION BASE statement.)

Storage for the array would be allocated as follows:

	<u>Column</u>	<u>1</u>	<u>2</u>	<u>3</u>
Row	1	A(1,1)	A(1,2)	A(1,3)
	2	A(2,1)	A(2,2)	A(2,3)
	3	A(3,1)	A(3,2)	A(3,3)

**Table 6-2**  
Multi-Dimensional Array  
Storage Allocation.

When reading from left to right, note that the second array subscript varied most rapidly. This is because BASIC-80 allocates array storage such that the right-most subscript varies the fastest.

String arrays can also be established in the same manner as numeric arrays. A string array is declared when the DIM statement is used.

```
DIM A$(100)
```

This statement will establish a 101 element string array. To access an element of the array, append an array subscript to the end of the variable name.

```
A$(20)="A STRING ARRAY"
```

## MATRIX MANIPULATION

The following is a collection of subroutines which are very useful for manipulating a matrix. The subroutine line numbers may have to be changed to be compatible with your main program.

### Matrix Input Subroutines

```
5000 'SUBROUTINE NAME -- MATIN2
5010 'ENTRY I% = # OF ROWS, J% = # OF COLUMNS
5020 DIM MAT(I%,J%)
5030 FOR K% = 1 TO I%
5040 PRINT "INPUT ROW #";K%
5050     FOR L% = 1 TO J%
5060         INPUT MAT(K%,L%)
5070 NEXT L%,K%
5080 RETURN
```

The above subroutine will accept data from the terminal and assign this data to the 2-dimensional array named MAT. Upon entry into this subroutine, the integer variable I% must contain the number of rows in the matrix and J% must contain the number of columns.

```
5000 'SUBROUTINE NAME -- MATIN3
5010 'ENTRY           I% = SIZE OF DIMENSION #1
5020 '                 J% = SIZE OF DIMENSION #2
5030 '                 K% = SIZE OF DIMENSION #3
5040 DIM MAT(I%,J%,K%)
5050 FOR L% = 1 TO I%
5060     FOR M% = 1 TO J%
5070         FOR N% = 1 TO K%
5080 READ MAT(L%,M%,N%)
5090 NEXT N%,M%,L%
6000 RETURN
```

This subroutine is used to read data from a DATA statement and assign this data to the 3-dimensional array named MAT. Upon entry into this subroutine, the integer variable I% must contain the number of elements for dimension 1, J% must contain the number of elements for dimension 2, and K% must contain the number of elements for dimension 3. The data must also be contained in a valid DATA statement.

## Scalar Multiplication (multiplication by a single variable)

```
5000 'SUBROUTINE NAME --- MATSCALE
5010 'ENTRY --- I% = SIZE OF DIMENSION #1
5020 '           J% = SIZE OF DIMENSION #2
5030 '           K% = SIZE OF DIMENSION #3
5040 '           A---ORIGINAL ARRAY
5050 '           X---SCALAR FACTOR
5060 '           B---NEW ARRAY
5070 FOR L% = 1 TO K%
5080   FOR M% = 1 TO J%
5090     FOR N% = 1 TO I%
6000       B(N%,M%,L%) = A(N%,M%,L%) *X
6010     NEXT N%
6020   NEXT M%
6030 NEXT L%
6040 RETURN
```

This subroutine will multiply each element in the 3-dimensional array A by the value assigned to X and produce a new 3-dimensional array B. Upon entry into this subroutine, I% must contain the size of dimension #1, J% must contain the size of dimension #2, K% must contain the size of dimension #3, X must be assigned the value to multiply by (scalar factor). Both arrays A and B must also have previously been defined by a DIM statement.

## Transposition of a Matrix

```
5000 'SUBROUTINE NAME --- MATTRANS
5010 'ENTRY I% = # OF ROWS, J% = # OF COLUMNS
5020 'TRANSPOSE A INTO B
5030 FOR K% = 1 TO I%
5040   FOR L% = 1 TO J%
5050     B(L%,K%) = A(K%,L%)
5060   NEXT L%
5070 NEXT K%
5080 RETURN
```

This subroutine will transpose the 2-dimensional matrix A into the 2-dimensional matrix B. Upon entry into the subroutine, I% must contain the number of rows and J% must contain the number of columns. The arrays A and B both must have previously been defined by a DIM statement.

## Matrix Addition

```
5000 'SUBROUTINE NAME -- MATADD
5010 'ENTRY -- I% = SIZE OF DIMENSION #1
5020 '           J% = SIZE OF DIMENSION #2
5030 '           K% = SIZE OF DIMENSION #3
5040 'ARRAY A+B = C
5050 FOR L% = 1 TO K%
5060   FOR M% = 1 TO J%
5070     FOR N% = 1 TO I%
5080       C(N%,M%,L%) = B(N%,M%,L%) + A(N%,M%,L%)
5090     NEXT N%
6000   NEXT M%
6010 NEXT L%
6020 RETURN
```

This subroutine will add the elements of arrays A and B to produce a new array C. A,B, and C must have previously been defined by a DIM statement.

## Matrix Multiplication

```
5000 ' SUBROUTINE NAME -- MATMULT
5010 'ENTRY -- ARRAY A MUST BE D1% BY D3% ARRAY
5020 '           ARRAY B MUST BE D3% BY D2% ARRAY
5030 '           ARRAY C MUST BE D1% BY D2% ARRAY
5040 FOR I% = 1 TO D1%
5050   FOR J% = 1 TO D2%
5060     C(I%,J%) = 0
5070       FOR K% = 1 TO D3%
5080         C(I%,J%)=C(I%,J%)+A(I%,K%)*B(K%,J%)
5090       NEXT K%
6000     NEXT J%
6010 NEXT I%
```

This subroutine will multiply the 2-dimensional array A by the 2-dimensional array B and produce C.

## Chapter Seven

# Functions

### OVERVIEW

BASIC-80 provides a full set of intrinsic functions for use by the BASIC-80 programmer. One group of intrinsic functions is the arithmetic functions. These functions are referenced by a symbolic name; when invoked, they return a single value. This single value will be either an integer or single-precision data type. The arguments to the arithmetic functions are enclosed in parentheses.

The BASIC-80 programmer also has a group of special functions that he may use. These special functions each have their own unique requirements for referencing.

Complete facilities for constructing and referencing user-written functions have also been included in BASIC-80.

## ARITHMETIC FUNCTIONS

Several arithmetic functions are available for use by the BASIC-80 programmer. These arithmetic functions are:

<u>FUNCTION</u>	<u>DEFINITION</u>
ABS(X)	absolute value
ATN(X)	arctangent
CDBL(X)	convert to double-precision
CINT(X)	round to integer
COS(X)	cosine
CSNG(X)	convert to single-precision
EXP(X)	e to the power of X
FIX(X)	truncate supplied argument
INT(X)	largest integer $\leq$ X
LOG(X)	natural log of X
RND(X)	random number between 0 and 1
SGN(X)	sign (+, - or 0) of X
SIN(X)	sine of X
SQR(X)	square root of X
TAN(X)	tangent of X

Table 7-1  
Arithmetic Functions.

### ABS (absolute value)

Form: ABS(X)

The ABS function returns the absolute value of the expression X.

Example:

```
PRINT ABS(7*(-5))  
35  
0k
```

### ATN (arctangent)

Form: ATN(X)

The ATN function will return the arctangent of X. X must be expressed in radians. The result will be in the range  $-\pi/2$  to  $\pi/2$ . The expression X may be any numeric type, but the evaluation of ATN is always performed in single-precision.

Example:

```
10 X = 3  
20 PRINT ATN(X)  
RUN  
1.24905  
0k
```

**CDBL (convert to double-precision)**

Form:            CDBL(X)

The CDBL function will convert X to a double-precision number.

Example:

```
10 X = 454.67
20 PRINT X;CDBL(X)
RUN
454.67  454.6700134277344
Ok
```

**CINT (convert to integer)**

Form:            CINT(X)

The CINT function will convert X to an integer. The fractional portion of X will be rounded to the nearest integer. If this function returns a result that is not in the range -32768 to 32767, an "Overflow" error will occur.

Example:

```
PRINT CINT(45.67)
46
Ok
```

## COS (cosine)

Form: COS(X)

The COS function will return the cosine of X. X must be expressed in radians.  
The calculation of COS is performed in single-precision.

Example:

```
10 X = 2 * COS(.4)
20 PRINT X
RUN
1.84212
0k
```

## CSNG (convert to single-precision)

Form: CSNG(X)

The CSNG function will convert X to a single-precision number.

Example:

```
10 A# = 975.3421#
20 PRINT A#;CSNG(A#)
RUN
975.3421 975.342
0k
```

NOTE: The # is used to declare the values as double-precision data types.

**EXP (e raised to a power)**

Form: EXP(X)

The EXP function will return e raised to the power of X. e is the natural logarithm's base value (2.71828...). X must be  $\leq 87.3365$ . If EXP overflows, the "Overflow" error message is displayed.

Example:

```
10 X = 5
20 PRINT EXP(X-1)
RUN
54.5982
0k
```

**FIX (truncate supplied argument)**

Form: FIX(X)

The FIX function will return the truncated integer part of X. The major difference between FIX and INT is that FIX simply removes any decimal portion of a number. INT will round a negative number to the next lowest number.

Examples:

```
PRINT FIX(58.75)
58
0k
```

```
PRINT FIX(-58.75)
-58
0k
```

### INT (round to integer)

Form: INT(X)

The INT function will return the largest integer  $\leq X$ . When a negative value is rounded, it will be rounded to the next smallest value.

Examples:

```
PRINT INT(99.89)
99
```

```
PRINT INT(-12.11)
-13
```

### LOG (natural logarithm)

Form: LOG(X)

The LOG function will return the natural logarithm of the supplied argument. X must be greater than zero. IF X is less than or equal to zero, an “Illegal function call” error message will be displayed.

Example:

```
PRINT LOG(45/7)
1.86075
```

**RND (random number generator)**

Form: RND(X)

The RND function will return a random number between 0 and 1. The same sequence of random numbers is generated each time the program is executed unless the random number generator is reseeded. The RANDOMIZE statement is used to reseed the random number generator.

If X<0, the sequence of numbers will be restarted. X>0 or X omitted will generate the next random number in the sequence. X=0 will repeat the last number generated.

Example:

```
10 RANDOMIZE PEEK(11)
20 FOR I = 1 TO 5
30 PRINT INT(RND*100);
40 NEXT
RUN
24 30 31 51 5
OK
```

NOTE: The sequence of numbers generated will be different every time this example program is executed.

**RANDOMIZE (reseed random number generator)**

Form RANDOMIZE <expression>

The RANDOMIZE statement is used to reseed the random number generator. <expression> is used as the random number seed value. If <expression> is omitted, BASIC-80 suspends program execution and asks for a value by printing:

Random Number Seed (-32768 to 32767) ?

The value input is used as the random number seed.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is executed.

To change the sequence of random numbers every time the program is executed, place a RANDOMIZE statement at the beginning of the program and change the argument with each run.

## SGN (sign of expression)

Form:            SGN(X)

The SGN function returns a result based on the numeric value of X.

If  $X < 0$ , SGN(X) will return -1. If  $X = 0$ , SGN(X) will return 0. If  $X > 0$ , SGN(X) will return 1.

You can create an arithmetic IF statement using this function:

```
ON SGN(X)+2 GOTO 100,200,300
```

If X is negative, the program will branch to 100. If X is zero, the program will branch to 200. If X is positive, the program will branch to 300.

Example:

```
10 INPUT X
20 ON SGN(X)+2 GOTO 50,60,70
50 PRINT"NEGATIVE":GOTO 10
60 PRINT"ZERO":GOTO 10
70 PRINT"POSITIVE":GOTO 10
RUN
? -10
NEGATIVE
? 0
ZERO
? 10
POSITIVE
Ok
```

**SIN (sine)**

Form: SIN(X)

The SIN function will return the sine of X. X must be expressed in radians. SIN(X) is calculated in single-precision.

Example:

```
PRINT SIN(1.5)
.997495
0k
```

**SQR (square root)**

Form: SQR(X)

The SQR function will return the square root of X. X must be  $\geq 0$ . If X is less than zero, an “Illegal function call” error will be displayed.

Example:

```
10 X = 25
20 PRINT X,SQR(X)
RUN
25      5
0k
```

**TAN (tangent)**

FORM: TAN(X)

The TAN function will return the tangent of X. X must be expressed in radians. TAN(X) will be calculated in single-precision. If TAN overflows, the “Overflow” error message will be displayed.

Example:

```
PRINT TAN(10)
.64836
0k
```

## MATHEMATICAL FUNCTIONS

Some functions that are not intrinsic to BASIC-80 may be calculated as follows:

<u>Function</u>	<u>BASIC-80 Equivalent</u>
SECANT	$\text{SEC}(X) = 1/\text{COS}(X)$
COSECANT	$\text{CSC}(X) = 1/\text{SIN}(X)$
COTANGENT	$\text{COT}(X) = 1/\text{TAN}(X)$
INVERSE SINE	$\text{ARCSIN}(X) = \text{ATN}(X/\text{SQR}(-X^*X+1))$
INVERSE COSINE	$\text{ARCCOS}(X) = -\text{ATN}(X/\text{SQR}(-X^*X+1))+1.570796$
HYPERBOLIC SINE	$\text{SINH}(X) = (\text{EXP}(X)-\text{EXP}(-X))/2$
HYPERBOLIC COSINE	$\text{COSH}(X) = (\text{EXP}(X)+\text{EXP}(-X))/2$
HYPERBOLIC TANGENT	$\text{TANH}(X) = (\text{EXP}(X)-\text{EXP}(-X))/\text{EXP}(X)+\text{EXP}(-X)$
HYPERBOLIC SECANT	$\text{SECH}(X) = 2/(\text{EXP}(X)+\text{EXP}(-X))$
HYPERBOLIC COSECANT	$\text{CSCH}(X) = 2/(\text{EXP}(X)-\text{EXP}(-X))$
HYPERBOLIC COTANGENT	$\text{COTH}(X) = (\text{EXP}(X)+\text{EXP}(-X))/(\text{EXP}(X)-\text{EXP}(-X))$

Table 7-2  
Mathematical Functions

## SPECIAL FUNCTIONS

Several special functions are available for use by the BASIC-80 programmer. These special functions are:

<u>Function</u>	<u>Definition</u>
FRE(X)	free memory space
INP(I)	input from port
LPOS(X)	position of print head
NULL(X)	set number of nulls
OUT I,J	output to port
PEEK(I)	read byte from memory
POKE I,J	write byte to memory
POS(X)	current cursor position
SPC(X)	print spaces
TAB(I)	tab carriage
VARPTR(X)	variable pointer
WAIT I,J,K	status of port
WIDTH I	set terminal line width
WIDTH LPRINT I	set printer line width

**Table 7-3**  
Special Functions.

### ( ) FRE (return amount of free memory)

Form: FRE(0) FRE(X\$)

The FRE function will return the number of bytes in memory that are not being used by BASIC-80. The arguments to FRE are dummy arguments.

FRE(" ") forces some system housekeeping before returning the number of free bytes. The housekeeping will take 1 to 2 minutes. BASIC-80 will not initiate housekeeping until all free memory has been used.

Example:

```
PRINT FRE(0)
```

### ( ) INP (input byte from I/O port)

Form: INP(I)

The INP function will return the byte read from port I. I must be in the range 0 to 255. INP is the complementary function to OUT.

Example:

```
10 A = INP(255)
```

**LPOS (return position of print head)**

Form: LPOS(X)

The LPOS function will return the current position of the line printer print head within the line printer buffer. This does not necessarily correspond to the actual physical position of the print head. X is a dummy argument.

Example:

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

**OUT (output byte to I/O port)**

Form: OUT I,J

The OUT statement will send a byte to an output port. I and J must be integer expressions in the range 0 to 255. The integer expression I is the port number, and the integer expression J is the data to be transmitted.

Example:

```
100 OUT 32,100
```

### PEEK (examine contents of memory location)

Form: PEEK(I)

The PEEK function will return the byte read from memory location I. The value returned will be a decimal integer in the range 0 to 255. I must be in the range 0 to 65536. PEEK is the complimentary function to the POKE function.

Example:

```
PRINT PEEK(34000)
234
0k
```

Note: You may not get the same result if you PEEK memory location 34000.

### POKE (change contents of memory location)

Form: POKE I,J

The POKE function will change the contents of a memory location. I and J must be integer expressions.

The integer expression I is the address of the memory location to be changed. I must be in the range 0 to 65535.

The integer expression J is the value to be placed into memory location I. J must be in the range 0 to 255.

POKE and PEEK are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

Example:

```
POKE 34000,1
0k
```

**POS (return current cursor position)**

Form: POS(I)

The POS function will return the current cursor position. The left-most position is 1. I is a dummy argument.

Example:

```
IF POS(I) > 60 THEN PRINT CHR$(13)
```

**SPC (print blanks)**

Form: SPC(I)

The SPC function is used to print blanks on the terminal or the line printer. The integer argument I specifies how many blanks are to be printed. I must be in the range 0 to 255. The SPC function may only be used with PRINT and LPRINT statements.

Example:

```
PRINT "OVER";SPC(15);"THERE"  
OVER                         THERE  
Ok
```

### TAB (tab carriage)

Form: TAB(I)

The TAB statement is used to space to position I on the terminal or line printer. If the current print position is already beyond space I, TAB goes to position I on the next line.

Position 1 is the left-most position, and the right-most position is the width minus one. I must be an unsigned integer expression in the range 1 to 255. TAB may only be used with PRINT and LPRINT statements.

Example:

```
10 PRINT "NAME";TAB(10);"AMOUNT"
20 READ A$,B$
30 PRINT A$;TAB(10);B$
40 DATA "WILLIAMS", "$20.00"
RUN
NAME          AMOUNT
WILLIAMS      $20.00
```

**VARPTR (variable pointer)**

Form#1:      VARPTR (<variable name>)  
Form#2:      VARPTR (#<file number>)

Form #1 of the VARPTR function is used to return an address-value which can be used to locate where the variable <variable name> is stored in memory. A value must have been previously assigned to <variable name> or an "Illegal function call" error will result.

Any type variable name may be used (numeric, string, array). The result returned will be an integer in the range -32768 to 32767. If a negative address is returned, add it to 65536 to obtain the actual address. This returned address (which we will refer to as A) has a different meaning depending upon on the data type of <variable name>.

NOTE: The results from these examples may vary depending on how much memory your system has, how much memory is being used for BASIC-80, etc.

If <variable name> is a string value:

- A — Contains the length of the string.
- A+1 — Contains the LSB (least significant byte) of the actual string starting address.
- A+2 — Contains the MSB (most significant byte) of the actual string starting address.

The actual address where the string value is stored can be calculated by:

$$\text{actual address} = (A+2)*256 + (A+1)$$

This address will most likely be in high RAM where the string values are stored. If the string value is a constant (a string literal), this address will represent the area of memory where the program line containing the string is stored.

(Remember, A is only the address of this information, you must PEEK(A) to obtain the actual value.)

Example:

```
X$="ABC" [you type]  
Ok  
PRINT VARPTR(X$) [you type]  
-23927  
Ok
```

If <variable name> is an integer value:

- A — Contains the LSB of the 2-byte integer
- A+1 — Contains the MSB of the 2-byte integer

To display this information (in two's complement decimal representation), execute a PRINT PEEK(A) and a PRINT PEEK(A+1).

Example:

```
I% = 1000 [you type]  
Ok  
PRINT VARPTR(I%) [you type]  
-29121  
Ok
```

If <variable name> is a single-precision value:

- A — Contains the LSB of value.
- A+1 — Contains next MSB of value.
- A+2 — MSB (most significant byte) with implied leading one.  
Most significant bit is the sign of the number.
- A+3 — Exponent of value in excess 128 notation  
(128 is added to the exponent).

If <variable name> is a double-precision value:

- A — Contains the LSB of value.
- A+1 — Next MSB.
- A+2 — Next MSB.
- A+3 — Next MSB.
- A+4 — Next MSB.
- A+5 — Next MSB.
- A+6 — MSB (most significant byte) with implied leading one.  
Most significant bit is the sign of the number.
- A+7 — Exponent of value in excess 128 notation.

The double and single-precision numbers are stored in a normalized exponent form, so that a decimal is assumed before the MSB. The exponent is stored in excess 128 notation (128 is added to the exponent). The high order bit of the MSB is used as a sign bit. It is 0 if the number is positive or 1 if the number is negative.

Example:

```
10 A = 23.4
20 B#=23.12345678
30 PRINT VARPTR(A),VARPTR(B)
RUN
-23888      -23880
```

Form #2 of the VARPTR function is used to return the address of the FIELD buffer for the specified random file.

Example:

```
10 OPEN "R",1,"OUT.DAT"
20 FIELD#1, 128 AS JUNK$
30 PRINT VARPTR(#1)
RUN
-2345
Ok
```

## WAIT (monitor port)

Form:            WAIT I,J,K

where I is the integer decimal number of the port being monitored and K and J are integer expressions. The WAIT function is used to suspend program execution while monitoring the status of a machine input port.

The WAIT function causes execution to be suspended until a specified machine input port develops a certain bit pattern. The data read at the port is XOR'ed with the integer expression K, and then AND'ed with the integer expression J.

If the result is zero, BASIC-80 loops back and reads the data at the port again. If the result is non-zero, execution resumes with the next executable statement. If K is omitted, it is assumed to be zero. I,J, and K must be in the range 0-255. (Remember, all numbers are decimal unless preceded by&H,&O, or&.)

Example:

```
WAIT 20,6
```

Execution stops until either bit 1 or bit 2 of port 20 are equal to 1. (Bit 0 is least significant, bit 7 is most.) Execution resumes at the next statement.

```
WAIT 10,255,7
```

Execution stops until any of the most significant five bits of port 10 are equal to 1, or any of the least significant three bits are 0. Execution resumes at the next statement.

**WIDTH (set line width)**

Form:            **WIDTH [LPRINT] <integer expression>**

The **WIDTH** function is used to set the printed line width for the terminal or line printer. The **LPRINT** option is used for the line printer width.

**<integer expression>** is the number of characters in the printed line. The default line width for the terminal is 72 and the default line width for the line printer is 132.

IF **<integer expression>** is 255 the line width is "infinite", that is, BASIC never inserts a carriage return. However, the position of the cursor or print head, as given by the **POS** or **LPOS** function, returns to zero after position 255.

**Example**

**WIDTH 80**        set terminal width at 80 characters.

**WIDTH LPRINT 96**    set printer width at 96 characters.

## USER-DEFINED FUNCTIONS

Sometimes it is necessary to execute the same sequence of program statements or mathematical formulas in several different places. BASIC-80 allows the programmer to define his own functions and then reference these functions in the same manner as the standard system functions, such as ABS, SIN, or SQR.

At times it may also be necessary to code a specific portion of a program in assembly language. Facilities have been provided for the BASIC-80 programmer to reference assembly language programs from a BASIC-80 program.

### DEF FN (define function)

Form: DEF FN<name>(<variable list>) = expression

The DEF FN statement is used to define an implicit function.

<name> must be a legal variable name. This name, preceded by the FN becomes the function name. The entries in the variable list are "dummy" variable names. The dummy variables represent the argument variables or values in the function call.

Any number of arguments are allowed, and any valid expression may appear on the right side of the equal sign. The length of the function definition is limited to one logical line (255 characters).

User-defined functions may be of any type. The type of a function is specified by inserting one of the type declaration characters (%,!,#,or \$) after the function name. If a type declaration character is not used, the definition (DEFSTR, DEFSNG, etc.) for that letter applies. If you have made no unique DEF's, then a numeric variable is assumed to be a single-precision data type.

If a type is specified for the function, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs. DEF FN is illegal in the Command Mode.

Example:

```
10 DEF FNAB(X,Y)=X+Y
20 SUM = FNAB(10,20)
30 PRINT SUM
RUN
30
0k
```

## ASSEMBLY LANGUAGE PROGRAMS

It is possible to invoke an assembly language program in either of two methods. The first method is to use the USR function, and the other method is with the CALL statement.

For more information, see Appendix E, "Assembly Language Subroutines."

### **DEF USR (define entry address for USR subroutine)**

Form:            **DEF USR<digit>=<expression>**

The DEF USR statement is used to define the entry points for up to 10 assembly language subroutines.

The <digit> is the number of the assembly language subroutine. <digit> may be any number from 0-9. If <digit> is omitted, it is assumed to be 0.

The value of expression is the starting address of the assembly language subroutine in decimal, unless the number is preceded by a special base specification character. A hexadecimal number is specified with the prefix &H and an octal number is specified with the prefix &O or &.

Examples:

```
DEFUSR1=&H22  
DEFUSR2=45000  
DEFUSR5=ADDRESS
```

## USR (invoke assembly language subroutine)

Form:      **USR<digit>(X)**

The USR function is used to invoke an assembly language subroutine. <digit> must be in the range 0-9 and corresponds to the digit supplied with the DEF USR statement. If <digit> is omitted, it is assumed to be zero. X is the argument to be passed to the assembly language subroutine.

Example:

Z = USR1(B/2)

A = USR2(1.23)

C = USR5(ARG1)

NOTE: A detailed description of how to define and reference USR functions is contained in Appendix E.

## CALL (call assembly language subroutine)

Form:      **CALL<variable name>[(argument list)]**

The CALL statement is used to call an assembly language subroutine.

<variable name> is assigned an address that is the starting point, in memory, of the assembly language subroutine. The address should be assigned before a CALL statement is executed. <variable name> may not be an array variable name. <argument list> contains the arguments that are passed to the assembly language subroutine.

The CALL statement generates the same calling sequence used by Microsoft's FORTRAN, COBOL and BASIC Compilers. This calling sequence is explained in Appendix E, "Assembly Language Subroutines."

Example:

```
110 MYROUT = &HD000
120 CALL MYROUT(I,J,K)
```



## *Chapter Eight*

# **Special Features**

## **OVERVIEW**

BASIC-80 provides the programmer with several special features. One of these features, Error Trapping, is useful for detecting errors during program execution. Another feature is the PRINT USING statement. This statement allows the programmer to specify the format of both numeric and string output.

Another important feature is the Trace flag, which allows the programmer to follow, line-by-line, the execution of a program.

BASIC-80 also provides the facilities for overlay management. The CHAIN and COMMON statement are used for this function.

## ERROR TRAPPING

BASIC-80 allows the programmer to write error detection and error handling routines which can attempt to recover from errors, or provide more complete explanations of the causes of errors. This facility has been added through the use of the ON ERROR GOTO, RESUME, and ERROR statements, and with the ERR and ERL variables.

### **ON ERROR GOTO (enable error trapping)**

Form:           **ON ERROR GOTO <line number>**

The ON ERROR GOTO statement is used to enable error trapping and specify the first line of the error handling subroutine.

Once error trapping has been enabled, all errors detected, including Command Mode errors (e.g., Syntax errors), will cause a jump to the specified error handling subroutine. If <line number> does not exist, an “Undefined line number” error results.

To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC-80 to stop and print the error message for the error that caused the trap. We recommend that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not trap errors within the error handling subroutine.

Example:

```
10 ON ERROR GOTO 1000
```

## **RESUME (continue execution)**

Forms: RESUME  
RESUME 0  
RESUME NEXT  
RESUME <line number>

The RESUME statement is used to continue program execution after an error recovery procedure has been performed:

Any one of the four formats shown above may be used, depending upon where execution is to resume:

**RESUME** Execution resumes at the statement  
or which caused the error.

## RESUME O

**RESUME NEXT** Execution resumes at the statement immediately following the one which caused the error.

RESUME<line number> Execution resumes at <line number>.

A RESUME statement that is not in an error trap routine causes a “RESUME without error” message to be printed.

#### Error Trap Example:

```
100 ON ERROR GOTO 500
200 INPUT "WHAT ARE THE NUMBERS TO DIVIDE";X,Y
210 Z=X/Y
220 PRINT "QUOTIENT IS";Z
230 GOTO 200
500 IF ERR=11 AND ERL=210 THEN 520
510 STOP
520 PRINT "YOU CAN'T HAVE A DIVISOR OF ZERO!"
530 RESUME 200
```

**ERROR (generate error)**

Form:      **ERROR <integer expression>**

The **ERROR** statement can be used either to simulate the occurrence of a BASIC-80 error, or to allow error codes to be defined by the user.

The value of **<integer expression>** must be greater than 0 and less than 255. If the value of **<integer expression>** equals an error code already in use by BASIC-80, the **ERROR** statement will simulate the occurrence of that error, and the corresponding error message will be printed.

To define your own error code, use a value that is greater than any used by BASIC-80's error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to BASIC-80.) This user-defined error code may then be conveniently handled in an error trap routine.

If an **ERROR** statement specifies a code for which no error message has been defined, BASIC-80 responds with the message "Unprintable error". Execution of an **ERROR** statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example:

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

Or, in Command Mode:

```
Ok
ERROR 15      (you type this line)
String too long (BASIC-80 types this line)
Ok
```

## ERR and ERL Variables

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF/THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a Command Mode statement, ERL will contain 65535. To test if an error occurred in a Command Mode statement, use IF 65535 = ERL THEN ... Otherwise, use

IF ERR = error code THEN ...

IF ERL = line number THEN ...

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement.

When the error handling subroutine is entered, the variable ERR contains the error code for the error. The error codes and their meanings are listed on the next page. See Appendix A, "Error Messages," for a more detailed discussion of the error messages.

## ERROR CODES

### General Errors

<u>CODE</u>	<u>ERROR</u>
1	NEXT WITHOUT FOR
2	SYNTAX ERROR
3	RETURN WITHOUT GOSUB
4	OUT OF DATA
5	ILLEGAL FUNCTION CALL
6	OVERFLOW
7	OUT OF MEMORY
8	UNDEFINED LINE
9	SUBSCRIPT OUT OF RANGE
10	DUPLICATE DEFINITION
11	DIVISION BY ZERO
12	ILLEGAL DIRECT
13	TYPE MISMATCH
14	OUT OF STRING SPACE
15	STRING TOO LONG
16	STRING FORMULA TOO COMPLEX
17	CAN'T CONTINUE
18	UNDEFINED USER FUNCTION
19	NO RESUME
20	RESUME WITHOUT ERROR
21	UNPRINTABLE ERROR
22	MISSING OPERAND
23	LINE BUFFER OVERFLOW
26	FOR WITHOUT NEXT
29	WHILE WITHOUT WEND
30	WEND WITHOUT WHILE

**Table 8-1**

Error Codes.

## Disk Errors

<u>CODE</u>	<u>ERROR</u>
50	FIELD OVERFLOW
51	INTERNAL ERROR
52	BAD FILE NUMBER
53	FILE NOT FOUND
54	BAD FILE MODE
55	FILE ALREADY OPEN
57	DISK I/O ERROR
58	FILE ALREADY EXISTS
61	DISK FULL
62	INPUT PAST END
63	BAD RECORD NUMBER
64	BAD FILE NAME
66	DIRECT STATEMENT IN FILE
67	TOO MANY FILES

**Table 8-1 (Cont'd.)**

Error Codes.

## FORMATTED OUTPUT

The PRINT USING statement can be used to output information in a specific format. This feature is useful in such applications as printing payroll checks or accounting reports.

### **PRINT USING (format output)**

Form:           PRINT USING<string exp>;<list of expressions>

The PRINT USING statement is used to print strings or numbers using a specified format.

<list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. <string exp> is a string literal (or variable) that is comprised of special formatting characters. These formatting characters (see below) determine the field, and the format, of the printed strings or numbers.

#### **String Fields**

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!"

This specifies that only the first character in the given string is to be printed.

### "\n spaces\"

This specifies that  $2+n$  characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!" ;A$;B$
30 PRINT USING "\  \";A$;B$
40 PRINT USING "\      \";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT !!
```

### "&"

The ampersand specifies a variable length string field. When the field is specified with "&", the string is output exactly as input.

Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!" ;A$
30 PRINT USING "&";B$
RUN
L
OUT
```

### Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

### "#"

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

"."

A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

Examples:

```
PRINT USING "##.##"; .78  
0.78
```

```
PRINT USING "###.##"; 987.654  
987.65
```

```
PRINT USING "##.##    "; 10.2, 5.3, 66.789, .234  
10.20 5.30 66.79 0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

"+"

A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

"-"

A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign. If the number is positive, a space will be printed.

```
PRINT USING "+##.##    "; -68.95, 2.4, 55.6, -.9  
-68.95    +2.40    +55.60    -0.90  
  
PRINT USING "##.##-  "; -68.95, 22.449, -7.01  
68.95-    22.45    7.01-
```

"\*\*"

A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The \*\* also specifies positions for two more digits.

Example:

```
PRINT USING "***#.#";12.39,-0.9,765.1
*12.4      *-0.9      765.1
```

"\$\$"

A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The \$\$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with \$\$ . Negative numbers cannot be used unless the minus sign trails to the right.

Example:

```
PRINT USING "$$###.##";456.78
$456.78
```

"\*\*\$"

The \*\*\$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. \*\*\$ specifies three more digit positions, one of which is the dollar sign.

Example:

```
PRINT USING "***$##.##";2.34
***$2.34
```

" , "

A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit on the left side of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^^^^) format.

Example:

```
PRINT USING "####,.##";1234.5  
1,234.50
```

```
PRINT USING "###.##,";1234.5  
1234.50,
```

" ^ ^ ^ ^ "

Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

Example:

```
PRINT USING "#.##^^^^";234.56  
2.35E+02
```

```
PRINT USING ".###^^^^-";888888  
.8889E+06
```

```
PRINT USING "+.#^ ^ ^";123  
+.12E+03
```

"\_\_"

An underscore in the format string causes the next character to be output as a literal character.

Example:

```
PRINT USING "_!##.##_!";12.34  
!12.34!
```

The literal character itself may be an underscore by placing “\_\_” in the format string.

### Errors

If the number to be printed is larger than the specified numeric field, a percent sign (%) is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

Example:

```
PRINT USING "##.##";111.22  
%111.22  
  
PRINT USING ".##";.999  
%1.00
```

If the number of digits specified exceeds 24, an “Illegal function call” error will result.

## TRACE FLAG

As a debugging aid, two statements are provided to trace the execution of program instructions.

### **TRON/TROFF (enable/disable trace flag)**

Forms:            TRON    TROFF

The TRON/TROFF statements are used to trace the execution of program statements.

As an aid in debugging, the TRON statement (executed in either the Command or Indirect Mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example:

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINTJ;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
[10] [20] [30] [40] 1 10 20
[50] [60] [30] [40] 2 20 30
[50] [60] [70]
Ok
TROFF
Ok
```

## OVERLAY MANAGEMENT

BASIC-80 provides two statements, CHAIN and COMMON, which are useful for manipulating overlays. With these two statements, it is possible to merge several programs during the execution of a program, as well as pass several or all the variables to another program.

### CHAIN (call overlay)

Form:      CHAIN [MERGE] <filename>[,,<line number exp>]  
[,ALL][,DELETE<range>]]

The CHAIN statement is used to call a program and pass variables to it from the current program.

<filename> is the name of the program that is called.

Example:

CHAIN"PROG1"

<line number exp> is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line.

Example:

CHAIN"PROG1", 1000

<line number exp> is not affected by a RENUM command.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to specify the variables that are passed.

Example:

CHAIN"PROG1", 1000, ALL

If the MERGE option is included, it allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGED.

Example:

```
CHAIN MERGE"OVRLAY",1000
```

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option. The line numbers in the <range> of the delete are affected by the RENUM command.

Example:

```
CHAIN MERGE"OVRLAY@",1000,DELETE 1000-5000
```

If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFNSG, DEFDL, DEFSTR, or DEFFN statement containing shared variables must be restated in the chained program.

### **COMMON (pass variables)**

Form:           **COMMON<list of variables>**

The COMMON statement is used to pass variables to a chained program.

The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though we recommend that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "( )" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:

```
100 COMMON A,B,C,D(),G$  
110 CHAIN "PROG3",10  
.  
.
```

## Chapter Nine

# Editing

### OVERVIEW

In Edit Mode, it is possible to edit portions of a line without retyping the entire line. Upon entering Edit Mode, BASIC-80 types the line number of the line to be edited. Then it types a space and waits for the Edit Mode subcommand.

Edit Mode subcommands are used to insert, delete, replace, or search for text within a line. The subcommands are not echoed to the terminal. Some of the Edit Mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When an integer is not specified, it is assumed to be one.

Edit Mode subcommands may be categorized according to the following functions:

1. Moving the cursor.
2. Inserting text.
3. Deleting text.
4. Finding text.
5. Replacing text.
6. Ending and restarting Edit Mode.

If BASIC-80 receives an unrecognizable command or illegal character while in Edit Mode, it sounds the bell (CTRL-G) and the command or character is ignored. You can invoke the Edit Mode by typing:

EDIT <line number>

Where <line number> is the number of the line to be edited. If no <line number> exists, an “Undefined line number” error will result.

The requested line number will be printed, followed by a space. The cursor will now be positioned to the left of the first character in the line.

Type in the following line:

100 FOR J = 1 TO 10:PRINT J:NEXT

This program line will be used to demonstrate the various Edit Mode commands.

## MOVING THE CURSOR

### In Space Bar

In Edit Mode, the Space Bar is used to move the cursor to the right. For example, using line 100 entered above, invoke the Edit Mode. The line number 100 should be displayed on your screen as such:

100 \_

Now press the Space Bar. The cursor will move over one space. The first character of the program line will now be displayed. If this character was a blank, then a blank will be displayed on your screen. Keep pressing the Space Bar until the first non-blank character is displayed. At this point, the screen should look like this:

100 F\_

It is also possible to move over more than one space at a time. Just type the number of spaces first, and then the Space Bar. For example, to move over five spaces, type 5 and then press the Space Bar once. The characters will be printed as you move over them.

100 FOR J=\_

(Your display may not look exactly like this, as it depends on how many blanks you inserted in the program line.)

### BACK SPACE

In Edit Mode, the BACK SPACE key moves the cursor one space to the left. The characters are not deleted as you move over them. To return to our example,

100 FOR J=\_

if the cursor were positioned after the = sign, pressing BACKSPACE once should move the cursor under the = sign. Thus:

100 FOR J\_=

## INSERTING TEXT

### I (Insert)

The I command will insert text beginning at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, press the ESC key. If you press the RETURN during the insert command, the effect is the same as typing ESC and then RETURN.

Use the Space Bar to move over to the 0 in the 10.

```
100 FOR J=1 TO 10_
```

Now, suppose you want to change the 10 to 100. Press the I key (you don't have to terminate the entry with a RETURN). You are now in Insert Mode. To make the necessary change, type a 0. The display should now look like this:

```
100 FOR J=1 TO 100_
```

Now that you have made the change, press the ESC key and you will exit Insert Mode. Now press the RETURN to save all your changes and return to BASIC-80 Command Mode. If you list line 100, it should look similiar to this:

```
100 FOR J=1 TO 100:PRINT J:NEXT
```

During an insert command, you can use the BACK SPACE key on the terminal to delete characters on the left of the cursor.

If you try to insert a character that will make the line longer than 255 characters, a bell (CTRL-G) will be typed and the character will not be printed.

### X (Extend Line)

The X command is used to extend the line. X moves the cursor to the end of a line. BASIC-80 then goes into the Insert Mode and allows text to be inserted as if an insert command had been given. When you are finished extending the line, type ESC or press RETURN and you will be returned to BASIC-80 Command Mode.

For example, to extend line number 100 you previously typed in, invoke Edit Mode with line number 100. The screen will show:

```
100 _
```

Now press the X key. The entire line will be displayed and the cursor will be at the end of the line:

```
100 FOR J=1 TO 100:PRINT J:NEXT..
```

Now you have been put into Insert Mode. You can now add another program statement to the end of this line. For example, type :PRINT“ALL DONE” and a RETURN. The line has now been extended to include this statement. If you were to LIST 100, it should look like this:

```
100 FOR J=1 TO 100:PRINT J:NEXT:PRINT"ALL DONE"
```

## DELETING TEXT

### nD (Delete)

nD deletes n characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than n characters to the right of the cursor, the remainder of the line will be deleted.

For example, enter Edit Mode with line number 100 you previously typed in. Now, using the Space Bar, move the cursor over to the end of the FOR statement. The screen should look something like this:

```
100 FOR J=1 TO 100:_
```

Now type 8D. This will delete eight characters to the right of the cursor. The screen should look something like this:

```
100 FOR J=1 TO 100:\PRINT J:\
```

(Note that the characters deleted are enclosed in backslashes.)

Now press RETURN and you will be back to the BASIC-80 Command Mode. If you LIST 100, you should notice that the PRINT J statement has been deleted from the program line.

### H (Hack and Insert)

H deletes all characters to the right of the cursor and then automatically enters Insert Mode. H is useful for replacing statements at the end of a line. For example, assume you wish to change the last statement of program line 100. First, you must enter Edit Mode with line number 100. Now move over to the NEXT statement with the Space Bar. The screen should look similiar to this:

```
100 FOR J=1 TO 100:NEXT:_
```

Press the H key and then type STOP. Type a RETURN to save this change and exit to BASIC-80 Command Mode.

Now list line number 100. If you've been following the editing changes in this Chapter, the line should look like this:

```
100 FOR J=1 TO 100:NEXT:STOP
```

## FINDING TEXT

### nS<ch>(Search)

The search subcommand searches for the nth occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor will stop at the end of the line. All characters passed over during the search are printed. NOTE: only characters to the right of the cursor are included in this search.

For example, using the current form of the sample line 100, enter Edit Mode with line 100. Next, type 2S:. This command will be used to search for the second occurrence of the colon character in program line 100. The display should look something like this:

```
.100 FOR J=1 TO 100:NEXT_
```

At this point you can execute any command you wish. You could enter a counter variable after the NEXT statement by first entering Insert Mode and then typing a space and the variable J. Now hit ESC to exit Insert Mode. Finally, press RETURN in order to exit back to the BASIC-80 Command Mode. Now, if you were to list line number 100, it would look similar to this (assuming you followed the editing changes in this chapter):

```
100 FOR J=1 TO 100:NEXT J:STOP
```

### nK<ch>(Search and “Kill”)

The search and kill subcommand is similar to the search subcommand except that all the characters passed over in the search are deleted. The cursor is positioned before <ch> and all the deleted characters are enclosed in backslashes.

For example, invoke the Edit Mode with the current version of line 100. Now type 2K:. This command will delete all of the characters in the line up to the second occurrence of the colon. The screen should look similar to this:

```
100 \FOR J=1 TO 100:NEXT J\_
```

The second colon still needs to be deleted, so type D. The screen should then look similar to this:

```
100 \FOR J=1 TO 100:NEXT J:\:
```

Now press RETURN and LIST line 100. It should look like this:

```
100 STOP
```

## REPLACING TEXT

### nC(Change)

The change subcommand changes the specified number of characters beginning at the current cursor position. If you type only a C without a preceding number, the computer assumes that you wish to change only one character. If you enter a number n before you type C, then it assumes that you wish to change the next n characters.

After you have entered n characters, the Change Mode will be exited. If you attempt to enter any more characters, the bell is sounded (CTRL-G) and the extra characters are ignored.

For example, first retype the original line 100 as:

```
100 FOR J=1 TO 100:PRINT J:NEXT
```

Next, enter Edit Mode with line 100. Your screen should look something like this:

```
100 _
```

Now let's assume that you want to change the terminal value in the FOR/NEXT loop from 100 to 150. You would have to move the cursor over to the first zero in 100. Use the Space Bar to move the cursor over. If you go too far, simply press the BACKSPACE key to move the cursor back.

```
100 FOR J=1 TO 1_
```

Now type C. BASIC-80 will assume that you wish to change only one character. Type 5 and then press RETURN. If you LIST 100, the new line should look like this:

```
100 FOR J=1 TO 150:PRINT J:NEXT
```

## ENDING AND RESTARTING EDIT MODE

### **R**ETURN(Save changes and Exit)

If you press a RETURN, remainder of the line is printed, the changes you made are saved, and the computer returns to the BASIC-80 Command Mode.

### **E**(Save Changes and Exit)

The E subcommand has the same effect as RETURN, except the remainder of the line is not printed.

### **Q**(Cancel and Exit)

The Q subcommand returns to the BASIC-80 Command Mode without saving any of the changes that were made to the line during Edit Mode.

### **L**(List Line)

The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in the Edit Mode. L is usually used to list the line when you first enter Edit Mode. For example:

```
EDIT 100
100 -
<you type L>
<BASIC-80 responds:>
100 FOR J=1 TO 100:NEXT:STOP
100 -
```

**A(Cancel and Restart)**

The A subcommand lets you begin editing a line over again. It discards any changes made so far and restores the original line, repositioning the cursor at the beginning. In order to use the A subcommand, you must not be currently executing any other subcommand. If you are executing another command (such as Insert), press the ESC, and then press the A. In the following example, the operator first lists the original line, then makes changes in Insert Mode, then decides to start over, using the A subcommand to restore the original line:

```
EDIT 100
100 -
<operator types L>
100 FOR J=1 TO 100:NEXT:STOP
100 -
100 for J=1 TO 10_ <operator types I and adds a zero>
<operator types ESC>
<operator types L>
100 FOR J=1 TO 1000:NEXT:STOP
100 -
<operator types A>
100 -
<operator types L; note how original line has been restored>
100 FOR J=1 TO 100:NEXT:STOP
100 -
```

## OTHER EDIT MODE FEATURES

### SYNTAX ERRORS

When it finds a syntax error during the execution of a program, BASIC-80 will automatically enter Edit Mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
Syntax Error in 10
Ok
10 _
```

When you finish editing the line and press RETURN (or the E subcommand), BASIC-80 reinserts the line. This causes all variable values to be lost and all open files to be closed. To preserve the variable values for examination, first exit Edit Mode with the Q subcommand. BASIC-80 will return to the Command Mode and all variable values will be preserved.

### CTRL-A

To enter the Edit Mode on the line you are currently typing, type CTRL-A. BASIC-80 will respond with a carriage return, an exclamation point, (!) and a space. The cursor will then be positioned at the first character in the line. At this point you may proceed by typing any Edit Mode subcommand.

### CURRENT LINE EDITING

You may use the period (.) to denote the current line when you invoke the Edit Mode. So, the command:

```
EDIT .
```

will invoke the Edit Mode at the current line. The line number symbol (.) always refers to the current line.



## Chapter Ten

# BASIC-80 Disk File Operations

## OVERVIEW

BASIC-80 provides several sets of statements for creating and manipulating program and data files.

The file manipulation commands are very useful for manipulating program files. Some of these commands can also be used with data files.

The file management statements are used to open and close data files, check for end-of-file, and to obtain information about the size of a file.

The sequential access statements are used to access sequential files. The sequential access file is easy to use, but the data must be accessed sequentially.

The random access statements are used to access and manipulate random access files. The random access file requires more program steps than the sequential access, but the records in the file can be read in any order.

## FILE MANIPULATION COMMANDS

This is a review of the commands and statements that are useful for manipulating program and data files. These statements and commands are also discussed in Chapter Three, "Command Mode Statements".

### **FILES ["<filename>"]**

The FILES command lists the names of the files that are residing on the current disk. If the optional <filename> string is included, the names of the files on any specified disk can be listed.

### **KILL "filename"**

The KILL command deletes the file from the disk. "filename" may be a program file, or a sequential or random access data file. If "filename" is a data file, it must be closed before it is killed.

### **LOAD "filename"[,R]**

The LOAD command loads the program from disk into memory. The option R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADING. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and can access the same data files.

### **MERGE "filename"**

The MERGE command loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory and BASIC-80 returns to Command Mode.

### **NAME "oldfile" AS "newfile"**

To change the name of a disk file, execute the NAME statement, NAME "oldfile" AS "newfile". NAME may be used with program files, random files, or sequential files.

### RESET

RESET reads the directory information off of a newly inserted disk which you have exchanged for the disk in the current default drive. RESET does not close files that were opened on the former default disk. Therefore, use RESET only after you have closed any open files and replaced the current default disk.

### RUN “filename”[,R]

RUN “filename” loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open.

### SAVE “filename”[,A]

The SAVE command writes to disk the program that is currently residing in memory. Thoption writes the program in ASCII format. (Otherwise, BASIC uses a compressed binary format.)

## Protected Files

If you wish to save a program in an encoded binary format, use the “Protect” option with the SAVE command. For example:

```
SAVE "MYPROG", P
```

A program saved this way cannot be listed or edited.

## FILE MANAGEMENT STATEMENTS

BASIC-80 provides a full set of I/O statements to be used for disk file management. These statements are listed below:

<u>Statement</u>	<u>Function</u>
OPEN	Opens a disk file and assigns a file number to the disk file.
CLOSE	Closes a disk file and de-assigns the file number from the disk file.
EOF	Returns $-1$ (true) if the end of a file has been reached.
LOF	Returns the number of records present in the last extent accessed.
LOC	Returns the next record to be accessed for a random file and the total number of sectors accessed for a sequential file.

**Table 10-1**  
File Management Statements.

The OPEN statement is used to assign a file number to a disk file name. Also, the OPEN statement is used to define the mode in which the file is to be used (sequential or random access).

The CLOSE statement performs the opposite function of the OPEN statement. It will de-assign the file number from a disk file name.

The EOF function will return  $-1$  (true) if the end of a sequential file has been reached. The EOF function can also be used with random files to determine the last record number.

The LOF function will return the number of records present in the last extent accessed.

The LOC function, when used with a random file, will return the next record to be accessed. When used with a sequential file, it returns the number of records accessed since the file was opened.

These statements are discussed on the following pages. For a detailed programming example that utilizes these statements, see "Appendix F."

## OPEN (open disk data file)

Form: OPEN "mode",[#]<filenumber>,"<filename>",[,<,reclen>]

where:

"mode" is a string expression whose first character is one of the following mode specification strings:

- O Specifies sequential output mode.
- I Specifies sequential input mode.
- R Specifies random input/output mode.

This string expression will be referred to as the "mode string".

<filenumber> is an integer expression which represents the file number associated with the file. This number will be used in subsequent I/O operations.

<filenumber> must not exceed the number of files that were set during the BASIC-80 initialization process. If no files were set during the initialization process, BASIC-80 will assume a maximum of 3. (See Chapter One, "System Introduction & General Information", for more information about this initialization process.)

"<filename>" is the fully qualified CP/M file name. No extensions are assumed, so the file name must include this information. If no drive is specified, the current default drive is assumed.

<reclen> is an integer expression which, if included, sets the record length for random files. The maximum record length is 256 bytes. The default record length is 128 bytes. If a record length greater than 128 bytes is desired, this length must also be specified when BASIC-80 is initialized. This record length option can only be used with random files. Any attempt to declare the size of a sequential record will result in a "Syntax error".

The OPEN statement is used to associate a file number with a file name. The OPEN statement also defines the mode in which the file will be used (sequential or random access). Subsequent I/O operations will reference the file number assigned to a file name. For example, assume that a file was opened using the following statement:

```
OPEN "I",2,"SAMPLE.DAT"
```

This statement will assign file number 2 to the file SAMPLE.DAT. Because no drive name was specified, BASIC-80 will assume that SAMPLE.DAT resides on the current default drive. The mode string for this file specifies "I" -- sequential input.

If SAMPLE.DAT does not exist on the current default disk, an error will be generated, since input can not be performed on a non-existent file. Now, to input data from this file, the following statement would be used:

```
INPUT#2,<variable list>
```

Note that this INPUT# statement references file number 2, and file number 2 was the number assigned to the file SAMPLE.DAT. (This is only a general form of the INPUT# statement. A detailed discussion of the INPUT# statement appears later in this Chapter.)

o

Now assume that the following OPEN statement is used:

```
OPEN "O",3,"B:OUTPUT.DAT"
```

This will assign file number 3 to the file OUTPUT.DAT. Since the file name does contain the drive specification B:, BASIC-80 will create this output file on drive B:. If this file already exists on drive B:, it will be destroyed, and all previous contents of the file will be lost. Now, to output data to this file, the following statement would be used:

```
WRITE#3,<variable list>
```

The WRITE# statement references file number 3, and file number 3 had been previously assigned to the file B:OUTPUT.DAT. So, the data specified in the <variable list> would be written to the file B:OUTPUT.DAT. (The WRITE# statement is discussed in more detail later in this Chapter.) A file can also be opened for random I/O. One OPEN statement can be used to open the file for both random input and random output. For example, the following statement will open a file for random I/O.

```
OPEN "R",1,"RANDOM.DAT"
```

The file, RANDOM.DAT, is opened for random I/O. If RANDOM.DAT does not exist, it will be created on the current default disk. Now, either random input or random output can be performed with this file. Note that no record size was specified with this OPEN statement. Therefore, BASIC-80 will assume the default record size of 128 bytes. A different record size can be specified with the OPEN statement. (But only for a random access file.)

For example, to open the file RANDOM.DAT for random access, and declare a record size of 32 bytes, the following statement would be used:

```
OPEN "R",1,"RANDOM.DAT",32
```

Now the record size would be 32 bytes. The CP/M sector size is 128 bytes. Therefore, four records would be stored in each CP/M sector. The record size can also be set during the initialization procedure with the /S switch. (See Chapter 1, "System Introduction & General Information," for the initialization procedure.)

It is important to note that the mode a file was opened under must be with the mode in which the file is accessed. For example, consider the following statement:

```
OPEN "I",1,"TEST.DAT"
```

The file TEST.DAT has been opened for sequential input and assigned to file number 1. Now an attempt to perform output on this file would be invalid and would generate an error message. For example:

```
WRITE#1,"HELLO THERE"
```

This WRITE# statement references file number 1. The previously executed OPEN statement has set the mode for file number 1 as sequential input. So this WRITE# would be invalid and would generate an error message.

However, there is an exception to this rule. Under certain circumstances several sequential I/O statements may be used with a random file. The conditions for using these sequential I/O statements with random files are explained in the last part of this chapter.

**CLOSE (close disk data file)**

Form:      CLOSE [#] [<filenumber>]

The CLOSE statement is used to conclude I/O activity to a disk data file.

<filenumber> is the number under which the file was opened. A CLOSE with no arguments will close all open files.

Assume the following OPEN statement appears in a program:

```
OPEN "O", 1, "ARTIST.DAT"
```

Now a sequential output statement may reference this file. When output to this file has concluded, it should be closed with the CLOSE statement.

```
CLOSE #1
```

This statement will disassociate file number 1 from the file ARTIST.DAT. Any reference to file number 1 would now be invalid. The file may then be reopened using the same or a different file number. For example:

```
OPEN "I", 3, "ARTIST.DAT"
```

The file ARTIST.DAT is now associated with file number 3, and is opened for sequential input. Now, a sequential input operation with this file would be valid. When the input operation has concluded, this file should be closed with the CLOSE statement.

```
CLOSE #3
```

The file could again be reopened:

```
OPEN "R", 3, "ARTIST.DAT"
```

The file number 3 has again been associated with the file ARTIST.DAT, but, this time the file has been opened for random I/O.

A CLOSE for a sequential output file writes the final buffer of output to the disk file. (This subject is covered in more detail later in this chapter.)

The END statement and the NEW command will close all disk files automatically. Any attempt to edit or modify a program will also automatically close all open disk files. (The STOP statement does not close disk files.)

**EOF (check for end-of-file)**

Form: EOF(<filenumber>)

<filenumber> is the file number assigned to a disk data file in a previously executed OPEN statement.

The EOF function will return -1 (true) if the end of a sequential file has been reached.

The EOF is useful for detecting when the end of a sequential file has been reached. The EOF function should be used in conjunction with the INPUT# statement and the LINE INPUT# statement to avoid "Input past end" errors.

The EOF function may also be used with random files. If a GET is done past the end of the last sector of the random file, the EOF function will return -1 (true). This may be used to find the size of a random file if record size equals sector size (128 bytes).

Example:

```
10 OPEN "I",1,"DATA"
20 IF EOF(1) THEN 100
30 INPUT#1,A$
40 GOTO 20
.
.
.
100 PRINT "END-OF-FILE REACHED"
```

**LOF (return number of records)**

Form: LOF(<filenumber>)

<filenumber> is the file number assigned to a disk data file in a previously executed OPEN statement.

The LOF Function returns the number of sectors present in the last extent that was accessed. If the file does not exceed one extent, and record length equals sector length (128 bytes), then LOF returns the true length of the file. (Refer to the "CP/M Application Programmer's Manual" for more information on extents.)

Example:

```
110 IF NUM% > LOF(1) THEN PRINT "INVALID ENTRY"
```

**LOC (return record number)**

Form: LOC(<filenumber>)

<filenumber> is the file number assigned to a disk data file in a previously executed OPEN statement.

When used with a random file, the LOC function returns the current record number. The current record number is the number of the last record accessed via GET or PUT. The first time a particular file is accessed, the current record is 1. The largest possible record number is 32767.

When used with a sequential file, the LOC function returns the number of sectors (128 byte blocks) accessed since the file was opened.

Examples:

```
10 OPEN "I",1,"TEST.DAT"
20 OPEN "R",2,"RANDOM.DAT"

.
.

200 PRINT"SECTORS READ---";LOC(1)
210 PRINT"NEXT REC#---";LOC(2)
```

## BASIC-80 SEQUENTIAL I/O

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored, one item after another (sequentially), in the order it is sent and it must be read back in the same order. The data is stored as a stream of ASCII characters.

### Sequential Access Statements

INPUT#	Input data from sequential file.
LINE INPUT#	Input entire line from sequential file.
PRINT#	Write data to sequential file.
PRINT# USING	
WRITE#	Write data to sequential file (with delimiters automatically inserted).

**Table 10-2**  
Sequential Access Statement.

#### INPUT# (input data from sequential file)

Form:      INPUT#<filenumber>,<variable list>

The INPUT# statement is used to read data items from a sequential disk file and assign them to program variables. The data will be read sequentially. When the file is opened, a pointer will be set to the beginning of the file. Each time data is read from the file, the pointer will advance. To start reading over from the beginning of a file, the sequential file must be closed and re-opened.

<filenumber> is the number used when the file was opened for input. <variable list> contains the variable names that the input data will be assigned to. (The variable data type must match the type specified by the variable name. It is invalid to read a string data value into a numeric variable.)

### Numeric Input

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces are ignored.

The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma.

For example, assume the following data image exists on a disk file:

(note: the b represents a blank or space - ASCII 32)

```
bb2.1234b-123.234bb456<carriage return>
```

Then the INPUT statement:

```
INPUT#1,X,Y,Z
```

or the sequence of INPUT statements:

```
INPUT#1,X:INPUT#1,Y:INPUT#1,Z
```

will assign the data values as follows:

```
X=2.1234  
Y=-123.234  
Z=456
```

The following discussion assumes the image on the disk is (note: the b represents a blank or space - ASCII 32):

```
bb2.1234b-123.234bb,456<carriage return>
```

And the INPUT statement used to access the data is:

```
INPUT#1,X,Y,Z
```

The two blanks before the value 2.1234 are leading spaces; therefore, they are ignored. The next character encountered is a 2, and this is considered the start of the first numeric field.

The BASIC-80 I/O processor now scans for the terminator of the first numeric field. The blank between 2.1234 and -123.234 is this terminator. So when BASIC-80 encounters this blank, it assumes that the first numeric field has ended. This first numeric field is assigned to the first item in the variable list, the variable X.

The BASIC-80 I/O processor now scans for the beginning of the second numeric field. The minus sign (-) is considered the start of the second numeric field. The BASIC-80 I/O processor will scan for the terminator of the second numeric field. The comma between -123.234 and 456 is this terminator. So, when BASIC-80 encounters this comma, it assumes that the second numeric field has ended. This second numeric field is assigned to the second item in the variable list, the variable Y.

The BASIC-80 I/O processor now scans for the beginning of the third numeric field. The number 4 is considered the start of the third numeric field. The BASIC-80 I/O processor will then scan for the terminator of the third numeric field. The carriage return after 456 is this terminator. So when BASIC-80 encounters this carriage return, it assumes that the third numeric field has ended. This third numeric field is assigned to the third item in the variable list, the variable Z.

At this point, all three variables in the variable list have values assigned to them, so execution of the INPUT statement has been completed. Execution continues with the next statement.

### String Input

When BASIC-80 scans the sequential data file for a string item, leading spaces, carriage returns, and line feeds are ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item.

This string is considered an unquoted string, and will terminate on a comma, carriage return or line feed (or after 255 characters have been read).

If this first character is a quotation mark, the string is considered a quoted string. The string item will consist of all characters read between the first quotation mark and the next quotation mark. Commas, blanks, and carriage return characters can be included in this string. A quoted string may not contain a quotation mark within the quoted string.

For example, assume the following data image exists on a disk file:

(b represents a blank or space -- ASCII 32)

```
BENTON,HARBOR,MI"49022"<carriage return>
```

Then the statement:

```
INPUT#1,A$,B$,C$
```

would assign the data values as follows:

```
A$=BENTON  
B$=HARBOR  
C$=MI"49022"
```

Note that the comma is used as the terminator in the above example. All three strings are considered to be unquoted strings.

In the last string field, the quotation mark is considered as part of the string. This is because the string starts with the letter M and is terminated by a carriage return.

Assume a comma is inserted between MI and "49022". The disk image would then look like this:

```
BENTON,HARBOR,MI,"49022"
```

Now there are a total of four string fields. The first three are unquoted strings fields, and the last is a quoted string field. These four fields could be input with the following statement:

```
INPUT #1,A$,B$,C$,D$
```

the variable values would be assigned as follows:

```
A$=BENTON  
B$=HARBOR  
C$=MI  
D$=49022
```

The variable D\$ would not contain the quotation marks because the quotation marks were used to terminate the field, and as such they do not represent data values.

**LINE INPUT# (input entire line from sequential file)**

Form: LINE INPUT#<filenumber>,<string variable>

The LINE INPUT# statement is used to read an entire line (up to 255 characters), without delimiters, from a sequential disk data file to a string variable.

<filenumber> is the file number assigned to the file with the OPEN statement. The file must be opened for sequential input (I mode). <variable> is the variable name to which the input will be assigned.

LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

If no carriage return is found, LINE INPUT# will read until 255 characters have been read. These 255 characters will then be assigned to the string variable.

LINE INPUT# is especially useful if each field of a data file has been terminated with a carriage return, or if a BASIC-80 program saved in ASCII mode is being read as data by another program.

For example, assume the following program exists in a disk file:

```
10 OPEN "0",1,"LIST" <carriage return>
20 INPUT C$ <carriage return>
30 PRINT #1, C$ <carriage return>
40 CLOSE #1 <carriage return>
```

then the statement:

```
LINE INPUT#1,Z$
```

could be repetitively used to read each program line, one line at a time.

## PRINT# AND PRINT# USING (write to sequential disk file)

Forms:

PRINT#<filenumber>,<list of expressions>

PRINT#<filenumber>,USING<string exp>;<list of expressions>

The PRINT# statement is used to write data to a sequential disk file.

<filenumber> is the number used when the file was opened for output. The expressions in <list of expression> are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal with a PRINT statement. (The PRINT statement is discussed in Chapter Four, "Program Statements.") For this reason, take care to delimit the data on the disk so it will be input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semi-colons.

For example:

PRINT#1,A;B;C;X;Y;Z

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let A\$="CAMERA" and B\$="93604-1".

The statement:

PRINT#1,A\$;B\$

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

PRINT#1,A\$;" ";"B\$

The image written to disk is:

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, CHR\$(34).

For example, let A\$="CAMERA, AUTOMATIC" and B\$=" 93604-1". The statement:

```
PRINT#1,A$;B$
```

would write the following image to disk:

```
CAMERA, AUTOMATIC 93604-1
```

and the statement:

```
INPUT#1,A$,B$
```

would input "CAMERA" to A\$ and "AUTOMATIC 93604-1" to B\$. To separate these strings properly on the disk, write double quotes to the disk image using CHR\$(34).

The statement:

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to disk:

```
"CAMERA, AUTOMATIC" " 93604-1"
```

and the statement:

```
INPUT#1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A\$ and " 93604-1" to B\$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$###.##.";J;K;L
```

The comma at the end of the format string serves to separate the items in the disk file. (For a complete discussion of the PRINT USING statement, refer to Chapter Eight, "Special Features.")

NOTE: The WRITE# statement will automatically insert the proper delimiters between data items in a sequential file.

### **WRITE#(write to sequential disk file)**

Form:            **WRITE#<filenumber>,<list of expressions>**

The WRITE# statement is used to write data to a sequential file.

<filenumber> is the number which was assigned to the file with an OPEN statement. The file must be open for sequential output ( O mode). The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the variable list is written to the disk file.

Example:    Let A\$="CAMERA" and B\$="93604-1". The statement:

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

would input "CAMERA" to A\$ and "93604-1" to B\$.

Note: The WRITE# statement is recommended for most applications using sequential output. Most problems arising from using sequential files are a result of not inserting the proper delimiters between data items. The WRITE# statement eliminates the need to be concerned with delimiting data items, thus eliminating most problems associated with sequential I/O.

In those cases where the WRITE# statement will not provide the flexibility needed for some unique sequential output application, use of the PRINT# or PRINT# USING statement should be considered. Care should be taken to insure that all the data items are separated by the proper delimiters.

## Sequential Access Techniques

### CREATING AND ACCESSING A SEQUENTIAL FILE

The following program steps are required to create a sequential file and access the data in the file:

---

**Open the file for sequential output.**

```
OPEN "O", #1, "DATA.DAT"
```

This step will associate the file number 1 with the file DATA.DAT. Because the O mode string was specified, the file will be opened for sequential output. Since no drive specification was included with the file name, the current default drive will be assumed.

If a file DATA.DAT already exists on the current default drive, contents of this file will be lost. This is due to the fact that, when a file is opened for sequential output, the BASIC-80 I/O processor will move the EOF marker to the beginning of the file. Thus, the previous contents of the file can no longer be accessed.

---

**Write data to the file**

```
WRITE#1, A$, B$, C$
```

This step assumes that some string value has been assigned to the string variables A\$, B\$ and C\$. The WRITE# statement will write data to the file with delimiters, so it is not necessary to insert any delimiters.

The PRINT# statement could have been used to write the data to this sequential file, but then it would have been necessary to insert delimiters between the data items. So for most applications using sequential output, it is more efficient to use the WRITE# statement.

---

**Close the file**

```
CLOSE#1
```

This statement will write any remaining data from the buffer to the disk file. Output to this file will then be terminated. The file must be closed before it can be reopened for sequential input.

---

**Reopen the file for input**

```
OPEN "I",#1,"DATA.DAT"
```

The file number 1 is again associated with the file DATA.DAT. This time, the file is opened for sequential input.

---

**Read the data**

```
INPUT#1,X$,Y$,Z$
```

The data will be read from the file DATA.DAT and assigned to the string variables X\$, Y\$ and Z\$

---

**NOTE:** The above example ignores the role of the I/O buffer in the sequential I/O process. Actually, BASIC-80 reads and writes in 128-byte blocks. So each INPUT# or WRITE# statement may not necessarily require a disk access.

With sequential output, each WRITE# or PRINT# will place the data in the buffer area. When the buffer is filled with data, the data will actually be written to the disk file.

With sequential input, 128 bytes will be read and placed in the buffer area. Then the BASIC-80 I/O processor will sort through the data in the buffer to satisfy the INPUT# statement variable list.

## ADDING DATA TO A SEQUENTIAL FILE

As soon as an existing sequential file is opened for output ("O" mode), the current contents of the file are destroyed. Thus, several program steps are required to add data to an existing sequential file. The following procedure can be used to add data to an existing file called "DATA.DAT"

---

### Open "DATA.DAT" for sequential input

```
OPEN "I", 1, "DATA.DAT"
```

This step associates file number 1 with the data file DATA.DAT. This file will be opened for sequential input. Since no drive specification was included with the file name, BASIC-80 will assume the current drive. If the file DATA.DAT can not be found on the current default drive, a "File not found" error will be generated.

---

### Open a second file called "TEMP.TMP" for sequential output

```
OPEN "O", 2, "TEMP.TMP"
```

The file, TEMP.TMP will be used as a temporary work file. After this process is completed, this file will be renamed and it will contain the original data as well as the newly created data.

---

### Read in the data in "DATA.DAT" and write it to "TEMP.TMP"

```
INPUT#1, A$, B$, C$  
WRITE#2, A$, B$, C$
```

This step must be repeatedly executed until all the data in file #1 is read.

---

**Close “DATA.DAT” and kill it.**

```
CLOSE#1  
KILL "DATA.DAT"
```

This file is no longer needed, as the information from this file has been copied into the file TEMP.TMP

---

**Write the new information to “TEMP.TMP”**

```
WRITE#2,A$,B$,C$
```

The data assigned to the string variables A\$,B\$ and C\$ will be written to the disk file.

---

**Close the file**

```
CLOSE#2
```

This step will terminate the output operation performed with this file.

---

**Rename “TEMP.TMP” as “DATA.DAT”**

```
NAME "TEMP.TMP" AS "DATA.DAT"
```

Now there is a file on disk called “DATA.DAT” that includes all the previous data plus the new data that was added to the file.

## BASIC-80 RANDOM I/O

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk because BASIC-80 stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk — it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

All data stored in a random file must be a string data type.

To store numeric values in a random file, the numeric values must be converted to strings. Several functions have been provided to convert numeric values to strings. These functions, (MKI\$, MKS\$, MKD\$), are explained later in this Chapter.

## Random Access Statements

<u>Statement</u>	<u>Function</u>
FIELD	Set up random file buffer.
LSET	Move data to random buffer. (left-justified)
RSET	Move data to random buffer. (right-justified)
GET	Read random record.
PUT	Write random record.
MKI\$	Make integer into 2-byte string.
MKS\$	Make single-precision number into 4-byte string.
MKDS\$	Make double-precision number into 8-byte string.
CVI	Convert 2-byte string to integer.
CVS	Convert 4-byte string to single-precision number.
CVD	Convert 8-byte string to double-precision number.

**Table 10-3**  
Random Access Statements.

### **FIELD (set up random file buffer)**

Form:

**FIELD#<filenumber>,<field width> AS <string variable>**

The FIELD statement is used to allocate space for variables in a random file buffer.

<filenumber> is the number assigned to the random file in the OPEN statement. <field width> is the number of characters (bytes) to be allocated to <string variable>.

For example:

**FIELD#1, 20 AS N\$, 10 AS ID\$, 40 AS ADD\$**

allocates the first 20 positions (bytes) in the random file buffer to the string variable N\$, the next 10 positions to ID\$, and the next 40 positions to ADD\$. FIELD does **not** place any data in the random file buffer, but instead defines the fields in the random file buffer.

A FIELD statement can only reference a file which has been opened for random I/O (R mode). The FIELD statement must also be executed prior to performing any I/O operation with the random file.

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was opened. Otherwise, a "Field overflow" error occurs. (The default record length is 128.)

If a number smaller than 128 is specified for the record length, the BASIC-80 I/O processor will take care of blocking and deblocking the record. For example, if a record length of 32 bytes is specified in the OPEN statement, the BASIC-80 I/O processor will block 4 of these logical records per physical record (sector). The user program is not responsible for blocking and deblocking these logical records.

If a number greater than 128 is specified for the record length, the BASIC-80 will also take care of blocking and deblocking the record. This number must be specified by using the /S switch when initializing BASIC-80. The largest record size allowed is 256 bytes.

With previous versions of Microsoft BASIC, the user program did have to assume responsibility for blocking and deblocking records.

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time. For example, the following FIELD statement could be used to define a 32-byte random buffer:

```
FIELD#1, 16 AS F1$, 16 AS F2$
```

This FIELD statement would allocate the first 16 characters (bytes) of the random buffer to the variable F1\$ and the next 16 characters (bytes) to the variable F2\$. Then, another FIELD statement could be used to redefine the buffer:

```
FIELD#1, 32 as BUFF$
```

So the variable BUFF\$ would refer to all 32 characters in the buffer. F1\$ would still refer to the first 16 characters and F2\$ would still refer to the second 16 characters.

Do **not** use a fielded variable name in an INPUT or LET statement. Once a variable name is fielded, it points to a specific address in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

Examples:

```
FIELD#1, 128 AS IBUFF$
```

```
FIELD#4, 10 AS A$(1), 10 AS A$(2), 10 AS A$(3)
```

```
FIELD#2, I AS STUFF$
```

(Note: the variable I must be assigned an integer value prior to the execution of this statement.)

### LSET/RSET (move data to random buffer)

Forms: LSET <fielded variable> = <string expression>

RSET <fielded variable> = <string expression>

The LSET/RSET statements are special assignment statements used to assign a string expression to a variable that has appeared in a FIELD statement (fielded variable).

The LSET/RSET statements are used to move data from memory to a random file buffer. This step is performed in preparation for a PUT statement. The only way to move data to a random buffer is by using the LSET/RSET statement.

If the <string expression> requires fewer bytes than were fielded to the <fielded variable>, LSET left-justifies the string in the field by adding spaces on the right. RSET is used to right-justify the string in the field by adding spaces on the left.

The only difference between LSET and RSET is the fact that LSET left-justifies the field and RSET right-justifies the field. If the string is too long for the field, characters are dropped from the right.

Numeric values must be converted to strings before they are LSET or RSET. Several special random I/O functions have been provided to perform this conversion. (Refer to the discussion of the MKI\$, MKS\$, and the MKD\$ functions later in this Chapter.)

#### Examples:

```
150 LSET A$=MKS$(AMT)
160 LSET D$=DESC$
170 LSET V$="LEFT-JUSTIFY AND PLACE IN BUFFER"
180 RSET G$="RIGHT-JUSTIFY AND PLACE IN BUFFER"
```

String variables A\$, D\$, V\$ and G\$ must have appeared in a previously executed FIELD statement.

**GET (read random record)**

Form:           **GET [#]<filenumber>[,<record number>]**

The GET statement is used to read a record from a random disk file into a random buffer. Before executing a GET statement, the file to be accessed must be opened for random I/O.

Additionally, the random file buffer must have been defined with a FIELD statement. If the random file buffer has not been defined, there will be no way to access the data after the GET has been executed.

<filenumber> is the number under which the file was opened. If <record number> is omitted, the current record is read into the buffer. The current record is the record number one greater than that of the last record accessed. The first time a particular file is accessed, the current record is 1. The largest possible record number is 32767.

If an attempt is made to GET a record whose number is higher than that of the last record number in the file, the buffer will be filled with NUL characters (ASCII Ø), although no error will be generated. The LOF function can be used to prevent this from occurring.

Examples:

GET#1,100

GET#2

GET FILE,IREC

GET#5,REC

## PUT (write random record)

Form:      PUT [#]<filenumber>[ ,<record number> ]

The PUT statement is used to write a record from a random buffer to a random disk file. Before executing a PUT statement, the file to be accessed must be opened for random I/O.

Additionally, the random file buffer must have been defined with a FIELD statement. If the random file buffer has not been defined, there will be no way to move data into the buffer before executing the PUT statement.

<filenumber> is the number under which the file was opened. If <record number> is omitted, the current record is written. The current record is the record number one greater than that of the last record accessed. The first time a particular file is accessed, the current record is 1. The largest possible record number is 32767.

If the <record number> is higher than the end-of-file record number, <record number> becomes the new end-of-file record number. Space will be allocated on the disk to accommodate the new end-of-file record, as well as all lower numbered records.

Before executing a PUT statement, the data to be written to a disk file must be moved into the buffer area. The LSET/RSET statements are used to move the data to the random file buffer.

Examples:

PUT#1

PUT#2,43

PUT I,J-1

PUT I,4

**MKI\$, MKS\$, MKD\$      (make a numeric value into a string)**

Forms:

MKI\$(<integer expression>)
MKS\$(<single-precision expression>)
MKD\$(<double-precision expression>)

The "make" functions, (MKI\$, MKS\$, MKD\$) are used to convert numeric value to string value. Any numeric value that is placed in a random file buffer must be converted to a string.

The MKI\$ function is used to convert an integer to a 2-byte string. The integer expression must be in the allowable range for integer values. If it is not, an "Illegal function call" error will be generated. Any fractional portion of the number will be truncated.

The MKS\$ function is used to convert a single-precision number to a 4-byte string. The MKD\$ function is used to convert a double-precision number to an 8-byte string.

These functions will not move the data to the random buffer. So after a numeric value is converted to a string, it still must be moved to the random file buffer. Additionally, the random file buffer must have been defined with a FIELD statement.

If the random file buffer has not been defined, there will be no way to access the data after the GET has been executed. The data must also be moved into the random buffer using LSET or RSET.

For example, to convert the integer variable IV% to a string and assign it to the field variable FV\$, the following single program statement could be used:

LSET FV\$ = MKI\$(IV%)

The variable FV\$ should have appeared in a previously executed FIELD statement.

Example:

```
90 AMT=(K+T)
100 FIELD #1, 8 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
```

## CVI, CVS, CVD (Converting string to numeric form)

Forms: CVI (<2-byte string>  
CVS (<4-byte string>  
CVD (<8-byte string>)

The CVI, CVS and CVD functions are used to convert string values to numeric values. These functions are generally used to convert numeric values that have been read from a random disk file. Data is always stored in random files as a string data type. Therefore, a numeric value read from a random disk file must be converted from a string back into a number.

The CVI function converts a 2-byte string to an integer. If the length of the string is greater than 2 bytes, only the first two characters in the string will be used. If the length of the string is less than 2 bytes, an “Illegal function call” error will result.

The CVS function converts a 4-byte string to a single-precision number. If the length of the string is greater than four bytes, only the first four characters in the string will be used. If the length of the string is less than four bytes, an “Illegal function call” error will result.

The CVD function converts an 8-byte string to a double-precision number. If the length of the string is greater than eight bytes, only the first eight characters in the string will be used. If the length of the string is less than eight bytes, an “Illegal function call” error will result.

Example:

```
PRINT CVS(A$)  
A#=CVD(BUFF$)  
I = I+CVI(I$)
```

## Random Access Techniques

### CREATING A RANDOM ACCESS FILE

The following program steps are required to create a random file.

---

#### OPEN the file for random access

```
OPEN "R", 1 "FILE.DAT", 32
```

In this example, the mode string specifies "R"—random access. File number 1 is assigned to the file FILE.DAT. Since no drive specification was included with this file name, the current default drive is assumed. This example also specifies a record length of 32 characters (bytes). If the record length is omitted, the default record length is 128 characters (bytes).

---

#### Set up the random file buffer

```
FIELD#1, 20 AS NAME$, 4 AS A$, 8 AS P$
```

Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file. The FIELD statement references file number 1, which has been opened for random input. (It is invalid to FIELD a file which has been opened for sequential input or output.)

This FIELD statement will allocate the first 20 characters of the random file buffer for the variable NAME\$, the next four characters for the variable A\$, and the next eight characters for the variable P\$.

---

### Move the data into the random buffer

```
LSET NAME$=X$  
LSET A$=MKS$( AMT )  
LSET P$=TEL$
```

Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the “make” functions: MKI\$ to make an integer value into a string, MKS\$ for a single-precision value, and MKD\$ for a double-precision value:

In this program step, the single-precision variable AMT is first converted to a string, and then it is assigned to the variable A\$. The variable A\$ has appeared in a previous FIELD statement. The FIELD statement was used to allocate four characters (bytes) to the variable A\$.

---

### Write data to disk

```
PUT#1
```

Write the data from the buffer to the disk using the PUT statement. No record number was specified with this PUT statement, so the current record number will be written. The current record is the record number one higher than the last record accessed. The first time a file is accessed, the current record is one.

---

Do not use a fielded string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

## ACCESSING A RANDOM ACCESS FILE

The following program steps are required to access a random file:

---

### OPEN the file for random access

```
OPEN "R",#1,"FILE.DAT",32
```

This step will open the file “FILE.DAT” for random access. The file can now be accessed by referring to file number 1.

---

### Set up random file buffer

```
FIELD#1, 20 AS NAME$, 4 AS A$, 8 AS P$
```

Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file. In this example, 20 characters (bytes) are allocated to the string variable NAME\$, four characters are allocated to the string variable A\$, and eight characters are allocated to the string variable P\$.

NOTE: In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

---

### Read data into buffer

```
GET#1
```

Use the GET statement to move the desired record into the random buffer. No record number was specified with this GET statement, so the current record number will be read. The current record is the record number one higher than the last record accessed. The first time a file is accessed, the current record is one.

---

### Access data in the buffer

The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single-precision values, and CVD for double-precision

```
PRINT NAME$  
AV=CVS(A$)  
DP#=CVD(P$)
```

---

## Additional Features

After a GET statement, INPUT# and LINE INPUT# may be used to read characters from the random file buffer. PRINT#, PRINT# USING, and WRITE# may also be used to put characters in the random file buffer before a PUT statement.

In the case of WRITE#, BASIC-80 pads the buffer with spaces (if necessary) and then inserts a carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.



## Chapter Eleven

# Microsoft BASIC-80 Summary

## OVERVIEW

This Chapter is a summary of the important concepts, ideas, keywords, etc. of the BASIC-80 programming language. The various intrinsic functions as well as the string functions are also included in this chapter.

## Abbreviations

<u>Abbreviation</u>	<u>Function</u>
?	Use in place of PRINT.
,	Use in place of REM.
.	"current line"; use in place of line number with LIST, EDIT, etc.

## Data Type Declaration Characters

<u>Character</u>	<u>Data Type</u>	<u>Examples</u>
\$	String	ZDS\$, WLW\$
%	Integer	I%, VALUE%
!	Single-Precision	V!, FLAG!
#	Double-Precision	DP#, PL#
D	Double-Precision (exponential notation)	1.23456789D-12
E	Single-Precision (exponential notation)	1.23456E+23

## Arithmetic Operators

<u>Operator</u>	<u>Operation Performed</u>
+	Addition
-	Subtraction
*	Multiplication
/	Division (floating point)
\	Integer division
^	Exponentiation

## String Operator

<u>Operator</u>	<u>Operation Performed</u>	<u>Example</u>
+	concatenate (string together)	"A"+“B”+“C”

## Relational Operators

<u>Operator</u>	<u>Numeric Expressions</u>	<u>String Expressions</u>
<	Less than	Precedes
>	Greater than	Follows
=	Equal to	Equals
<= or =<	Less than or equal to	Precedes or equals
>= or =>	Greater than or equal to	Follows or equals
<> or ><	Does not equal	Does not equal

## Logical Operators

<u>Operator</u>	<u>Function</u>
NOT	Bitwise negation
AND	Bitwise disjunction
OR	Bitwise conjunction
XOR	Bitwise exclusive OR
IMP	Bitwise implication
EQV	Bitwise equivalence

## Commands

<u>Command/Function</u>	<u>Examples</u>
-------------------------	-----------------

AUTO <line number>,<increment>

Enable automatic line numbering starting at <line number> and incrementing by <increment>.

AUTO  
AUTO 10  
AUTO 5,5

CLEAR

Set numeric values to zero, strings to null.

CLEAR

CLEAR,<expression>

Same as CLEAR, but <expression> is used to set the high memory limit for use by BASIC-80.

CLEAR ,32768

CLEAR,<expression1>,<expression2>

Same as CLEAR<expression> but <expression2> is used to set the amount of stack space for use by BASIC-80.

CLEAR ,32768 ,2000

CONT

Continues program execution after a BREAK or STOP.

CONT

DELETE <line number>

Deletes the specified line number in the current program.

DELETE 100

DELETE -<line number>

Deletes every line of the current program up to and including <line number>.

DELETE -500

<u>Command/Function</u>	<u>Examples</u>
DELETE <line number>-<line number>	
Deletes all lines of the current program up to and including the second number.	DELETE 10-1000
EDIT <line number>	
Enter Edit Mode at the specified line number.	EDIT 100
FILES "<filename>"	
List names of files residing on the current disk.	FILES "* .BAS"
LIST	
List the program currently in memory starting with the lowest numbered line.	LIST
LIST <line number>	
List the specified line number.	LIST 100
LIST <line number>-<line number>	
List all lines from the first line up to and including the second.	LIST 10-100
LLIST	
List all or part of the program currently in memory. The listing will be printed on the line printer. The options for the LLIST command are the same as for the LIST command.	LLIST LLIST 500 LLIST 150- LLIST -100 LLIST 150 — 400

Command/FunctionExamples

LOAD &lt;"filename"&gt;,R

Load a program file from disk into memory. The R is optional, and if used will run the program after it is loaded.

```
LOAD"B:GAME"
LOAD"PROG.ASC",R
```

MERGE &lt;"filename"&gt;

Merges a disk file into a program in memory.

```
MERGE"B:TEST.BAS"
```

NEW

Deletes the current program and clears all variables.

```
NEW
```

RENUM &lt;nn&gt;,&lt;mm&gt;,&lt;ii&gt;

Renumerates program lines starting at line <mm>, as line <nn>, with increments of <ii>.

```
RENUM
RENUM 300,,5
RENUM 1000,900,20
```

RESET

Changes disk in default drive.

```
RESET
```

RUN &lt;line number&gt;

Executes the current program starting with specified line number. If line number is not specified, execution starts at the lowest line number.

```
RUN 100
RUN
```

RUN &lt;"filename"&gt;,R

Loads a program from disk and executes it. R keeps all data files open.

```
RUN "PROG1"
RUN"B:GAME",R
```

Command/Function

SAVE "filename",A  
SAVE "filename",P

Saves the current program on disk. If A is used, the file is saved in ASCII format. If P is used, the file is saved in a protected format. If neither the P or A is used, the file is saved in a compressed binary format.

Examples

SAVE"COM2",A  
SAVE"TEST1"  
SAVE"INVEN",P

**SYSTEM**

Closes all files and performs a CP/M warm start.

**SYSTEM**

## Edit Mode Subcommands and Functions

<u>Command</u>	<u>Function</u>
RETURN	End editing and return to Command Mode.
<i>Space Bar	Move cursor <i> spaces to the right.
<i>Back Space	Move cursor <i> spaces to the left.
L	List remainder of program line and return cursor to the beginning of the program line.
X	List remainder of program line, move cursor to the end of the line, and go into Insert Mode.
I	Insert text beginning at the current position of the cursor. Use ESC to exit Insert Mode.
A	Cancel editing changes and return cursor to beginning of line.
E	End editing, save all changes and return to Command Mode.
Q	End editing, cancel all changes and return to Command Mode. -
H	Delete remainder of line and then enter Insert Mode.
<i>D	Delete specified number of characters <i> beginning at current cursor position.
<i>C	Change (or replace) the specified number of characters <i> using the next <i> characters entered.
<i>S<c>	Move the cursor to the <i>th occurrence of character <c>, counting from the current cursor position.
<i>K<c>	Delete all characters from the current cursor position up to the <i>th occurrence of character <c>.

# Print Using Format Field Specifiers

<u>Numeric Specifier</u>	<u>Function</u>	<u>Example</u>
#	Numeric field.	###
.	Decimal point position.	##.##
+	Print leading or trailing signs (plus for positive numbers, minus for negative numbers).	+##.##
-	Print trailing sign only if value printed is negative.	##.##-
**	Fill leading blanks with asterisks.	**##.##
\$\$	Place dollar sign immediately to left of leading digit.	\$\$###.##
**\$	Asterisk fill and floating dollar sign.	**\$###.##
,	Use comma every three digits (left of decimal point only).	##,###.##
^^^^	Exponential format. Number is aligned so leading digit is non-zero.	.##^^^^

<u>Literal Specifier</u>	<u>Function</u>	<u>Example</u>
—	Literal character string field.	

## Program Statements

### Statement/Function

### Examples

#### DATA TYPE DEFINITION

DEFINT <letter range>

Declare range of variable names as integer data types.

DEFINT I-N

DEFSNG <letter range>

Declare range of variable names as single-precision data types.

DEFSNG A-H, O-P

DEFDBL <letter range>

Declare range of variable names as double-precision data types.

DEFDBL X, Y, Z

DEFSTR <letter range>

Declare range of variable names as string variables.

DEFSTR A-C, Z

#### ASSIGNMENT AND ALLOCATION

DIM <list of subscripted variables>

Allocate storage for array.

DIM A(20), B(12,2)

OPTION BASE n

Declare minimum value for array subscript. The default base is 0. This may be changed to 1.

OPTION BASE 1

Statement/FunctionExamples

ERASE &lt;list of array names&gt;

Remove an array from the program.

ERASE A,B

LET &lt;variable&gt; = &lt;expression&gt;

Assign value of expression to variable.

LET SUM = A+B+C

REM &lt;remark&gt;

Insert remark into program.

REM GRP IS GROSS PAY

SWAP &lt;variable&gt;,&lt;variable&gt;

Exchange the values of two variables.

SWAP A,B

**SEQUENCE OF EXECUTION**

END

Terminate program execution, close all files and return to Command Mode.

100 END

FOR &lt;V&gt;=&lt;X&gt; TO &lt;Y&gt; STEP &lt;Z&gt;

Allows repetitive execution of a series of statements.

FOR I = 1 TO 100

GOSUB &lt;line number&gt;

Branch to subroutine beginning at <line number>.

GOSUB 100

GOTO &lt;line number&gt;

Branch to specified line number.

GOTO 400

NEXT &lt;variable&gt;

Terminates a FOR loop.

NEXT I

Statement/FunctionExamples

ON <expression> GOTO line1,...linek

Evaluate expression. If  
INT(<expression>) equals  
one of the numbers 1-k,  
branch to appropriate  
line number. If it is  
not equal, go to the  
next statement.

ON L1 GOTO 10,20,30

ON <expression> GOSUB line1,...linek

Same as ON...GOTO except  
branch is to a subroutine.

ON L GOSUB 300,400

RETURN

Terminates a subroutine.  
Branches to the statement  
following the most recent  
GOSUB.

RETURN

STOP

Terminates program execution  
and returns to Command Mode.

STOP

## CONDITIONAL EXECUTION

IF <expression> THEN <statement(s)>  
> ELSE <statement(s)>

Evaluate <expression>: If true,  
execute THEN clause. If false,  
execute ELSE clause. (if present)

IF A=0 THEN A=1  
ELSE A=0

<u>Statement/Function</u>	<u>Examples</u>
<pre>WHILE &lt;expression&gt;     &lt;loop statements&gt; WEND</pre> <p>Executes a series of statements in a loop as long as a given condition is true.</p>	<pre>WHILE A=0 PRINT "ZERO"</pre>

### NON-DISK I/O STATEMENTS

**INPUT <\*> <"prompt string">;<list of variables>**

Inputs data from the terminal during program execution.

```
INPUT "AGE";A
```

**LINE INPUT <\*> <"prompt string">;<string variable>**

Inputs an entire line (up to 255 characters) to a string variable, without the use of delimiters.

```
LINE INPUT J$
```

**DATA <list of constants>**

Stores numeric and string constants. These constants are assigned to variables by using the READ statement.

```
DATA 34,23.1,45.0
DATA "HELLO","BYE"
```

**PRINT <list of expressions>**

Outputs data on the terminal.

```
PRINT "HELLO"
PRINT A$,Z,C
```

**READ <list of variables>**

Reads data into specified variables from a DATA statement. -

```
READ I,A,B
READ A$,B$
```

<u>Statement/Function</u>	<u>Examples</u>
RESTORE <line number>	RESTORE
Resets DATA pointer so that data may be reread.	
LPRINT <list of expressions>	LPRINT "HELLO"
Prints data on the line printer.	

## String Functions

<u>Function</u>	<u>Operation</u>	<u>Example</u>
ASC(X\$)	Returns ASCII code of first character in string argument.	ASC("B") ASC(H\$)
CHR\$(I)	Returns a one-character string whose character has the ASCII code of I.	CHR\$(66) CHR\$(N)
HEX\$(X)	Converts a number to a Hexadecimal string.	HEX\$(100) HEX\$(A)
INKEY\$	Reads one character from the keyboard.	A\$=INKEY\$
INPUT\$(X,Y)	Reads X characters from the keyboard or from file number Y.	INPUT\$(1,1)
INSTR(I,X\$,Y\$)	Returns the position of the first occurrence of Y\$ in X\$ starting at position I.	INSTR(A\$,"")
LEFT\$(X\$,I)	Returns left-most I characters of the string expression X\$.	LEFT\$(A\$,1) LEFT\$(C\$,3)
LEN(X\$)	Returns length of string X\$.	LEN(A\$)
MID\$(X\$,I,J)	Returns string of length J characters from X\$ beginning with the Ith character.	MID\$(X\$,5,10)

<u>Function</u>	<u>Operation</u>	<u>Example</u>
MID\$(X\$,I,J)=Y\$	Replaces the characters in X\$, beginning at position I, with the characters in Y\$. J is the number of characters to use in the replacement.	MID\$(A\$,1,2)="Z"
OCT\$(X)	Converts the numeric expression X to an octal string.	OCT\$(24)
RIGHT\$(X\$,I)	Returns the right-most I characters of string X\$.	RIGHT\$(X\$,8)
SPACE\$(X)	Returns a string of X spaces.	SPACE\$(20)
STR\$(X)	Converts a numeric expression to a string.	STR\$(100)
STRING\$(I,J)	Returns a string of length I containing characters with the ASCII code J.	STRING\$(20,33)
STRING\$(I,X\$)	Returns a string of length I containing the first character of string X\$.	STRING\$(20,"!")
VAL(X\$)	Converts the string X\$ to a numeric value.	VAL("3.14")

## Arithmetic Functions

<u>Function</u>	<u>Operation</u>	<u>Example</u>
ABS(X)	Returns absolute value.	ABS(-1)
ATN(X)	Returns arctangent of X. (X must be in radians.)	ATN(3)
CDBL(X)	Converts X to double-precision.	CDBL(A)
CINT(X)	Converts X to an integer by rounding.	CINT(46.6)
COS(X)	Returns the cosine of X. (X must be in radians)	COS(A+B)
CSNG(X)	Converts X to single-precision.	CSNG(V)
EXP(X)	Returns e to the power of X.	EXP(34.5)
FIX(X)	Returns truncated integer portion of X.	FIX(23.2)
INT(X)	Returns largest integer not greater than X.	INT(-12.11)
LOG(X)	Returns the natural logarithm of X. X must be greater than zero.	LOG(45/7)
RND(X)	Returns a random number between 0 and 1.	RND(0)
SGN(X)	Returns -1 for negative X, 0 for zero X, +1 for positive X.	SGN(C/A)
SIN(X)	Returns the sine of X. (X must be in radians.)	SIN(A*1.3)
SQR(X)	Returns the square root of X. X must be non-negative.	SQR(A*B)
TAN(X)	Returns the tangent of X. (X must be in radians.)	TAN(X+Y+Z)

## Special Functions

<u>Function</u>	<u>Operation</u>	<u>Example</u>
FRE(X)	Returns memory space not used by BASIC-80.	FRE(0)
INP(I)	Returns the byte read from port I.	INP(255)
LPOS(X)	Returns current position of line printer print head within the line printer buffer.	LPOS(0)
NULL(X)	Sets the number of nulls to be printed at the end of each line.	NULL(3)
OUT I,J	Sends byte J to port I.	OUT 127,255
PEEK(I)	Reads a byte from the specified memory address.	PEEK(8192)
POKE I,J	Puts byte J into memory location I.	POKE(8192,200)
POS(X)	Returns current cursor position.	POS(1)
SPC(I)	Prints I spaces on the terminal.	PRINT SPC(5)
TAB(I)	Tabs carriage to specified position.	PRINT TAB(20)
VARPT(X)	Returns address of variable in memory.	VARPTR(V)
WAIT I,J[,K]	Status of port I is XOR'ed with K and AND'ed with J. Continued execution awaits non zero result.	WAIT 21,1
WIDTH I	Sets the printed line width.	WIDTH 80
WIDTH LPRINT I	Sets the line printer width.	WIDTH LPRINT 132

## Special Features

### ERROR TRAPPING

<u>Statement/Function</u>	<u>Example</u>
ON ERROR GOTO <line number>	ON ERROR GOTO 100
Enables error trapping and specifies the first line of the error trapping subroutine.	
RESUME <line number>	RESUME. RESUME NEXT RESUME 100
Continues program execution after an error recovery procedure has been performed.	
ERROR <integer expression>	ERROR 10
Simulates the occurrence of an error, also allows error codes to be defined by user.	
ERL	PRINT ERL
Error line number.	
ERR	PRINT ERR
Error code number.	
<b>TRACE FLAG</b>	
TRON	TRON
Enables trace flag.	
TROFF	TROFF
Disables trace flag.	

Statement/FunctionExample**OVERLAY MANAGEMENT**

CHAIN [MERGE]"<filename>"[,<line number>]  
[,ALL][,DELETE<range>]]

Calls program and passes  
variables from the current  
program.

CALL "PROG"

COMMON <list of variables>

Pass variables to a chained  
program.

COMMON A,B

## Disk Input/Output Statements

### Statement/Function

### Example

**CLOSE#[<filenumber>[,<filenumber>]]**

Closes disk files. If no argument is supplied, all open files are closed.

CLOSE #6

**FIELD# <filenumber>,<field size>  
AS <string variable>**

Allocates random buffer space to <string variable>, where <file number> is the random buffer referenced, and <field size> is the space reserved for a given <string variable>.

FIELD #1,3 AS A\$

**GET#<file number>[,<record number>]**

Transfers data from the <record number> of the random file <file number> to the random buffer. If <record number> is omitted, the next record is transferred.

GET #1,I

**INPUT#<filenumber>,<variable list>**

Reads data from file <filenumber> and assigns the input to the elements of <variable list>.

INPUT #3,A,B

**KILL "<filename>"**

Deletes a disk file.

KILL "A:GAME.BAS"

**LINE INPUT#<file number>,<string variable>**

Read an entire line from a file <file number> and assigns it to <string variable>.

LINE INPUT #1,A\$

Statement/FunctionExample

LSET <string variable> = <string expression>

Stores data in random file buffer,  
left justified.

LSET A\$="HELLO"

OPEN <mode>,[#]<filenumber>,<"filename">

Opens a disk file, where <mode> is  
the file type,<filenumber> is the  
I/O label, and <file name> is the  
disk directory entry.

OPEN "O",1,"GM.DAT"

PRINT#<file number>,<list of expressions>

Writes data to a sequential disk file.

PRINT #1,A\$,B

PUT [#]<filenumber>[,<record number>]

Transfers data from the random file  
buffer to random file <file number>.  
If <record number> is omitted,  
the next record is written.

PUT #2,3

RSET <string variable> = <string expression>

Stores data in a random file buffer,  
right justified.

RSET B\$="BYE"

WRITE#<file number>,<list of expressions>

Writes data to a sequential disk  
file. Delimiters are inserted  
between items in the I/O list.

WRITE #2,A,B\$

## Disk Input/Output Functions

<u>Function</u>	<u>Operation</u>	<u>Example</u>
CVD(X\$)	Converts 8-character string to double precision number.	A#=CVD ( A\$ )
CVI(X\$)	Converts 2-character string to an integer.	I%=CVI ( I\$ )
CVS(X\$)	Converts 4-character string to single precision number.	B=CVS ( B\$ )
EOF(file no.)	Returns true (-1) if a file is positioned at its end.	IF EOF(1)
LOC(file no.)	Returns next record number to read (random file). Returns number of sectors accessed (sequential file).	X=LOC(1)
MKD\$(Z#)	Converts double-precision number to an 8-character string.	A\$=MKD\$ ( A# )
MKI\$(I%)	Converts an integer to a 2-character string.	I\$=MKI\$ ( I% )
MKS\$(B)	Converts a single-precision number to a 4-character string.	B\$=MKS\$ ( B )

## Appendix A

# Error Messages

After an error occurs, BASIC-80 returns to the Command Mode and types Ok. (Although overflow and division by zero errors will not cause BASIC-80 to stop execution.) Variable values and the program text remain intact, but you cannot continue the program with the CONT command. However, execution can be continued with a Command Mode GOTO.

The formats of error messages are:

Direct Statement	<error message>
Indirect Statement	<error message> in nnnnn

where nnnnn is the line number where the error occurred. When an error occurs in a direct statement, no line number is printed.

The error messages are listed on the next few pages, along with the error number. If an error should occur for which there is no error code, BASIC-80 will print the message "Unprintable error".

## GENERAL ERRORS

### 1      NEXT without FOR

The variable in a NEXT statement corresponds to no previously executed FOR statement.

### 2      Syntax error

A line has been encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled statement or command, incorrect punctuation, etc.).

### 3      RETURN without GOSUB

A RETURN statement has been encountered before a GOSUB was executed.

### 4      Out of data

A READ statement was executed but all of the DATA statements in the program have already been read.

### 5      Illegal function call

The parameter passed to an arithmetic or string function was out of range. Illegal function calls can occur due to:

1. A negative array subscript (LET A(-1)=0).
2. An unreasonably large array subscript (>32767).
3. LOG with a negative or zero argument.
4. SQR with a negative argument.
5. A^B with A negative and B not an integer.
6. A call to a USR function before the address of a machine language subroutine has been entered.
7. Calls to MID\$, LEFT\$, RIGHT\$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING\$, SPACE\$, INSTR, or ON...GOTO with an improper argument.

## 6 Overflow

The result of a calculation was too large to be represented in BASIC-80's number format. If an underflow (i.e..a number is too small to be represented) occurs, zero is given as the result and execution continues without any error message being printed.

## 7 Out of memory

A program is too large, has too many variables, too many FOR loops, too many GOSUB's, or too complicated expressions.

## 8 Undefined line number

The line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE was to a non-existent line.

## 9 Subscript out of range

An attempt was made to reference an array element which is either outside the dimensions of the array, or with the wrong number of subscripts.

## 10 Duplicate Definition

After an array was dimensioned, another dimension statement for the same array was encountered. The error often occurs if an array was given the default dimension of 10 and later in the program the same array is specified in a DIM statement.

## 11 Division by zero

A division by zero has been encountered in an expression, or the evaluation of an expression results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

## 12 Illegal direct

A statement that is illegal in Direct Mode has been entered as a Direct Mode command.

**13      Type mismatch**

A string variable has been assigned a numeric value or vice versa; a function that expects a numeric argument has been given a string argument or vice versa.

**14      Out of string space**

String variables have caused BASIC-80 to exceed the amount of free memory remaining. BASIC-80 will allocate string space dynamically, until it runs out of memory.

**15      String too long**

An attempt was made to create a string more than 255 characters long.

**16      String formula too complex**

A string expression was too long or too complex. The expression should be broken into smaller expressions.

**17      Can't continue**

An attempt has been made to continue a program that:

1. Has halted due to an error.
2. Has been modified during a break in execution.
3. Does not exist.

**18      Undefined user function**

A reference was made to a user-defined function which had never been defined.

**19      No RESUME**

BASIC-80 entered an error trapping routine, but the program ended before a RESUME statement was encountered.

**20      RESUME without error**

A RESUME statement was encountered, but no error trapping routine had been entered.

21 Unprintable error

An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.

22 Missing operand

During evaluation of an expression, an operator was found with no operand following it.

23 Line buffer overflow

An attempt has been made to input a line that has too many characters.

26 FOR without NEXT

A FOR was encountered without a matching NEXT.

29 WHILE without WEND

A WHILE statement has been encountered without a matching wend.

30 WEND without WHILE

A WEND was encountered without a matching WHILE.

## DISK RELATED ERRORS

50        Field overflow

An attempt was made to allocate more bytes than were specified for the record length of a random file.

51        Internal error

An internal malfunction has occurred in BASIC-80. Report conditions under which error occurred and all relevant data to Zenith Data Systems Customer Service.

52        Bad file number

A statement or command has referenced a file number that is not OPEN or is out of the range of numbers specified at initialization.

53        File not found

A LOAD, KILL, or OPEN statement referenced a file that did not exist.

54        Bad file mode

An attempt was made to perform a PRINT or WRITE on a random file, to OPEN an already open random file for sequential output, to perform a GET or PUT on a sequential file, to load from a random file, or to execute an OPEN statement where the file mode is not I,O, or R.

55        File already open

A sequential output mode is issued for a file that is already open; or a KILL is given for a file that is open.

57        Disk I/O error

An I/O error occurred on a disk I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error.

58        File already exists

The file name specified in a NAME statement is identical to a file name already in use on the disk.

61 Disk full

All disk storage space is in use.

62 Input past end

An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.

63 Bad record number

In a PUT or GET statement, the record number is either greater than the maximum allowed (32768) or equal to zero.

64 Bad file name

An illegal form is used for the file name with LOAD, SAVE, KILL, or OPEN.

66 Direct statement in file

A direct statement is encountered while an ASCII-format file is being loaded. The LOAD is terminated.

67 Too many files

An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

## RESERVED WORDS

Some words are reserved by BASIC-80 for use as statements, commands, operators, and so on, and therefore may not be used in variable or function names. The reserved words are listed below. Note that all intrinsic functions are considered to be reserved.

ABS	AND	ASC	ATN
AUTO	BASE	CALL	CHAIN
CINT	CDBL	CHR\$	CLEAR
CLOSE	COMMON	CONT	COS
CSNG	CVD	CVI	CVS
DATA	DEF	DEFDBL	DEFINT
DEFSNG	DEFSTR	DEFUSR	DELETE
DIM	EDIT	ELSE	END
EOF	ERASE	ERL	ERR
ERROR	EXP	FIELD	FILES
FIX	FN	FOR	FRE
GET	GOSUB	GOTO	HEX\$
IF	IMP	INKEY\$	INP
INPUT	INSTR	INT	KILL
LEFT\$	LEN	LET	LINE
LIST	LLIST	LOAD	LOC
LOF	LOG	LPOS	LPRINT
LSET	MERGE	MID\$	MKD\$
MKI\$	MKS\$	MOD	NAME
NEW	NEXT	NOT	NULL
OCT\$	ON	OPEN	OPTION
OR	OUT	PEEK	POKE
POS	PRINT	PUT	RANDOMIZE
READ	REM	RENUM	RESET
RESTORE	RESUME	RETURN	RIGHT\$
RND	RSET	RUN	SAVE
SGN	SIN	SPACE\$	SPC
SQR	STEP	STOP	STR\$
STRING\$	SWAP	SYSTEM	TAB
TAN	THEN	TO	TROFF
TRON	USR	VAL	VARPTR
WAIT	WEND	WHILE	WIDTH
WRITE	XOR		

## Appendix B

# ASCII Codes

### DECIMAL TO OCTAL HEX TO ASCII CONVERSION

<b>I</b>				<b>II</b>				<b>III</b>				<b>IV</b>			
DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII
0 .	000 .	00 .	NUL	32 .	040 .	20 .	SPACE	64 .	100 .	40 .	@	96 .	140 .	60 .	'
1 .	001 .	01 .	SOH	33 .	041 .	21 .	!	65 .	101 .	41 .	A	97 .	141 .	61 .	a
2 .	002 .	02 .	STX	34 .	042 .	22 .	"	66 .	102 .	42 .	B	98 .	142 .	62 .	b
3 .	003 .	03 .	ETX	35 .	043 .	23 .	#	67 .	103 .	43 .	C	99 .	143 .	63 .	c
4 .	004 .	04 .	EOT	36 .	044 .	24 .	\$	68 .	104 .	44 .	D	100 .	144 .	64 .	d
5 .	005 .	05 .	ENQ	37 .	045 .	25 .	%	69 .	105 .	45 .	E	101 .	145 .	65 .	e
6 .	006 .	06 .	ACK	38 .	046 .	26 .	&	70 .	106 .	46 .	F	102 .	146 .	66 .	f
7 .	007 .	07 .	BEL	39 .	047 .	27 .	'	71 .	107 .	47 .	G	103 .	147 .	67 .	g
8 .	010 .	08 .	BS	40 .	050 .	28 .	(	72 .	110 .	48 .	H	104 .	150 .	68 .	h
9 .	011 .	09 .	HT	41 .	051 .	29 .	)	73 .	111 .	49 .	I	105 .	151 .	69 .	i
10 .	012 .	0A .	LF	42 .	052 .	2A .	*	74 .	112 .	4A .	J	106 .	152 .	6A .	j
11 .	013 .	0B .	VT	43 .	053 .	2B .	+	75 .	113 .	4B .	K	107 .	153 .	6B .	k
12 .	014 .	0C .	FF	44 .	054 .	2C .	,	76 .	114 .	4C .	L	108 .	154 .	6C .	l
13 .	015 .	0D .	CR	45 .	055 .	2D .	-	77 .	115 .	4D .	M	109 .	155 .	6D .	m
14 .	016 .	0E .	SO	46 .	056 .	2E .	PERIOD	78 .	116 .	4E .	N	110 .	156 .	6E .	n
15 .	017 .	0F .	SI	47 .	057 .	2F .	/	79 .	117 .	4F .	O	111 .	157 .	6F .	o
16 .	020 .	10 .	DLE	48 .	060 .	30 .	Ø	80 .	120 .	50 .	P	112 .	160 .	70 .	p
17 .	021 .	11 .	DC1	49 .	061 .	31 .	1	81 .	121 .	51 .	Q	113 .	161 .	71 .	q
18 .	022 .	12 .	DC2	50 .	062 .	32 .	2	82 .	122 .	52 .	R	114 .	162 .	72 .	r
19 .	023 .	13 .	DC3	51 .	063 .	33 .	3	83 .	123 .	53 .	S	115 .	163 .	73 .	s
20 .	024 .	14 .	DC4	52 .	064 .	34 .	4	84 .	124 .	54 .	T	116 .	164 .	74 .	t
21 .	025 .	15 .	NAK	53 .	065 .	35 .	5	85 .	125 .	55 .	U	117 .	165 .	75 .	u
22 .	026 .	16 .	SYN	54 .	066 .	36 .	6	86 .	126 .	56 .	V	118 .	166 .	76 .	v
23 .	027 .	17 .	ETB	55 .	067 .	37 .	7	87 .	127 .	57 .	W	119 .	167 .	77 .	w
24 .	030 .	18 .	CAN	56 .	070 .	38 .	8	88 .	130 .	58 .	X	120 .	170 .	78 .	x
25 .	031 .	19 .	EM	57 .	071 .	39 .	9	89 .	131 .	59 .	Y	121 .	171 .	79 .	y
26 .	032 .	1A .	SUB	58 .	072 .	3A .	:	90 .	132 .	5A .	Z	122 .	172 .	7A .	z
27 .	033 .	1B .	ESC	59 .	073 .	3B .	;	91 .	133 .	5B .	[	123 .	173 .	7B .	{
28 .	034 .	1C .	FS	60 .	074 .	3C .	<	92 .	134 .	5C .	\	124 .	174 .	7C .	
29 .	035 .	1D .	GS	61 .	075 .	3D .	=	93 .	135 .	5D .	]	125 .	175 .	7D .	}
30 .	036 .	1E .	RS	62 .	076 .	3E .	>	94 .	136 .	5E .	Δ	126 .	176 .	7E .	~
31 .	037 .	1F .	US	63 .	077 .	3F .	?	95 .	137 .	5F .	-	127 .	177 .	7F .	DELETE

## Control Character Definitions

NUL	Null: Tape feed,
SOH	Start of Heading; Start of Message
STX	Start of Text; End of Address
ETX	End of Text; End of Message
EOT	End of Transmission; Shuts off TWX machines
ENQ	Enquiry; WRU
ACK	Acknowledge; RU
BEL	Rings Bell
BS	Backspace
HT	Horizontal TAB
LF	Line Feed or Space (New Line)
VT	Vertical TAB
FF	Form Feed (PAGE)
CR	Carriage Return
SO	Shift Out
SI	Shift In
DLE	Data Link Escape
DC1	Device Control 1; Reader on
DC2	Device Control 2; Punch on
DC3	Device Control 3; Reader off
DC4	Device Control 4; Punch off
NAK	Negative Acknowledge; Error
SYN	Synchronous Idle(SYNC)
ETB	End of Transmission Block; Logical End of Medium
CAN	Cancel (CANCL)
EM	End of Medium
SUB	Substitute
ESC	Escape
FS	File Separator
GS	Group Separator
RS	Record Separator
US	Unit Separator

Refer to the chart on Page B-1. Note that any print control character defined above and listed in column I of the chart can be produced from the combination of CTRL and the alphabetical character in column III or IV which is on the same line and to the right of the print control character. That is, DLE is CTRL-P or ^P, BEL is CTRL-G or ^G, and so on.

## Appendix C

# New Features in BASIC-80

### New Reserved Words

BASIC-80 has new reserved words: CALL, CHAIN, COMMON, WHILE, WEND, WRITE, OPTION BASE, RANDOMIZE.

### Type Conversions

Conversion from floating point to integer values results in rounding. (Previous versions of Microsoft BASIC would truncate the value.) This affects not only assignment statements (e.g., I% = 2.5 results in I% = 3), but also affects function and statement evaluations [ e.g., TAB(4.5) goes to the fifth position, A(1.5) yields A(2), and X = 11.5 MOD 4 yields 0 ]

### FOR/NEXT Loop Evaluation

The body of FOR/NEXT loop is skipped if the initial value of the loop exceeds the terminal value (or if a negative STEP is specified and the initial value is less than the terminal value). See Chapter Four, "Program Statements," for more information about FOR/NEXT loops.

### Division by Zero and Overflow

Division by zero and overflow no longer produce fatal errors. See Chapter Two, "Expressions," for more information.

### RND Function

The RND function has been changed so that RND with no argument is the same as RND with a positive argument. The RND function generates the same sequence of random numbers each time it is executed. The RANDOMIZE option should be used to reseed the random number generator. See Chapter Seven, "Functions," for more information.

## Printing Numeric Values

The rules for PRINTing single-precision and double-precision numbers have been changed. See Chapter Four, "Program Statements," for more information about the PRINT statement.

## String Space Allocation

String space is allocated dynamically, so the CLEAR statement is no longer used to set aside memory for string storage. The first argument in a CLEAR statement is used to set the end of memory, and the second argument is used to set the amount of stack space.

## Invalid Input

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.), or with only a carriage return causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

## PRINT USING Characters

There are two new field formatting characters for use with PRINT USING. An ampersand is used for variable length string fields, and an underscore signifies a literal character in a format string.

## WIDTH Statement

If the expression supplied with the WIDTH statement is 255, BASIC-80 uses an "infinite" line width; that is, it does not insert carriage returns. WIDTH LPRINT may be used to set the line width at the line printer.

## EDIT Characters

The at-sign and underscore are no longer used as editing characters.

## Variable Names

Variable names are significant up to 40 characters and can contain embedded reserved words. However, reserved words must now be delimited by spaces. To maintain compatibility with earlier versions of BASIC, spaces will be automatically inserted between adjoining reserved words and variable names. This insertion of spaces may cause the end of a line to be truncated if the line length exceeds 255 characters.

## Protected Binary Format

BASIC-80 programs may be saved in a protected binary format so that they may not be LISTed or EDITed.

## Appendix D

# Programming Hints

As your level of programming experience increases, you will eventually have to concern yourself with program efficiency. The two main resources you will have to conserve are: memory space and execution time. This Appendix has been included to aid in your programming effort.

## CONSERVING MEMORY SPACE

To conserve memory space, make sure that you do the following:

**Place multiple program statements on a single line.**

BASIC-80 must keep track of each program line as well as the program line number. If you place multiple statements on a single line, less space will be used for program line overhead.

**Remove all unnecessary REM statements.**

When you use a REM statement, BASIC-80 will store the one-byte code which represents the REM keyword plus the ASCII representation of the actual remark. This can result in a lot of memory being used simply for remarks. (You will have to consider the trade-off of program documentation vs. memory space when you remove these REM statements.)

**Use a subroutine call (GOSUB) only when a GOTO won't work.**

The GOSUB statement should be used only when a routine must be called from several different places within the main program. If a routine is to be called from the same place every time, then use a GOTO. Each active GOSUB will consume memory space (to update the stack), but a GOTO will not.

**Use as few parentheses in an expression as possible.**

Structure your arithmetic expressions so they use as few parentheses as possible. Each time BASIC-80 has to evaluate an expression enclosed in parentheses, it will consume more memory space. BASIC-80 will also have to store the result of this evaluation in a temporary storage location, thus using more memory space.

**Use integer variables whenever possible.**

This is very important, as integer variables only consume two bytes of memory. A single-precision variable will take four bytes, and a double-precision will take eight bytes.

**Dimension arrays sparingly.**

Make sure that you only allocate as much space for an array as you will use. For example, if you allow BASIC-80 to establish the 11-element default array size, and then only use four of these elements, you have wasted more space than you have used. So always set the array size with a dimension statement, never let BASIC-80 assume the default size of 11 elements. (Unless your array size is only 11 elements.)

**Split large programs into smaller modules.**

BASIC-80 will allow you to CHAIN between programs, as well as pass variables between programs. This makes it very easy to write a large program as several small programs and pass variables between them.

**Use DEF statements to declare variable types.**

This will prevent you from having to use the type declaration characters, thus saving you one byte for every variable that is not a single-precision data type.

**Reduce the number of simultaneously open data files.**

Every data file requires a buffer area, so it is more efficient to use the same buffer for several different files. To do this, open the first file as file #1, and then access it as needed. Then close this file and open the second file as file #1. Although you will not be able to simultaneously access both files, you will still be able to access both files as needed.

**Reduce the number of variables and arrays in a program.**

You can accomplish this by reusing variables and arrays in a program when they are no longer needed. Or, you can establish one variable to be used as a FOR/NEXT counter, and then use it for every FOR/NEXT loop.

## SAVING EXECUTION TIME

To save execution time make sure you do the following:

**Define the most commonly used variables first.**

The variables are placed in the BASIC-80 variable table as they are encountered. When a variable is referenced, the table is searched sequentially. Thus, if a variable is near the top of the table, it will take less time to access.

**Use integer variables in FOR/NEXT loops.**

This is very important and can result in a significant time savings. If you wish to try an experiment, set up a FOR/NEXT with a single-precision loop counter and time the execution. Then simply define the loop counter as an integer data type and time the execution again. (Make sure you set the loop for at least 10,000 iterations.) You will notice a significant difference in the execution times.

**Use variables instead of constants in arithmetic expressions.**

BASIC-80 uses a floating point decimal representation for numeric values. It takes less time for BASIC-80 to access a variable than to convert a constant to this representation. If you have a constant you are planning to use quite often in a program, assign it to a variable and use the variable instead.

This list is by no means exhaustive, but if you adhere to the above suggestions, you will be well on the way to generating efficient code.



## *Appendix E*

# **Assembly Language Subroutines**

BASIC-80 provides two methods for calling assembly language subroutines from a BASIC-80 program. The first method uses the USR function, which allows assembly language subroutines to be called in the same way BASIC-80's intrinsic functions are called. The second method uses the CALL statement, which generates the same calling sequence as the Microsoft FORTRAN, COBOL, and BASIC Compilers.

Since assembly language subroutines bypass some of the built-in safeguards of BASIC-80, calling assembly language subroutines renders BASIC-80 vulnerable to and defenseless against the errors in those subroutines. Therefore, write your subroutines with caution.

## MEMORY ALLOCATION

When using assembly language subroutines with BASIC-80, an important consideration is memory space allocation. Memory space must be set aside for an assembly language subroutine before it can be loaded.

During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s). The /M switch can be used during initialization to set the top of memory. (See Chapter One, "System Introduction & General Information," for more information about the initialization procedure.) BASIC-80 uses all memory available from its starting location up, so only the topmost locations in memory can be set aside for user subroutines.

After an assembly language subroutine is called, the stack pointer is set up for eight levels (16 bytes) of stack storage. If more stack space is needed, BASIC-80's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC-80's stack must be restored, however, before the program returns from the subroutine.

The assembly language subroutine may be loaded into memory by means of the CP/M system monitor, or by using the BASIC-80 POKE statement. Assembly language subroutines may also be assembled with the MACRO-80 assembler and loaded using the LINK-80 linking loader. (These programs are not provided with BASIC-80, they must be purchased separately.)

## USR FUNCTION CALLS

Before a USR function is called, the entry address for the USR subroutine must be defined in a DEF USR statement.

### DEF USR

(define entry address for USR subroutine)

Form:      DEF USR<digit>=<expression>

The DEF USR statement is used to define entry points for up to 10 assembly language subroutines.

The <digit> is the number of the assembly language subroutine. <digit> may be any number from 0-9. If <digit> is omitted, it is assumed to be 0.

The value of <expression> is the starting address of the assembly language subroutine. This address is assumed to be in decimal unless a special base specifier character is used. Hexadecimal numbers are specified with the prefix &H and octal numbers are specified with the prefix &O or &.

The format of the USR function call is:

USR[<digit>](argument)

where <digit> is from 0 to 9 and the argument is any numeric or string expression. <digit> specifies which USR subroutine is being called, and corresponds with the digit supplied in the DEF USR statement for that subroutine. If <digit> is omitted, USR0 is assumed. The address given in the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the data type of the argument that was given. The value in A will be one of the following:

<u>Value in A</u>	<u>Type of Argument</u>
2	Two-byte integer (two's complement)
3	String
4	Single-precision floating point number
8	Double-precision floating point number

**Table E-1**  
Register Values Used to Specify Data Types.

If the argument is a numeric data type, the [H,L] register pair will point to the Floating Point Accumulator (FAC) where the argument is stored. The FAC occupies eight bytes in memory — enough for a double-precision number.

## NUMERIC STORAGE FORMAT

### Integer Storage Format

An integer argument is stored as a 2-byte data value. The integer is stored in a two's complement representation. (In the following discussion, the Floating Point Accumulator will be referred to as the FAC.) An integer argument will be stored in the FAC as follows:

FAC-3 — Contains the lower 8 bits of the argument  
(the least significant byte)

FAC-2 — Contains the upper 8 bits of the argument  
(the most significant byte)

### Single-Precision Storage Format

A single-precision argument is stored as a 4-byte data value. The first byte will be the exponent. The exponent will be stored in excess 128 (200 octal) notation. This means that 200 (octal) represents an exponent of 0, 201 (octal) represents an exponent of 1, 177 (octal) represents an exponent of -1, and so forth. A single-precision number will be stored in the FAC as follows:

FAC-3 — Contains the lowest eight bits of the mantissa.

FAC-2 — Contains the middle eight bits of the mantissa.

FAC-1 — Contains the highest seven bits of the mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).

FAC — Contains the exponent stored in "excess 128" (200 octal) format

### Double-Precision Storage Format

A double-precision argument is stored using the same format as the single-precision number, only four more bytes are used to store the mantissa. A double-precision number is stored in the FAC in the same manner as a single-precision number, except:

FAC-7 through FAC-4 contain four more bytes of the mantissa (FAC-7 contains the lowest eight bits).  
(least significant).

## STRING STORAGE FORMAT

If the argument is a string, the [D,E] register pair points to three bytes called the "string descriptor". Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes one and two, respectively, are the lower and upper eight bits of the string starting address in string space.

**CAUTION:** If the argument is a string literal in the program, the string descriptor will point to the program text where the string appears. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program.

Example:

```
A$ = "BASIC-80"+""
```

This will force BASIC-80 to copy the string literal into string space and will prevent alteration of program text during a subroutine call.

## Data Type Conversions

Usually, the value returned by a USR function is the same type (integer, string, single-precision or double-precision) as the argument that was passed to it. However, calling the MAKINT subroutine returns the integer in [H,L] as the value of the function, thus forcing the value returned by the function to be integer.

To execute MAKINT, use the following sequence to return from the subroutine:

```
MAKINT EQU 105H ;address of MAKINT for CP/M
PUSH H ;save value to be returned
LHLD MAKINT ;get address of MAKINT subroutine
XTHL ;save return on stack and
;get back [H,L]
RET ;return
```

Also, the argument of the function, regardless of its type, may be forced to an integer value of the argument in [H,L]. Execute the following subroutine:

```
FRCINT EQU 103H ;address of FRCINT for CP/M
LXI H ;get address of subroutine
;continuation
PUSH H ;place on stack
LHLD FRCINT ;get address of FRCINT
PCHL
```

## CALL STATEMENT

BASIC-80 user function calls may also be made with the CALL statement. The calling sequence used is the same as that in Microsoft's FORTRAN, COBOL and BASIC compilers.

The general format of the CALL statement is:

```
CALL <variable name>[(argument list)]
```

<variable name> is assigned an address that is the starting point in memory of the assembly language subroutine. The address should be assigned to <variable name> before a CALL statement is executed. <variable name> may not be an array variable name. <argument list> contains the arguments that are passed to the assembly language subroutine.

A CALL statement with no arguments generates a simple "CALL" instruction. The corresponding subroutine should return via a simple "RET." (CALL and RET are 8080 opcodes - consult an 8080 reference manual for details.)

A subroutine CALL with arguments results in a somewhat more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine. That parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each, regardless of data type.

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

- A. 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).
- B. 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 subroutine 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. There are no checks for correct number or type of parameters.

If a subroutine expects more than three parameters, and needs to transfer them to a local data area, there is a system subroutine named \$AT (located in the FORTRAN library, FORLIB.REL) which will perform the transfer. If you do not have FORTRAN, the \$AT argument transfer subroutine is listed on Page E-9.

\$AT is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to be transferred (i.e., the total number of arguments minus 2). Your subroutine is responsible for saving the first two parameters before calling \$AT.

For example, if a subroutine expects five parameters, it should use the following general procedure:

```
SUBR:    SHLD    P1      ;SAVE PARAMETER 1
          XCHG
          SHLD    P2      ;SAVE PARAMETER 2
          MVI     A,3      ;NO. OF PARAMETERS LEFT
          LXI     ;POINTER TO LOCAL AREA
          CALL    $AT      ;TRANSFER THE OTHER 3 PARAMETERS
          .
          .
          .
          .
          .
          [body of subroutine]
          .
          .
          .
          .
          .
          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 parameters are accessed in a subprogram, remember that they are only pointers to the actual arguments passed.

It is entirely up to the programmer to insure that the arguments in the calling program correspond in number, type, and length with the parameters expected by the subprogram.

A listing of the argument transfer subroutine \$AT follows.

```
00100    ;      ARGUMENT TRANSFER
00200    ;[B, C]  POINTS TO 3RD PARAMETER
00300    ;[H, L]   POINTS TO LOCAL STORAGE FOR PARAMETER 3
00400    ;[A]      CONTAINS THE # OF PARAMETERS TO XFER(TOTAL-2)
00500
00600
00700          ENTRY    $AT
00800    $AT:     XCHG      ;SAVE [H,L] IN [D,E]
00900          MOV       H,B
01000          MOV       L,C      ;[H,L] = PTR TO PARAMETERS
01100    AT1:     MOV       C,M
01200          INX       H
01300          MOV       B,M
01400          INX       H      ;[B,C] = PARAM ADR
01500          XCHG      ;[H,L] POINTS TO LOCAL STORAGE
01600          MOV       M,C
01700          INX       H
01800          MOV       M,B
01900          INX       H      ;STORE PARAM IN LOCAL AREA
02000          XCHG      ;SINCE GOING BACK TO AT1
02100          DCR       A      ;TRANSFERRED ALL PARAMS?
02200          JNZ       AT1    ;NO, COPY MORE
02300          RET      ;YES, RETURN
```

## INTERRUPTS

Assembly language subroutines can be written to handle interrupts. All interrupt handling subroutines should save the stack, registers A-L, and the PSW. Interrupts should always be re-enabled before returning from the subroutine, since an interrupt automatically disables all further interrupts once it is received. It is also very important to choose the proper interrupt vector. With CP/M BASIC-80, all interrupt vectors are free.



## Appendix F

# Random and Sequential I/O Programming Examples

A directory application, such as a computerized telephone book, is a practical use of random files. The following two sample programs illustrate this technique. The first program, "DIRECTORY", accepts the data required to build the random file and a sequential directory file. The second program, "QUERY", retrieves the data from the directory file.

To fully understand this method of random I/O, you should look at what information is contained in the directory file. The directory file has a key created from putting together the individual's first and last names. The other field in the directory is the record number. The record number is used as an index, and points to that particular individual's entry in the random file.

When you run the "QUERY" program, you will supply the first and last name of a person. If it is a valid name (that is, if it is an entry in the directory), the record number will be used. This will point to the proper record in the random file, so the telephone number can be retrieved.

Note that these examples are NOT intended to be efficient examples of random file usage. They are designed to show how to use the random and sequential file commands.

The example does not show how to add to the data in the file once it has been created. This was done to keep the example simple. If you want to add more names to the file, you will need to modify or rewrite the build program.

As it stands, the build program assumes that there is no pre-existing directory file and starts building one. If it were changed to read in the old directory file, then new entries could be added. (Lines 50 to 80 in the query program read the file.)

If you want to do this, first open A:TABLE.EXT for input and read all of it into an array such as NP\$ and SP. Then close the file, but reopen it for output before you write out the directory.

Again, this example is not designed to be efficient. An efficient program would put the directory as the first or last few records of the file A:RFILE.EXT. In addition, the directory would be kept in alphabetical order for efficient searching.

You will understand these examples best if you type them in and use them.

```
5 REM "DIRECTORY PROGRAM"
10 OPEN "O",1,"A:TABLE.EXT"
20 OPEN "R",2,"A:RFILE.EXT"
30 FIELD #2,12 AS LN$,9 AS SN$,12 AS SR$<operator types LINE FEED>
    12 AS CI$, 10 AS SZ$,2 AS CD$,2 AS EX$,2 AS PN$
40 REC=REC+1
50 LINE INPUT "LAST NAME? ":N1$
60 LINE INPUT "FIRST NAME? ":N2$
70 LINE INPUT "STREET ADDRESS? ";N3$
80 LINE INPUT "CITY? ";N4$
90 LINE INPUT "STATE ZIP? ";N5$
100 INPUT "PHONE NUMBER (XXX,XXX,XXXX) ";N%,N1%,N2%
110 LSET LN$=N1$:LSET SN$=N2$:LSET SR$=N3$:<operator types LINE FEED>
    LSET CI$=N4$:.LSET SZ$=N5$
120 LSET CD$=MKI$(N%):LSET EX$=MKI$(N1%)<operator types LINE FEED>
    :LSET PN$=MKI$(N2%)
130 KEY$=N1$+N2$
140 PRINT #1,KEY$;,",REC
150 PUT #2,REC
160 LINE INPUT "MORE INPUT (Y OR NO)";MI$<operator types LINE FEED>
    :IF MI$="Y" GOTO 40
170 CLOSE
180 END
```

Line Number	Explanation
10	Open directory file and label it "A:TABLE.EXT".
20	Open a random file and label it as "A:RFILE.EXT."
30	Reserve space in the random file buffer for directory entries.
	LN\$=Last Name SN\$=First Name SR\$=Street Address CI9=City SZ\$=State and Zip Code CD\$=Area Code EX\$=Telephone Exchange PN\$=Last 4 digits of telephone number
40	Increment record number counter.
50 - 100	Accept input data.
110	Left-justify the string input for the random buffer.
120	Left-justify and convert integers to string values. (You must convert to strings before PUTting values into the buffer.)
130	Construct the key from first and last names.
140	Output data to the directory file.
	KEY\$=Key for directory REC=Record number of random file
150	Put the record in the random buffer.
160	Check for more data.
170	Close all files.
180	End the program and return to MBASIC Command Mode.

```
5 REM "QUERY PROGRAM"
10 CLEAR 200
20 OPEN "I",1,"A:TABLE.EXT"
30 OPEN "R",2,"A:RFILE.EXT"
40 FIELD #2,12 AS LN$,9 AS SN$,12 AS SR$,12 AS CI$,<operator types LINE FEED>
      10 AS SZ$,2 AS CD$,2 AS EX$,2 AS PN$
50 IF EOF(1) THEN GOTO 90
60 CT=CT+1
70 INPUT #1,NP$(CT),SP(CT)
80 GOTO 50
90 INPUT "NAME (LAST,FIST)":L$,F$
100 KEY$=L$+F$
110 FOR I%=1 TO CT
120 IF KEY$=NP$(I%) THEN GO TO 150
130 NEXT I%
140 PRINT "NO RECORD EXIST":GOTO 170
150 GET #2,SP(I%)
160 PRINT LN$,SN$,CVI(CD$);"-";CVI(EX$);"-";CVI(PN$)
170 INPUT "MORE QUERIES? (Y OR N)":M$:IF M$="Y"GOTO 90
180 CLOSE
190 END
```

Line Number	Explanation
10	Set up string storage space.
20	Open directory file for input.
30	Open the random file.
40	Reserve space in random file buffer.
50	Check for end-of-file condition.
60	Increment directory record counter.
70	Read directory into string.
80	Loop back for EOF check.
90	Supply the name for which you want the telephone number.
100	Create key from the first and last names.
110	Set up loop to search for record in the directory.
120	Compare input key to directory key.
130	If no match on first comparison, try the next key.
140	If no match is found after comparing all keys, print the message.
150	If match is found, put the requested record in the random buffer.
160	After converting the requested record back to integer, print it.
170	Check for more queries.
180	Close all files.
190	End the program and return to Microsoft BASIC's prompt.



# Index

- ABS, 7-3
  - absolute value function, 7-3
- accessing a random access file, 10-36
- accessing a sequential file, 10-21
- Adding Data to a Sequential File, 10-23
- Additional considerations for IF statements, 4-16
- additional features of random access files, 10-37
- address, entry for USR routine, 7-24
- allocation of
  - string space, 3-3
  - stack space, 3-3
- arccosine, 7-11
- arcsine, 7-11
- arctangent function, 7-3
- Arithmetic Functions, 7-2
- Arithmetic Operators, 2-8
- Array
  - Declarator, 6-2
  - Subscript, 6-3
  - Vertical, 6-4
- Arrays, 6-1
- ASC, 5-5
  - ASCII to numeric conversion, 5-5
  - ASCII to string conversion, 5-5
- Assembly Language
  - Programs, 7-24
  - subroutines, 7-25,E-1
- assign value to a variable, 4-5
- associate file number with file name, 10-5
- ATN, 7-3
- AUTO, 3-2
  - automatic insertion of delimiters in disk file, 10-18
  - automatic line numbering, 3-2
  - avoiding Input past end errors, 10-9
- Bad file mode, A-6,3-9
- Bad file name, A-7
- Bad file number, A-6
- Bad record number, A-7
- base specification characters, 7-24
- BASIC-80
  - new features, C-1
  - Random I/O, 10-25
  - Sequential I/O, 10-11
- BEL character, 5-5
- branch to subroutine, 4-11
- buffer, moving data to, 10-29
- buffer, sequential file, 10-22
- buffer, random file, 10-27
- build string, 5-12
- call overlay, 8-15
- CALL statement, E-7
- calling sequence, E-7
- Can't continue, A-4,3-4
- cancel and quit (Edit Mode), 9-10
- CDBL, 7-4
- CHAIN, 8-15
  - change contents of memory location, 7-15
  - change sequence of random number, 7-8
  - change text (Edit Mode), 9-8
  - character pending at terminal, 5-6
- Character Set, 1-13
- check for end-of file, 10-9
- CHR\$, 5-5
- CINT, 7-4
- CLEAR, 3-3
- close disk data file, 10-8
- CLOSE, 10-8
- Command Mode Statements, 3-1

COMMON, 8-16  
concatenation, 5-3  
conclude I/O activity to disk file, 10-8  
Conditional Execution, 4-14,4-15,4-17  
Conserving Memory Space, D-1  
Constants, 2-2  
    Fixed Point Constants, 2-2  
    Floating Point Constants, 2-2  
    Hex Constants, 2-3  
    Integer Constants, 2-2  
    Octal Constants, 2-3  
    Single and Double-Precision Numeric Constants, 2-3  
    String Constants, 2-2  
CONT, 3-4  
continue execution after error trap, 8-3  
continue program execution, 3-4  
Control Characters, 1-14  
Control Statements, 4-7  
Conversion, Type, 2-6  
conversion from ASCII to numeric, 5-5  
conversion from ASCII to string, 5-5  
conversion from decimal to hexadecimal, 5-6  
convert  
    decimal to octal, 5-10  
    numeric values to string, 10-32  
    string to numeric form, 10-33  
    string to numeric value, 5-13  
    to double-precision, 7-4  
    to integer, 7-4  
    to single-precision, 7-5  
COS, 7-5  
cosecant, 7-11  
cosine function, 7-5  
cotangent, 7-11  
CP/M extents, 10-9  
CP/M file name, 10-5  
Creating a Sequential file, 10-21  
CSNG, 7-5  
Current Line Editing, 9-11  
CVD, 10-33  
CVI, 10-33  
CSV, 10-33  
DATA, 4-18  
data file, opening, 10-5  
Data Type Conversion, 2-6  
Data Type Definition, 4-2  
debugging aid, 8-14  
decimal to hexadecimal conversion, 5-6  
decimal to octal conversion, 5-10  
declare variable  
    as double-precision, 4-3  
    as integer, 4-2  
    as single-precision, 4-2  
    as string, 4-3  
DEF FN, 7-23  
DEF USR, 7-24  
default  
    extension, 3-13,3-8  
    printer line width, 7-22  
    record length, 10-5  
    terminal line width, 7-22  
DEFDBL, 4-3  
define entry address for USR routine, 7-24  
define function, 7-23  
defintion of data types, 4-2  
DEFINT, 4-2  
DEFSNG, 4-2  
DEFSTR, 4-3  
default drive, 10-5  
DELETE, 3-4  
delete current program, 3-9  
delete program lines, 3-4  
Deleting Text (Edit Mode), 9-6  
delimiters in sequential files, 10-13  
DIM, 4-4  
Dimension statement, 6-2  
Direct statement in file, A-7  
disable error trapping, 8-2  
disable trace flag, 8-14  
disk file, opening, 10-5  
Disk File Operations, 10-1  
Disk full, A-7  
Disk I/O error, A-6  
Division by zero, A-3,2-9  
double-precision, 4-3  
Double-Precision Storage Format, E-5  
Duplicate definition, A-3,4-4,6-3

e raised to a power, 7-6  
EDIT, 3-5  
Editing, 9-1  
ELSE, 4-15  
enable automatic line numbering, 3-2  
enable Edit Mode, 9-2  
enable error trapping, 8-2  
enable trace flag, 8-14  
Ending and Restarting Edit Mode, 9-10  
END, 4-7  
enter Edit Mode, 3-5  
entry address for USR routine, 7-24  
EOF, 10-9  
ERASE, 4-5  
ERL variable, 8-5  
ERR variable, 8-5  
Error Codes, 8-6  
error simulation, 8-4  
Error Trapping, 8-2  
ERROR, 8-4  
examine contents of memory location, 7-15  
Example of  
    Error Trap, 8-3  
    input from terminal, 4-19  
    INPUT\$, 5-7  
    integer to string conversion, 10-32  
    LINE INPUT, 4-20  
    numeric input, 10-12  
    RESTORE statement, 4-24  
    WHILE/WEND loop, 4-17  
BASIC-80 Variables Names, 2-5  
FOR/NEXT loop, 4-9  
IF statements, 4-15  
Nested IF statement, 4-16  
Nested Loops, 4-9  
    numeric output, 4-22  
excess 128 storage format, E-5  
exchange variable values, 4-6  
execute program, 3-12  
exit BASIC-80, 3-13  
Expressions and Operators, 2-8  
Expressions, 2-1  
EXP, 7-6  
extend line (Edit Mode), 9-5  
FIELD, 10-27  
Field overflow, A-6, 10-27  
fields in sequential files, 10-13  
File already exists, A-6  
File already open, A-6  
File Management Statements, 10-4  
File Manipulation Commands, 10-2  
File not found, A-6  
FILES, 3-6  
Finding Text (Edit Mode), 9-7  
FIX, 7-6  
FOR without NEXT, A-5  
FOR/NEXT Loop Evaluation, C-1  
FOR/NEXT, 4-8  
formatted  
    numeric fields, 8-9  
    output, 8-8  
    output errors, 8-13  
    string fields, 8-8  
formatting characters, 8-7  
FRE, 7-13  
function, user-defined, 7-23  
Functional Operators, 2-14  
Functions, 7-1  
  
generate error, 8-4  
GET, 10-30  
GOSUB, 4-11  
GOTO, 4-12  
  
hack and insert (Edit Mode), 9-6  
hard copy device output, 4-21  
HEX\$, 5-6  
high-order byte, 7-18  
hints, programming, D-1  
hyperbolic cosecant, 7-11  
    cosine, 7-11  
    cotangent, 7-11  
    secant, 7-11  
    sine, 7-11  
    tangent, 7-11

I/O port, monitoring of, 7-21  
I/O port, input from, 7-13  
I/O Statements (Non-Disk), 4-18  
IF/THEN/ELSE, 4-15  
Illegal direct, A-3  
Illegal function call, A-2  
illegal input, 4-19  
incremental value of loop counter, 4-8  
infinite line width, 7-22  
initial value of loop counter, 4-8  
initialize variables, 3-3  
INKEY\$, 5-6  
INP, 7-13  
INPUT, 4-19  
INPUT#, 10-11  
INPUT\$, 5-7  
input  
  byte from I/O port, 7-13  
  data from sequential file, 10-11  
  entire line from sequential file, 10-16  
  entire line, 4-20  
  from terminal, 4-19  
  past end, 10-9  
Input past end, A-7, 10-19  
insert (Edit Mode), 9-4  
insert remark, 4-6  
inserting delimiters in sequential files, 10-17  
Inserting Text (Edit Mode), 9-4  
Installation Guide, 1-2  
INSTR, 5-8  
Integer, 4-2  
Integer Division, 2-9  
Integer Storage Format, E-5  
Internal error, A-6  
INT, 7-7  
Invalid Input, C-2  
inverse cosine, 7-11  
inverse sine, 7-11  
Initialization of BASIC-80, 1-13  
invoke assembly language subroutine, 7-25  
invoking Edit Mode, 9-2  
largest record number, 10-10  
least significant byte (LSB), 7-18  
LEFT\$, 5-8  
left-justify and place in random buffer, 10-29  
LEN, 5-9  
length of file, 10-9  
LET, 4-5  
Line buffer overflow, A-5  
Line Format, 1-17  
LINE INPUT, 4-20  
LINE INPUT#, 10-16  
Line numbers, 1-17  
line printer, outputting data to, 4-21  
list line (Edit Mode), 9-9  
list names of files, 3-6  
list program on line printer, 3-7  
list program on terminal, 3-7  
listing a program, 3-7  
LIST, 3-7  
LLIST, 3-7  
load and execute program, 3-12  
load overlay, 8-15  
load program file from disk, 3-8  
LOAD, 3-8  
LOC, 10-10  
LOF, 10-9  
LOG, 7-7  
Logical Operators in Relational Expressions, 2-14  
Logical Operators, 2-11  
logical record size, 10-27  
logical records, 10-27  
loop counter, 4-8  
loop, 4-8  
low-order byte, 7-18  
LPOS, 7-14  
LPRINT, 4-21  
LSET, 10-29

make numeric value into string, 10-32  
 Manual Scope, 1-9  
 Mathematical functions, 7-11  
 Matrix  
     Addition, 6-8  
     Input Subroutine, 6-6  
     Manipulation, 6-6  
     Multiplication, 6-8  
 maximum record number, 10-10  
 Memory Allocation E-2  
 memory location, examining contents of, 7-15  
 memory space conservation D-1  
 MERGE, 3-9  
 merge programs, 3-9  
 MID\$ function, 5-9  
 MID\$ statement, 5-10  
 minimum subscript, 6-3  
 Missing operand, A-5  
 MKD\$, 10-32  
 MKI\$, 10-32  
 MKS\$, 10-32  
 mode string, 10-5  
 Modes of Operation, 1-14  
 Modulus Arithmetic, 2-9  
 monitor port, 7-21  
 most significant byte (MBS), 7-18  
 move data to random buffer, 10-29  
 Moving the Cursor (Edit Mode), 9-3  
 Multi-dimensional arrays, 6-5  
 multi-dimensional array subscripts, 6-5  
 multiple statements in an IF, 4-15

natural logarithm base value, 7-6  
 natural logarithm function, 7-7  
 Nested IF statements, 4-16  
 Nested Loops, 4-8  
 New features in BASIC-80, C-1  
 New Reserved Words, C-1  
 NEW, 3-9  
 NEXT without FOR, A-1,4-10  
 NEXT, 4-8  
 No RESUME, A-4  
 numeric fields, formatted, 8-9  
 Numeric Input (from sequential disk file), 10-12  
 Numeric Storage Format, E-5

OCT\$, 5-10  
 ON ERROR GOTO, 8-2  
 ON/GOSUB, 4-13  
 ON/GOTO, 4-13  
 one-dimensional arrays, 6-4  
 ON, 4-13  
 open disk data file, 10-5  
 OPEN, 10-5  
 Operator  
     Arithmetic, 2-8  
     Logical, 2-11  
     Functional, 2-14  
     Relational, 2-10  
 Option Base statement, 6-3  
 OPTION BASE, 4-4  
 Other Edit Mode Features, 9-11  
 Out of data, A-2,4-23  
 Out of memory, A-2  
 Out of string space, A-3,3-3  
 output byte to I/O port, 7-14  
 output data to line printer, 4-21  
 output data to terminal, 4-25  
 Overflow, A-3,2-9,7-4,7-6  
 Overlay Management, 8-15

passing variables to a chained program, 8-16  
 PEEK, 7-15  
 pending character at terminal, 5-6  
 POKE, 7-15  
 port, output to, 7-14  
 port, input from, 7-13  
 port, monitoring of, 7-21  
 POS, 7-16  
 Precedence of Arithmetic Operators, 2-8  
 Preparing the Diskette 1-11  
 print blanks, 7-16  
 print line number as its executed, 8-14  
 PRINT# USING, 10-17  
 Print Positions, 4-21  
 PRINT USING, 8-8  
 print zones, 4-21  
 printed line longer than terminal width, 4-21

printer line width, 7-22  
printing data on the line printer, 4-21  
printing numeric values, 4-22  
program editing, 9-1  
Program Statements, 4-1  
Programming Hints, D-1  
prompt string, 4-19  
protected files, 10-2  
Protected File, 3-13  
PUT, 10-31

random access  
    file, creation of, 10-34  
    record size, 10-5  
    Statements, 10-26  
    Techniques, 10-34  
random number generator, 7-8  
random record, reading, 10-30  
random record, writing, 10-31  
RANDOMIZE, 7-8  
range of a FOR/NEXT loop, 4-8  
READ, 4-23  
read one character from keyboard, 5-6  
read random record, 10-30  
read values from DATA statement, 4-23  
reading a random access file, 10-34  
record length, 10-5  
Redo from start, 4-19  
register values, E-4  
Relational Expressions using Logical Operators, 2-14  
Relational Operators, 2-10  
REM, 4-6  
renumber program lines, 3-10  
RENUM, 3-10  
repetitive execution loop, 4-8  
replace portion of a string, 5-10  
Replacing Text, 9-8  
reserved words, A-8  
reserved words, new, C-1  
reset data pointer, 4-24

RESET, 3-11  
RESTORE, 4-24  
RESUME, 8-3  
RESUME without error, A-4,8-3  
return  
    address of FIELD buffer, 7-20  
    address of variable, 7-18  
    amount of free memory, 7-13  
    current cursor position, 7-16  
    current record number, 10-10  
    from subroutine, 4-11  
    leftmost characters, 5-8  
    length of string, 5-9  
    number of records, 10-9  
    number of sectors accessed, 10-10  
    numerical representation, 5-13  
    position of print head, 7-14  
    rightmost characters, 5-11  
    string of spaces, 5-11  
    string representation, 5-12  
return substring, 5-9  
RETURN without GOSUB, A-2  
RETURN, 4-11  
RIGHT\$, 5-11  
right-justify and place in random buffer, 10-29  
RND function, new features, G-1  
RND, 7-8  
round to integer, 7-6  
RSET, 10-29  
RUN, 3-12

save changes and exit (Edit Mode), 9-9  
SAVE, 3-13  
Saving Execution Time, D-1  
Scalar Multiplication, 6-7  
scaled format, 4-22  
search (Edit Mode), 9-7  
search and "kill" (Edit Mode), 9-7  
search for substring, 5-8  
secant, 7-11  
seed random number generator, 7-8  
send special character to terminal, 5-5  
Sequence of Execution, 4-7

sequence of random numbers, 7-8  
Sequential  
    Access Statements, 10-10  
    Access Techniques, 10-21  
    data pointer, 10-11  
    disk file, writing to, 10-16  
    disk file, reading from, 10-11  
    file, accessing a, 10-22  
    file, I/O buffer, 10-22

sequential file, creation of, 10-21  
sequential file input, 10-12  
set  
    line width 7-22  
    random access record size, 10-5  
    random file buffer, 10-27

set-up array, 4-4  
SGN, 7-9  
sign of expression, 7-9  
simulate occurrence of error, 8-4  
sine function, 7-10  
Single-Precision Storage Format, E-5  
single-precision, 4-2  
SIN, 7-10  
SPACE\$, 5-11  
SPC, 7-16  
Special Features, 8-1  
Special functions, 7-12  
SQR, 7-10  
square root function, 7-10  
stack space allocation, 3-3  
STEP, 4-8  
STOP, 4-14  
store constants, 4-18  
STR\$, 5-12  
stream of ASCII characters, 10-10  
string  
    arrays, 6-5  
    fields, formatted, 8-8  
    formula too complex, A-4  
    Functions, 5-4  
    Input (from sequential disk file), 10-14  
    Input/Output, 5-2  
    of spaces, 5-11  
    Operations, 5-3  
    space allocation, 3-3,C-1

String Storage Format, E-6  
String too long, A-4  
STRING\$, 5-12  
Strings, 5-1  
string, 4-3  
Subscript out of range, A-3,4-4,6-2  
substring search, 5-8  
suspend execution, 4-14  
SWAP, 4-5  
Syntax error, A-2,4-3,10-5  
SYSTEM, 3-13  
System Software Requirements, 1-10

TAB 7-17  
tab carriage, 7-17  
tangent function, 7-10  
TAN, 7-10  
terminal  
    line width, 7-22  
    value of loop counter, 4-8  
    width, 4-21

terminators in sequential files, 10-13  
text insertion (Edit Mode), 9-4  
THEN, 4-15  
Too many files, A-7  
Trace Flags, 8-14  
Transposition of a Matrix, 6-7  
trapping error, 8-2  
TROFF, 8-14  
TRON, 8-14  
truncate supplied argument, 7-6  
Type Conversion, 2-6,C-1  
Type mismatch, A-4,4-5,7-23

unconditional branch, 4-12  
Undefined line number, A-3,4-16,4-12,8-2  
Undefined user function, A-4  
unmatched WEND, 4-17  
unmatched WHILE, 4-17  
Unprintable error, A-5,8-4  
unscaled format, 4-22  
user-defined errors, 8-4  
User-Defined Functions, 7-23  
USR function calls, E-3  
USR function data type conversions, E-6  
USR, 7-25

VAL, 5-13  
variables, 2-4  
Variable Names and Declaration Characters, 2-4  
variable pointer, 7-18  
VARPTR, 7-18  
Vertical Arrays, 6-4

WAIT, 7-21  
WEND without WHILE, A-5,4-17  
WEND, 4-17

WHILE without WEND, A-5,4-17  
WHILE/WEND, 4-17  
WIDTH LPRINT, 7-22  
WIDTH, 7-22  
write  
    data to sequential disk file, 10-19  
    directory information to disk, 3-11  
    program to disk, 3-13  
    random record, 10-31  
    to sequential disk file, 10-16

WRITE, 4-25  
WRITE#, 10-19