Microsoft BASIC-80

Software Reference Manual

for HEATH/ZENITH 8-bit digital computer systems

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

595-2538-02
Printed in the United States of America

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

Table of Contents

Chapter One — System Introduction and General Information

Chapter Two — Expression

Chapter Three — Command Mode Statements

Chapter Four — Program Statements

Chapter Five — Strings

Chapter Six — Arrays

Chapter Seven — Functions

Chapter Eight — Special Features

Chapter Nine — Editing

Chapter Ten — BASIC-80 Disk File Operations

Chapter Eleven — Microsoft BASIC-80 Summary

Appendix A — Error Messages

Appendix B — ASCII Codes

Appendix C — New Features in BASIC-80

Appendix D — Programming Hints

Appendix E — Assembly Language Subroutines

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

Index

Tables

Table

INSERT

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 microprocessors.

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 develop. 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

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 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 Distribution Disk I

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).

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 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 may differ due to the different algorithms used to manipulate data.

BOUNDS ON PI — DOUBLE PRECISION BIONOMIAL 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 BIONOMIAL 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.

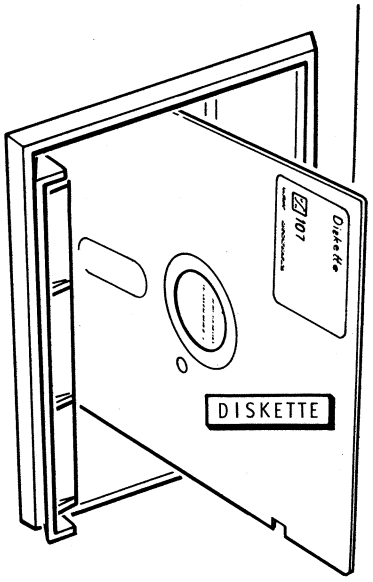


Figure 1-1A

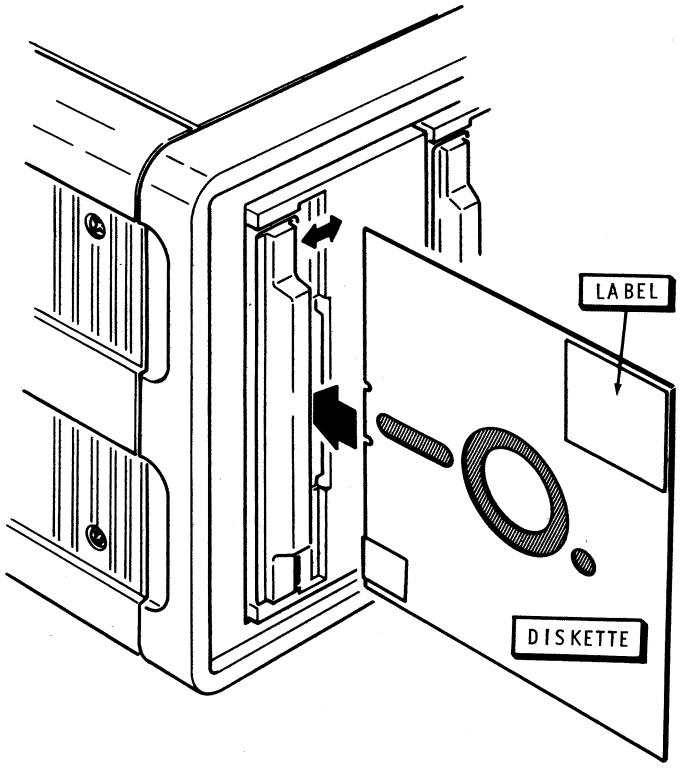


Figure 1-1B

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 magentic 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.

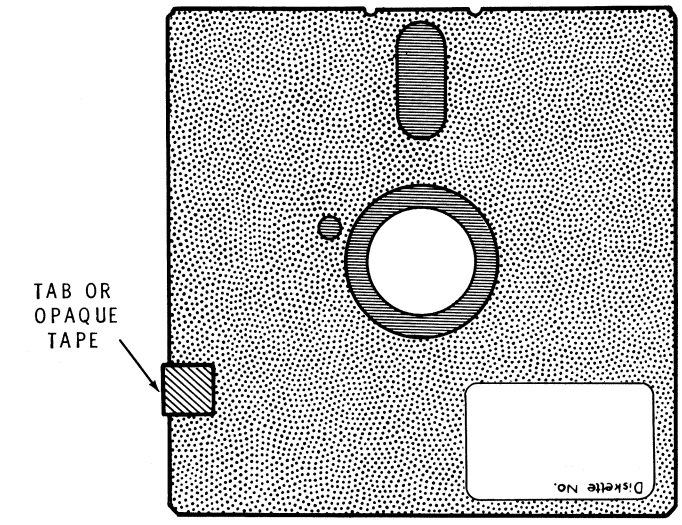
7. Do not touch the diskette surface. Abrasions can alter stored data.

WRITE-PROTECTION

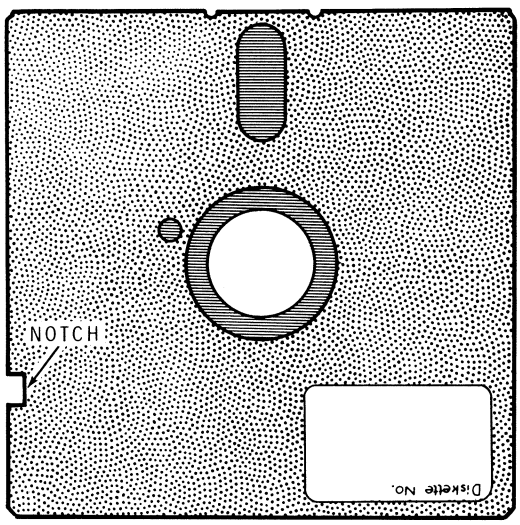
The diskette can be write-protected so that data cannot be written to it. (All distribution diskettes are shipped write-protected). How a disk is write-protected depends on the size of the diskette.

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

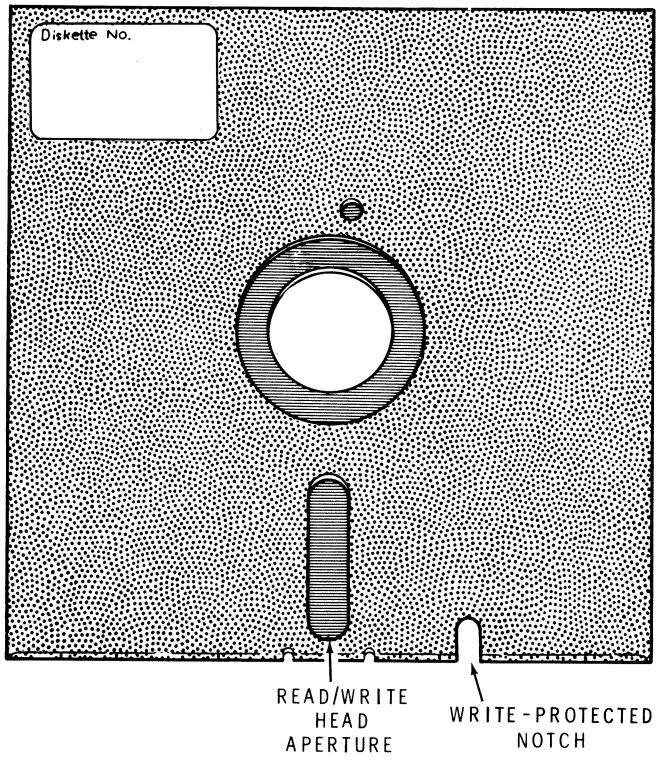


Figure 1-3A

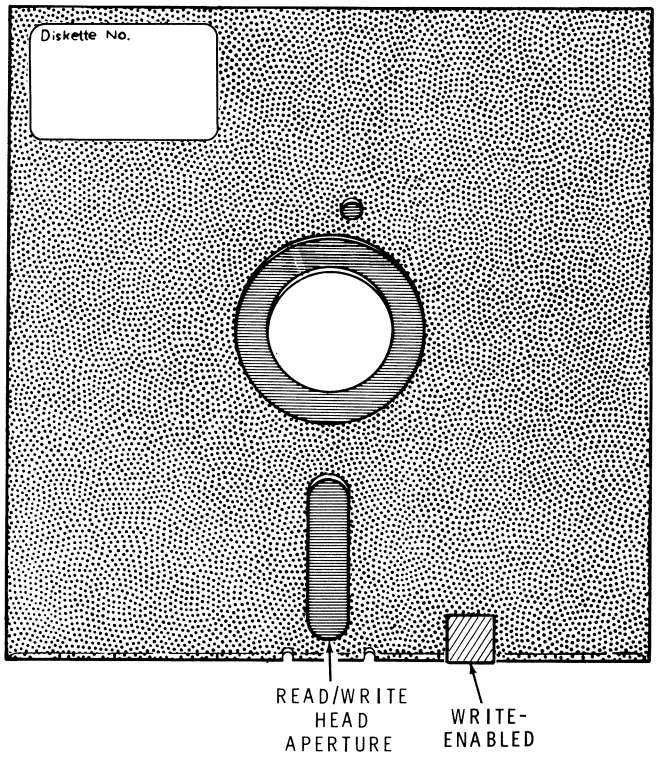


Figure 1-3B

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△SAMPLE
```

The space between MBASIC and SAMPLE is required. (Throughout this manual, we will use the symbol △ 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:

Character	Name
	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 Interpeter, 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 decribed 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:

line number	statement keyword	statement text	line terminator
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:

```
RUN△100
```

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 progam 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.

INSERT

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	Double-Precision Constants
46.8	345692811
-7.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	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 expresssions. 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-precison 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:

Operator	Operation	Sample Expression
^	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:

```
10\4 = 2
25.68\6.99 = 3
```

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:

```
10.4 MOD 4 = 2 (10\4=2 with a remainder 2)
25.67 MOD 6.99 = 5 (26\7=3 with a remainder 5)
```

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.

Operator	Relation Tested	Expression
=	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.

OPERATOR	EXAMPLE	EXPLANATION
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).

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
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 AND 16=16

63 = binary 111111 and 16 = binary 10000, so 63 and 16 = 16

15 AND 14=14

15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 binary 1110)

−1 AND 8=8

−1 = binary 1111111111111111 and 8 = binary 1000, so −1 AND 8 = 8

4 OR 2=6

4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)

10 OR 10=10

10 = binary 1010, so 1010 OR 1010 = 1010 (10)

−1 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.

NOT X=−(X+1)

The two's complement of any integer is the bit complement plus one.

6 IMP 2 =−5

6 = binary 110 and 2 = binary 10, so 6 IMP 2 = −5

3 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."

INSERT

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△<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 immmediately 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 varibles)

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△<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: EDIT△<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△<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△<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: RENUM△<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 yyyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy 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: RUN△<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]
      (assumming A: is the current default disk)
```

INSERT

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△<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△<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.

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.

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△BASE△n

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△<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△<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
```

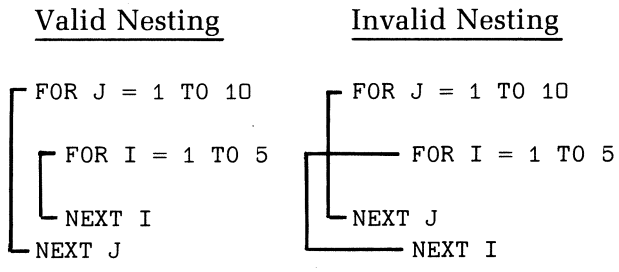
In this example, the loop executes 10 times. The terminal value for the loop is evaluated first. The terminal value (I+5) is 10. Next, the initial value is evaluated. The initial value is 1. The loop counter is then set equal to the initial value. Because the STEP value has been omitted, the incremental 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.



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 inadvertant 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
Ok
```

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 continous 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
Ok
```

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 delimeters.

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 betweeen 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"
Ok
```

INSERT

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" THENW GH.N Z$
```

STRING FUNCTIONS

The string functions avaliable to the BASIC-80 programmer are:

Function	Definition
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) MID$(X$,I,J)=Y$	return substring 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) STRING$(I,X$)	build string
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

Statement	Resultant A$
`MID$(A$,3,4)="ABCDE"`	12ABCD7
`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 chraracters)

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 VAL(X$) = 0. The & is used to specify an octal value. The VAL function will convert this octal value to decimal when VAL(X$) 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
```

INSERT

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 subscipt 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:

Array element	Subscripted variable
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 × 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:

Column		1	2	3
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.

INSERT

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:

FUNCTION	DEFINITION
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 <= 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
Ok
```

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
Ok
```

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
Ok
```

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
Ok
```

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 <= 87.3365. If EXP overflows, the "Overflow" error message is displayed.

Example:

```
10 X = 5
20 PRINT EXP(X-1)
RUN
 54.5982
Ok
```

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
Ok

PRINT FIX(-58.75)
-58
Ok
```

INT (round to integer)

Form: INT(X)

The INT function will return the largest integer <=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
Ok
```

SQR (square root)

Form: SQR(X)

The SQR function will return the square root of X. X must be >=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
Ok
```

TAN (tangent)

FORM: TAN(X)

The TAN function will return the tangent of X. X must be in 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
Ok
```

MATHEMATICAL FUNCTIONS

Some functions that are not intrinsic to BASIC-80 may be calculated as follows:

Function	BASIC-80 Equivalent
SECANT	SEC(X) = 1/COS(X)
COSECANT	CSC(X) = 1/SIN(X)
COTANGENT	COT(X) = 1/TAN(X)
INVERSE SINE	ARCSIN(X) = ATN(X/SQR(−X*X+1))
INVERSE COSINE	ARCCOS(X) = -ATN(X/SQR(-X*X+1))+1.570796
HYPERBOLIC SINE	SINH(X) = (EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X) = (EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X) = (EXP(X)-EXP(-X))/EXP(X)+EXP(-X)
HYPERBOLIC SECANT	SECH(X) = 2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X) = 2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X) = (EXP(X)+EXP(-X))/(EXP(X)-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:

Function	Definition
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
Ok
```

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
Ok
```

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:

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
Ok
```

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 = &HDOOO
120 CALL MYROUT(I,J,K)
```