Data General

Software Documentation

MP/BASIC
Reference Manual

093-400005-01

MP/BASIC Reference Manual

093-400005-01

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-400005

Printed in the United States of America
Revision 01, October 1983

NOTICE

DGC reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN DGC AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

MP/BASIC
Reference Manual
093-400005

Revision History:	Effective with:
Original Release - August 1980	
First Revision - October 1983	MP/BASIC Rev. 2.0

Preface

We have produced this manual with the needs of a diverse readership in mind. In the first nine chapters, the detailed discussion and examples giving hands-on practice should help the new programmer master the rudiments of programming in MP/BASIC. Chapters 10, 11, and 12 describe programming options for the advanced MP/BASIC programmer: segmentation, exception handling, and use of assembly language subroutines, respectively. The experienced BASIC programmer will find speedy, efficient reference tools in the summary chapter (Chapter 13), the detailed dictionary (Chapter 14), and the appendixes.

MP/BASIC

MP/BASIC is an implementation of the BASIC language as developed by Dartmouth College and standardized by the American National Standards Institute (American National Standard for Minimal Basic, ANSI X3.60-1978).

The following major enhancements to the ANSI Standard are made available in MP/BASIC (in every instance, enhancements are clearly identified as such in the text):

- Long variable names
- String dimensioning
- String concatenation
- Substrings
- Long array names
- RESTORE to a specific data line
- Eight additional math functions
- Nine string functions
- Fixed and variable-length file manipulation
- MP/BASIC assembly language interface.
- Program segmentation
- Exception handling
- ON...GOSUB statement
- ELSE clause in ON...GOTO, ON...GOSUB, and IF...THEN statements

In addition, MP/BASIC supports integer as well as real and string data types.

Future revisions of MP/BASIC will continue conforming to ANSI standards.

MP/BASIC can be used under other Data General operating systems in addition to MP/OS. This manual includes the commands necessary to use MP/BASIC from MP/AOS, MP/AOS-SU, AOS, and AOS/VS.

Organization of the Manual

Chapters 1 through 9 provide a detailed introduction to MP/BASIC.

Chapter 10 describes program segmentation of large programs.

Chapter 11 describes an optional method of exception handling.

Chapter 12 discusses calling assembly language subroutines from MP/BASIC.

Chapter 13 summarizes the material in the preceding 12 chapters, thus serving as review and overview.

Chapter 14 is a dictionary of all MP/BASIC statements, commands, and functions.

Appendix A lists the text of the error messages, and describes exception codes.

Appendix B describes overlay and non-overlay MP/BASIC.

Appendix C describes MP/BASIC internal data formats.

Appendix D describes run-only MP/BASIC.

Using This Manual

This document addresses both the new and the experienced programmer. Use the following table to determine which portions of this manual will be most helpful to you.

If You...	Then Read...
are learning to program in BASIC	Chapters 1 through 9
are an experienced BASIC programmer	Chapters 13 and 14
are writing a very long program and need to segment it	Chapter 10
want to handle exception codes within your program	Chapter 11 and Appendix A
want to use assembly language subroutines with MP/BASIC	Chapter 12
want to learn about overlay versus non-overlay MP/BASIC	Appendix B
are interested in internal data formats	Appendix C
receive an error message	Appendix A

Related Manuals

This section describes software manuals related to MP/BASIC. The manuals cover operating systems, utilities, editors, languages, and communications software.

System Topics

The following manuals describe the MP/AOS, MP/AOS-SU, and MP/OS operating systems.

MP/AOS-SU

MP/AOS-SU Programmer's Manual (DGC No. 093-000348) documents the MP/AOS-SU structure and provides a dictionary of system calls and library routines.

MP/AOS-SU Command Line Interpreter (CLI) (DGC No. 093-000349) describes the CLI program, the user's primary interface with the MP/AOS-SU system. The manual provides a command dictionary containing descriptions of command functions, formats, and examples.

Loading and Generating MP/AOS-SU (DGC No. 093-000354) describes how to install MP/AOS-SU software on microECLIPSE(TM) and DESKTOP GENERATION^{TM} computers. The manual also describes the following utilities, including sample dialogs, as appropriate:

- SYSGEN, which generates custom tailored systems.
- DINIT, which initializes disks.
- FIXUP, which repairs disks.
- MAKEBOOT, which prepares stand-alone programs and systems for booting.

MP/AOS Macroassembler, Binder, and Library Utilities (DGC No. 069-400210) documents the MP/AOS macroassembler and binder as well as the library file editor (LED) and system cross-reference analyzer (SCAN). The manual includes programming examples and a dictionary of assembler pseudo-ops.

MP/AOS-SU Debugger (DGC No. 093-000350) describes DEBUG, the system utility that aids in detecting and correcting program runtime errors. The manual provides a command dictionary that contains command functions, formats, and examples.

MP/AOS and MP/AOS-SU Advanced Program Development Utilities (DGC No. 069-400208) describes the following utilities:

- TCS (Text Control System), which can maintain multiple versions of a file.
- BUILD, which creates a new version of a file from existing files, thus minimizing effort and errors in program development.
- FIND, which locates occurrences of strings in text files.

MP/AOS and MP/AOS-SU File Utilities' (DGC No. 093-000351) describes the following utility programs, providing sample dialogues for each:

- AOSMIC, which allows manipulation of MP/AOS, MP/AOS-SU, and MP/OS disks and files on an AOS system.
- FDISP, which can display the address and data content of a file or compare two files, displaying parts that differ.
- FLED, a disk file editor that allows examination and modification of executable and data files, using a variety of formats.
- FOXFIRE, which permits the transfer of files among MP/0S, MP/AOS, MP/AOS-SU, and AOS operating systems.

- MOVE, which allows the transfer of files among directories and performs the system backups.
- REFIT, which cross-references in multi-module symbols on high-level-language sources or listing files.
- SCMP, which can compare two source programs, line by line.
- TCOPY, which allows the transfer of data to and from tapes.
- VAMP, a user-oriented file patch utility for building patch files and installing patch code.

MP/AOS

MP/AOS Concepts and Facilities (DGC No. 069-400200) provides a concise but thorough introduction to the MP/AOS operating system for users who want to assess the system's advantages.

MP/AOS System Programmer's Reference (DGC. No. 093-400051) documents MP/AOS system structure and provides a complete dictionary of system calls and library routines.

Loading MP/AOS (DGC No. 069-400207) describes how to install MP/AOS software on ECLIPSE®-line computers.

MP/AOS Command Line Interpreter (CLI) (DGC No. 069-400201) describes the interactive CLI program, the user's primary interface to the MP/AOS system. A command dictionary provides command descriptions, formats, and examples.

MP/AOS System Generation and Related Utilities (DGC No. 069-400206) describes the generation of an MP/AOS system tailored to specific applications. It also describes the following utilities, including sample dialogs as appropriate:

- SYSGEN, the interactive system generation utility;
- DINIT, the disk initializer;
- FIXUP, the disk repair utility;
- SPOOLER, which controls line printer operations;
- ELOG (error logger), the utility for interpreting the system log file.

MP/AOS Debugger and Performance Monitoring Utilities (DGC No. 069-400205) describes the following utilities, providing a dictionary of debugger commands and sample dialogues as appropriate:

- FLIT, the process debugger;
- PROFILE, which measures execution-time performance;
- OPM, the process monitor that displays current system resource allocation and status.

MP/AOS Macroassembler, Binder, and Library Utilities (DGC No. 069-400210). See description under MP/AOS-SU.

MP/AOS and MP/AOS-SU Advanced Program Development Utilities (DGC No. 069-400208) See description under MP/AOS-SU.

MP/AOS and MP/AOS-SU File Utilities (DGC No. 093-000351) See description under MP/AOS-SU.

MP/OS

An Introduction to MP/OS (DGC No. 014-000658) presents an overview of the MP/OS operating system's structure and capacities.

Installing Your Microproducts System (DGC No. 069-400001) provides instructions both for installing Microproducts hardware and for beginning to run the MP/OS software and making backup copies of it.

MP/OS System Programmer's Reference (DGC No. 093-400001) describes the MP/OS operating system in detail and tells you how to call system routines from your programs. It includes a dictionary of system calls and library routines.

MP/OS Utilities Reference (DGC No. 093-400002) describes the utility programs available with the MP/OS system.

MP/OS File Management Utilities Reference (DGC No. 093-400009) describes the use of the following file management programs for assembly language programmers:

- MP/ISAM (Micro Products/Indexed Sequential Access Method), a data base management system
- ACORN, the MP/ISAM file cration utility
- SORT/MERGE, an MP/OS utility for sorting records within a single file and merging records from two or more files into a single file

MPT Software System Guide (DGC No. 093-400011) provides a summary of system hardware and software for the experienced programmer who will be developing programs by use by other users on MPT and Enterprise workstations.

Editors

MP/AOS and MP/AOS-SU Slate Text Editor (DGC No. 069-400209) documents the features of SLATE, a screen- and line-oriented text editor.

MP/AOS SPEED Text Editor (DGC No. 069-0400202) documents the features of SPEED, the MP/AOS and MP/AOS-SU character- oriented text editor.

Languages

SP/Pascal Programmer's Reference (DGC No. 069-400203) documents an extended Pascal for system programmers. SP/Pascal has all of the features of MP/Pascal as well as special features targeted for the MP/AOS, MP/AOS-SU, and AOS operating systems.

MP/FORTRAN IV Programmer's Reference (DGC No. 069-400033) documents for system programmers a language based on American National Standard (ANS) FORTRAN (X3.9-1966) plus extensions.

MP/Pascal Programmer's Reference (DGC No. 069-400031) documents for system programmers a Pascal-based language for the MP/OS operating system.

Communications

MP/HASP Reference (DGC No. 069-400050) describes the MP/HASP II Workstation Emulator, a program that supports the simultaneous transmission of up to five files between two computers linked by telecommunication lines.

MP/RJE80 Reference (DGC No. 069-400040) describes the Remote Job Entry Program that supports the batch transfer of files between two computers linked by telecommunication lines.

MP/3270 Reference (DGC No. 069-400041) describes a program that permits terminals on any Data General system to emulate IBM Model 3277 terminals and exchange data with remote IBM or IBM-emulating systems.

MSCP Programmer's Reference (DGC No. 093-400012) describes the MP Synchronous Communications Package (MSCP), a set of program calls that allow communication with a remote station over a synchronous line. MSCP is required for MP/RJE80, MP/HASP, MP/3270, and user-defined communications programs using the Binary Synchronous Communications protocol.

Typesetting Conventions

We use these conventions for command formats in this manual:

COMMAND required *[optional]* ...

Where **Means**

COMMAND You must enter the command as shown.

required You must enter some argument (such as a filename). Sometimes, we use:

required1 | required2

which means you must enter one of the arguments. Don't enter the vertical bar; it only sets off the choice.

[optional] You have the option of entering this argument. Don't enter the brackets; they only set off what's optional.

... You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

Other conventions we use are as follows:

ESC Escape character (ESC on your keyboard). (See Chapter 1 or Chapter 13 for instructions on using this character.)

CTRL-X Control character. Depress and hold CTRL key while striking another key (represented by X). Echoed as ^X on your terminal.

↵ NEW LINE, carriage return, or line feed symbol.

* BASIC prompt.

Abbreviations

We use the following abbreviations in the descriptions of BASIC keywords:

Abbreviation	Term
var	numeric variable
expr	numeric or string expression
str-expr	string expression
log-expr	logical expression
str lit	string literal
val	numeric value
line no.	line number
col	column
control var	control variable
svar	string variable
filename	a disk filename or device

Contacting Data General

- If you have comments on this manual, please use the prepaid Remarks Form that appears after the Index. We want to know what you like and dislike about this manual.

- If you need additional manuals, please use the enclosed TIPS order form (USA only) or contact your Data General sales representative.
- If you experience software problems, please notify Data General Systems Engineering.

End of Preface

Contents

Chapter 1 - Fundamentals of Programming

Chapter 2 - Numeric Expressions: Variables, Constants and Operators

Chapter 3 - Character Strings

Chapter 4 - The INPUT and READ DATA Statements

Chapter 5 - Control Statements: Branching and Loops

Chapter 6 - Subroutines

Chapter 7 - Arrays and Subscripted Variables

Chapter 8 - Functions

Chapter 9 - File Input and Output

Chapter 10 - Program Segmentation

Chapter 11 - Exception Handling

Chapter 12 - Using Assembly Language Subroutines with MP/BASIC

Chapter 13 - Summary

Chapter 14 - Dictionary of Statements, Commands, and Functions

Appendix A - Run-Time Error Messages

Appendix B - MP/BASIC with Overlays

Appendix C - Data Formats

Appendix D - Run-only MP/BASIC

Illustrations

Figure

Tables

Table

Chapter 1
Fundamentals of Programming

Programs are collections of statements to the computer. Statements in turn are composed of characters which you generate from your keyboard. This chapter describes how programs are created, used, and maintained. We have organized this information as follows:

1. We identify the different characters available to you.
2. We explain the structure and syntax of program statements.
3. We describe how you interact with your terminal to create, modify, run, and save your programs for later use.

Characters

As a computer user, you have at your disposal a total of 128 characters with which to communicate with the system.

Each of these characters is represented inside the computer by a numeric code. There are several such codes called character codes or character sets.

MP/BASIC is designed to utilize the American Standard Code for Information Interchange, commonly referred to as the ASCII code. Figure 1-1 shows the complete set of ASCII character codes.

The first 32 characters are known as control characters. You can generate these characters (NEW LINE, for example) from your keyboard, but you cannot print them out visibly, in the same way that you can print out a letter, digit, or punctuation mark. These characters control the operation of the terminal (for instance, a line feed or a carriage return) or the information exchange between two terminals. The latter category is of no direct concern to us here.

Control characters are described in the leftmost group of columns in Figure 1-1. Within this group, the fourth column from the left shows the key symbol corresponding to each character, while the fifth column shows the mnemonic for that character. For example, BS (backspace) is the mnemonic for key symbol ↑H.

It is important to understand that, although the control characters are not visible in print, they are considered as any other character and are included in the total character count of a given word or line.

The remaining 96 characters are all printable and consist of alphabetic, numeric, and punctuation characters, as well as such symbols as the percent sign (%), dollar sign ($), and arithmetic operators.

Internal Representation of Characters

The numeric value of each character in the ASCII set is expressed in decimal, octal, and hexadecimal notation in the leftmost three columns of each group of columns in Figure 1-1.

Notice that ASCII has different representations for uppercase and lowercase letters of the alphabet. The former run from decimal 65 through 90, while the latter run from decimal 97 through 122.

Also note that the digits 0 through 9 are internally represented by codes with decimal values ranging from 48 through 57.

DECIMAL	OCTAL	HEX	KEY SYMBOL	MNEMONIC
0	000	00	↑@	NUL
1	001	01	↑A	SOH
2	002	02	↑B	STX
3	003	03	↑C	ETX
4	004	04	↑D	EOT
5	005	05	↑E	ENQ
6	006	06	↑F	ACK
7	007	07	↑G	BEL
8	010	08	↑H	BS (BACKSPACE)
9	011	09	↑I	TAB
10	012	0A	↑J	NEW LINE
11	013	0B	↑K	VT (VERT.TAB)
12	014	0C	↑L	FORM FEED
13	015	0D	↑M	CARRIAGE RETURN
14	016	0E	↑N	SO
15	017	0F	↑O	SI
16	020	10	↑P	DLE
17	021	11	↑Q	DC1
18	022	12	↑R	DC2
19	023	13	↑S	DC3
20	024	14	↑T	DC4
21	025	15	↑U	NAK
22	026	16	↑V	SYN
23	027	17	↑W	ETB
24	030	18	↑X	CAN
25	031	19	↑Y	EM
26	032	1A	↑Z	SUB
27	033	1B	ESC	ESCAPE
28	034	1C	↑\	FS
29	035	1D	↑]	GS
30	036	1E	↑↑	RS
31	037	1F	↑—	US

DECIMAL	OCTAL	HEX	KEY SYMBOL
32	040	20	SPACE
33	041	21	!
34	042	22	" (QUOTE)
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	' (APOS)
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	, (COMMA)
45	055	2D	-
46	056	2E	. (PERIOD)
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	>
63	077	3F	?
64	100	40	@

DECIMAL	OCTAL	HEX	KEY SYMBOL
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	↑ OR ∧
95	137	5F	← OR —
96	140	60	` (GRAVE)

DECIMAL	OCTAL	HEX	KEY SYMBOL
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~ (TILDE)
127	177	7F	DEL (RUBOUT)

DG-05495

Figure 1-1. ASCII Character Codes

Statements

Statements are the computer equivalent of English sentences. They are used to instruct the computer to perform some desired operation. Like sentences, statements follow a special syntax.

Each BASIC statement performs a single, well-defined step of an operation. A statement may be up to 156 characters long. Statements of this length, however, should be the exception rather than the rule. You terminate each program statement by striking the NEW LINE key.

Keywords

Statements always begin with a special word called a keyword. The keyword identifies the type of the statement and also tells BASIC the kind of operation the statement is to perform and how to treat the data (if any) following the keyword.

BASIC, like other computer languages, has a special list of keywords, which we summarize for you in Chapter 14. As we move through the chapters of this manual, we discuss specific keywords and their operation in detail.

Immediate and Delayed Execution

BASIC distinguishes between those statements intended for immediate execution and those intended for delayed execution. The distinction is based entirely on the absence or presence of a line number in front of the statement. A statement preceded by a number, for instance,

```
* 150 PRINT X, Y ↓
```

is identified as being intended for delayed execution. Programs are composed of a series of numbered statements, which are not executed until the appropriate command is typed.

Conversely, the absence of a line number in a statement such as

```
* LIST ↓
```

causes the computer to execute it as soon as you type the keyword LIST and strike the NEW LINE key. We say such statements are executed in *immediate mode.*

Some keywords can be used only in statements appearing within a program (for example, END). Others are used only in immediate mode (e.g., ENTER, LIST, NEW); sometimes we refer to keywords of this type as commands. Still other keywords may be used within a program as well as in immediate mode (e.g., PRINT, LET). In practice, you will find it easy to use each keyword appropriately, once it has been explained.

The list of keywords at the end of each chapter identifies the appropriate context for each keyword.

The syntax of BASIC recognizes some program statements in which the keyword stands by itself. For example,

```
* 10 PRINT ↓
```

is a legal statement. Other statements contain keywords that require arguments. For example, in the line

```
* 10 INPUT X,Y ↓
```

X and Y are arguments to the keyword INPUT.

Line Numbers

Each BASIC statement within a program is always preceded by a line number that you assign to it. This number identifies the line and is unique to it; you cannot assign the same line number to two different statements.

The line number is followed by the keyword (with a space before it for legibility) and by the rest of the statement. When you have finished typing the line, you signal that fact by striking the NEW LINE key. The downarrow symbol (↓) you will find in the examples we use in this chapter denotes the NEW LINE key. Thus,

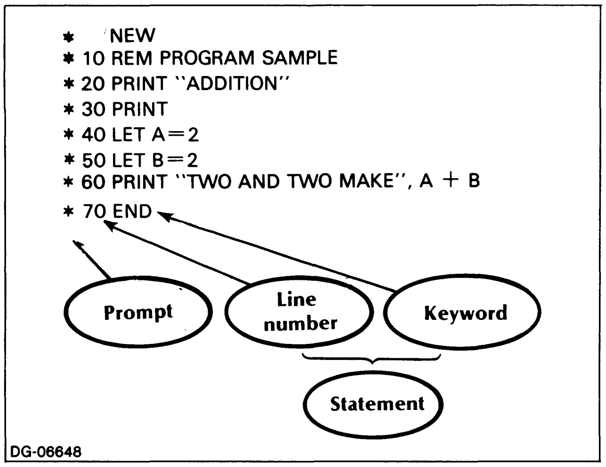
```
* 150 PRINT "HELLO" ↓
```

means that the NEW LINE character follows after the word "HELLO".

Your program statements are executed according to the order of their line numbers, moving from low to high. In special cases, this sequence is temporarily interrupted by control statements such as GOTO... and GOSUB.... (See Chapters 5 and 6.)

You should, therefore, number your lines in the order in which you wish to have them executed; observe the ascending sequence of line numbers in the program displayed in Figure 1-2.

You need not necessarily enter the program lines in exact order into your terminal. If we had typed line 40 before line 20 in Figure 1-2, the program would automatically rearrange these lines in proper sequence.



```
*  NEW
* 10 REM PROGRAM SAMPLE
* 20 PRINT "ADDITION"
* 30 PRINT
* 40 LET A=2
* 50 LET B=2
* 60 PRINT "TWO AND TWO MAKE", A + B
* 70 END
```

Figure 1-2. Sample MP/BASIC Program

Line numbers can range from 1 to 65534. Note that you may not begin a line with zero. No space should precede the number or be included within it. For example,

```
7 5
```

is not a valid line number. For the sake of legibility, however, a space should follow the line number. (In fact, the system will supply the space if you forget to type it.)

It is generally good practice to number your program lines at intervals of 10, as we have done in Figure 1-2. This allows you to insert forgotten or additional lines without retyping or otherwise renumbering your entire program. See "Inserting New Lines into the Program."

Programs

BASIC programs are collections of statements containing instructions and functioning as self-contained units.

Each program statement includes a keyword, preceded by a line number, and followed as necessary by an argument. As we have said, the instructions embodied in program statements are not executed at the time you type them: you call on commands which are not part of the program proper to perform actions such as executing, saving, and retrieving the program.

Programming involves several steps, such as writing the program (also described as coding the instructions), typing the program into the terminal, checking for errors and correcting them (known as debugging), and running or executing the program.

Below, we discuss the organization of a program and the conventions involved in performing all operations from typing through saving a given program.

Our sample program in Figure 1-2 illustrates most of the concepts we consider in this chapter. We suggest you copy this program on your terminal as soon as you are familiar with the log-on procedures and that you try out the various modifications of it we will be suggesting as we go along.

Logging On

When you work on your terminal you are said to be *on-line.* Logging on to the computer is the first step in an on-line session. Logging on puts you in communication with the CLI (Command Line Interpreter), which is the program that gives you access to all other programs. To enter BASIC once you are logged on and in the CLI mode, type the command

```
BASIC ⤶
```

NOTE: If you are using MP/BASIC from an operating system other than MP/OS, you will need to type a slightly different version of this command as follows:

For this operating system	Type this name to reach MP/BASIC
MP/OS	BASIC
MP/AOS or MP/AOS-SU	OBASIC
AOS	MBASIC
AOS/VS	MVSBASIC

The Asterisk Prompt

Once you have entered MP/BASIC, an asterisk (*) will appear at the beginning of the line. This symbol is a prompt, indicating that you can now enter a command or a program statement. The prompt appears at the beginning of each line throughout your BASIC on-line session.

If you are logged on at this time, practice at the keyboard by copying the sample program in Figure 1-2.

Logging Off: The BYE Command

Logging off means signalling your terminal that you are ending your current session.

If you have just copied the sample program, you may want to postpone trying the log-off procedure until we have shown you how to save programs; otherwise, logging off will erase everything you have typed.

To log off from BASIC, type the command BYE followed by a NEW LINE character, as follows:

```
* BYE ↲
```

Clearing Memory Area: The NEW Command

NEW is the first word you encountered as you copied our sample program. You should type this command every time you begin a BASIC session, and every time you switch from one program to another during a session.

The NEW command clears your work area in computer memory by erasing all data not explicitly saved. This ensures that no leftover statements or data from previous programs will intrude accidentally into your current program.

Format

NEW does not become part of the program; nor does it require a line number. It appears on a line by itself, followed immediately by a NEW LINE character, as follows:

```
* NEW ↲
```

Program Organization

Having dealt with the preliminaries involved in beginning an on-line session, we are now ready to consider the overall structure of a program.

In general, programs contain three broad types of numbered statements:

1. Explanatory statements, which are not executed;
2. The body of the program, consisting of statements which the system reads and executes;
3. An END statement terminating the program.

Below, we discuss explanatory remarks and the END statement before describing the various executable statements that compose the body of a program.

Explanatory Remarks: The REM Statement

The abbreviation REM stands for remark. This keyword, preceded by a line number, can be followed by any comment or explanatory remark you wish to insert into a program. (See line 10 in Figure 1-2.)

Very often REM statements are placed at the beginning of a program to indicate its general purpose. But you can insert remarks throughout the program to explain the purpose of any statement sequence. Appropriate use of remarks will clarify your program structure, and make it easy for other people to understand it.

As we have said, REM statements do not affect the way your program is executed. They will not appear as part of the output when the program is run; but if you request BASIC to display a list of your program, REM statements will be included in the list along with the executable program statements.

Format

Remarks may extend over more than one line (each line must start with a line number and the keyword REM). Remarks need not be enclosed in quotation marks, and they may contain any characters you wish. For example,

```
* 10 REM REMARKS THROUGHOUT A PROGRAM ↲
* 20 REM HELP EXPLAIN THE PURPOSE OF ↲
* 30 REM COMMANDS AND STATEMENTS. REMARK ↲
* 40 REM LINES ARE NOT EXECUTED. ↲
* 50 PRINT "END." ↲
*
```

Terminating the Program: The Keyword END

As its name implies, this keyword signals the end of the program, and terminates execution when the program is run.

When used, END must appear as the last (highest numbered) line in a program, followed by a NEW LINE character. For example,

```
.
.
.
* 150 END ↲
```

ANSI Minimal BASIC requires the use of END to terminate each program. In MP/BASIC, however, use of this keyword is optional.

Listing the Program: The LIST Command

This command writes to the screen or to a file a complete list of your program, but with the lines appearing in correct sequence (ascending order).

The LIST command does not execute your program, but merely displays it for your information. You can then check its text and logic to determine what changes, if any, are needed.

After you have finished copying the sample program, type

```
* LIST ↲
```

The entire program will appear, followed by a prompt which signifies that the system is now ready to accept additional commands.

The computer will tidy up each line in your program (for example, by inserting spaces, inserting or deleting parentheses, converting any keywords you entered in lowercase to uppercase, and so forth). This will not change the substance of your program, but will just make it easier for you to read.

Sometimes you will want to look at individual lines, rather than at an entire program. In this case, type the LIST command followed by the line number you wish to see. For example,

```
* LIST 60 ↓
```

will result in a list of line 60.

The related keywords, FIRST, LAST, and TO, enable you to ask for a list of several consecutive lines. FIRST and LAST refer to the first and last lines of a program, respectively. Thus, the command

```
* LIST ↓
```

which produces a list of the entire program, is the equivalent of

```
* LIST FIRST TO LAST ↓
```

The command,

```
* LIST FIRST TO 60 ↓
```

will output all lines from the first line through line 60 in the current program, and the command,

```
LIST 20 TO LAST ↓
```

will result in a list of lines 20 through the final line in the current program.

To list any intermediate lines in a program, you separate the first and last lines you want listed with the keyword TO, for example,

```
* LIST 20 TO 60 ↓
```

You can request a list any time while typing a program. The portions you have typed so far will then be displayed.

The LIST command is also used to save programs. See "Saving and Retrieving the Program."

Modifying the Program

It is normal for programs to undergo a great deal of restructuring as they are tested and used. Such alterations may range from simple typographical corrections to substantive changes involving deletions and insertions of entire statement blocks.

Correcting Typographical Errors

If you misspell a keyword, or if you make a typographical error in typing a line number, the system will not accept the line on which the mistake was made and will print a message to notify you of the error. (See "Error Messages.")

You can correct this type of error by retyping the entire line, including the line number and the NEW LINE character at the end. When you do this, your old line is erased, because the new line is written over it. The system retains only the latest version of a line.

Spelling mistakes and typographical errors unrelated to MP/BASIC statement syntax may not be detected by the system. It is, therefore, essential that you check the typed program as well as the program output carefully for typing and logic errors.

To alter the text of a typed line:

1. If you have already struck the NEW LINE key and are no longer on the line which contains the error, retype the entire line.
2. If you have not yet struck the NEW LINE key, either delete the entire current line (see "Deleting Program Lines") or use the DEL key to back up to the place where the error occurred.

   Suppose, for instance, that in line 20 of the sample program, you typed "Asdition" instead of "Addition". Your cursor is now positioned just beyond the closing quote of the word. If you strike the DEL key, the cursor will erase the character immediately to its left (i.e., the quotation marks) and back up one space. Continue striking the DEL key until the cursor is under the letter "s". At this point, all the characters to the right of the letter "A" have been erased, and you can retype the word correctly.
3. If you are using the AOS or AOS/VS versions of MP/BASIC, and you have not yet struck the NEW LINE key, you may use the following AOS screen-editing keys to correct the line:

Keys*	Function
←	Move cursor one character to the left
→	Move cursor one character to the right
CTRL-A	Move cursor to end of line or, on a new line, retype the last statement
CTRL-B	Move cursor to the last character of the previous word
CTRL-E	Insert one or more characters before the cursor. Press ^E again to stop inserting
CTRL-F	Move cursor to the first character of the next word
CTRL-H or HOME	Move cursor to beginning of line
CTRL-K or ERASE EOL	Erase all characters to the right of the cursor
CTRL-U	Erase all characters you have typed since the last NEW LINE

*CTRL-X means that you press and hold down the CTRL key while you press the character following the dash (-);i.e., CTRL-F means press and hold the CTRL key while pressing the F key.

Deleting Program Lines: The DELETE Command

This command will erase one or more lines whose numbers you specify.

Format

To delete a single line, type the command followed by the number of the line you want to delete and, of course, the NEW LINE character. For example,

```
* DELETE 10 ⤶
```

will delete line 10. Actually, you can omit the DELETE command and simply type the line number to be deleted. The command

```
* 10 ⤶
```

is the equivalent of

```
* DELETE 10 ⤶
```

Note that this shorthand notation works only when you delete a single line.

The related keywords, FIRST, LAST, and TO, which we have already encountered in connection with the LIST command, enable you to delete several consecutive lines. FIRST and LAST refer to the first and last lines of a program, respectively. Some examples are

```
* DELETE FIRST TO 40 ⤶
* DELETE 40 TO LAST ⤶
* DELETE 20 TO 40 ⤶
```

You can use the DELETE command during and after the typing of a program, as well as during interruptions in program execution.

In a later section, we discuss the deletion of an entire program, after you have named and saved it.

Other Methods of Deletion

If you are on the line you want deleted and have not yet struck the NEW LINE key, you can hold down the CTRL key while striking the C and A keys in succession. This will delete the entire current line. Our notation for this operation is CTRL-C, CTRL-A. On the terminal, this command will be displayed as ^C^A.

BASIC will respond with a prompt after the CTRL-C, CTRL-A sequence has been executed.

For example, type

```
* 20 PRINT "TESD CTRL-C CTRL-A ⤶
```

You will see this echoed on your terminal as

```
* 20 PRINT "TESD ^C^A ⤶
```

To check whether line 20 has indeed been deleted, type

```
* LIST
```

The resulting list should display any previous program lines you may have typed, with the exception of line 20. If line 20 was the only line you typed so far, then only a prompt should appear in response to the LIST command.

If you wish to use the ESC key instead of ^C^A, type the following while in the CLI (before bringing up BASIC):

```
CHARACTERISTICS/ON/ESC
```

Another option to delete a line before you have struck the NEW LINE key is to use CTRL-U.

Inserting New Lines into the Program

It is often necessary to insert additional statements into a program after it has been typed or executed.

Assume, for example, that you wish to insert a line into our sample program to print out the message

THIS IS THE END

between lines 60 and 70. The line number for this insertion should be a number between 60 and 70.

Since we have numbered our lines by increments of 10, nine intermediate line numbers are now available for insertions between any two statements throughout the program. Thus, our proposed one-line insertion presents no particular problem: we merely choose a number between 60 and 70, say, 65, and type our new line as follows:

```
* 65 PRINT "THIS IS THE END" ⤸
```

List the program to check whether it contains the above addition.

If the number of new statements exceeds the line numbers available between existing statements, you can make more intermediate line numbers available by renumbering the existing lines.

Renumbering Program Lines

The command RENUMBER and its related keywords enable you to change the increments between line numbers as well as the line numbers themselves. This command is only used in immediate mode; i.e., it cannot constitute a program statement.

There are several ways of renumbering, and we suggest that you try them all with the current sample program. List the program after executing each version of the command to see the difference in the results produced.

RENUMBER

This command will renumber all the lines in your program, assigning 100 to your first line and incrementing subsequent line numbers by 10.

If you try this with the current program, to which, as you recall, you added a line numbered 65, you will find that line number changed to 160, and the last line of the program, formerly line 70, changed to 170. In this way, you are back to increments of 10 after an insertion, and you have created additional space for future insertions should they become necessary.

What if you felt like starting your first program line at 50 and incrementing succeeding line numbers by 25? The related keywords, AT and STEP allow you to replace the preset starting and increment values of 100 and 10 by any other values you wish to use.

RENUMBER AT n1

This command will renumber the entire program, assigning the number you have specified as n1 to the first line, and incrementing subsequent lines by 10.

If you type

```
* RENUMBER AT 20 ⤸
```

beneath the sample program, the first line will now be numbered 20 and the others will follow by increments of 10. The last line number should be 90.

RENUMBER STEP n2

This command assigns the number 100 to the first line in the program and increments subsequent lines by the number you have specified in n2. For instance,

```
* RENUMBER STEP 5 ⤸
```

If you have been trying these commands as we have gone along, your first and last program lines are now numbered 100 and 135, respectively.

RENUMBER AT n1 STEP n2
RENUMBER STEP n2 AT n1

Either of these versions renumbers program lines so that the first one corresponds to the number you have specified in n1 and line increments correspond to the interval you have specified in n2. For example,

```
* RENUMBER AT 150 STEP 50 ⤸
```

or

```
*RENUMBER STEP 50 AT 150 ⤸
```

Your first line is now changed to 150 with subsequent program lines incremented by 50, as you specified. As you can now see, the command

```
* RENUMBER ⤸
```

is equivalent to the command

```
* RENUMBER AT 100 STEP 10
```

The AT and STEP portions need not be spelled out, since they automatically assume the values of 100 and 10 in the absence of user-specified values. Such preset values are known as default values, and we will encounter several more of them later on.

As a final move in this renumbering session, try returning to a line sequence which begins with 10 and proceeds by increments of 10.

See Chapter 14 for a more detailed description of the renumbering command.

Saving and Retrieving the Program

If you have written a program that you'll want to reuse, you need a way of saving it so you won't need to retype it before each use. It would be uneconomical and ultimately impossible to store all programs not presently in use in the computer's working memory; hence, we resort to disk storage for programs and data not in active use.

These programs or data are organized as separate files in the disk device, and they are transferred to the computer's main memory whenever they are needed for execution. The process of saving a program in disk memory involves creating a new disk file, which will be explained soon.

Naming the Program

Assigning a name to a program is part of the process of saving it, and these two actions occur on the same command line; naming a program permits you to create a file capable of being stored and retrieved. We refer to program names as *filenames.*

Legal Filenames

Program names can be from 1 to 15 characters long if you are using MP/OS or MP/AOS; names can be 31 characters long for other operating systems. Program names may include a combination of alphabetic, numeric, and other characters, as follows:

A through Z

a through z (this will automatically be converted to uppercase)

0 through 9

. (period)

$ (dollar sign)

_ (underscore)

? question mark

The following are examples of legal filenames:

TESTFILE

SURVEY.79

ACCOUNT$.REC

As you see from the second and third sample filename, you can create filenames containing a period and a suffix. Such suffixes, also known as file extensions, often serve as an efficient means of identifying a given file's contents. (For example, the name SURVEY.79 easily distinguishes the survey file for 1979 from a similar file for the year 1978.) The only restriction on filenames extensions is that they must be made up of the legal characters we have listed.

Saving the Program

Once you have chosen a name for your program, you can save it in a disk file by typing the command

```
* LIST "filename" ⤶
```

where filename (which should be enclosed in quotation marks) is the name of your program.

Suppose we decide on SAMPLE as the name for the sample program we have been working with so far; to save that program, we would type

```
* LIST "SAMPLE" ⤶
```

This will simultaneously name and save the program.

To delete a saved program, use the DELETE command followed by the filename in quotation marks and a NEW LINE. For example,

```
* DELETE "SAMPLE" ⤶
```

(Do not delete program SAMPLE right now, as you still need it for the rest of this chapter; if you have deleted it by mistake, just retype, name and save it.)

Once a filename has been saved, the system will not allow you to duplicate it; that is, you cannot use an existing filename for a new program. You will see shortly why this is important.

Saving Program Revisions

If you modify a saved program and wish to preserve the revised version, you must resave the program; otherwise your changes will be lost at the end of your current session.

Suppose you have just deleted the END statement from your saved program, SAMPLE, and you want to keep this changed text of the program.

If you now type

```
* LIST "SAMPLE" ⤶
```

the system will not accept this command; instead, you will receive an error message stating

File already exists.

You are, in effect, trying to identify two different programs by the same filename. To replace your former program with a revised version, you must first discard the original text (i.e., the old program file), after which you can reassign the filename to your present text. As we mentioned before, you delete a file by typing the DELETE command, followed by the filename. So, if you type

```
* DELETE "SAMPLE" ⤶
```

your original file, SAMPLE, will no longer exist. You can then save your changed program in a new file, SAMPLE, in the usual way by typing

```
* LIST "SAMPLE" ↲
```

Another way to save program files is with the SAVE command. This creates a different type of file, however, and should only be used for program segmentation and run-only BASIC (see Chapter 10).

Recalling Saved Programs: The ENTER Command

To retrieve a program that you have saved, you begin, as always, by typing

```
* NEW ↲
```

followed on the next line by the command ENTER and the name of the program you are recalling, enclosed in quotation marks.

If you do not type the NEW command before ENTER, any program lines already in your work area will be merged with the lines in the program you are recalling.

To recall our sample program, you would type:

```
* NEW ↲
* ENTER "SAMPLE" ↲
```

The name must correspond exactly to the filename you used when saving the program. Any imprecision, such as an extra space between characters, a misspelling, or the omission of any character, will prevent the system from finding the file, and it will send you a message to that effect.

If the system succeeds in finding your file, it will enter it into the computer's main memory, and respond with a prompt when it is ready. The program is now available for you to work with.

Note that the text of the program will not appear on the screen or line printer: you have merely instructed the system to load the program internally.

You can now type the LIST command (not followed by a filename) to see the text of your program displayed.

Executing the Program

The RUN Command

When you are ready to execute your program, type the command

```
* RUN ↲
```

BASIC then runs the program, starting from the statement with the lowest number, performs all operations you specified (provided it encounters no program errors), and outputs to your screen or to a file all the results you requested by means of PRINT statements.

When run, our sample program will produce the output shown below.

```
* RUN ↲
ADDITION

TWO AND TWO MAKE 4.

*
```

Try to run this program on your own terminal. Note the difference between the output produced by the LIST command and that produced by the RUN command.

You can run a program any number of times.

Interrupting Program Execution: The Keyword STOP

You may sometimes want the program to stop execution before it reaches the END statement, so that you can check your results or make some program changes.

You can build an interruption into your program code with a line containing only a line number and the keyword STOP; thus, for example, you could insert the line

```
* 35 STOP ↲
```

into program SAMPLE (see Figure 1-2)

When you run this program, you will see the following:

```
* RUN ↲
ADDITION

Stop at line 35.
*
```

The program carried out the instructions in lines 20 and 30 and then printed a message to tell us the last line number it executed. This is a useful reminder, since a long program may contain several STOP commands.

Next, the system will produce a prompt to indicate that it is ready to accept further instructions. You may then modify your program, have material printed out, or perform calculations. (See "Immediate Mode.") When you are ready to resume normal execution, you instruct the program to continue. (See "Resuming Program Execution.")

We can try making a simple change in SAMPLE. After execution of the program has been interrupted, type the following:

```
* 40 LET A = 3 ↲
```

This version of line 40 will replace the original version of that line for the duration of this session (as well as permanently, if you resave the changed program). You will see the result of the change when the program continues.

Aborting Program Execution

To stop a program at any time during execution, use the CTRL-C, CTRL-A sequence we discussed earlier. (See "Other Methods of Deletion.")

Used during a program run, CTRL-C, CTRL-A will stop execution. When you want to run the program again, you can use the RUN command, and execution will restart from the beginning. Or you can use the CON command (described in the next section) or RUN n (described in Chapter 14) to continue execution at the line following the interruption.

CTRL-C, CTRL-A is particularly useful for interrupting reiterated program operations such as loops. (See Chapter 5.) Short programs are normally executed so rapidly as to forestall intervention by this command.

Errors and Error Messages

In addition to the interruptions you have programmed, you will often get interruptions from the system. Such interruptions are caused by errors that prevent BASIC from completing a program run.

When this happens, the system prints a message indicating the type of error that occurred. Appendix A contains a complete list of MP/BASIC error messages.

In many error situations, the next line will display your erroneous program statement with a caret (^) placed beneath the area where the mistake occurred, to help you localize your search. We illustrate an error message at the end of this section.

When you have found and corrected your error, you may be able to continue running the program (see "Resuming Program Execution" below), or you can restart at the beginning.

Many errors are detected as soon as you enter your program into the terminal; in such cases, the system displays an error message immediately after you have terminated the erroneous statement with a NEW LINE. (Appendix A lists all MP/BASIC Run-time Error Messages.)

Type the following program statement (missing closing quotation mark), and note the resulting error message sequence.

```
* 10 PRINT "BYE ⤶
End of string requires " in line 10.
10 PRINT "BYE
             ^
*
```

While the caret may not point precisely to the location of the error, it will draw your attention to the general area, thereby making it easy for you to identify the error (in this example, the caret points to the unmatched quotation mark). Retype the entire line correctly, and then proceed as usual.

Resuming Program Execution: The CON Command

CON is an abbreviation of CONTINUE. Type

```
CON ⤶
```

whenever you are ready to have a program continue execution after an interruption. In the previous section, for instance, we interrupted our program at line 35, after which we changed line 40. Now we will continue running the program.

The original interrupted run, the modification of line 40, the CON command, and the subsequent run are shown below.

```
* RUN ⤶
ADDITION

STOP AT LINE 35.
* 40 LET A = 3 ⤶
* CON ⤶
TWO AND TWO MAKE 5.
*
```

The program responded to the CON command by resuming execution where it left off, that is, immediately after line 35. It took account of the change we made in line 40, (two and two now make five!), and since it encountered no other STOP commands, it ran without interruption until it reached the end.

If you use CON when the program is not running, the system will respond with a prompt.

Printed Output: PRINT

This command makes the result of computer operations visible to you by displaying it on your terminal.

PRINT is both a keyword and a command. Used on a numbered line within a program, it is a program statement; used on an unnumbered line, it functions as a command that is executed as soon as it has been typed and followed by the NEW LINE character.

NOTE: MP/BASIC allows you to use a semicolon (;) as a shorthand notation for PRINT. This is an extension to ANSI Minimal BASIC, which requires the user to type out the word PRINT.

PRINT as a Keyword

Below, we take examples from SAMPLE to show the major ways in which this keyword functions within a program.

1. Print a message enclosed in quotation marks.

Line 20 in SAMPLE contains such a message, i.e.,

```
* 20 PRINT "ADDITION" ↲
```

This command could also have been typed as

```
* 20 ; "ADDITION" ↲
```

The quotation marks are not part of the message; they serve only as delimiters to indicate its beginning and end. When the program is run the quotation marks are removed, and the message appears exactly as you typed it.

Print messages enclosed by quotation marks can contain any characters you wish (letters, digits, punctuation characters, spaces, and so on) except for quotation marks. The following line, for instance, would be illegal:

```
* 20 PRINT "CALLING "SO LONG", SHE LEFT." ↲
```

Try changing the message in line 20 in different ways; then run the program to see what is printed out.

The length of a line with a print message may not exceed 156 characters. If your message is longer, continue it by typing a NEW LINE followed by a new line number, a new PRINT command and the remainder of your message.

2. Print the result of a computation.

If we split the PRINT command of line 60 in SAMPLE into two separate commands, we might have a line such as

```
* 60 ; "TWO AND TWO MAKE"; ↲
65 PRINT A + B ↲
```

Line 65 would result in a printed output of the sum of values represented by A and B.

You can instruct the program to print out the result of any computation, or the current value of several variables and constants. (See Chapter 2.)

3. Print both a message and the result of a computation.

A single PRINT statement may serve to output several items per line, provided they do not exceed 156 characters. Such items may include a quoted message as well as the result of calculations, or current values of variables and constants.

The example we gave above (from line 60 of SAMPLE) shows such a combined operation.

See "Spacing of Output" for ways to separate items on a single print line.

4. Print a blank line.

As you have observed from the run of SAMPLE, there is an empty line between the heading

```
ADDITION
```

and the message

```
TWO AND TWO MAKE 4
```

We created this space for the sake of legibility by typing a PRINT command without a list of print items or punctuation on line 30, as follows:

```
* 30 PRINT ↲
```

When executed, this printed a blank line, creating space between two adjacent lines of output. You can create additional space by adding as many PRINT or ; commands as you wish. Try it.

PRINT in Immediate Mode

A line such as

```
* 50 PRINT "HELLO!" ↲
```

will be executed only when the entire program of which it is a part is run.

If this line is changed to read

```
* PRINT "HELLO!" ↲
```

or

```
* ; "HELLO!" ↲
```

then the message

```
HELLO!
```

will appear immediately after you strike the NEW LINE key. You have thus used PRINT in the immediate mode of execution.

In this mode, the PRINT command allows you to interact with a program before, during, or after execution to obtain computation results or current program values.

Assume you have programmed a STOP command into SAMPLE. Upon interruption of the run you could type

```
* PRINT A ↵
```

and the program would immediately respond with

```
2
```

Or you could type

```
* ; A, 5 + A ↵
```

and the program would print

```
2      7
```

In a long program, you could build STOP statements at several key points, print out current values and test calculations, make any necessary corrections, and then resume execution. This feature makes it possible to track down errors and correct them while the program is actually being executed; we refer to this process as dynamic debugging.

In addition, you can use PRINT to perform and print calculations unrelated to any particular program; hence, we sometimes refer to immediate mode as the *desk calculator* mode.

Spacing the Output: Printing Numbers

BASIC prints numbers in the following form:

sign number

The sign is either minus (−) or plus (a blank).

Unlike nonnumeric characters, which are printed without intervening spaces, positive numbers are always preceded by a blank space; in this way, they may be printed in closely packed format without running into each other.

Compare, for example, the printing of the characters A and B and the numbers 5 and 6 in the output of this short example:

```
.
.
* 20 PRINT "A";"B" ↵
* 25 LET X=5 ↵
* 30 LET Y=6 ↵
.
* 40 PRINT X;Y ↵
.
* RUN ↵
AB
5  6
*
```

Zone Spacing of Printed Output: The Comma

The spacing of items on a print line is determined by the punctuation separating them. Any spaces you insert in the print statement itself will not affect the appearance of the output, unless they are enclosed in quotation marks. For example, regardless of whether you write

```
* 40 PRINT X;Y ↵
```

or

```
* 40 PRINTX;Y ↵
```

or

```
* 40 PRINT X;Y ↵
```

the output will appear spaced as follows:

```
5  6
```

The determining factor is the punctuation between X and Y, and the fact that this punctuation is a semicolon (;) rather than a comma (,).

As Figure 1-3 shows, the print line on a terminal is divided into five print zones, each of which is 14 characters wide.

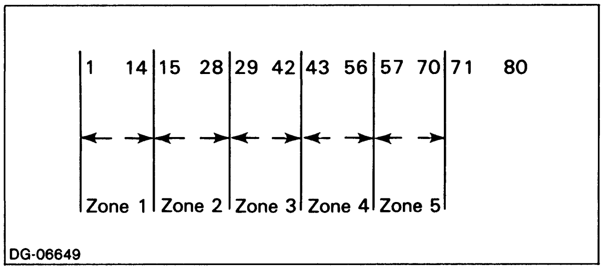


Figure 1-3. Print Zones

A comma (,) between items in a PRINT statement list causes the next item to be printed in the leftmost position of the next printing zone available. For example,

```
* 10 LET A = 3 ↵
* 20 LET B = 2 ↵
* 30 PRINT A, B, A+B, A-B, A*B ↵
* 40 END ↵
```

When executed, this program would produce the output shown in Figure 1-4.

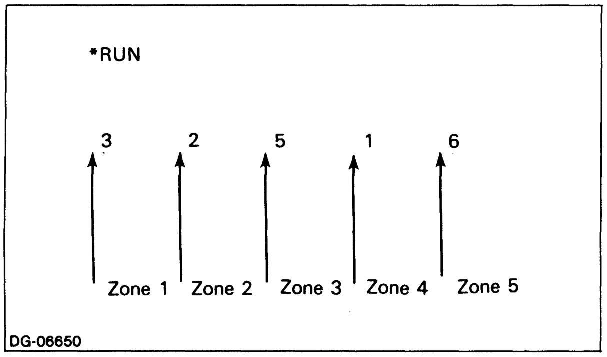


Figure 1-4. Zone Spacing of Printed Output

If there are no more printing zones on the current line, or if the item to be printed is longer than the remaining space on that line, printing continues in the first printing zone on the next line.

If, for example, line 30 had read

```
* 30 PRINT A, B, A+B, A-B, A*B, B-A ↵
```

the output would be as shown in Figure 1-5.

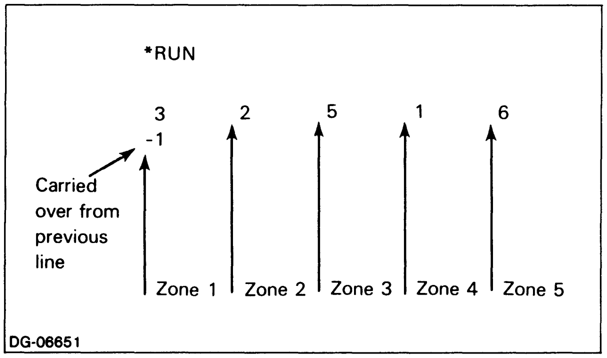


Figure 1-5. Zone Spacing of Printed Output: Output Continued to Next Line

If an item requires more than one print zone, the next item in the list is printed in the next free print zone. For example, the program below would generate the output shown in Figure 1-6.

```
* 10 LET A = 3 ↵
* 20 LET B = 2 ↵
* 30 PRINT "THIS IS A LONG MESSAGE", A, B ↵
* 40 END ↵
```

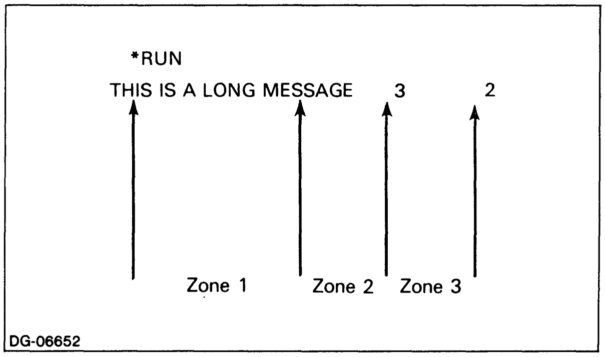


Figure 1-6. Zone Spacing of a Long Item

If the PRINT command is followed by more than one comma, the corresponding number of print zones is skipped. For example,

```
* 10 PRINT "DOUBLE",,"SPACE" ↵
```

would print the word "space" in zone three, rather than in zone two, as shown in Figure 1-7.

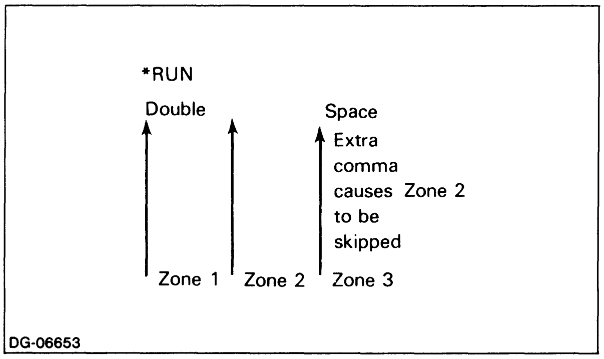


Figure 1-7. Effect of Commas on Zone Spacing

If, however, skipping print zones would cause the next item to be printed beyond the allowable line width, the item is printed on the next line.

Compact Spacing of Output: The Semicolon

A semicolon (;) between items in the PRINT statement list causes the next item to be printed at the next character position. If the item is a positive number, a blank space is reserved for the plus (+) sign.

In line 60 of Figure 1-2, we used a semicolon to separate the two items on our print line, and this caused the number 4 to appear adjacent to the printed message. The program statement was

```
* 60 PRINT "TWO AND TWO MAKE" ; A+B ⤶
```

and the printed output was

```
TWO AND TWO MAKE 4.
```

Look at the following program lines.

```
* 10 LET A = 3 ⤶
* 20 LET B = 5 ⤶
* 30 LET C = 4 ⤶
* 40 PRINT A ; B ; A+1 ; B+1 ; C ; C+1  ⤶
* 50 PRINT A, B, C ⤶
* RUN ⤶
```

The output of line 40 will be spaced as follows:

```
3 5 4 6 4 5
```

The output of line 50 will be spaced as follows:

```
3          5          4
```

Using a combination of commas and semicolons is an easy way to tabulate your output. Consider how the layout is handled in this example. (Don't worry about any parts of line 20 that may, at this point, seem unfamiliar; just focus on the spacing instructions.)

```
* 20 PRINT "TOY #:" ; N, "COLOR: " ; B$, "# IN
STOCK:" ; S ⤶
.
.
* RUN ⤶
TOY #: 859       COLOR: RED#        IN STOCK: 675
```

Spacing with TAB

When spacing with commas is too wide, and spacing with semicolons too compact, you can use the TAB function to set the exact intervals you need. Functions are discussed in Chapter 8, but we are anticipating a little to allow you greater flexibility in controlling the appearance of your output.

TAB resembles a tabulator in a typewriter, in that it advances the cursor on the line to the column (character position) specified by the user. This function is always used in conjunction with the PRINT keyword. For example,

```
* 50 PRINT "TAB 25 IS AT" ; TAB(25) ;"A" ⤶
```

The word TAB must always be followed by a numeric argument in parentheses that tells the system in what column to start printing. Column count begins with 1; i.e., the first column on a line is number 1. The column count specified by the numeric argument of TAB is always relative to column 1.

You can set the argument of TAB to any number you wish, with the following qualifications:

- If the argument is less than 1, the system will automatically set it to 1.
- If your TAB call brings you beyond the end of the line, it will wrap-around and print in the corresponding column on the same line;
- If that column already contains a a printed character, printing will resume in the corresponding column of the next line.

The TAB call should be followed by a semicolon (;).

Assuming that no other PRINT statements preceded the TAB function in our example, and assuming that the cursor is presently on a lower column number than 25, line 50 would result in the following output:

```
* RUN ⤶
TAB 25 IS AT            A
*
```

If the cursor happened to be on a higher column number than 25, printing would be executed on column 25 of the next line. If, for example, your line read

```
* PRINT TAB(10); "A"; TAB(30); "C"; TAB(25); "B" ⤶
```

A and C would appear on the same line, while B would appear in column 25 of the next line, as follows:

```
* RUN ⤶
         A                   C
                        B
*
```

As you gathered from the previous example, several TAB commands can be combined on the same print line, just like ordinary PRINT statements. For example, a statement such as

```
*                  50                  PRINT
X;TAB(1);Y;TAB(30);Z;TAB(38);"OUNCES" ⤶
```

would produce a tabulated output, provided the system has values for X, Y, and Z.

Spacing to the Next Line

When the last item in a print list has been printed, BASIC moves to the line immediately below by outputting a carriage return and a line feed, unless the last item in a list is followed by a comma (,) or semicolon (;).

In this case, the carriage return and line feed are not output, and the next item in a subsequent PRINT command is printed on the same line according to the spacing dictated by the comma or semicolon punctuation. Notice, for example, the effect of the semicolon and the comma following lines 40 and 50 below.

```
* 10 LET A = 5 ↲
* 20 LET B = 6 ↲
* 30 LET C = 3 ↲
* 40 PRINT "A =" ; A ; ↲
* 50 PRINT "B =" ; B , ↲
* 60 PRINT "AND C =" ; C ↲
* 70 END ↲
* RUN ↲
A = 5   B = 6       AND C = 3
```

If, however, the comma or semicolon would cause the next item to be printed beyond the allowable line width, a carriage return and line feed are output, and printing continues on the next line.

Formatted Output: The PRINT USING Command

Printed output can also be formatted with the PRINT USING statement. In this statement, you specify the image of the output as you want it to appear, in addition to the information you want to display. The image specification lets you left- or right-justify information, insert dollar signs, asterisks, commas, and decimal points into numbers, suppress or print leading zeros, indicate positive and negative numbers, and control the spacing of your information display. Note that PRINT USING is an enhancement to ANSI standard.

For example, to replace a leading zero with a space, use the number sign (#) in the image. To print 010 without the leading zero, type

```
* PRINT USING "###":010 ↲
```

The result is :

```
 10
```

Let's take a moment to look at the PRINT USING statement in the above example. The asterisk is the MP/BASIC prompt. It is followed by the PRINT USING keywords. If this were a program statement, we would have inserted a line number after the asterisk. The characters within quotation marks (in this case ###) are the image. 010 is the information to be output. A colon separates the image from the output information.

In addition to suppressing leading zeros, the number sign reserves space for the output value. Because we specified three number signs, three characters (including spaces) will be displayed. For example:

```
* PRINT USING "###":10 ↲
 10
```

As you can see, BASIC displayed a space and then 10.

You can also use the number sign to reserve space in an image for alphabetic characters. For example:

```
* PRINT USING "####":"NO" ↲
 NO
```

Notice that BASIC displayed a space before the word NO. This is because BASIC will center alphabetic characters within the image. To left-justify the word NO, type:

```
* PRINT USING "<###":"NO" ↲
NO
```

To right-justify the word NO within four spaces reserved for the word, type:

```
* PRINT USING ">###":"NO" ↲
  NO
```

Notice that the right-justify image character (>) also reserves a space for the word.

To print two values on a line type:

```
* PRINT USING "<###>###":"NO","YES" ↲
NO      YES
```

You can also specify, within the image, additional information to be displayed. For example:

```
* PRINT USING "ENTER <## OR <##":"NO","YES"
↲
ENTER NO   OR YES
```

There are several more characters (summarized in Table 1-1) used to format the printing of numbers. These are the asterisk (*), the percent sign (%), the dollar sign ($), the plus sign (+), the minus sign (-), the comma (,), and the decimal point (.).

The asterisk replaces any leading zeros with asterisks. For example,

```
* PRINT USING "****":1 ↲
***1
```

The percent sign displays any leading zeros. For example,

```
* PRINT USING "%%% %%%":01,1 ↲
001 001
```

A dollar sign can be placed only in the left-most position of the image. The result is a floating dollar sign preceding the first displayed digit. For example,

```
* PRINT USING "$###":10 ↲
$10
```

The plus sign displays a plus sign if the number is positive, and a minus sign if the number is negative. Type:

```
* PRINT USING "+### +###":10,−10 ↲
+10     −10
```

The minus sign displays a blank if the number is positive, and a minus sign if the number is negative. For example:

```
* PRINT USING "−### −###":10,−10 ↲
 10  −10
```

The comma inserts appropriate commas in numbers, while the decimal point forces a decimal point. Consider the following example:

```
* PRINT USING "##,###.## ##,###.## ##,###.##":-10,10,1000.10 ↲
   −10.00     10.00     1,000.10
```

Now, let's combine the dollar sign, asterisk, comma, and decimal point to format the display of dollar amounts.

```
* PRINT USING "$*,***.## $*,***.##":1000.10,222.22 ↲
$1,000.10   $**222.22
```

You can also use variables within the PRINT USING statement. For example:

```
* LET A$="$*,***.##" ↲
* LET B=1000.10 ↲
* PRINT USING A$:B ↲
$1,000.10
```

Notice that the variable name of the image is A$. This means that it is a string variable; it contains alphanumeric characters. String variables are described in Chapter 3.

Table 1-1 contains a brief description of the image characters used in the PRINT USING statement. For more information about this statement, refer to Chapter 14.

Table 1-1. Image Characters for PRINT USING Statement

Image Character	Meaning
#	Replaces leading zeros with spaces
*	Replaces leading zeros with asterisks
%	Displays leading zeros
$	Places a dollar sign before the left-most digit
+	Prints a plus sign immediately to the left of a positive number; prints a minus sign immediately to the left of a negative number
−	Prints a minus sign immediately to the left of a negative number; prints a space to the left of a positive number
,	Prints a comma in the specified position if there is a significant digit to the left of that position
.	Prints a decimal point in the specified position

Determining Memory Used: The MEMORY Command

To determine how much memory space a given program takes up, type the command

```
MEMORY ↲
```

In response, BASIC displays the number of words of memory used by the current program, as well as the number of words of memory still available for use.

(As you may know, a word of memory consists of two units called bytes; a byte in turn consists of eight units called bits [binary digits]. Thus, one word of memorv is composed of two 8-bit bytes, or 16 bits.)

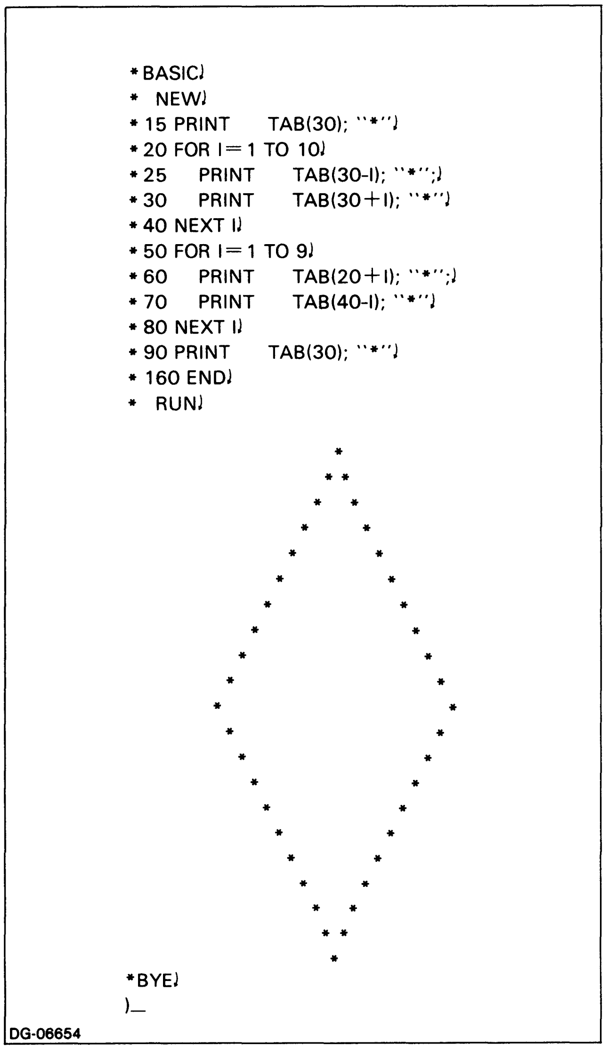
For example:

```
* MEMORY ↲

PROGRAM AREA: 70 words.
DATA AREA: 24 words.
AVAILABLE AREA: 12535 words.
```

The program area and data area together make up the total memory space used by your current program. Available area indicates how many words of memory are remaining and available for use.

A Complete BASIC Session

Figure 1-8 shows an entire BASIC session: logging on, typing a new program, running the program, and logging off. Observe that this program makes generous use of the TAB function. Its other statements may mystify you at the moment, but the next few chapters will supply the necessary information. Before long, you may find yourself returning to this program and tinkering with improvements in its design.



```
*BASIC↵
*  NEW↵
*15 PRINT    TAB(30); "*"↵
*20 FOR I=1 TO 10↵
*25    PRINT    TAB(30-I); "*";↵
*30    PRINT    TAB(30+I); "*"↵
*40 NEXT I↵
*50 FOR I=1 TO 9↵
*60    PRINT    TAB(20+I); "*";↵
*70    PRINT    TAB(40-I); "*"↵
*80 NEXT I↵
*90 PRINT    TAB(30); "*"↵
*160 END↵
*  RUN↵

                *
               * *
              *   *
             *     *
            *       *
           *         *
          *           *
         *             *
        *               *
       *                 *
      *                   *
       *                 *
        *               *
         *             *
          *           *
           *         *
            *       *
             *     *
              *   *
               * *
                *

*BYE↵
)_
```

DG-06654

Figure 1-8. Complet BASIC Session

Keywords in Chapter 1

Table 1-2 lists the keywords introduced in Chapter 1.

Table 1-2. Keywords in Chapter 1

Keyword	Can be used in	
	Program Statement	Immediate Mode
BYE	No	Yes
CON	No	Yes
DELETE	No	Yes
DELETE FIRST TO LAST	No	Yes
END	Yes	No
ENTER	No	Yes
LIST	No	Yes
LIST FIRST TO LAST	No	Yes
MEMORY	No	Yes
NEW	No	Yes
PRINT	Yes	Yes
PRINT USING	Yes	Yes
REM	Yes	No
RENUMBER	No	Yes
RENUMBER AT...STEP...	No	Yes
RUN	No	Yes
STOP	Yes	No

End of Chapter

Chapter 2
Numeric Expressions: Variables, Constants and Operators

A numeric expression is a combination of one or more data elements placed in relationship to each other by means of arithmetic operators.

Numeric expressions in MP/BASIC look slightly different from conventional algebraic expressions, because they must be written in a straight line; that is, expressions like $^A/_B$ (A divided by B) and T^2 (T squared), must be written A/B and T^2 respectively, so that everything appears on the same level on the line.

The data elements within an expression can be variables, constants, and function references (discussed in Chapter 8). With the addition of strings (see Chapter 3), these elements constitute the main building blocks with which you will be constructing your MP/BASIC programs.

In this chapter we will deal in order with each of the ingredients of an expression, i.e. numeric variables, numeric constants, and arithmetic operators.

NOTE: In the programming examples we use in this and subsequent chapters, we omit both the prompt (*) and the NEW LINE symbol (⤶).

Variables

A variable is a data item whose value can be changed during the execution of the program.

There are several kinds of variables, namely:

- simple numeric variables (discussed in this chapter)
- simple character string variables (see Chapter 3)
- subscripted numeric and string variables, or arrays (see Chapter 7)

You are allowed a maximum of 255 variables in one program (each numeric variable, string variable, numeric array and string array counts as one variable).

Numeric Variables

A numeric variable is represented by a name to which a value is assigned during the execution of the program.

Suppose we have the following program lines:

```
10 LET X = 15
20 PRINT X
30 LET X = X + 3
40 PRINT X
```

The statement in line 10 above creates a variable, X, and assigns the value 15 to it.

What this actually means is that the machine puts the value 15 into a location in memory to which it gives the name X, as Figure 2-1 shows.

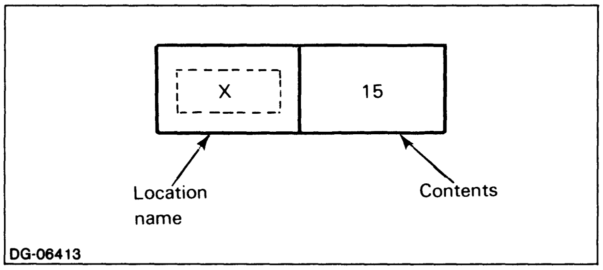


Figure 2-1. Numeric Variables

This location is now the exclusive preserve of X, and no values other than values specifically assigned to X may be placed into it.

A statement such as

```
15 LET Y = 35
```

for example, will place 35 into a completely different location in memory called Y. The two pieces of data in X and Y are totally separate from each other.

If we now execute line 20 of our program,

```
20 PRINT X
```

we will get a value of 15 for X.

Line 30 on the preceding page

```
30 LET X = X + 3
```

says that we want the new value of X to be equal to the old value of X plus 3.

In executing this command, the computer will cause the new value of X (i.e., 18) to be written into location X, as shown in Figure 2-2.

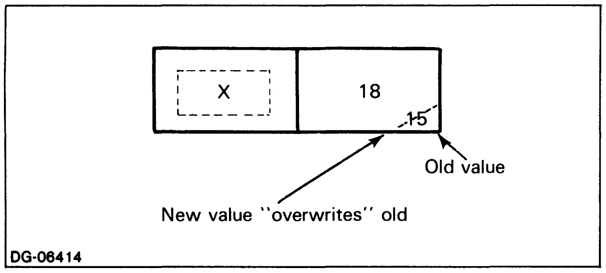


Figure 2-2. Changing Variable Values During Execution

What has happened to 15, the previous value of X? As you might have guessed, any new values which X assumes are written on top of the old values; as soon as this happens, the old values are erased and are no longer available to us.

To put it another way, at any given moment during the execution of a program, a numeric variable is associated with a single numeric value, although that value can be changed by the execution of statements within the program. Any new values then *overwrite* (replace) those which preceded them.

The command

```
40 PRINT X
```

will now print 18 as the value of X.

Naming Numeric Variables

A numeric variable name must begin with an alphabetic character, which can be followed by any combination of:

- alphabetic characters, a through z or A through Z (there is no difference between upper- and lowercase characters, i.e., area3 and AREA3 are the same variable)
- digits (0 through 9)
- the underscore character (_)

The length of the variable name is limited by the computer's memory. We suggest that you keep your variable names short so that they are easier to type and to prevent line length problems. For compatibility with AOS/VS BASIC, we suggest a maximum of 32 characters in your variable names.

A few examples of valid and invalid numeric variable names are listed in Table 2-1.

Table 2-1. Numeric Variable Names

Valid Numeric Variable Names	Invalid Numeric Variable Names
P1	C 7 (space not valid)
A	10_FOLD (digit as first character)
payroll_amount	

Please note that ANSI presently supports only a single letter, or a single letter followed by a digit, for variable names.

Assigning Variable Values: The LET Statement

The LET keyword (of which you had a sneak preview in the last section) is one command you can use to assign a value to a variable. (The INPUT and READ keywords are two additional ways as you will see in Chapter 4.)

As you already know, a command such as

```
10 LET X = 15 + 3 * Y
```

means: let whatever value X previously held be replaced by the value(s) listed to the right of the equals sign.

Note the format of the command line: you can have only one variable to the left of the equals sign; but you can have several numbers and/or variables to the right of the equals sign.

Once assigned, the value of X will not change until the program encounters another LET statement (or any of the other keywords discussed in later chapters).

Values for variables can be generated by the program as a result of calculations executed within it.

Consider, for instance, the following:

```
10 LET X = 5
20 LET Y = 7
 .
 .
 .
60 LET Z = X + Y
 .
 .
80 LET Z = Z - 2
```

As Figure 2-3 shows, until the program reaches line 60, location Z will contain no value.

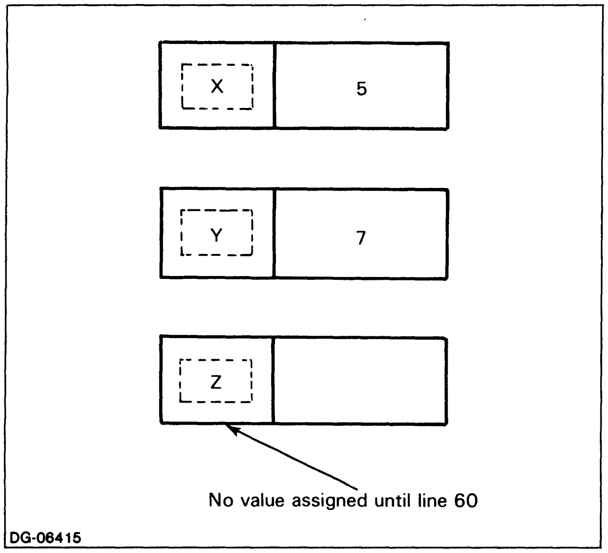


Figure 2-3. Variable Has No Value Before Assignment Statement

When the program reaches line 60, it reads the values of X and Y from their respective locations in memory, adds them up, and places the resulting value (12) into variable location Z. Figure 2-4 shows how the variables now look.

Note that X and Y have so far retained their original values, since we have only read these values from their location in memory, without writing anything else in.

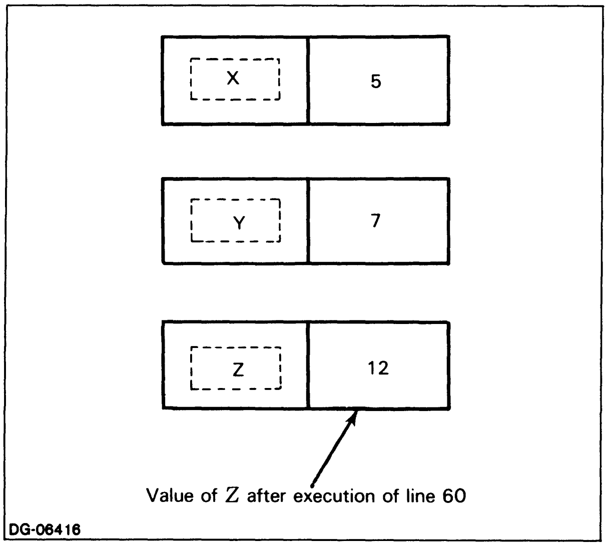


Figure 2-4. Variable After Assignment of a Value

In line 80, the value of Z changes once again. The program is asked to subtract 2 from the current value of Z and to assign the result (10) to Z as its new value.

At the end of the program run, variables X, Y, and Z will hold the values indicated in Figure 2-5.

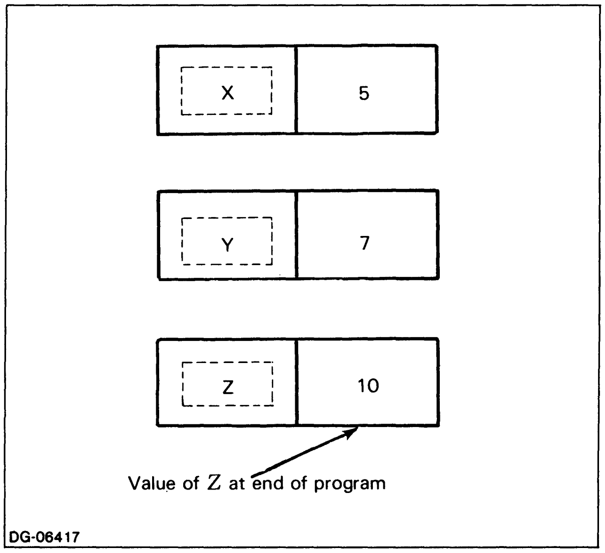


Figure 2-5. Change in Variable Z's Value

Initializing Variables

The term *initializing variables* refers to assigning beginning (initial) values to all variables used within a program. To explain this, let's refer back to the program excerpt we used earlier.

In that example, we created three variables, X, Y, and Z. X and Y were assigned their values fairly early in the program (lines 10 and 20 respectively), but Z, as you recall, received no value at all until the program reached line 60. Had we tried to use Z before we assigned it a value, we would have received an error.

A very important general rule is that you should not reference a variable before you have assigned it a value. Sometimes you will use an initializing statement near the beginning of your program to set a variable to an initial value, just to assure that it will have a value when it is referenced.

The initializing statement works just like an ordinary assignment statement, through the use of the keyword LET. The following could be the initializing statement for variable Z:

```
5 LET Z = 0
```

Numeric Constants

A constant is an entity whose value does not change during the execution of a program.

Numbers within a program are called numeric constants.

If you write, for example

```
LET N = 3 + X
```

the value of N and X may change during the execution of the program, since N and X are variables. But the value 3 will remain unchanged.

Constants can be positive or negative. For example, +2 and −56 are acceptable constants.

The positive (+) sign is optional, that is, in the absence of a plus (+) sign, the number is understood to be positive. The minus (−) sign, on the other hand, must always be written.

The range of numeric constants on MP/BASIC is −7.237E + 75 to +7.237E + 75

Representing Numeric Constants

You can write numeric constants in two different ways: decimal notation and scientific notation.

In decimal notation you write decimal digits, with or without the decimal point. Some examples are:

5.5978
−296.00
50

A program can contain numeric constants that have an arbitrary number of digits. MP/BASIC truncates the values of such constants to a precision of 15 decimal digits.

Scientific notation is a shorthand for writing very large or very small numbers.

If, for example, we ask the computer to print out the result of

```
50 * 30 * 100000
```

it will print

1.5E + 08

This means 1.50000 x 10^8, or 150000000.

Breaking this down, we know that

10^2 = 10 x 10 = 100
10^3 = 10 x 10 x 10 = 1000

and so on.

Hence,

1.5 x 10^2 = 150
1.5 x 10^3 = 1500

and so on.

Thus we see that multiplying 1.5 x 103 is the same as moving the decimal point three places to the right:

1.5 x 10^3 = 1500.

In the same way, 1.5 x 10^8 = 150000000.

In scientific notation, this is expressed as

1.5E + 08

where

- The letter E means "times 10 to the power."
- The number following the letter E is called the exponent;
- E followed by an exponent N indicates "times 10 to the power of N." (In our previous example, the exponent is positive: N = 8.)

E08 means "move the decimal point eight places to the right." E followed by any positive number means "move the decimal point to the right the number of spaces the exponent indicates."

You will notice that scientific notation illustrates the convention of writing expressions on a straight line: instead of writing 10^8, the computer writes E08.

In the case of a positive exponent, we don't need to write out the plus (+) sign; we can, instead, write the exponent number immediately after the E, thus: 1.5E08.

If you have a negative exponent, e.g.,

1.5 x 10^{-3}

it is the same as moving the decimal point three places to the left. So,

1.5 x 10^{-3} = .0015

As with positive exponents, 1.57828E−08 means 1.57828 x 10^{-8} which is the same as .0000000157828.

Note that the negative sign must always be written out.

You can use exponents as high as +99 or as low as −99 when entering constants in scientific notation as long as the limits of −7.237E + 75 to +7.237E + 75 are observed.

To summarize:

- A positive exponent is written with an optional plus sign (+) after the E, and it indicates moving the decimal point to the right;
- A negative exponent is written with a minus sign (−) immediately after the E, and it indicates moving the decimal point to the left.

Numeric Data Types

MP/BASIC supports two types of numeric data, INTEGER and REAL. (ANSI presently supports numeric data of type REAL only.)

As its name implies, an INTEGER is a positive whole number (1, 2, 3, ...), a negative whole number (−1, −2, −3, ...) or zero (0). An integer does not have a decimal point and cannot have a fractional part. The range of integer values is −32,767 to 32,767, inclusive.

A REAL number is a signed number which is written with a decimal point and may contain a fractional part.

MP/BASIC, without special declarations to the contrary, treats all numbers as REAL numbers. To declare part or all of your data as type INTEGER, use one of the following statement forms:

```
10 DECLARE INTEGER C, F, X(25), Z(12,15), C$ * 6
10 DECLARE ALL INTEGER
```

The variables following the first statement are a data list of the specific numeric and array variables you want treated as integers. All other data will remain unchanged and will be treated as REAL. (Note that this statement may also serve as your DIM statement for arrays and strings. See Chapters 3 and 7.)

The second form of the statement specifies that all data be treated as integers. Hence, there is no need for a data list following the keyword.

Working with integers offers several advantages:

- Integers occupy less memory space.
- Arithmetic operations are performed faster.
- Execution of FOR...NEXT loops is considerably speeded up when the initial value, the control variable, the increment and the limit are all integers.

Corresponding forms of these statements allow you to declare some or all of your data as real. The forms are:

```
10 DECLARE REAL X, Y, Z, A(50), B(20,20)
10 DECLARE ALL REAL
```

In the first statement, the keyword is followed with a data list; in the second, all data are declared as real.

These statements declare single-precision (6-digit) real numbers. You can achieve the same result by using the following forms:

```
DECLARE REAL*4 X,Y,Z,A(50),B(20,20)
DECLARE ALL REAL*4
```

To declare double-precision (15-digit) real numbers, use the forms:

```
DECLARE REAL*8 X,Y,Z,A(50),B(20,20)
DECLARE ALL REAL*8
```

The differences between single- and double-precision numbers is described in Appendix C.

As just explained, the declaration statement affects only variables following it.

Once a variable has been declared to be of a particular data type (either implicitly by the computer or explicitly through a DECLARE statement) in your program, you cannot change it to another data type.

Arithmetic Operators

MP/BASIC makes it possible for you to perform numerous kinds of calculations. This is done through the use of the standard arithmetic operations of addition, subtraction, multiplication, division, and exponentiation. (MP/BASIC also includes a number of mathematical functions, discussed in Chapter 8.)

The standard arithmetic operations are denoted by symbols called operators, which we summarize in Table 2-2.

Notice that MP/BASIC uses special symbols for multiplication and exponentiation.

You cannot write PRINT AB nor can you write PRINT AxB; rather, you must use the asterisk (*) which is the special symbol for multiplication.

You must use the circumflex to denote exponentiation.

Table 2-2. Arithmetic Operators

Operation	MP/BASIC Operator	Sample MP/BASIC Expression
Addition	+	A + 9
Subtraction	-	M - B
Multiplication	*	C * 6
Division	/	X/Y
Exponentiation	^	Z ^ 3

Operator Precedence

When an expression contains two or more operations, the sequence which MP/BASIC follows in the execution of these operations is determined by the rules of operator precedence.

These rules are summarized in Table 2-3, and explained in the text following the table.

It is important that you understand these rules, so you will be able to write your expressions in a way that will give you the results you want.

Table 2-3. Rules of Operator Precedence

Operator	Operation	Order of Precedence
( )	Parentheses	Evaluated first
^	Exponentiation	Evaluated second (if there are several, they are evaluated from left to right)
Unary + Unary -	Identity Sign inversion	Evaluated third Evaluated third
* and /	Multiplication and division	Evaluated fourth (if there are several, they are evaluated from left to right)
+ and -	Addition and subtraction	Evaluated last (if there are several, they are evaluated from left to right)

Parentheses

Expressions or parts of expressions which are contained inside pairs of parentheses ( ) are executed first.

For example, the program would execute the expression (5 + 6) * 2 in the sequence indicated in Figure 2-6.

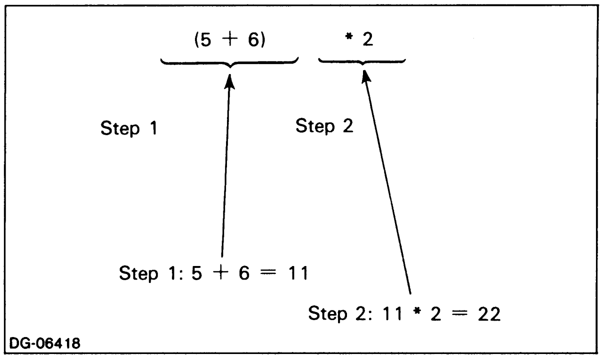


Figure 2-6. Operations within Parenthesis are Executed First

In the absence of parentheses, however, the order of execution would be different, as shown in Figure 2-7.

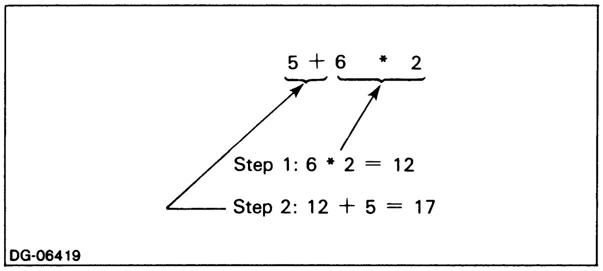


Figure 2-7. Sequence of Operations with Parenthesis Removed

In effect, then, the use of ( ) enables you to bypass the normal rules of precedence and thus to control the order in which you wish to have operations executed.

In cases where you have more than one set of parentheses (nested parentheses), the innermost set of parentheses is evaluated first.

Thus, the expression

(3 + 5 * (2 + 6)) * 2

would be evaluated as follows:

Step 1 : 2 + 6 = 8
Step 2 : 5 * 8 = 40
Step 3 : 3 + 40 = 43
Step 4 : 43 * 2 = 86

When you type in an expression as part of a program statement and then LIST that line, the LISTed version may have a different configuration of parentheses. As we mentioned in Chapter 1, this does not change the substance of the statement; in this case, it just shows you exactly how MP/BASIC will evaluate the expression.

Exponentiation

This operation is second in order of precedence. If several cases of exponentiation occur within an expression, they are executed in order, moving from left to right.

For example,

2^4 + 10 * 3^2

would be executed in the following sequence:

Step 1 : 2^4 = 16
Step 2 : 3^2 = 9
Step 3 : 10 * 9 = 90
Step 4 : 16 + 90 = 106

Multiplication and Division – These two operations are of equal priority; they have the same level of precedence. If there are several of them in an expression, they are executed in order, moving from left to right.

For example,

10/2 * 5

yields 25 if executed according to the rules of precedence:

Step 1 : 10/2 = 5
Step 2 : 5 * 5 = 25

If your intent is to divide 10 by 2*5 (which, in effect, would make you reverse the order of precedence and move from right to left), then you must use parentheses and write the expression thus:

10/(2 * 5)

Addition and Subtraction

These come last in the order of precedence. If there are several of them, their order is determined by moving from left to right.

Keywords in Chapter 2

Table 2-4 lists the keywords introduced in Chapter 2.

Table 2-4. Keywords in Chapter 2

Keyword	Can be used in	
	Program Statement	**Immediate Mode**
DECLARE [ALL] INTEGER	Yes	Yes
DECLARE [ALL] REAL	Yes	Yes
LET	Yes	Yes

End of Chapter

Chapter 3
Character Strings

So far you have familiarized yourself with numerical data. MP/BASIC also allows you to process information in the form of character strings.

A character string is a sequence of characters which is treated as a unit. Strings may be composed of any of the following, in any combination:

- uppercase alphabet characters, A through Z
- lowercase alphabet characters, a through z
- digits 0 through 9
- any of the characters in Table 3-1

Each of the items below is a string:

"BILLY THE KID"
"PT.109"
"AUGUST"
"30 RIVERSIDE DRIVE"
"*/$+=&?#!"

Note that each of these strings begins and ends with quotation marks. The quotation marks are not a part of the string, but are used to indicate to the computer that the intervening characters compose a string.

You have already encountered one use of character strings in REM comment statements. In this chapter we introduce you to the use of strings as data in the form of string variables and string constants. Chapters 5 and 8 describe string relational operations and string functions, respectively.

String Variables

A string variable, like a numeric variable, is a data item whose value can change during the execution of a program.

The program line

```
10 LET ST1$="FOOT"
```

creates a string variable, ST1$, and assigns a current value (the character string consisting of the four characters F, O, O, and T) to it. Just as with numeric variables, what this means is that the machine puts the string "FOOT" into a location in memory to which it gives the name ST1$, as shown in Figure 3-1.

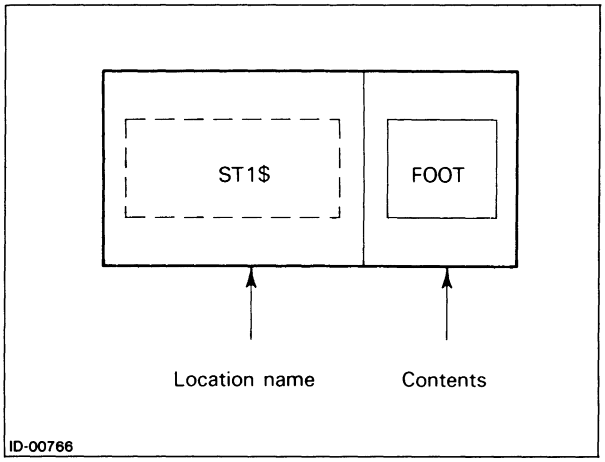


Figure 3-1. String Variable

Table 3-1. Symbols Used in Character Strings

Character	Symbol
Space	
Exclamation point	!
Number sign	#
Dollar sign	$
Percent sign	%
Ampersand	&
Apostrophe	’
Left parenthesis	(
Right parenthesis	)
Asterisk	*
Plus	+
Comma	,
Minus	−
Period	.
Slash	/
Colon	:
Semicolon	;
Less than	<
Equal	=
Greater than	>
Question mark	?
Circumflex	^
Underline	_

String Variable Names

As with numeric variables, a string variable is represented by a name to which a value is assigned during the execution of the program. String variable names must begin with an alphabetic character and end with a dollar sign ($). The dollar sign tells MP/BASIC that the variable is a string. Between the initial alphabetic character and the ending $, you can use any combination of uppercase and lowercase alphabetic characters, digits and the underscore character (_).

The length of the variable name is limited by the computer's memory. We suggest that you keep your variable names short so that they are easier to type and to prevent line length problems. For compatibility with AOS/VS BASIC, we suggest a maximum of 32 characters in variable names. The use of long variable names is an extension provided by MP/BASIC: the ANSI standard permits a string variable name to consist of only a single alphabetic character, plus the dollar sign ($), with an optional intermediate single digit (for example A$ or P3$).

Remember that the computer treats uppercase and lowercase characters the same. Thus, the names title1$ and TITLE1$ represent the same variable.

You can use the same base name for a numeric and a string variable within one program. MP/BASIC recognizes that COUNT and COUNT$, for example, are different. However, see Chapter 7 for restrictions on using the same variable name in single and double subscripted arrays.

Assigning a Variable Value

You can assign values to string variables through the LET statement. For example,

```
20 LET A$ = "APPLEPIE"
```

The INPUT and READ statements discussed in Chapter 4 are two additional ways of assigning values to string variables.

MP/BASIC also allows you to concatenate string variables. That is, you can combine the values of two or more string variables and assign the resulting value to another string variable. For example,

```
40 LET A$ = "BREAD"
50 LET B$ = "BASKET"
60 LET C$ = A$ & B$
70 PRINT C$
* RUN BREADBASKET
```

Notice that the ampersand (&) serves as the operator performing the string concatenation.

Length of String Variables

The ANSI standard sets the size of string variables in the range of null (0 characters) to a maximum of 18 characters. MP/BASIC expands this provision by allowing you to write string variables of any length.

Declaring String Variable Length: The DIM Statement

In order to write a string variable exceeding 18 characters, you must use a special statement called the DIM statement (short for dimensioning) to specify the dimension, i.e., the maximum number of characters your string may contain.

Assume, for instance, that you wish to work with a string variable, W$, which is to contain up to 23 characters. Your dimensioning declaration for this string would read as follows:

```
20 DIM W$ * 23
```

The effect of the DIM statement is to reserve memory space corresponding to the number of characters you have specified for your string. We have just reserved 23 such memory locations for string W$ in our sample dimensioning declaration above. In the absence of a DIM statement, the program automatically reserves a maximum of 18 memory locations for your string variables. We refer to this number as the default size of string variables (i.e., their size in default of a user-specified length).

Each string variable larger than 18 characters must be dimensioned separately, although several such statements may be combined on a single line, i.e.,

```
130 DIM A$*26, C1$*22, T3$*30
```

You may also wish to dimension string variables smaller than 18 characters in order to save memory space. If, for example, you are working with string A$, whose value may be either YES or NO, 15 of the 18 default memory locations made available for this string variable will remain unused. To ensure that you work only with the amount of memory you will actually need, it is good practice to dimension strings shorter than 18 characters, i.e.,

```
130 DIM A$*3
```

although this is not required by the system.

To repeat, you must dimension all string variables exceeding 18 characters; you may dimension string variables shorter than 18 characters in order to save memory.

In the case of strings that have been explicitly dimensioned, the program should encounter the DIM declaration before you ask it to execute any operations involving those strings.

Initializing String Variables: The Null String

In our discussion of numeric variables, we stressed the necessity of initializing all your variables to zero at the beginning of each program.

String variables, however, are automatically initialized by MP/BASIC. Their starting value is set to null (a string of 0 length) until such time as you supply other values to the program. You represent a null string by two adjacent quotation marks " ".

Writing Character Strings: The Use of Quotation Marks

A major convention in the writing of character strings relates to the use of quotation marks: all character strings must be within quotation marks (" ") except in REM, LINPUT, and DATA statements (see Chapter 4). Therefore, we suggest that you always use quotation marks to surround a string and, thus, prevent confusion.

You cannot imbed a quotation mark within a character string, except in the REM and LINPUT statements. In all other cases, MP/BASIC requires that you use quotation marks only to indicate the beginning and end of a string. To embed a quote within a string, use an apostrophe (') for the quoted portion, and enclose the whole string in quotation marks. For example,

```
20 PRINT "The missing word is 'money'."
```

would be printed as: The missing word is 'money'.

Referencing Substrings

Once you have created a string variable, MP/BASIC allows you to refer to selected portions or substrings of that string.

This means you can extract any part of a string by specifying its first and last character positions within the string. For example, assume you have created a string, C$, and assigned it the value "CASHFLOW". To reference the substring FLOW, you would write

```
40 PRINT C$(5:8)
```

BASIC then prints a string of four characters, starting with the fifth character in "CASHFLOW", the F, and ending with the eighth, the W.

To extract the substring "CASH", you would write

```
50 PRINT C$(1:4)
```

The system will respond with a four-character substring, beginning at the first character of string C$ and ending with the fourth.

You may have noticed that "CASHFLOW" contains only eight characters. If you request a substring whose last character position exceeds the length of your string, for example,

```
50 ; C$(2:10)
```

BASIC will ignore the erroneous ending reference and will print

ASHFLOW

a substring ranging from position 2 of C$ through it' entire length, in this case, from 2 through 8.

If you request a substring beginning at zero character of your string, for example,

```
50 ; C$(0:4)
```

BASIC will ignore the zero and extract the substring beginning from character 1 of your string, namely,

CASH

Finally, should you request a substring beginning after the last character of your string, for example,

```
50 ; C$(9:12)
```

the system will respond with a null string.

Similarly, if the starting and ending character positions of your substring are reversed for example,

```
50; C$ (5:4)
```

the system will return a null.

Assigning Values to Substrings

You can also assign a value to a selected portion of a string. This is an extension of MP/BASIC; it is not allowed in the ANSI standard. As with extraction of a substring, you must specify the first and last character position. For example,

```
LET STRING$="ABCDEF"
LET STRING$(3:4)="XX"
PRINT STRING$
ABXXEF
```

As you can see, the characters previously in positions 3 and 4 (CD) are replaced with XX.

If you specify a starting position which is greater than the number of characters currently in your string, your specified values are appended to your string. For example,

```
LET STRING$(8:9)="YY"
PRINT STRING$
ABXXEFYY
```

If you use character zero as your starting position, MP/BASIC will ignore the zero and begin the replacement at the first character. For example

```
LET STRING$(0:4)="ZZZZ"
PRINT STRING$
ZZZZEFYY
```

String Constants

A string constant (also called a literal) is a string whose value does not change during program execution.

We have already encountered string constants (although we didn't identify them as such) when we tried simple PRINT messages in Chapter 1.

Type the two program lines below, and substitute your name for the string variable assigned to N$:

```
30 LET N$ = "JAMIE"
40 PRINT "MY NAME IS "; N$
```

Here the message (MY NAME IS) printed by line 40 is a string constant, whereas N$ is, clearly, a string variable.

Notice that we left a space after "is" at the end of line 40; if we had omitted it, the run would have produced an output like this:

MY NAME ISJAMIE

Length of String Constants

A string constant can only be as long as the length of a program line; in MP/BASIC the program line, as we have noted earlier, is 156 characters long.

Keywords in Chapter 3

Table 3-2 lists the keywords in Chapter 3.

Table 3-2. Keywords in Chapter 3

Keyword	Can Be Used In	
	Program Statements	Immediate Mode
DIM	Yes	Yes

End of Chapter

Chapter 4
The INPUT and READ DATA Statements

In Chapter 2 we saw that a statement such as

```
LET X = 20
```

is one way of assigning values to program variables. We will now show you two additional ways of accomplishing the same thing.

The INPUT Statement

Suppose you have been given $5,000, and you want to find out how much yearly interest this money would earn you under each of several savings plans offered by your local bank.

Your formula for computing interest income is

I = C * R/100

where

I = Interest income
C = Your capital
R = Interest rate

Using one of the available options, say a 5% interest rate, you could then write the following program:

```
10 LET C = 5000
20 LET R = 5
30 LET I = C*R/100
40      PRINT      "YEARLY      INCOME
FROM";C;"AT";R;"%=";I
```

and then run the program:

```
RUN
Yearly Income From 5000 At 5% = 250
```

You'd probably think that the return on your investment is too low, so you'd try the 6% and 7% options, even though these would tie up your money for prolonged periods. To try these options, you must now rerun your program twice more, each time retyping line 20 so as to assign R its proper value:

first

```
20 LET R = 6
```

then

```
20 LET R = 7
```

Copying a whole line when only a small portion of it requires changing does seem like overkill, though. And what if you were trying out ten or fifteen different interest rates, instead of only three? Clearly this approach is impractical.

Here the INPUT statement neatly resolves our difficulties, because it will allow us to change values during the execution of a program without retyping any program lines.

To see how this works, change line 20 to read

```
20 INPUT R
```

and run the program.

What will happen now is that the computer prints a prompting question mark (?) when it reaches line 20.

This question mark tells you that the computer is waiting for you to feed it a piece of information. It will continue to wait until you supply (INPUT) the value requested (the value of R), followed by a NEW LINE character.

After you've typed in the value of R, the program will store it and use it as needed in continuing execution. The value of R you have just INPUT will thus supersede any previous value of R stored in your program.

So, when you now run the program your dialog with the computer will be as follows:

```
RUN
?6
Yearly income from 5000 At 6% = 300
```

After you typed 6 in response to the prompting *?* and struck the NEW LINE key, the program proceeded as before, and printed out the result of its calculation using a 6% interest rate.

Now try running the program while you input a 7% interest rate:

```
RUN
?7
Yearly income from 5000 At 7% = 350
```

In this way, you can painlessly try endless values for R by simply running the program and responding to the prompting ? with new values each time around.

You should type your value on the same line as the question mark, but if you strike the NEW LINE key before you input your value, the computer will considerately print another question mark and wait for your response.

Using Prompt Lines

To remind yourself during a run exactly what information your program is expecting you to supply, it is a good idea to have it print out a message spelling out exactly what data are needed at this time. The INPUT statement may easily be extended to include such a prompt line. This facility is an extension to the ANSI standard.

To illustrate with the interest calculating program we are working with, an INPUT line containing a prompt message might look as follows:

```
15 INPUT PROMPT "TYPE IN THE RATE OF INTEREST": R
```

Instead of simply typing

```
15 INPUT R
```

you type

```
15 INPUT PROMPT
```

followed by your message within quotes. A colon (:) follows the ending quotes; after the colon you list the variable(s) you wish to input.

At this point we should list our complete program to see where everything is (notice that, in line 30 below, the computer has added extra parentheses to clean up the format):

```
LIST

10 LET C = 5000
20 INPUT PROMPT "TYPE IN THE RATE OF INTEREST": R
30 LET I = (C*R)/100
40 PRINT "YEARLY INCOME FROM";C;" AT";R;"% = ";I
```

If we run the program now, the prompt line will be printed before the input is accepted.

```
RUN

Type in the rate of interest6
Yearly income from 5000 at 6 = 300
```

Notice how efficiently our prompt line reminds us precisely how to respond to the input request which follows it. When you write lengthy programs containing many INPUT statements, you will come to depend a great deal on such aids to memory.

A further advantage of this device is that it allows even persons unfamiliar with programming techniques to use any of your programs: all they need to do is to respond to the instructions you have built into the prompt lines.

Your message following INPUT PROMPT can be any string expression. In the example above, the message was a string constant, but you could have used a string variable or a concatenation of string variables and constants. For example, the statements

```
12 LET MSG1$="TYPE IN "
14 LET MSG2$="THE RATE OF INTEREST"
20 INPUT PROMPT MSG1$ & MSG2$: R
```

would produce the same prompt message as the statement on line 20 of the former example.

You may have noticed also that no question mark followed the INPUT PROMPT statement; if you want a question mark displayed, you simply include it in your prompt message.

The INPUT Statement with Several Variables

A program may contain more than one INPUT statement; also, you can use a single INPUT statement to enter a number of different variables at once. These variables may be of several types; that is, they may be string variables as well as numeric variables. Moreover, they may be listed in any order you please.

To illustrate this point, if several persons want to use your interest program, you can change it so as to input the person's name, the capital amount, and the interest rate.

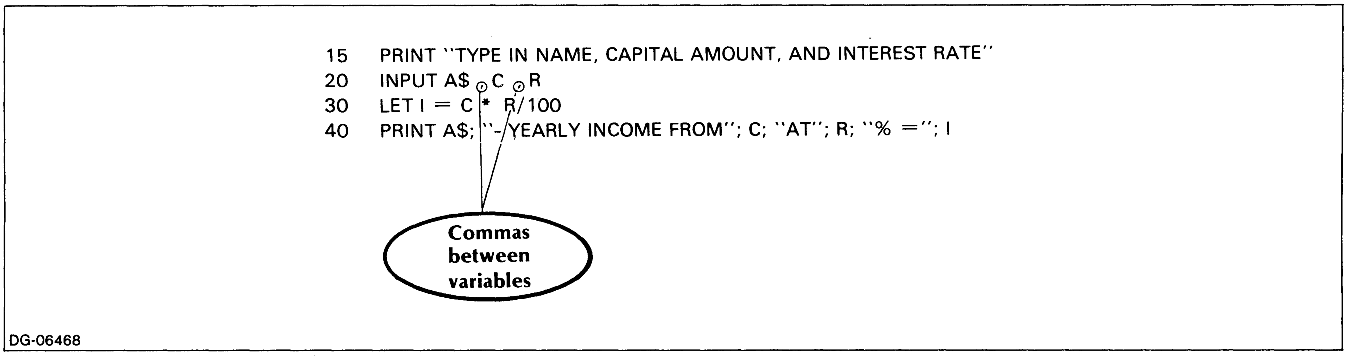
Your program may now look somewhat similar to Figure 4-1. (Incidentally, lines 15 and 20 of this program show you a different way to write prompt lines. This version separates the prompt message from the INPUT statement.)

Notice that we typed a comma after each of the variables in line 20, except for the last one, which is followed by a NEW LINE character. If you look at the output of the program (see Figure 4-2), you will see that we also use commas to separate the items we typed in response to the prompting question mark.

The reason is that commas and the NEW LINE character act as delimiters: that is, they allow the computer to distinguish the individual items within a list.

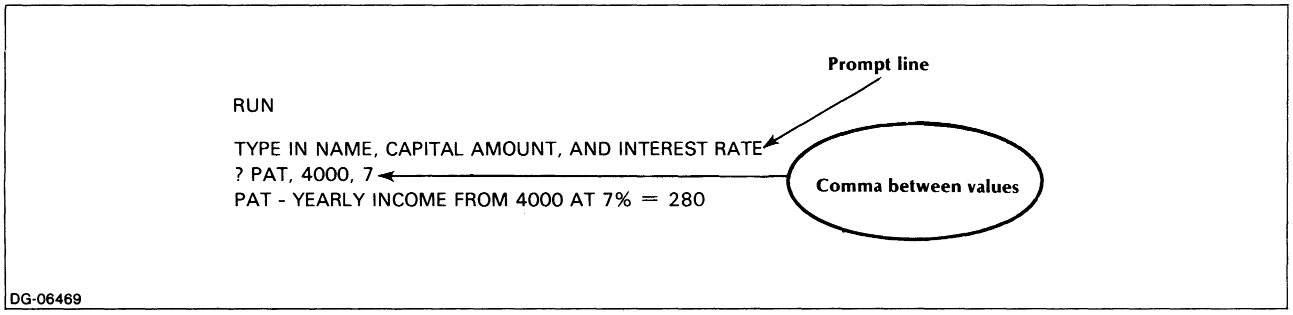
As a general rule, then, we must always use commas to separate items in a list.

As we said before, in line 20 of your program you could have listed the variables you wanted to input in any order whatsoever: you could, for instance, have written



```
15   PRINT "TYPE IN NAME, CAPITAL AMOUNT, AND INTEREST RATE"
20   INPUT A$ , C , R
30   LET I = C * R/100
40   PRINT A$; "- YEARLY INCOME FROM"; C; "AT"; R; "% ="; I
```

Figure 4-1. INPUT with Multiple Variables



```
RUN

TYPE IN NAME, CAPITAL AMOUNT, AND INTEREST RATE
? PAT, 4000, 7
PAT - YEARLY INCOME FROM 4000 AT 7% = 280
```

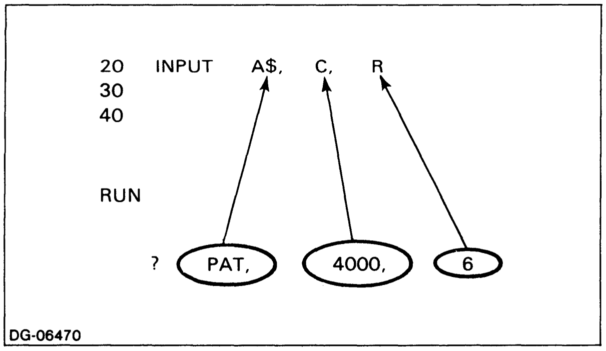
Figure 4-2. Dialog from Running Program in Figure 4-1

20 INPUT C, A$, R

or,

20 INPUT R, C, A$

But once you list the variables you'll want to input, the computer will expect to receive their values in exactly the same order as you listed them. Figure 4-3 shows how the values you input are associated with their corresponding variables when the program is run.



```
20   INPUT   A$,   C,   R
30
40

RUN

  ?  PAT,   4000,   6
```

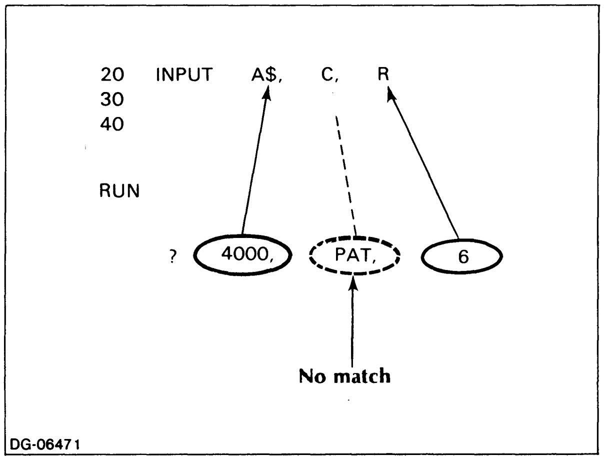
Figure 4-3. Correspondance Between Items as Input and Output

As you see, the process is very much like matching socks. Since A$ is the first item on your list, the computer expects the first value it sees in response to the question mark to be a character string constant which it can store in A$.

Moving from left to right, the computer will then expect numeric constants so that it can store them in C and R respectively.

Figure 4-4 shows what will happen if you mistakenly type

4000, PAT, 6



```
20   INPUT   A$,   C,   R
30
40

RUN

  ?  4000,   PAT,   6
```

Figure 4-4. An Error Condition from Mismatched Data Items

Since 4000 may be considered as a character string, the computer will assume that this is the value of A$. Next, it sees PAT, but since that value does not correspond to a numeric constant, the computer is unable to make the proper match between your second input and variable C. It is equally unable to assume the initiative of rearranging your values in proper order. You will therefore get the following error message:

Illegal data type in line 20

A new question mark (?) will then appear immediately below the error message; you must now re-input the entire data list from the beginning.

The best way to avoid such mishaps is to have your program print out prompt lines such as those we discussed in the previous section.

Figure 4-2 on the preceding page shows how the prompt line *Type in name, capital amount, and interest rate* in that program jogs your memory about the values required and the sequence in which you should list them.

In addition to inputting the values in their proper order, it is, of course, very important to supply the same number of values as the number of variables which your program specifies.

If you input

- more values than variables, or
- fewer values than variables for which values are needed

you will receive an error message, followed by the display of a ?. If that happens, you will need to re-input the entire list of your values, just as though you were beginning the process for the first time.

The LINPUT Statement

The LINPUT (line input) statement enters an entire line of string data and assigns it to a string variable. LINPUT differs from the INPUT statement in the following ways:

- LINPUT accepts only string data (if you erroneously give LINPUT a number as data, it will interpret the component digits as characters in a string);
- LINPUT accepts an entire line of data as it stands, including characters normally treated in special ways, such as quotation marks, commas, and apostrophes.

Normally you would terminate strings entered with LINPUT by pressing the NEW LINE character. LINPUT also accepts a carriage return, form-feed, and the null character as terminators.

If you do use quotation marks to enclose the character string for LINPUT, the quotation marks will be included in the string. Thus,

```
*10 PRINT "WHAT IS YOUR DESTINATION";
*20 LINPUT DESTINATION$
*30 PRINT DESTINATION$
*RUN
WHAT IS YOUR DESTINATION? "KANSAS"
"KANSAS"
```

To use one LINPUT line to accept several lines of input, specify variable names separated by commas in the LINPUT statement. For example:

```
* 10 DIM A$*80, B$*80, C$*80, D$*80
* 15 DIM E$*80, F$*80, G$*80
* 20 LINPUT A$, B$, C$, D$, E$, F$, G$
* 30 ;
* 40 ; A$
* 50 ; B$
* 60 ; C$
* 70 ; D$
* 80 ; E$
* 90 ; F$
* 100 ; G$
* RUN ?THE DODO SAT FOR A LONG TIME IN
THOUGHT,
?WITH ONE FINGER PRESSED AGAINST ITS
?FOREHEAD (THE POSITION IN WHICH YOU
?USUALLY SEE SHAKESPEARE, IN THE PIC-
TURES
?OF HIM,) WHILE THE REST WAITED IN SI-
LENCE.
?AT LAST THE DODO SAID, "EVERYBODY HAS
?WON, AND ALL MUST HAVE PRIZES."

THE DODO SAT FOR A LONG TIME IN
THOUGHT,
WITH ONE FINGER PRESSED AGAINST ITS
FOREHEAD (THE POSITION IN WHICH YOU
USUALLY SEE SHAKESPEARE, IN THE PIC-
TURES
OF HIM,) WHILE THE REST WAITED IN SI-
LENCE.
AT LAST THE DODO SAID, "EVERYBODY HAS
WON, AND ALL MUST HAVE PRIZES."
```

Using LINPUT PROMPT will allow you to combine a prompt message with your line input. The format is the same as that for the INPUT PROMPT command.

In a slightly different format, the INPUT and LINPUT statements may also be used with data files. (See Chapter 9.)

LINPUT is an extension to the ANSI standard.

READ Data

This is yet another way of feeding information to your computer: you list all your data in special lines called DATA lines, which you then instruct the computer to READ.

Let's illustrate this with a short program to read and process your weekly expenses. Since you'll want to classify these expenses by category, you need to read two kinds of variables:

- a string variable, indicating category of expense (food, etc.),
- a numeric variable, indicating dollar amount spent on a given category.

To simplify, we'll use only three expense categories and the variables shown in Table 4-1.

Table 4-1. Variables for Weekly Expenses Program

Expense Category	String Variable (Category)	Numeric Variable ($'s Spent)
Food	F$	F
Gas	G$	G
Recreation	R$	R

After it reads these variables, the program will tabulate your expenses by category, add them up, and calculate each individual expense as a percentage of the total amount you spent.

You'll need the following formulas:

S = F + G + R and
P1 = F * 100/S
P2 = G * 100/S
P3 = R * 100/S

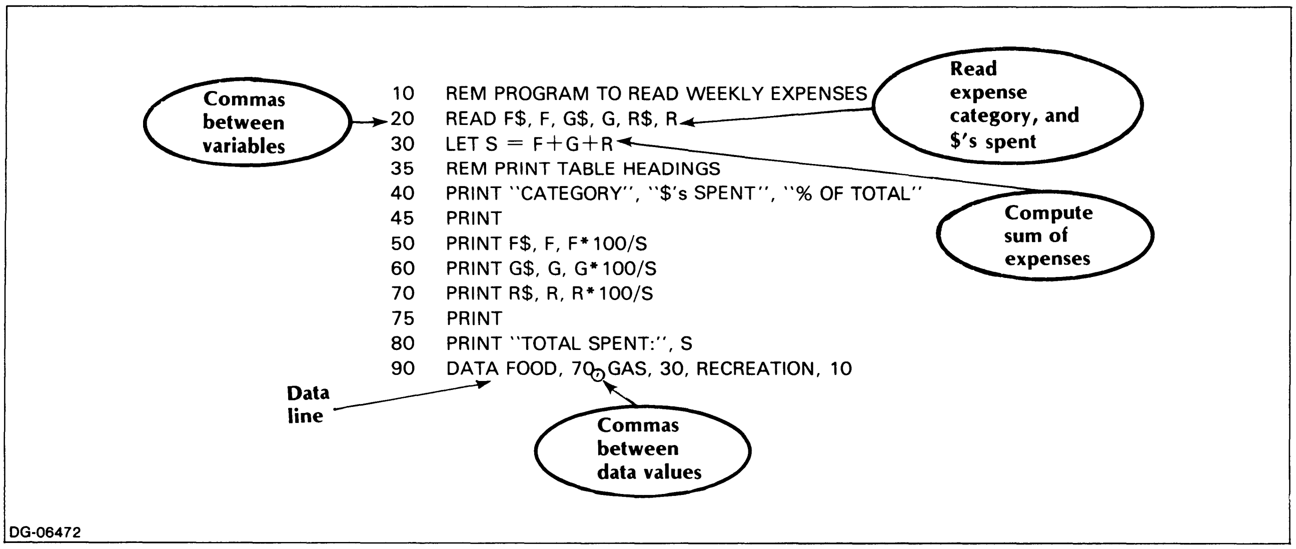
where

S = the sum of your expenses

P1, P2, and P3 = individual cost as percentage of total cost in each category.

Your program is now shown in Figure 4-5.

Before we run this program, we should point out some things about the format of the READ and DATA lines.



```
10   REM PROGRAM TO READ WEEKLY EXPENSES
20   READ F$, F, G$, G, R$, R
30   LET S = F+G+R
35   REM PRINT TABLE HEADINGS
40   PRINT "CATEGORY", "$'s SPENT", "% OF TOTAL"
45   PRINT
50   PRINT F$, F, F*100/S
60   PRINT G$, G, G*100/S
70   PRINT R$, R, R*100/S
75   PRINT
80   PRINT "TOTAL SPENT:", S
90   DATA FOOD, 70, GAS, 30, RECREATION, 10
```

Figure 4-5. Weekly Expenses Program

Use of Commas and Quotation Marks

We use commas to separate the variables in line 20, and the data in line 90. (See the discussion in the INPUT section.)

We use no quotation marks around strings in our DATA line. Even if a string consists of several words (e.g., Food and Drink), no quotation marks are necessary as long as the string does not include any of the characters requiring quotation marks (see Table 4-2).

If, for example, you want a list such as Food, Drink, Cigarettes to be read as a single string, you must enclose it in quotation marks.

As you can guess, without the quotation marks, the computer, seeing the commas between the words would read three separate strings instead of one. (See the discussion in the INPUT section.) But when you type

"Food, Drink, Cigarettes"

the quotation marks around these words will lead the machine to ignore the commas and to see a single string.

Table 4-2. Quoted Strings in DATA Statement

Character	Symbol
Ampersand	&
Apostrophe	'
Asterisk	*
Circumflex	^
Right parenthesis	)
Colon	:
Comma	,
Dollar sign	$
Equal sign	=
Exclamation point	!
Greater than	>
Less than	<
Number sign	#
Left parenthesis	(
Percent sign	%
Question mark	?
Semicolon	;
Slash	/
Underline	_

Proper Matching of Data; Multiple DATA Lines

During execution, a program reads data in exactly the same way as it reads values which you input. Moving from left to right, much like a pointer, the computer will match the first variable on its READ list with the first value on the DATA line, the second variable with the second value, and so on. Figure 4-6 illustrates this.

Hence, values must be listed in the DATA line(s) in exactly the same order as in the READ statement. This does not mean, however, that all your data must be listed on a single line: you can list data on as many lines as you wish, provided the order of their enumeration matches the order in which you will want to have the data read. The usefulness of this will become apparent in the next section. When a program encounters a DATA line it does not need at the moment, it will simply ignore it and move on to the following line. But regardless of how many DATA lines it has bypassed, the program will always find its way back to the DATA line whose turn it is to be read next.

It is, nonetheless, good practice to place your DATA lines close to their corresponding READ statements.

If your DATA line contains more values than the program needs to read, the excess data is simply ignored (see Figure 4-7).

If, however, you ask the program to read variables for which it can find no values in the DATA line (i.e., if, as in Figure 4-8, your READ list is longer than your data list), then the program will stop executing, and you will get an error message.

The computer will not accept gaps in DATA lines: even if you intend the value of, say, your last numeric variable to be a zero, you must type that zero into your DATA line, so that the program can read it.

Having finished reading the data, the program will continue normal execution. The output in our example might be as shown in Table 4.3.

Table 4-3. Output from Program in Figure 4-5

Category	$'S Spent	% of Total
FOOD	70	63.6364
GAS	30	27.2727
RECREATION	10	9.09091
TOTAL SPENT:	110	

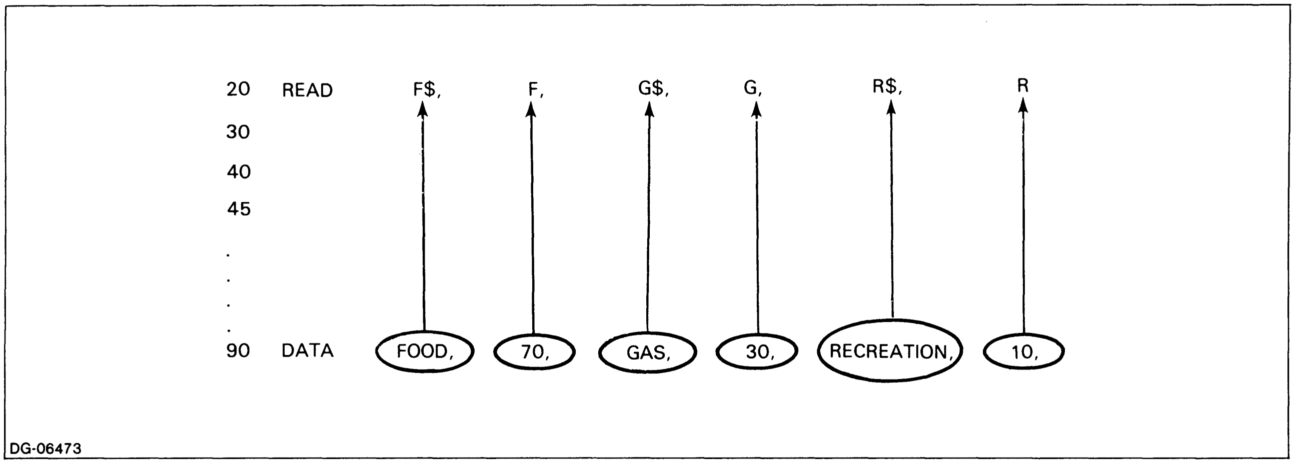

20 READ F$, F, G$, G, R$, R
30
40
45
90 DATA FOOD, 70, GAS, 30, RECREATION, 10,
DG-06473

Figure 4-6. Proper Matching of Data

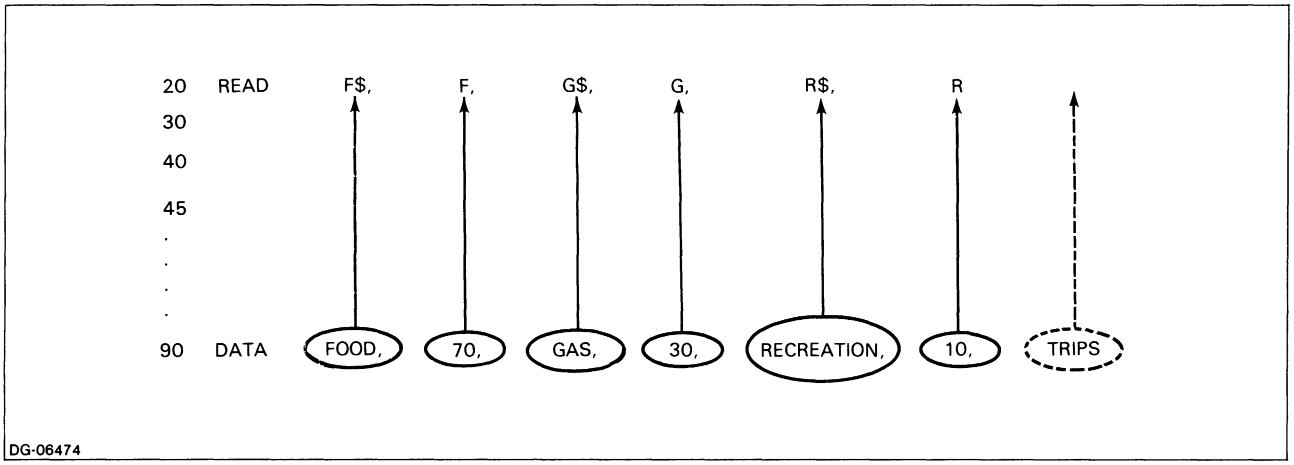

20 READ F$, F, G$, G, R$, R
30
40
45
90 DATA FOOD, 70, GAS, 30, RECREATION, 10, TRIPS
DG-06474

Figure 4-7. Data Mismatched: Excess Data Ignored

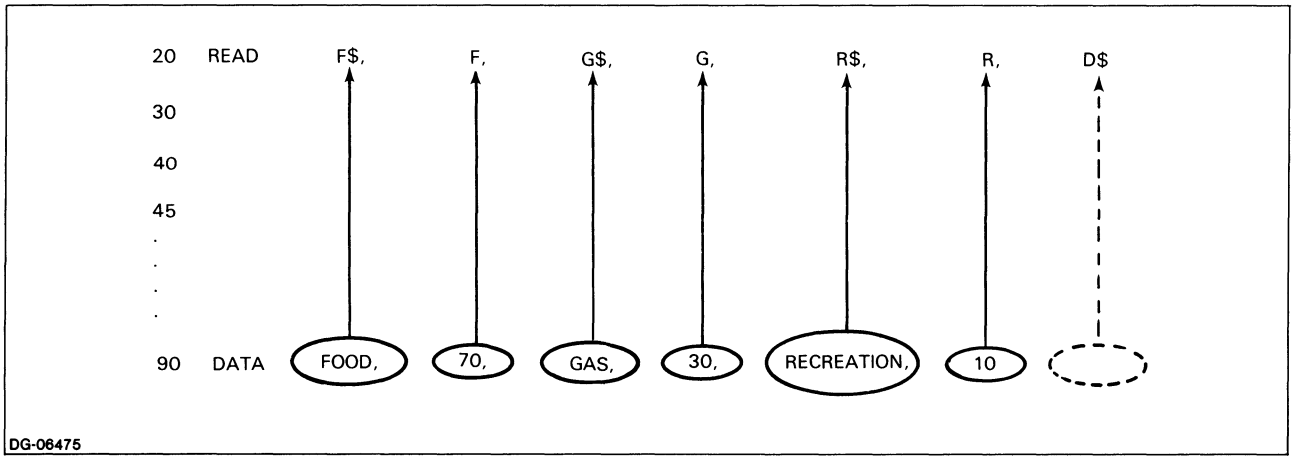
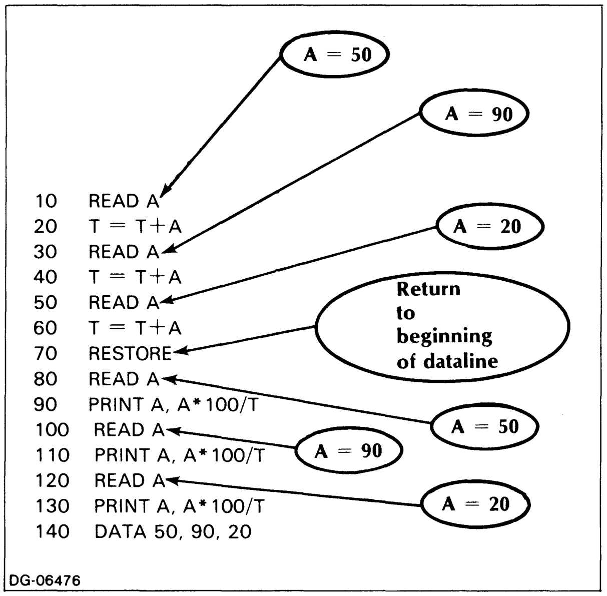

20 READ F$, F, G$, G, R$, R, D$
30
40
45
90 DATA FOOD, 70, GAS, 30, RECREATION, 10
DG-06475

Figure 4-8. Data Mismatched: Missing Data Causes an Error Condition

RESTORE

An interesting feature of READ DATA is that you can repeatedly re-read data by means of a RESTORE command. This command resets the pointer from the position it has reached on a given DATA line all the way back to the first DATA statement in the program. In this way, you are then able to read the data once again. Figure 4-9 illustrates this.



```
10   READ A
20   T = T+A
30   READ A
40   T = T+A
50   READ A
60   T = T+A
70   RESTORE
80   READ A
90   PRINT A, A*100/T
100  READ A
110  PRINT A, A*100/T
120  READ A
130  PRINT A, A*100/T
140  DATA 50, 90, 20
```

Figure 4-9. Using RESTORE to Re-read Data

Here we are using a single variable, A, and reading three values for it, without retaining them in memory. In other words, each time we read A, it loses its previous value and takes on the value which is next on the data line.

As we read each value of A we add it to a variable, T, which computes a cumulative total of A.

Once we have that total, we want to see what percentage of the total each value of A represents. Using the RESTORE command, we then re-read A, and print its value, as well as the percentage of the total it represents (lines 70 – 130).

RESTORE is useful in cases where you read a very large number of values without retaining them after you have performed the desired calculations with them. You might then want to reread your data in order to perform further analysis which would utilize the result of your previous calculations, and the RESTORE command allows you great flexibility in doing this.

Another useful feature of this command is that it can take a line number indicating which DATA statement is to be used for the next READ. (This is an MP/BASIC extension which is not at present included in ANSI Minimal BASIC.)

You can only RESTORE to the beginning of a DATA line. But by splitting data across lines (a practice we mentioned earlier), the successive READs can be made to start anywhere you wish.

In the example below, for instance, the line

```
40 RESTORE 30
```

causes the READ to begin on line 30. This way you can have only the value(s) on that DATA line re-read and assigned.

```
10 READ A,B,C
20 DATA 1,2
30 DATA 3
40 RESTORE 30
50 READ D
```

Keywords in Chapter 4

Table 4-4 lists the keywords introduced in Chapter 4.

Table 4-4. Keywords in Chapter 4

Keyword	Can be used in	
	Program Statement	Immediate Mode
DATA	Yes	No
INPUT	Yes	No
INPUT FILE	Yes	No
INPUT PROMPT	Yes	No
LINPUT	Yes	No
LINPUT PROMPT	Yes	No
READ	Yes	No
RESTORE	Yes	No

End of Chapter

Chapter 5
Control Statements: Branching and Loops

So far, our programs have been executed straight down in the order of their line numbers. Most of the time, however, you will need the flexibility of building alternate execution options into your programs. You might, for example, want to branch temporarily to other parts of a program, or you might want to keep re-executing a given set of commands for a specified number of times.

To do any of these things, you need a mechanism which allows you to control the flow of your program, and to interrupt and resume straight sequential execution at will.

Collectively, the control statements we discuss in this chapter constitute such a mechanism: temporary branching away from the main program is made possible by unconditional and conditional branching commands, while repeated execution of specific commands is accomplished by means of loops (single, consecutive, or nested).

In this chapter we also introduce you to some basic aids to good programming, such as flowcharting, and we use flowcharts to illustrate the new material presented here.

Unconditional Branching

The GOTO... Statement

Suppose you want to write a program that will let you input the daily temperature (in Fahrenheit) for each of the first seven days of a given month, after which it will convert that temperature to Centigrade and print out both the Fahrenheit and its Centigrade equivalent.

The formula for converting Fahrenheit to Centigrade is:

C = (F-32) * 5/9

where

C = Centigrade

F = Fahrenheit

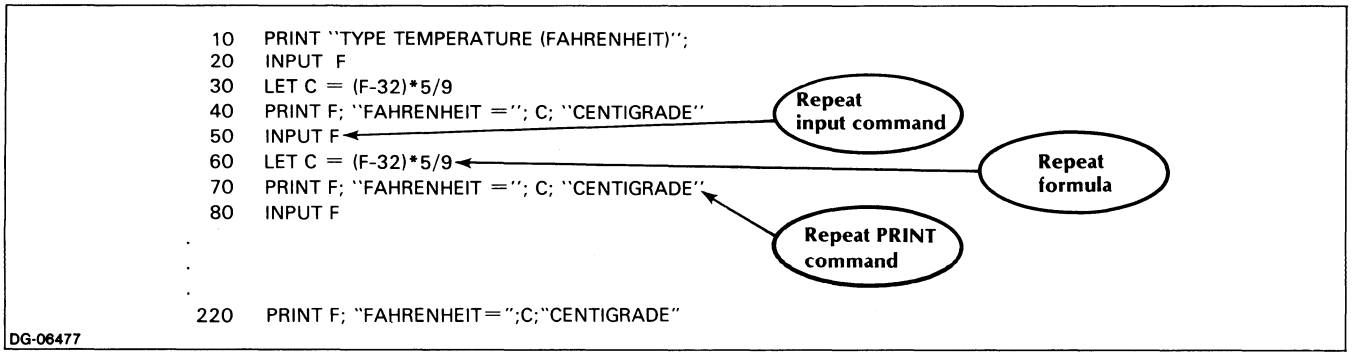
One way to write this program would be as shown in Figure 5-1.

As you see, the basic program consists of only three one-line instructions, namely: input the temperature, convert it, print the result. However, in order for us to process each of our seven temperatures, we must retype these three instructions (lines 20, 30, and 40) seven times, thus expanding the original three lines of our program to twenty-one.

We need a way to keep recycling our original set of instructions in lines 20–40 as many times as necessary, without having to retype them.

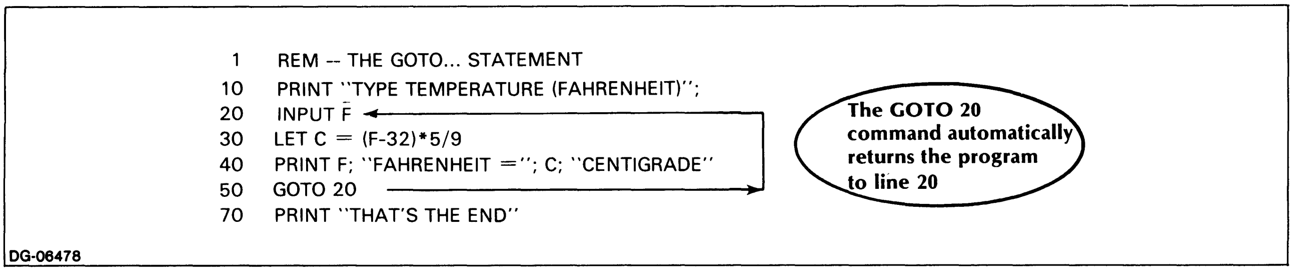
One solution is to build in some signal which would return us to line 20 (INPUT F), each time we reach the end of line 40 (PRINTF,C).

The GOTO... statement is such a signal. It tells the program to interrupt its normal sequence of execution, and to branch to a line number which you specify; this line number may refer to a line occurring much earlier or much later in the program.



```
 10   PRINT "TYPE TEMPERATURE (FAHRENHEIT)";
 20   INPUT F
 30   LET C = (F-32)*5/9
 40   PRINT F; "FAHRENHEIT ="; C; "CENTIGRADE"
 50   INPUT F
 60   LET C = (F-32)*5/9
 70   PRINT F; "FAHRENHEIT ="; C; "CENTIGRADE"
 80   INPUT F
 .
 .
 .
220   PRINT F; "FAHRENHEIT=";C;"CENTIGRADE"
```

Figure 5-1. Program to Convert Degrees Fahrenheit to Degrees Centigrade



```
1    REM -- THE GOTO... STATEMENT
10   PRINT "TYPE TEMPERATURE (FAHRENHEIT)";
20   INPUT F
30   LET C = (F-32)*5/9
40   PRINT F; "FAHRENHEIT ="; C; "CENTIGRADE"
50   GOTO 20
70   PRINT "THAT'S THE END"
```

The GOTO 20 command automatically returns the program to line 20

Figure 5-2. Temperature Conversion Program Incorporating a GOTO Statement

In our example, a GOTO... command (written as a single word), would appear immediately after line 40 and would look as follows:

50 GOTO 20

Our revised, and radically shortened, program is now as shown in Figure 5-2.

Figure 5-3 illustrates the effect of the GOTO statement: each time the program reaches line 50, it is redirected back to line 20. This certainly solves our original difficulty of having to retype all instructions the same number of times as the number of data we want to process.

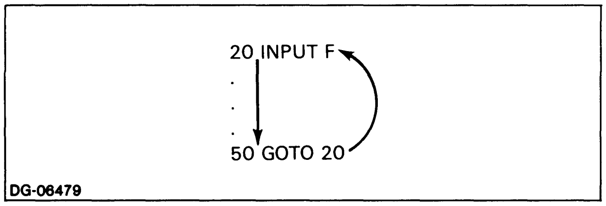


Figure 5-3. Effect of the GOTO Statement

Looking at this figure, though, you might well wonder how you de-activate the GOTO 20 command, once you have input all your data: judging by this diagram, we seem to be locked into an endless circle here. That is indeed the case, and we will deal with it very shortly. But first, let us illustrate the GOTO command yet another way, by introducing the concept of the flowchart.

Flowcharts

Flowcharts are a visual layout of the logic of your program, represented in a set of boxlike symbols. As your programs get more complex, you will find that diagramming them in flowcharts will help you avoid any number of procedural and logical mistakes.

In Figure 5-4 we have drawn a flowchart of our sample program in Figure 5-2.

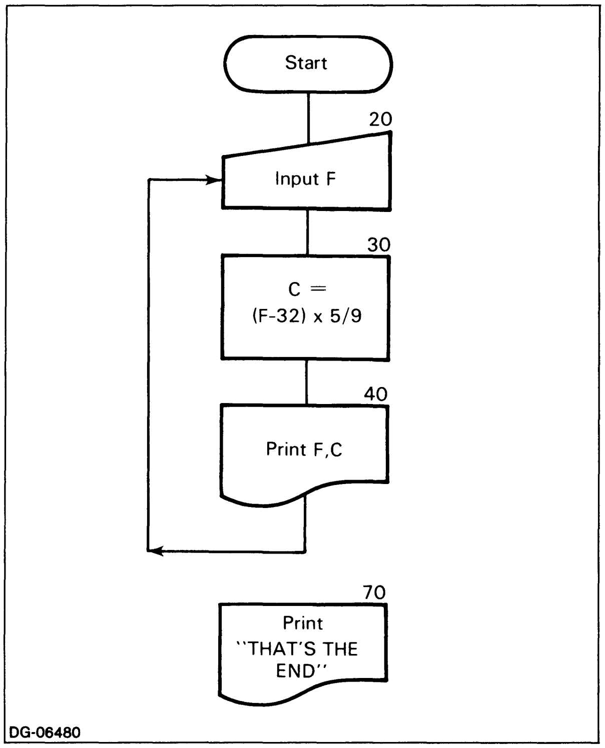


Figure 5-4. Flowchart for Temperature Conversion Program

If you compare the flowchart with the program, you will see that each box in the diagram represents a major logical step in the program; notice also that over each box we place the corresponding program line number(s) for ready reference. The arrows connecting the boxes show the order in which program execution proceeds.

Disregard for the moment the shapes of the flowchart boxes. These are determined by the type of process the box indicates (input, print, etc.) and are explained in Figure 5-9. For now, follow the arrows and note that the program flows sequentially from its start to the PRINT F, C box representing line 40.

At this point, rather than having a separate box for the GOTO 20 statement, we use an arrow to indicate the direction of the branch from line 40 back up to line 20.

Infinite Loops

Notice that no arrows connect the last two boxes in the flowchart (lines 40 through 70). This highlights the problem we have mentioned earlier: as now written, the program will never reach line 70, the line which should print out *THAT'S THE END.* Instead, it will be endlessly redirected to line 20, where it will keep printing question marks in expectation of input.

The reason for this is that we have specified no set of conditions that would render the GOTO 20 statement inoperative and prevent it from being executed.

A GOTO... of this type is called an unconditional control statement; the exitless, endlessly circular program flow it created in this example is known as an infinite loop.

To exit from an infinite loop, you must stop the program's execution by striking the CTRL-C, CTRL-A key combination. (As we mentioned in Chapter 1, you can use the ESC key instead of the CTRL-C, CTRL-A combination by typing the CHARACTERISTICS/ON/ESC command from the CLI, prior to entering MP/BASIC.)

We see from the above that, in order to make a control statement really useful, we should always use it in conjunction with specific conditions under which it can be deactivated. One way of achieving this is through conditional branching.

Conditional Branching

The IF...THEN... Statement

The IF...THEN... statement is a built-in test that allows the program to determine automatically which of two alternative routes it should choose during execution.

This statement can be translated as follows:

- IF condition X is True, THEN do Z;
- IF condition X is False, THEN do Y.

Any valid BASIC statement may serve as the object of a THEN clause, e.g.,

```
IF X = 10 THEN PRINT X
IF X = 10 THEN LET Y = 0
IF X = 10 THEN READ Y
```

The Decision Box

Flowcharts illustrate the operation of the IF...THEN... statement by means of a diamond shaped box called the decision box, as shown in Figure 5-5.

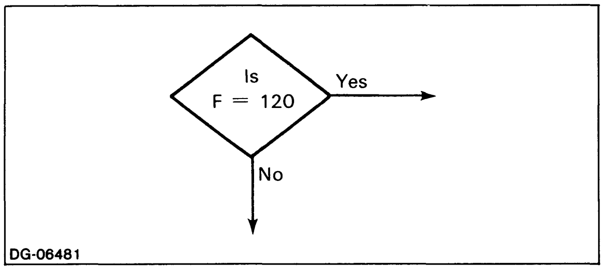


Figure 5-5. Decision Box

As you can see, the decision box functions much like the roadside signpost which points the traveller to two alternate destinations: if the statement in the box is True, the YES arrow points the program to one branch; if False, the NO arrow directs the program to a different set of operations.

NOTE: In our example, the NO arrow will cause the program to continue normal sequential execution, whereas the YES arrow diverts it to a branch; but there is no set rule about this placement which depends entirely upon your own convenience.

Relational Operators

The test of whether or not a given condition is True is performed by means of relational operators.

These operators allow us to perform comparisons that determine the relationship of variables, constants, or expressions to each other.

Relational operators (as well as logical operators, which we will discuss below) connect constants, variables, and expressions to form logical expressions. A logical expression has one of two values: True or False.

There are six relational operators, and we summarize them in Table 5-1.

Table 5-1. Relational Operators

Relational Operator*	Example	Meaning
=	A = B	A is equal to B
<>	A <> B	A is not equal to B
>	A > B	A is greater than B
<	A < B	A is less than B
>= or =>	A >= B A => B	A is greater than or equal to B
<= or =<	A <= B A =< B	A is less than or equal to B

*The relational operators => and =< are an MP/BASIC feature not presently included in ANSI Minimal BASIC.

String Comparisons

You can use the relational operators to perform comparisons between two strings, asking, for example, whether A$ = B$. In normal alphabetizing, the reader would order words according to the sequence of the letters in the alphabet, i.e., BACON before BATTER, CHERRY before CHOCOLATE, and so on.

You recall that all characters are represented inside the computer by numeric codes. These are listed in the ASCII character set. (See Figure 1-1.)

The sequence in which these characters are ordered is known as the ASCII collating sequence. In the case of letters, this sequence corresponds to conventional alphabetic sequence.

In performing string comparisons, the computer does a left-to-right comparison based on the ASCII collating sequence of the numeric codes making up the characters of the strings being compared (including such characters as leading and trailing spaces).

It follows that any two character strings can be compared, regardless of whether or not they are alphabetic characters. The computer will, for example, consider the character ! to be less than the character % since ! precedes % in the ASCII collating sequence. A% is thus greater than A!.

The following considerations apply in the comparison of strings:

1. To be considered equal, the two strings must be of the same length and must contain an identical sequence of characters, including spaces. For example,

   "ALPHABET"

   would not be equal to

   "ALPHABET "

2. If one string is equivalent to the leftmost portion of another string, the shorter precedes the longer in alphabetical sequence. If, for example,

   A$ = "PIN"
   B$ = "PINPOINT"

   then A$ will precede B$ in alphabetical sequence. In other words the condition A$<B$ will be True.

3. Upper and lowercase alphabetic characters are not considered equal. Notice that the upper case alphabetic characters precede the lower case alphabetic characters in ASCII collating sequence. For example,

   "PIN"

   would precede

   "pin"

   and

   "ZEBRA"

   would precede

   "antelope"

String comparisons are useful in connection with control statements. You will find an example of this in Figures 5-10 and 5-11 later in this chapter.

Using IF...THEN

If you use the IF...THEN... statement in conjunction with a branching command, you will make the branch conditional on the result of the relational operation test, rather than automatic (unconditional), as it was in our previous program.

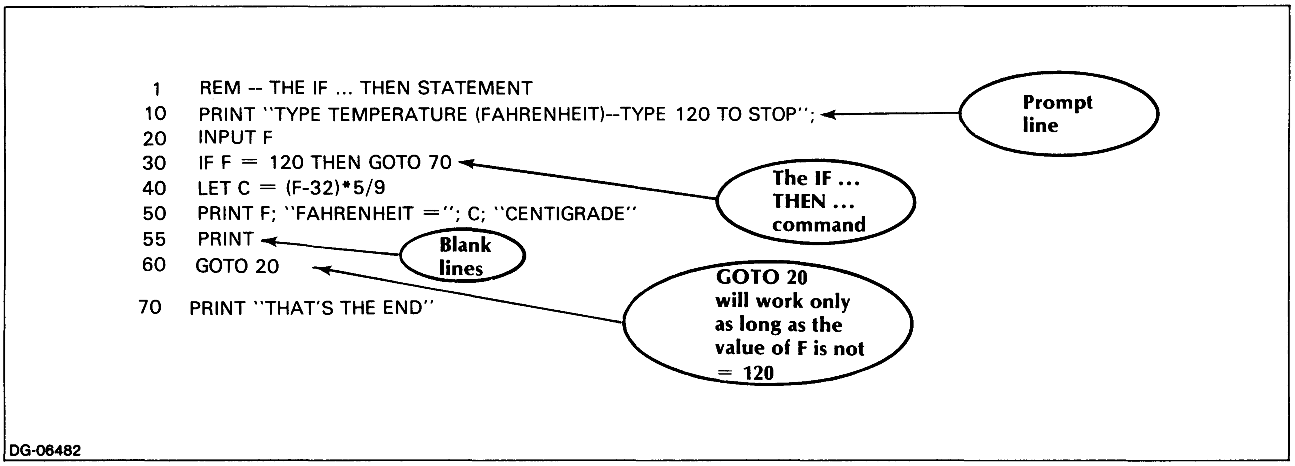
We will reuse the conversion program (Figure 5-2) to illustrate.

As you recall, we input daily temperatures in Fahrenheit, converted them to Centigrade, and printed out the results, using the GOTO 20 command to repeat this sequence.

Now we will add one further instruction, causing the program to recognize a given value of F, say 120, as the condition for ignoring the GOTO 20 command and resuming normal execution instead.

We write this as follows:

```
.
.
```



```
1    REM -- THE IF ... THEN STATEMENT
10   PRINT ''TYPE TEMPERATURE (FAHRENHEIT)--TYPE 120 TO STOP'';
20   INPUT F
30   IF F = 120 THEN GOTO 70
40   LET C = (F-32)*5/9
50   PRINT F; ''FAHRENHEIT =''; C; ''CENTIGRADE''
55   PRINT
60   GOTO 20
70   PRINT ''THAT'S THE END''
```

Prompt line
The IF ... THEN ... command
Blank lines
GOTO 20 will work only as long as the value of F is not = 120
DG-06482

Figure 5-6. Temperature Conversion Program Using an IF...THEN

```
20 INPUT F
30 IF F = 120 THEN GOTO 70
40 LET C=(F-32)*5/9
50 PRINT F;" FAHRENHEIT =";C;" CENTIGRADE"
60 GOTO 20
70 PRINT "THAT'S THE END"
```

Now, each time the program reaches line 30, it will test the value (2) of F you have just input in line 20, by asking:

Is F = 120?

If this condition is True, i.e., if you have just input 120 as the value of F, the program reads and executes the second part of line 30, i.e., the THEN clause, which directs it to branch to line 70.

Thus, the program will now go directly to line 70, bypassing lines 40 through 60 and exiting from what was previously an infinite loop.

If you have input a value other than 120 for F, the condition F= 120 is False. In such a case, the program will not reach the THEN clause of line 30. It will, instead, move to the line immediately following line 30, eventually reaching and executing the GOTO 20 statement (2) of line 60, which will repeat the entire cycle once again.

Note that there are only two possibilities: the condition can be either True or False. Hence, the program interprets an IF... THEN... statement as follows:

IF condition X is True, THEN do Z; otherwise, proceed normally.

This does not mean that you must formulate your statement so as to test for equality only; you could, for example, also write line 40 as follows to test for inequality:

```
30 IF F <> 120 THEN GOTO 20
```

(Note that in this version we need to rephrase the instructions, contingent upon the test results: the GOTO 20 instruction of line 60 has now been incorporated into the second half of line 30; hence, line 60 can be eliminated. The LET and PRINT statements must also be moved from lines 40 and 50 back to line numbers preceding line 30, in order to get a printed output of daily temperatures.)

Now the program will test each value of F you input to see if the condition

F is not equal to 120

is True. If it is, the THEN clause of line 30 will be executed, sending the program back up to line 20.

If the condition is False, and you have input 120 as the value of F, the program will ignore the rest of line 30, moving in normal sequence directly to line 70.

In addition to testing for equality and inequality, you can, with appropriate program modifications, formulate your IF...THEN... line so as to apply any of the six relational operators in Table 5-1.

The GOTO statement is optional in line 30; you can just type

```
30 IF F <> 120 THEN 20
```

When the system formats your program, it will insert the keyword GOTO after THEN, so that a listing of the program will display the IF...THEN... statement in its extended form.

Example

In Figure 5-6 we relist our former program, now modified through use of the IF...THEN... statement.

A flowchart for Figure 5-6 could look like the one shown in Figure 5-7.

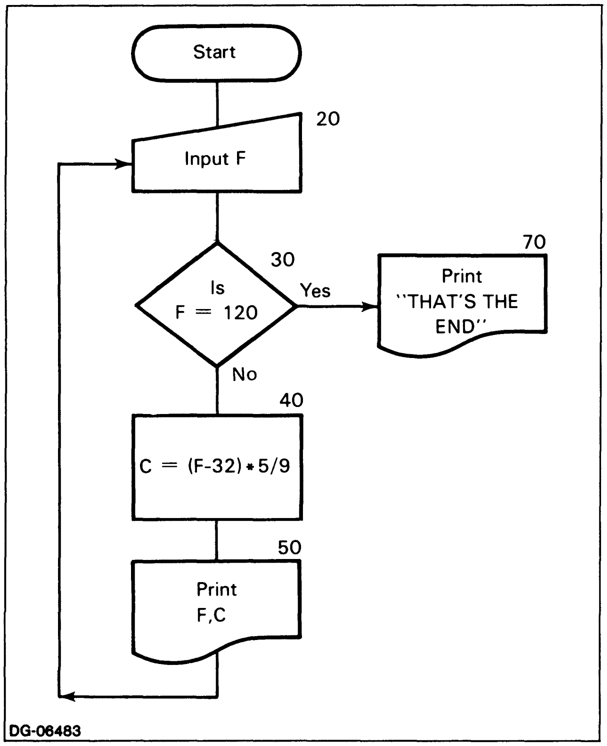


Figure 5-7. Flowchart for IF...THEN Version of Temperature Conversion

Line 30, which tests whether F= 120, is represented by the decision box in the flowchart. The YES and NO arrows point to the alternative flow directions, contingent upon the result of the relational operation performed on F.

We will test the program by running it and typing in temperature values for 10 days of any given month (see Figure 5-8).

The IF...THEN... statement worked as expected: as soon as we input 120 for the value of F, the program ignored lines 40 through 60, proceeded to line 70, printed *THAT'S THE END*, and terminated the program.

You can build into your programs as many of these conditional branching statements as you wish; be wary, however, of overusing this device, since too many deviations from normal sequence may make your program logic difficult to follow.

The IF...THEN...ELSE Statement

You may want to perform one action if a condition is True, perform another if it is False, and then proceed with normal program execution in both cases. You can do this by adding an ELSE clause to the IF... THEN statement. The ELSE clause can contain any BASIC statement that can be in the THEN clause.

For example, assume you want to input two numbers, subtract the second from the first, and print a message to tell you if the result is positive or negative. You could write it as follows:

```
10 INPUT PROMPT "ENTER A NUMBER:":A
20 INPUT PROMPT "ENTER A NUMBER:":B
30 IF (A-B)>=0 THEN PRINT "POSITIVE"
40 PRINT "NEGATIVE"
50 PRINT "FINISHED"
```

If the result is positive, however, you will receive both messages because the computer executes the THEN clause and then executes the next statement. To avoid this problem, you can combine statements 30 and 40 as follows:

```
30 IF (A-B)>=0 THEN PRINT "POSITIVE" ELSE PRINT
"NEGATIVE"
```

Now, the program works properly.

Note that both the THEN and ELSE clauses could contain GOTO statements; in such a case, the next sequential line would not be executed unless it is used in another GOTO statement somewhere in the program. You will see the use of this type of IF...THEN...ELSE statement when we discuss subroutines in Chapter 6.

Logical Operators

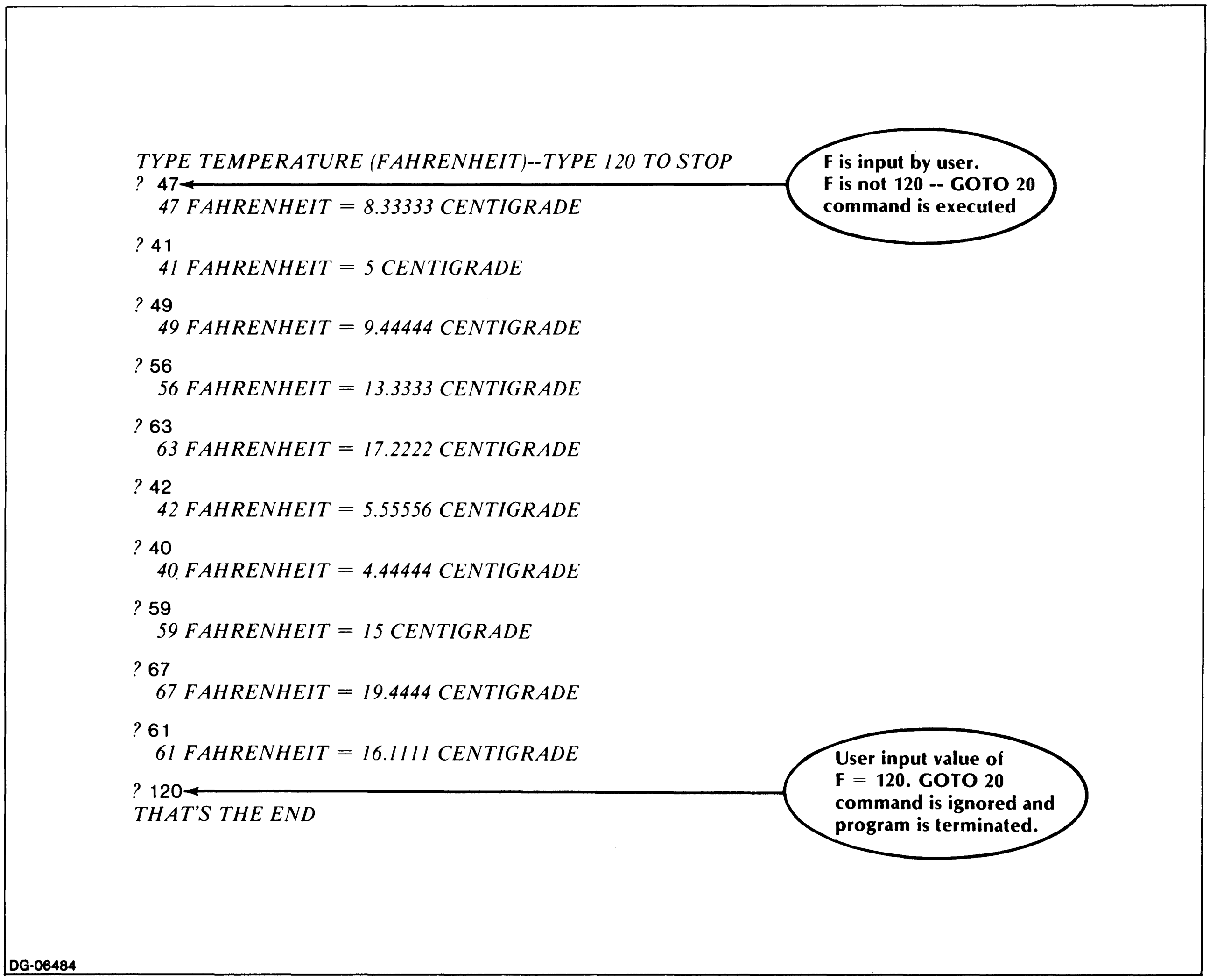
As we mentioned earlier in this chapter, you can use logical operators and relational operators to connect constants, variables, and expressions into logical expressions. The logical operators are AND, OR, and NOT.

Suppose, in our temperature conversion program, we want to see if the temperature entered is a reasonable fahrenheit number (we'll define reasonable to be between 0 and 100). We could add the following statements to our program in Figure 5-6:

```
32 IF F>=0 THEN GOTO 34 ELSE GOTO 36
34 IF F=<100 THEN GOTO 40
36 PRINT "UNREASONABLE TEMPERATURE - TRY
AGAIN"
38 GOTO 20
```

In these statements, we are checking to see if the number entered is greater than or equal to zero and also less than or equal to 100 (a number which is greater than zero could also be greater than 100). The logical operator AND would permit us to combine the effect of the four statements above into the following:

```
32 IF F>=0 AND F<=100 THEN GOTO 40
34 PRINT "UNREASONABLE TEMPERATURE - TRY
AGAIN"
36 GOTO 20
```



```
TYPE TEMPERATURE (FAHRENHEIT)--TYPE 120 TO STOP
? 47
  47 FAHRENHEIT = 8.33333 CENTIGRADE

? 41
  41 FAHRENHEIT = 5 CENTIGRADE

? 49
  49 FAHRENHEIT = 9.44444 CENTIGRADE

? 56
  56 FAHRENHEIT = 13.3333 CENTIGRADE

? 63
  63 FAHRENHEIT = 17.2222 CENTIGRADE

? 42
  42 FAHRENHEIT = 5.55556 CENTIGRADE

? 40
  40 FAHRENHEIT = 4.44444 CENTIGRADE

? 59
  59 FAHRENHEIT = 15 CENTIGRADE

? 67
  67 FAHRENHEIT = 19.4444 CENTIGRADE

? 61
  61 FAHRENHEIT = 16.1111 CENTIGRADE

? 120
THAT'S THE END
```

F is input by user.
F is not 120 -- GOTO 20
command is executed

User input value of
F = 120. GOTO 20
command is ignored and
program is terminated.

DG-06484

Figure 5-8. Sample Run of Temperature Conversion Program (IF...THEN Version)

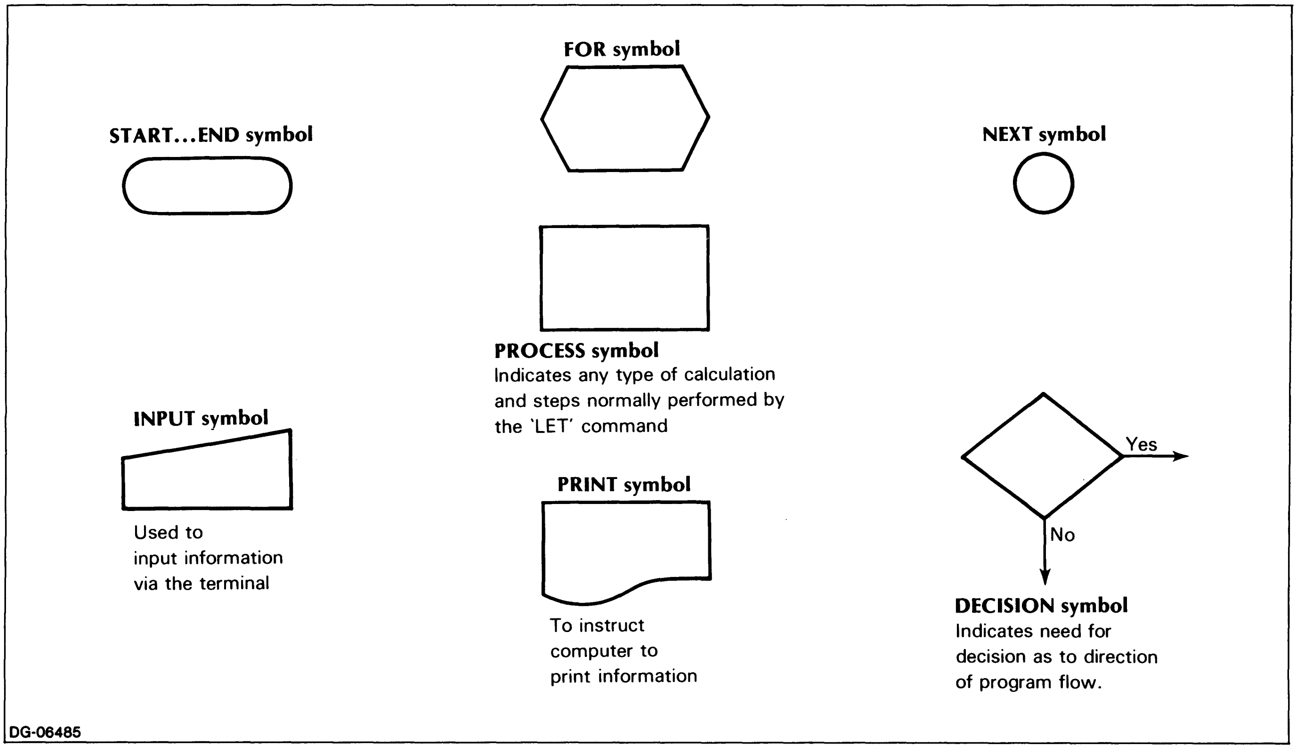

FOR symbol
START...END symbol
NEXT symbol
PROCESS symbol
Indicates any type of calculation and steps normally performed by the 'LET' command
INPUT symbol
Used to input information via the terminal
PRINT symbol
To instruct computer to print information
Yes
No
DECISION symbol
Indicates need for decision as to direction of program flow.
DG-06485

Figure 5-9. Flowchart Symbols

The logical expression formed by connecting two or more relational comparisons by the logical operator AND is True only if all of the component relational comparisons are True, and is False in all other cases. Thus, the logical expression

F>=0 AND F<=100

is True for all numbers between 0 and 100 (including 0 and 100), and is False otherwise.

A logical expression consisting of two or more logical expressions connected by the logical operator OR is True if at least one of the component logical expressions is True, and is False only if all components are False. Thus, the logical expression

F<0 OR F>100

is False for all numbers between 0 and 100 (inclusive) and is True otherwise. Using the OR operator, another alternative to the lines we just wrote would be:

```
32 IF F<0 OR F>100 THEN GOTO 34 ELSE GOTO 40
34 PRINT "UNREASONABLE TEMPERATURE - TRY
AGAIN"
36 GOTO 20
```

The logical operator NOT reverses the truth of an expression. For example, if A=B then the logical expression

A=B

is True, and the expression

NOT (A=B)

is False. Notice the use of parentheses in this example. NOT is considered a unary operator and requires parentheses in much the same way as the unary + and - arithmetic operators.

Thus, the NOT operator suggests yet another variation on our example:

```
32 IF NOT (F<0 OR F>100) THEN GOTO 40
34 PRINT "UNREASONABLE TEMPERATURE - TRY
AGAIN"
36 GOTO 20
```

Flowchart Symbols

At this point we have already shown you the most common box shapes used in flowcharting. These are summarized in Figure 5-9. You should be aware, however, that these symbols are not yet consistent throughout the industry. Figure 5-9 thus represents our usage, rather than an agreed-upon standard.

In Figure 5-9, only the FOR and NEXT symbols should be unfamiliar to you at this point. You will encounter them later in this chapter.

Multiple Branching

The ON...GOTO... Statement

As we have seen, the IF...THEN... statement is efficient for testing whether a given condition is True or False. But to take advantage of this statement, we must formulate our problem in terms of only two alternatives (such as, is F= 120, or is it not?)

The ON...GOTO... statement gives us more leeway, since it permits the program to respond to multiple choices. The branching here is not determined by a True/False test. Instead, the program identifies a given alternative among a sequentially arranged series of choices, and matches up that specific choice with the set of instructions pertinent to it.

To illustrate, we use a very simple music quiz program, giving the user a choice between three types of questions, which we code as follows:

Easy	= 1
Intermediate	= 2
Advanced	= 3

We'll use the following variables:
N = Question type selected (1 out of possible 3)
A$= User's answer to quiz question
After it asks the user to indicate his choice of question by typing either 1, 2, or 3, we want the program to branch automatically to the appropriate line containing the question type the user selected.

If we use the IF...THEN... test to accomplish this, the beginning of our program will look as follows:

```
10 PRINT "TYPE 1 FOR EASY, 2 FOR INTERMEDIATE,"
11 PRINT "3 FOR ADVANCED"
12 INPUT N
15 IF N = 1 THEN 30
20 IF N = 2 THEN 120
25 IF N = 3 THEN 200
30 PRINT "EASY QUESTION"
.
.

120 PRINT "INTERMEDIATE QUESTION"
.
.

200 PRINT "ADVANCED QUESTION"
```

As you see, we needed three test lines to determine the value of N and to send the program to line 30, 120, or 200, depending on whether N = 1, 2, or 3.

The ON...GOTO... statement, on the other hand, eliminates the necessity of separate lines for each of our alternatives, a process which would become cumbersome if we had a larger number of alternatives than the three we presently have.

Instead, we can condense everything into one single line, as follows:

```
20 ON N GOTO 30, 120, 200
```

When the program reaches line 20, it has the value of N which you, the user, have input in response to line 10.

Line 20 instructs the program as follows:

- If N = 1 [THEN] GOTO the first of the line numbers listed here.
- If N = 2 [THEN] GOTO the second of the line numbers listed here.
- If N = 3 [THEN] GOTO the third of the line numbers listed here.

To put it another way, the statement says

GOTO the Nth line of those lines listed.

If you type 1, line 20 will send the program to line 30, where the sequence for the easy quiz begins.

If you type 2, the program will bypass everything from line 20 up to line 120, where the intermediate sequence begins.

Finally, if you type 3, the program will branch directly to the advanced quiz sequence beginning on line 200 and will ignore all the intervening lines.

What if you mistakenly input a number other than 1, 2, or 3? Suppose, for example, that you typed in 5 as your question type choice.

The program will then look for a fifth branch on the line containing the ON...GOTO... statement, and, not finding it (since there are only three branches), it will print out an error message, as follows:

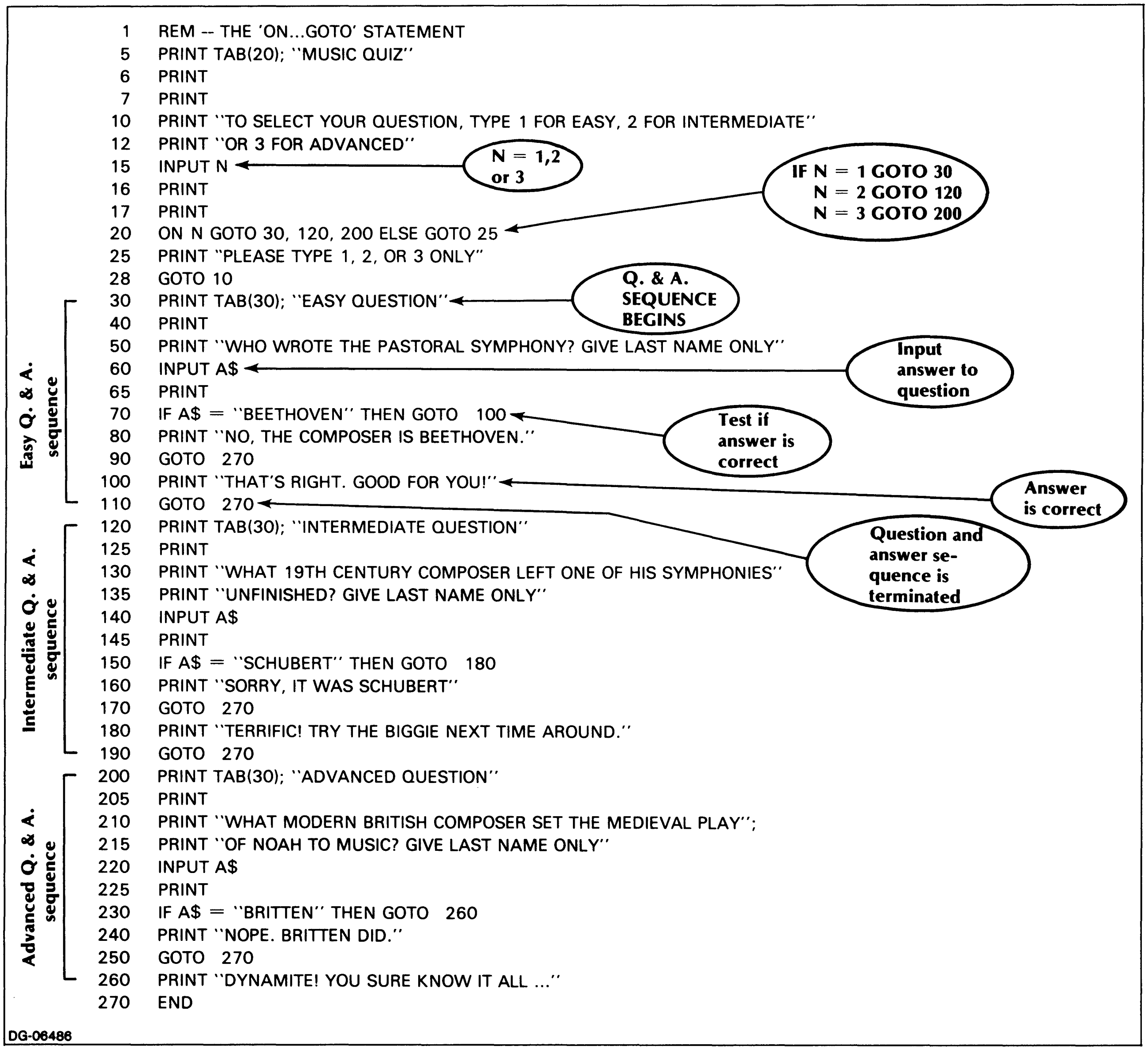
Invalid number of ON...GOTO line numbers in line 20.
Stop at line 20

To avoid receiving such an error, add an ELSE clause to your ON... GOTO...statement. An example is

```
20 ON N GOTO 30, 120, 200 ELSE GOTO 25
25 PRINT "PLEASE TYPE 1 2 OR 3 ONLY"
28 GOTO 10
```

The ELSE clause in statement 20 is executed if a number other than 1, 2, or 3 is entered. In line 25, we clarify the choices, and in line 28, we return to the original question.

It follows that, like the READ DATA statement, the ON...GOTO... is a relatively uncomplicated matching process, which, however, must be set up with absolute precision in order to function.



```
  1  REM -- THE 'ON...GOTO' STATEMENT
  5  PRINT TAB(20); "MUSIC QUIZ"
  6  PRINT
  7  PRINT
 10  PRINT "TO SELECT YOUR QUESTION, TYPE 1 FOR EASY, 2 FOR INTERMEDIATE"
 12  PRINT "OR 3 FOR ADVANCED"
 15  INPUT N
 16  PRINT
 17  PRINT
 20  ON N GOTO 30, 120, 200 ELSE GOTO 25
 25  PRINT "PLEASE TYPE 1, 2, OR 3 ONLY"
 28  GOTO 10
 30  PRINT TAB(30); "EASY QUESTION"
 40  PRINT
 50  PRINT "WHO WROTE THE PASTORAL SYMPHONY? GIVE LAST NAME ONLY"
 60  INPUT A$
 65  PRINT
 70  IF A$ = "BEETHOVEN" THEN GOTO  100
 80  PRINT "NO, THE COMPOSER IS BEETHOVEN."
 90  GOTO  270
100  PRINT "THAT'S RIGHT. GOOD FOR YOU!"
110  GOTO  270
120  PRINT TAB(30); "INTERMEDIATE QUESTION"
125  PRINT
130  PRINT "WHAT 19TH CENTURY COMPOSER LEFT ONE OF HIS SYMPHONIES"
135  PRINT "UNFINISHED? GIVE LAST NAME ONLY"
140  INPUT A$
145  PRINT
150  IF A$ = "SCHUBERT" THEN GOTO  180
160  PRINT "SORRY, IT WAS SCHUBERT"
170  GOTO  270
180  PRINT "TERRIFIC! TRY THE BIGGIE NEXT TIME AROUND."
190  GOTO  270
200  PRINT TAB(30); "ADVANCED QUESTION"
205  PRINT
210  PRINT "WHAT MODERN BRITISH COMPOSER SET THE MEDIEVAL PLAY";
215  PRINT "OF NOAH TO MUSIC? GIVE LAST NAME ONLY"
220  INPUT A$
225  PRINT
230  IF A$ = "BRITTEN" THEN GOTO  260
240  PRINT "NOPE. BRITTEN DID."
250  GOTO  270
260  PRINT "DYNAMITE! YOU SURE KNOW IT ALL ..."
270  END
```

Figure 5-10. Music Quiz Program

If you are very careful to observe the following rules, you will avoid any problems in working with the ON...GOTO...statements.

- Enumerate as many possible branches as the alternative choices available.
- List the branch lines in precisely the same order as you want them identified by the program; i.e., list the first branch first, the second branch second, and so on.
- Add an ELSE clause to handle unanticipated alternatives.

Here, now, is our program (Figure 5-10), followed immediately by its flowchart (Figure 5-11) for easier interpretation.

Follow the arrows in the flowchart and identify each of our three main branches, i.e., lines 30, 120, 200.

Next, note that each of these locations contains a minisequence of its own which follows these identical steps:

- Identify the question type chosen.
- Print the question.
- User inputs answer.

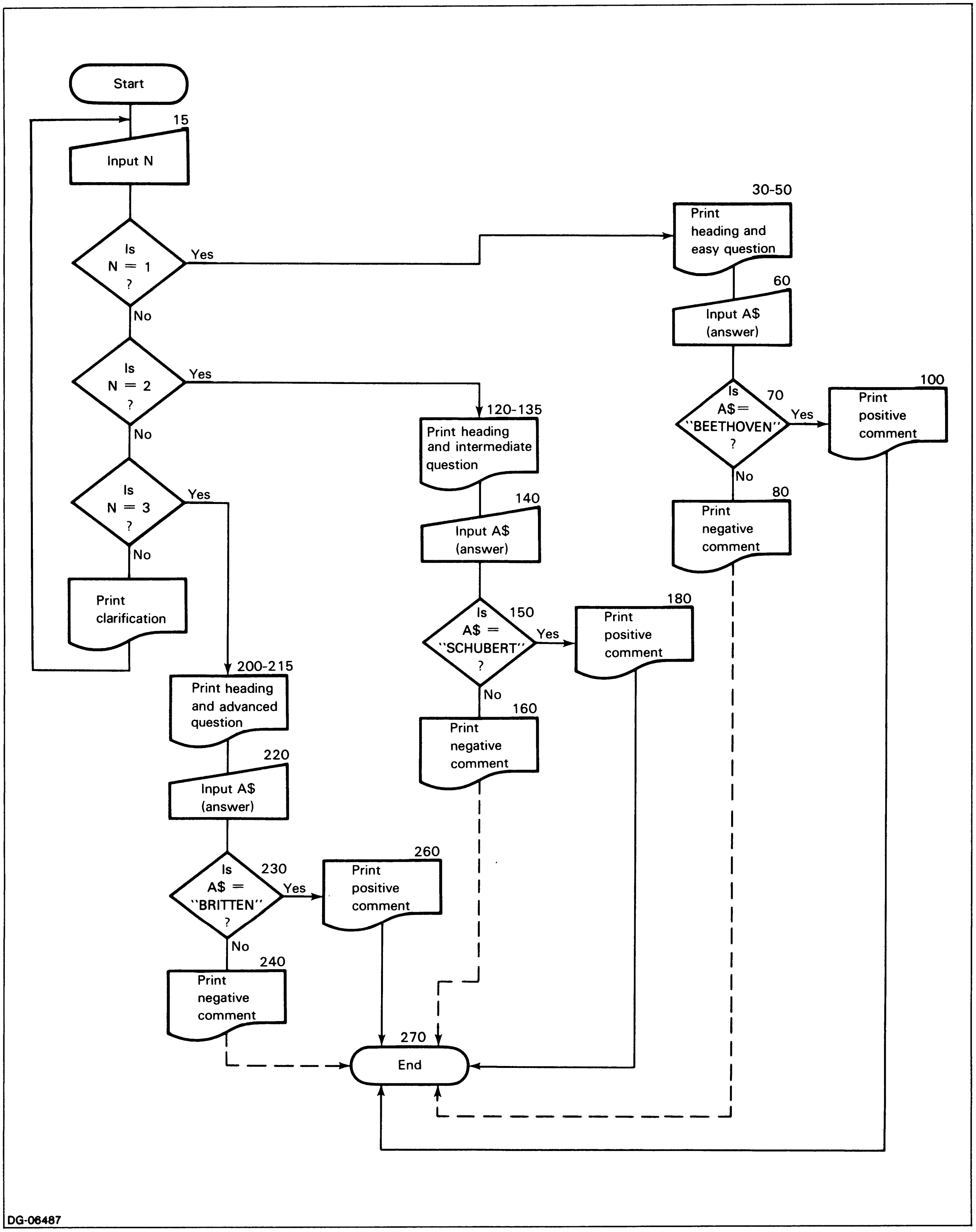


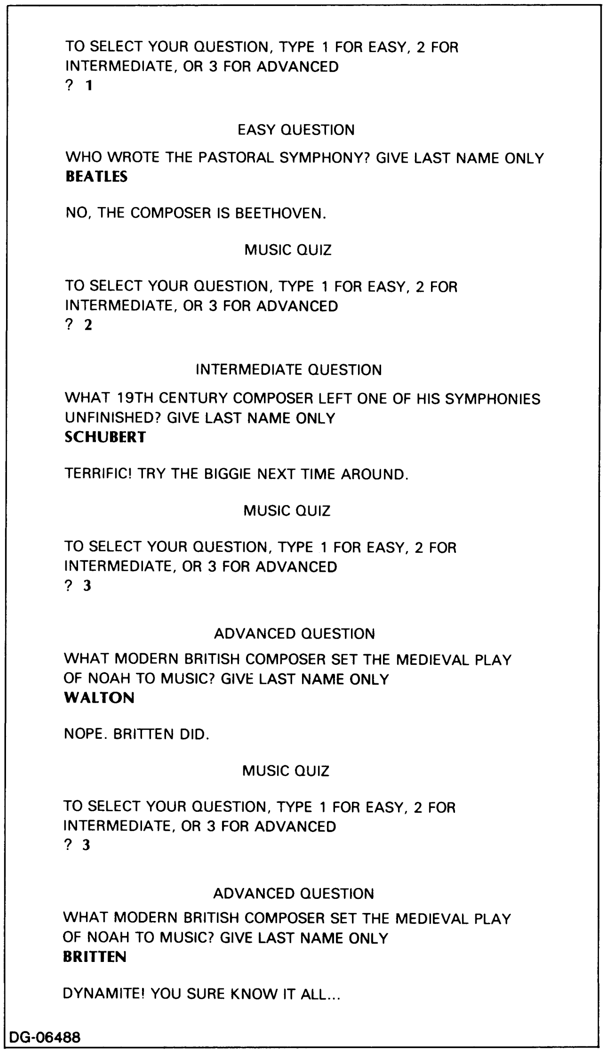
Figure 5-11. Flowchart of Music Quiz Program

- Test whether answer = correct answer. (Note that we used a string comparison for this purpose.)
- Also note: since the answer can be only right or wrong, we use a simple IF...THEN... to determine the next appropriate sequence.
- If the answer is correct, branch to a line which prints congratulatory message and then branch to end of program.
- If the answer is incorrect: print a negative comment and branch to end of program.

When you run this program, your output will be as shown in Figure 5-12. (Actually, we show you several consecutive runs here, so that you can go through all of the question types, as well as through some right and wrong answers. As the program is written, a single run will, of course, take you through only one question/answer/comment sequence.) Remember that string comparisons are sensitive to the difference between upper and lowercase, so if you type in your answers as lowercase, you're sure to get the wrong answer!

This program still contains some redundancies (you will see how to get rid of them in Chapter 6); it does, however, demonstrate the increased range of possibilities available through use of the ON...GOTO... statement either by itself or, as here, in combination with other control statements.

We also wish to point out that this statement does not always require a user input, as it did here. In many instances, the program itself will be able to generate the data it needs to use in an ON...GOTO... statement.



```
TO SELECT YOUR QUESTION, TYPE 1 FOR EASY, 2 FOR
INTERMEDIATE, OR 3 FOR ADVANCED
? 1

                EASY QUESTION

WHO WROTE THE PASTORAL SYMPHONY? GIVE LAST NAME ONLY
BEATLES

NO, THE COMPOSER IS BEETHOVEN.

                 MUSIC QUIZ

TO SELECT YOUR QUESTION, TYPE 1 FOR EASY, 2 FOR
INTERMEDIATE, OR 3 FOR ADVANCED
? 2

            INTERMEDIATE QUESTION

WHAT 19TH CENTURY COMPOSER LEFT ONE OF HIS SYMPHONIES
UNFINISHED? GIVE LAST NAME ONLY
SCHUBERT

TERRIFIC! TRY THE BIGGIE NEXT TIME AROUND.

                 MUSIC QUIZ

TO SELECT YOUR QUESTION, TYPE 1 FOR EASY, 2 FOR
INTERMEDIATE, OR 3 FOR ADVANCED
? 3

              ADVANCED QUESTION
WHAT MODERN BRITISH COMPOSER SET THE MEDIEVAL PLAY
OF NOAH TO MUSIC? GIVE LAST NAME ONLY
WALTON

NOPE. BRITTEN DID.

                 MUSIC QUIZ

TO SELECT YOUR QUESTION, TYPE 1 FOR EASY, 2 FOR
INTERMEDIATE, OR 3 FOR ADVANCED
? 3

              ADVANCED QUESTION
WHAT MODERN BRITISH COMPOSER SET THE MEDIEVAL PLAY
OF NOAH TO MUSIC? GIVE LAST NAME ONLY
BRITTEN

DYNAMITE! YOU SURE KNOW IT ALL...
```

DG-06488

Figure 5-12. Sample Dialog with Music Quiz Program of Figure 5-10

Repeated Operations: Loops

As we have already indicated, there are occasions when you need to repeat the same operation a specified number of times. Here we automate the process by applying the concept of the True/False test to check whether or not the desired number of repeats has been performed.

In an earlier program (Figure 5-6), we input the temperatures in Fahrenheit and used the IF...THEN... statement to test whether the value we input was a termination signal for the program (F = 120).

Now we will show you yet another way of formulating your program so as to control its branching. (Figure 5-13 incorporates this technique into our original program.)

Looping with Counters

Let's assume that, instead of testing each of your input values until you get the value signalling that you have run out of data, you want to terminate the program after you have processed a specific number of values.

In our case, we know that we have data for 10 days; thus, we want to perform our input, conversion, and print operations a total of 10 times.

If we could get the program to keep count of the number of times it has performed the above operations, then we could use the IF...THEN statement to test whether or not our count has reached the desired limit of 10 (or 20, or whatever).

How do we build a *counter* into our program?

This is a very simple process: we begin by taking a variable letter we have not yet used in our program, for example, I, and designating it as the variable to hold our counter values.

First we will initialize the value of I to 0 to ensure that it will hold no other values than those generated during the execution of our program. (See Chapter 2.)

```
30 LET I = 0
```

As you recall, the other variables in our program are

F = Temperature in Fahrenheit

C = Temperature in Centigrade

We will make one further minor but convenient change in the program: rather than using INPUT to enter our temperature values for each of the 10 days, we will enter all our values at once into a DATA line to be read by the program. This will eliminate the need for interacting with the program during execution.

Here are our 10 daily temperatures in their proper sequence:

```
160 DATA 47, 41, 49, 56, 63, 42, 40, 59, 67, 61
```

When the program reads the first value of F (i.e., 47), our counter should reflect this fact by also changing its value (i.e., from 0 to 1). Increasing the value of the counter is called incrementing the counter. Line 100 of the program shown in Figure 5-13 instructs the program to increment our counter I:

```
100 LET I = I + 1
```

This causes the value of 1 to be added to the previous value of I. Since the previous value of I was 0, the value of I after we have reached line 100 for the first time is 1.

Each time we read an additional data item, we move through line 100, where the value of I is incremented by one. This means that, at any point in the program, the current value of I reflects the number of data items we have already read.

Thus, when

F = 47 : I = 1
F = 41 : I = 2
F = 49 : I = 3
F = 56 : I = 4
F = 63 : I = 5
F = 42 : I = 6
F = 40 : I = 7
F = 59 : I = 8
F = 67 : I = 9
F = 61 : I = 10

Now that we have a counter to keep track of the number of items we have processed, all we need to do is to build in an IF...THEN... which asks:

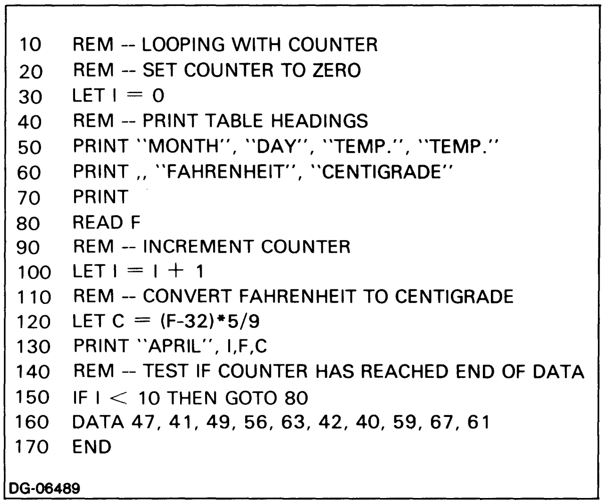
Is I $<$ 10?

If the value of counter I is less than 10, more data remain to be read; the program returns to line 80 to read the next value of F. If the counter value is not less than 10, all data have been read and the program terminates.

The IF...THEN... statement might read as follows:

150 IF I < 10 THEN GOTO 80

Figure 5-13 displays our modified program with explanatory REM lines added.

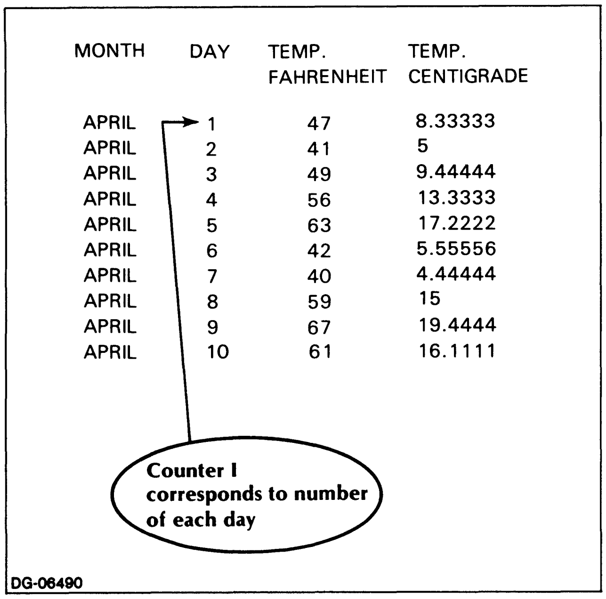


```
10   REM -- LOOPING WITH COUNTER
20   REM -- SET COUNTER TO ZERO
30   LET I = 0
40   REM -- PRINT TABLE HEADINGS
50   PRINT "MONTH", "DAY", "TEMP.", "TEMP."
60   PRINT ,, "FAHRENHEIT", "CENTIGRADE"
70   PRINT
80   READ F
90   REM -- INCREMENT COUNTER
100  LET I = I + 1
110  REM -- CONVERT FAHRENHEIT TO CENTIGRADE
120  LET C = (F-32)*5/9
130  PRINT "APRIL", I,F,C
140  REM -- TEST IF COUNTER HAS REACHED END OF DATA
150  IF I < 10 THEN GOTO 80
160  DATA 47, 41, 49, 56, 63, 42, 40, 59, 67, 61
170  END
```

Figure 5-13. Temperature Conversion Program Using a Counter

So as to illustrate the operation of the counter more vividly, we directed the program in line 130 to print out the counter's value (the value of I), along with the values of F and C.

As you see from the run (Figure 5-14) this results in a neat tabulation, whereby the days of the month (we picked April), are automatically numbered from 1 to 10 before each day's temperature is printed out.



```
MONTH    DAY   TEMP.         TEMP.
               FAHRENHEIT    CENTIGRADE

APRIL    1      47           8.33333
APRIL    2      41           5
APRIL    3      49           9.44444
APRIL    4      56           13.3333
APRIL    5      63           17.2222
APRIL    6      42           5.55556
APRIL    7      40           4.44444
APRIL    8      59           15
APRIL    9      67           19.4444
APRIL    10     61           16.1111
```

Counter I corresponds to number of each day
DG-06490

Figure 5-14. Sample Run of Temperature COnversion Program (Counter Version)

As the run shows, the program correctly performed a total of 10 times the operations READ F, increment counter, convert F to C, and PRINT.

Such a repeated set of identical operations is a loop, in our case, a loop performed by means of a counter. (In Figure 5-2 we executed a loop by means of a GOTO... statement, while in Figure 5-6 we looped with an IF...THEN... statement.)

The flowchart shown in Figure 5-15 illustrates looping with a counter.

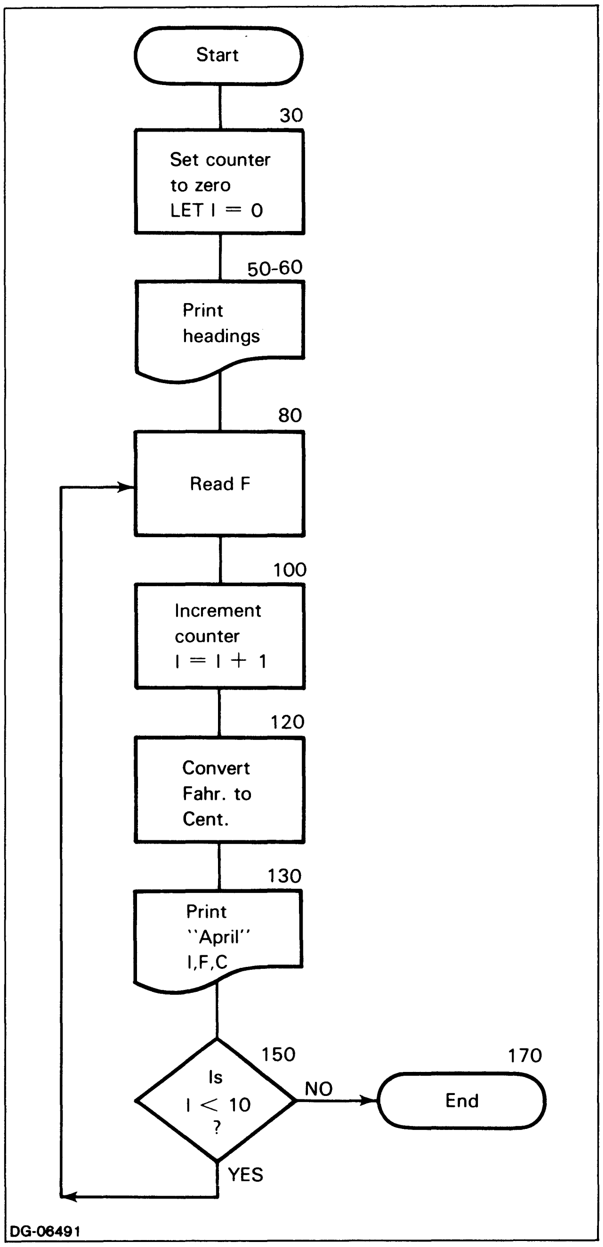
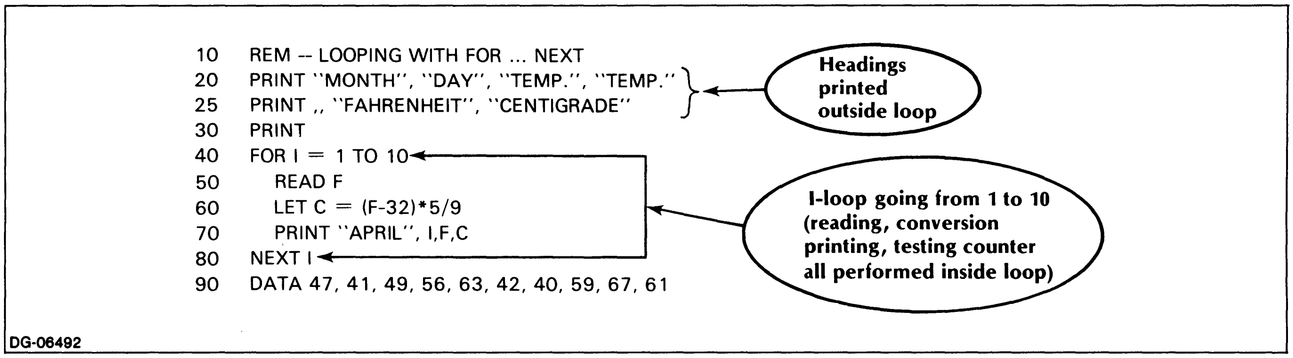


Figure 5-15. Flowchart for Counter Version of Temperature Conversion Program

Looping with FOR...NEXT

To execute our loop with counter, we needed three special lines:

- Line 30 to set the counter at 0.
- Line 100 to increment the counter after value was read.



```
10   REM -- LOOPING WITH FOR ... NEXT
20   PRINT "MONTH", "DAY", "TEMP.", "TEMP."
25   PRINT ,, "FAHRENHEIT", "CENTIGRADE"
30   PRINT
40   FOR I = 1 TO 10
50     READ F
60     LET C = (F-32)*5/9
70     PRINT "APRIL", I,F,C
80   NEXT I
90   DATA 47, 41, 49, 56, 63, 42, 40, 59, 67, 61
```

Headings printed outside loop
I-loop going from 1 to 10 (reading, conversion printing, testing counter all performed inside loop)
DG-06492

Figure 5-16. Temperature Conversion Program with FOR...NEXT loop

- Line 150 to test the value of the counter and to redirect the program to line 80 if the counter value was less than 10.

All of this works perfectly, and there are many cases where it is useful to have a separate counter built into the program.

But under normal conditions we can execute a loop by condensing all the above lines into two:

- The first to specify the counter's beginning and ending values;
- The second to signal the end of the operation(s) to be repeated.

In this version, the counter will increment itself automatically; similarly, the IF...THEN... test checking whether the maximum value has been reached is automatic without being explicitly written in a separate instruction line.

The program we are working with requires the counter, I, to increment itself from 1 to 10. To have this happen automatically, we formulate our instruction as follows:

```
40 FOR I = 1 TO 10
          .
          .
          .
80 NEXT I
```

Translated, the FOR statement means using I as your counter, and beginning your count with 1, perform all the instructions you will find listed below, until you find the instruction

```
80 NEXT I
```

Then increment I and check to see if I > 10.

If I is not greater than 10, go back to the first line following line 40 and repeat all the operations as previously.

Each time you reach line 80, increment I and test whether I > 10. When you reach the point where I > 10, proceed to the line immediately below the NEXT and follow the new instructions listed therein.

When I = 10, the loop will still be executed, since the value of I has not yet exceeded the specified limit of 10. This will only happen at the following NEXT statement when I is incremented to 11. The value of I upon termination of the loop will thus be 11 rather than 10.

Everything between the FOR line and the NEXT line is called the loop body. In our example, the body of the loop is contained between lines 40 and 80:

```
40 FOR I = 1 TO 10
          .
          .
          .
80 NEXT I
```

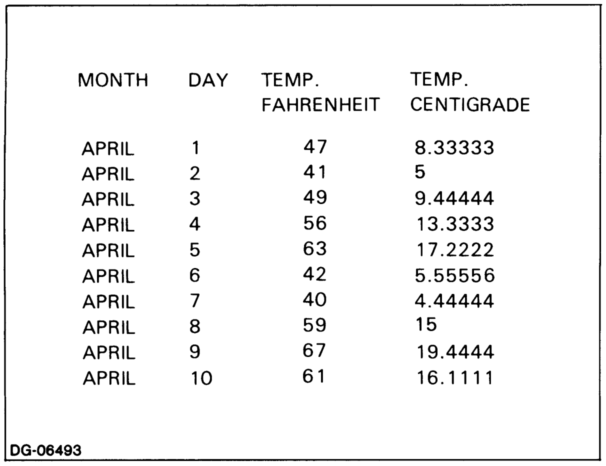
The FOR and NEXT statements are intended to appear together; you shouldn't have one without the other. (Without the FOR you wouldn't have a loop; without the NEXT you would not be returned to the beginning of the loop to repeat the required operations.)

The program in Figure 5-16 illustrates how the loop with automatic counter works.

Notice that the instructions in the body of the loop (lines 50, 60, and 70) in Figure 5-16 are indented, for clarity. You might want to get into the habit of indenting loop bodies as you type them in. In any case, MP/BASIC will do this formatting for you, so the proper indentation will appear whenever you get a copy of your program typed out by the LIST command.

Lines 40 and 80 set up the loop counting from 1 to 10, by means of the FOR...NEXT statements we have discussed.

The rest of the loop sequence is exactly as before. You can verify this by comparing Figure 5-17 with Figure 5-14. The flowchart in Figure 5-18 further underscores the similarity in process.



```
MONTH      DAY    TEMP.          TEMP.
                  FAHRENHEIT     CENTIGRADE

APRIL      1        47           8.33333
APRIL      2        41           5
APRIL      3        49           9.44444
APRIL      4        56           13.3333
APRIL      5        63           17.2222
APRIL      6        42           5.55556
APRIL      7        40           4.44444
APRIL      8        59           15
APRIL      9        67           19.4444
APRIL      10       61           16.1111
```

DG-06493

Figure 5-17. Sample Run of Temperature Conversion Program (FOR...NEXT Version)

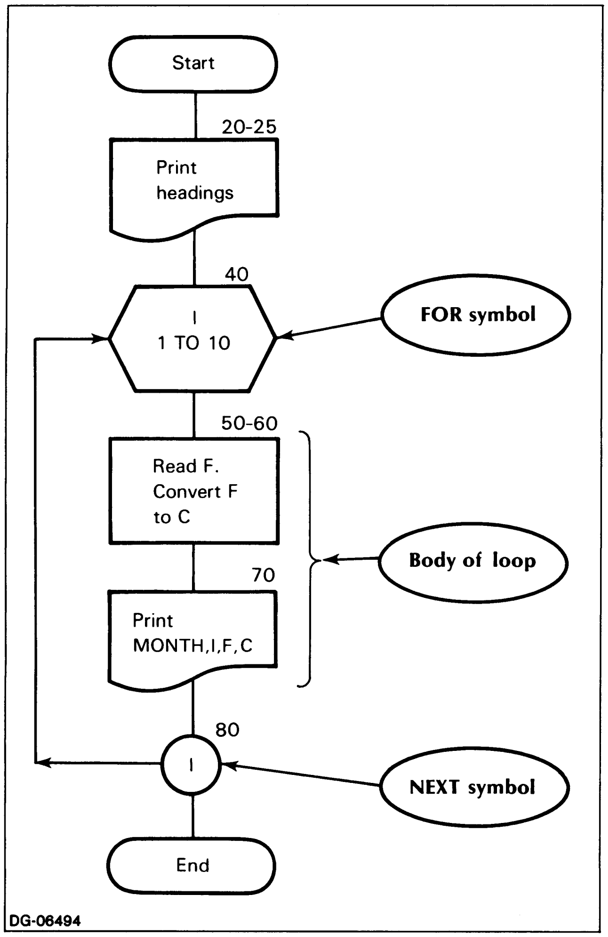


Figure 5-18. Flowchart for Temperature Conversion Program (FOR...NEXT Version)

The program

- Reads the value of F from the data line;
- Converts that value to its Centigrade equivalent;
- Prints the month (April), day (value of I), daily temperature in Fahrenheit and Centigrade;
- Increments the counter
- Tests whether the counter exceeds 10 (I > 10). If the answer is Yes, it terminates the loop; if the answer is No, it returns to re-execute the body of the loop.

This sequence will be repeated 10 times; yet, as you see, this program is far shorter than its predecessor in which we used a counter and the IF...THEN statement.

If you look at the program or the flowchart, you will notice that we placed the instructions for our headings (lines 20-25) outside our loop. If we had included these instructions inside the body of the loop, the headings would have been printed 10 times, once in each succeeding pass of the program.

Determining what you must exclude from the loop body is, therefore, just as important as knowing what you want to include in it.

Loop execution is considerably speeded up if you use the MP/BASIC facility of declaring your initial value, counter variable, increment and limit values to be of type INTEGER. (See Chapters 2 and 14.)

Flowchart FOR...NEXT Symbols

As you looked at our last flowchart, you probably noticed that we used two new symbols, the FOR symbol, and the NEXT symbol,which we illustrate for you in Figure 5-19. (See also Figure 5-9.)

These symbols are useful merely as a form of shorthand, since the logic of the program is conveyed quite as accurately by the longer flowchart of Figure 5-15, which does not use them.

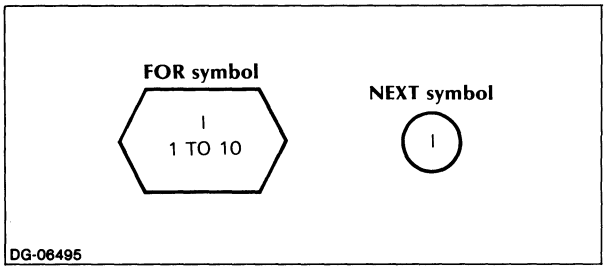


Figure 5-19. FOR and NEXT Symbols

Looping with STEP for Increments Different from 1

An added feature of looping with FOR...NEXT... is that we need not confine our count to one-step increments; rather, we can set up a counter to increment itself by any intervals we choose, such as by increments of 3, 10, or whatever.

Suppose we wanted only the temperature for every second day in April, starting with April 1. We would then write something like

```
40 FOR I = 1 to 9 STEP 2
```

The STEP... statement tells the program by how many units to increment the counter after each operation. In the absence of such a statement, the counter automatically increments itself by 1.

With a loop as set up by line 40 above,

For day #1 I = 1

For day #3 I = 3

For day #5 I = 5

For day #7 I = 7

For day #9 I = 9

When you are stepping forward, the smaller number you begin with comes first, and the larger number comes next. (We move from 1 to 9).

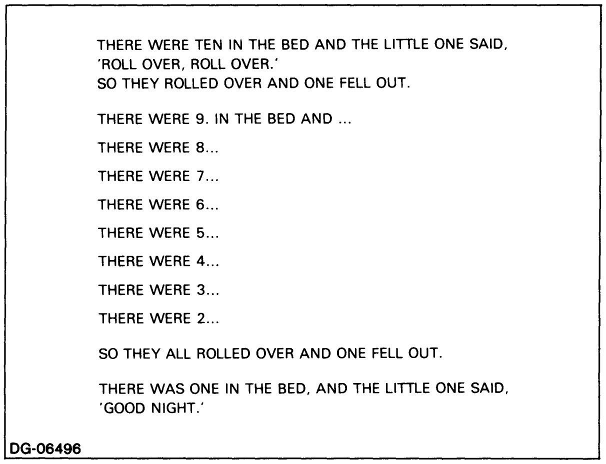
But you need not always count forwards. Use a STEP command with a negative increment, and the computer will count backward. For instance,

```
40 FOR I = 9 TO 1 STEP -2
```

will take you from 9 down to 1 by twos.

NOTE: In STEP with a negative increment, the larger number you are counting down from must be listed first, followed next by the smaller number you are moving towards. That is exactly the opposite of the way you list the numbers when counting forward.

A popular children's song cleverly acts out backward counting. Figure 5-20 reproduces the first and last stanzas together with the intervening countdown.



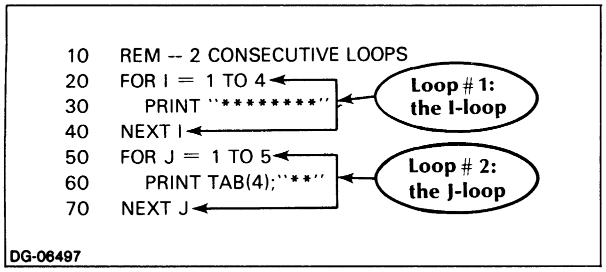
```
THERE WERE TEN IN THE BED AND THE LITTLE ONE SAID,
'ROLL OVER, ROLL OVER.'
SO THEY ROLLED OVER AND ONE FELL OUT.

THERE WERE 9. IN THE BED AND ...

THERE WERE 8...

THERE WERE 7...

THERE WERE 6...

THERE WERE 5...

THERE WERE 4...

THERE WERE 3...

THERE WERE 2...

SO THEY ALL ROLLED OVER AND ONE FELL OUT.

THERE WAS ONE IN THE BED, AND THE LITTLE ONE SAID,
'GOOD NIGHT.'
```

Figure 5-20. Backward Counting

You might enjoy reconstructing the program which produced this output. Ours appears at the end of this chapter.

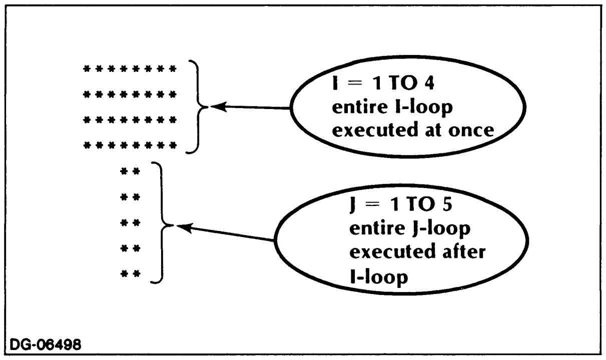
Nested Loops

Figure 5-21 represents a program with two loops which will print out the design shown in Figure 5-22.



```
10   REM -- 2 CONSECUTIVE LOOPS
20   FOR I = 1 TO 4
30     PRINT "********"
40   NEXT I
50   FOR J = 1 TO 5
60     PRINT TAB(4);"**"
70   NEXT J
```

Figure 5-21. Program with Two Consecutive Loops



```
********
********
********
********
    **
    **
    **
    **
    **
```

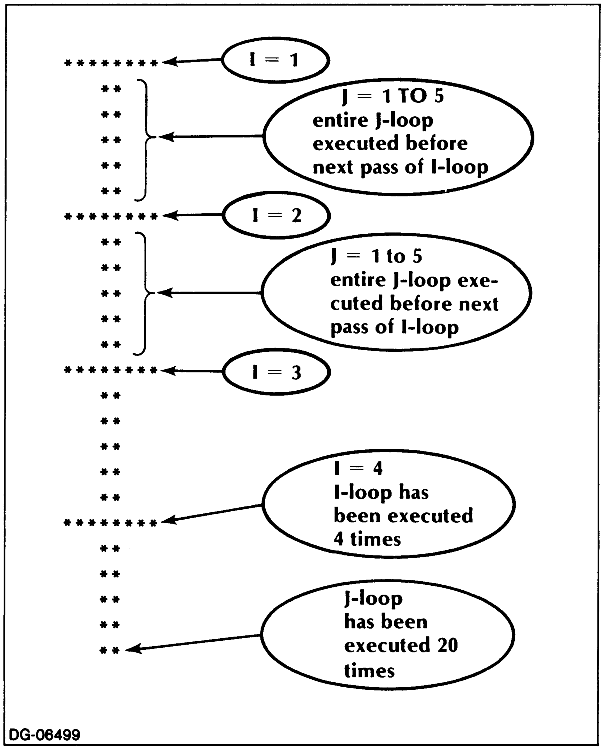
Figure 5-22. Output from Program in Figure 5-21

This design consists of the following:

- Eight asterisks forming a horizontal line, which the I-loop is instructed to print four consecutive times;
- Two asterisks which the J-loop is instructed to print five consecutive times, thus forming a vertical line. (Strictly for looks, we use a tab to center the two asterisks beneath the horizontal line.)

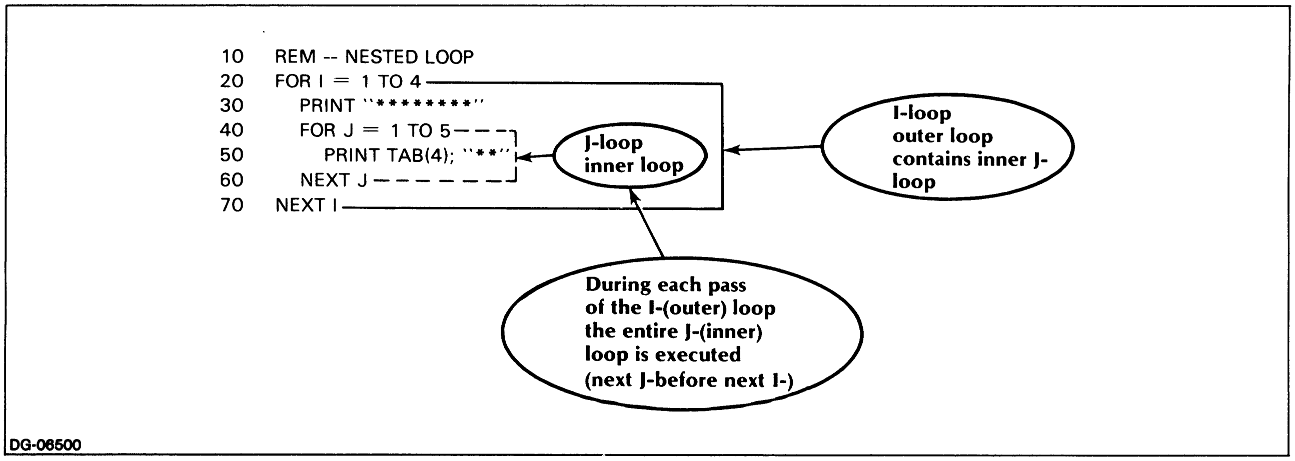
Since the J-loop is executed only after the entire I- loop is finished, these two loops are described as being consecutive.

Suppose now that instead of four horizontal lines and a single vertical, we want our design to alternate horizontals and verticals four times each, as in Figure 5-23.



```
********
    **
    **
    **
    **
    **
********
    **
    **
    **
    **
    **
********
    **
    **
    **
    **
    **
********
    **
    **
    **
    **
    **
```

Figure 5-23. Variation in Output from Loop Program



```
10   REM -- NESTED LOOP
20   FOR I = 1 TO 4
30     PRINT "********"
40     FOR J = 1 TO 5
50       PRINT TAB(4); "**"
60     NEXT J
70   NEXT I
```

Figure 5-24. Program with Nested Loops

To produce such a design, we need the same two loops as before, but we must change their relationship to each other. Instead of allowing the I-loop to be completely executed, we must intercept it during each of its single passes and cause it to detour through the complete J-loop,printing a vertical line before resuming its own re-execution. In this way, we can alternate the printing of horizontal and vertical lines.

We can accomplish this very easily, by moving the J-loop from its former independent position after the end of the I-loop, placing it instead within the body of the I-loop.

This type of arrangement, whereby a loop incorporates one or more subsidiary loops within itself, is referred to as nesting.

In Figure 5-24 we modify our program of Figure 5-21 so as to nest the J-loop inside the I-loop.

We will now take a closer look at the way nested loops work.

As you see in Figure 5-24, the I-loop is now the main or outer loop, since its body, which goes from line 20 to line 70, contains within it a second loop, the J-loop.

The J-loop is now the inner loop, since its body (lines 40 to 60) is entirely contained within the body of the I-loop.

Again, note the indentations used to set off the loop bodies; MP/BASIC will do this indenting for you as part of its formatting.

Let us work through the instructions of this program as though we were duplicating the computer's steps.

As we begin the I-loop, we are instructed to

```
30 PRINT "*********"
```

As soon as we have carried out that instruction, we encounter our inner loop, the J-loop, which instructs us to

```
50 PRINT TAB(4); "***"
```

and which immediately thereafter directs us to

```
60 NEXT J
```

causing us to reprint "**" until such time as we have printed "**" five times (until J = 6).

Only at this point, when the entire J-loop has been executed, do we reach the

```
70 NEXT I
```

command, which returns us to the next pass of our I-loop, allowing us to print another horizontal line as specified by line 30.

Thereupon we re-encounter the J-loop, and re-execute it five more times before returning to the next pass of our I-loop, and so on.

Thus, when I = 1

J = 1

J = 2

J = 3

J = 4

J = 5

When I = 2

J = 1

J = 2

J = 3

J = 4

J = 5

When I = 3

J = 1

J = 2

J = 3

J = 4

J = 5

When I = 4

J = 1

J = 2

J = 3

J = 4

J = 5

The I-loop goes through four passes (I=1 TO 4). The J-loop goes through 20 passes (J=1 TO 5 for each of the four passes of the outer I-loop).

We will use a flowchart one final time in this chapter (Figure 5-25) as an additional illustration of nested loops.

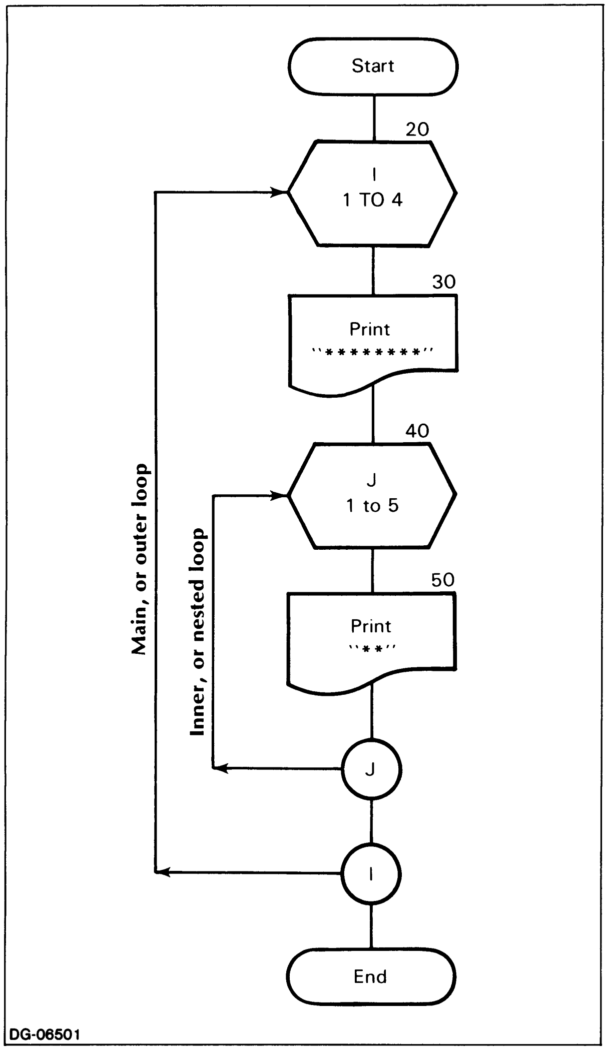


Figure 5-25. Flowchart of Program with Nested Loops

Keeping in mind that the inner loops are always executed before the outer loops, you must be sure to have your NEXT... commands in proper sequence, inner before outer or, as here,

60 NEXT J

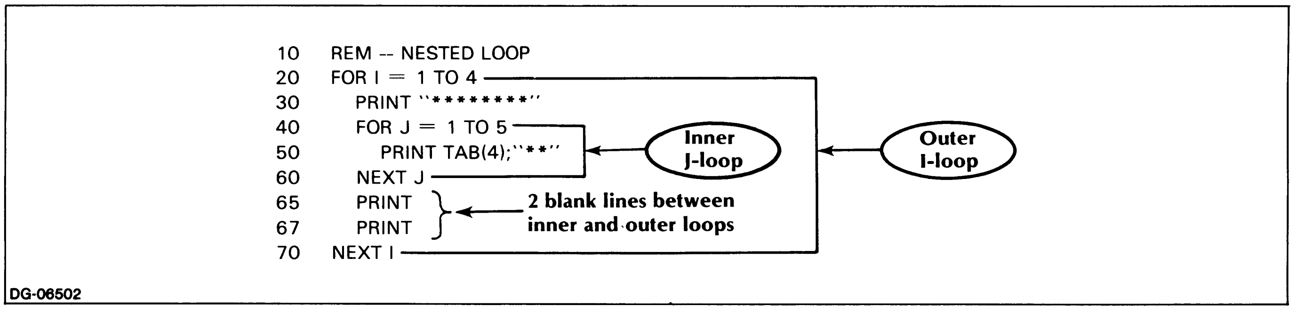
before

70 NEXT I

Confusion of this sequence is a frequent cause of difficulty with loop execution; hence, great care and doublechecking are always advisable.

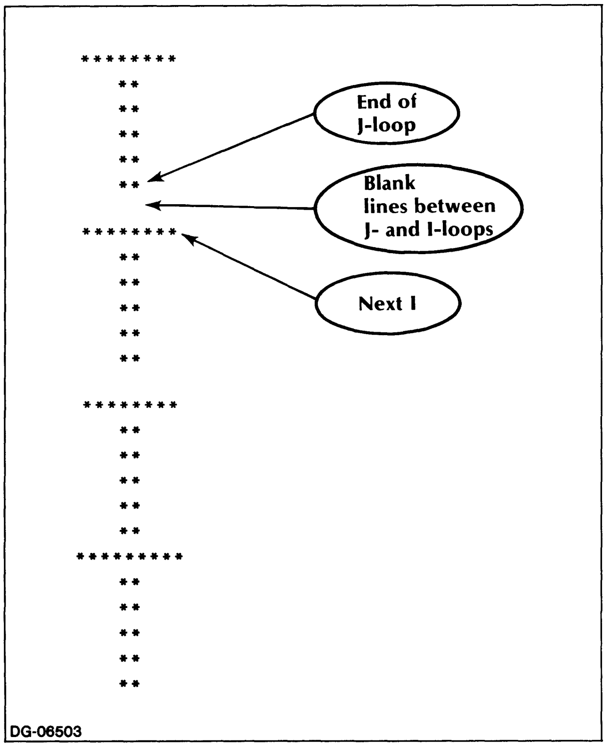
You can vary the shape of your design by means of a few simple modifications in your nested loops.

Would you like a T-shaped figure? All you have to do is to insert two blank lines between your outer and inner loops as in Figure 5-26. This will separate your vertical line from the next horizontal line and produce the Ts shown in Figure 5-27.



```
10   REM -- NESTED LOOP
20   FOR I = 1 TO 4
30     PRINT "*********"
40     FOR J = 1 TO 5
50       PRINT TAB(4);"**"
60     NEXT J
65     PRINT
67     PRINT
70   NEXT I
```

Figure 5-26. Modification of Loop Program to Produce T-shape

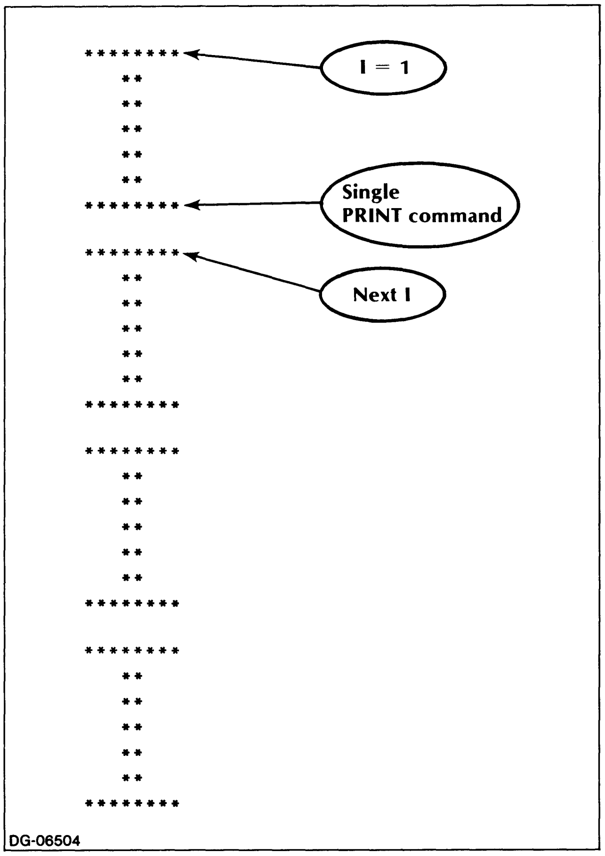


```
********
   **
   **
   **
   **
   **

********
   **
   **
   **
   **
   **

********
   **
   **
   **
   **
   **
*********
   **
   **
   **
   **
   **
```

Figure 5-27. Output of Program in Figure 5-26

Don't, however, insert your blanks within your J-loop, as this will simply create spaces between your successive vertical asterisks. As we've already pointed out, deciding what belongs inside or outside a loop is often tricky business. Working the steps of your program out on paper and flowcharting are good ways of catching any misplaced instructions.

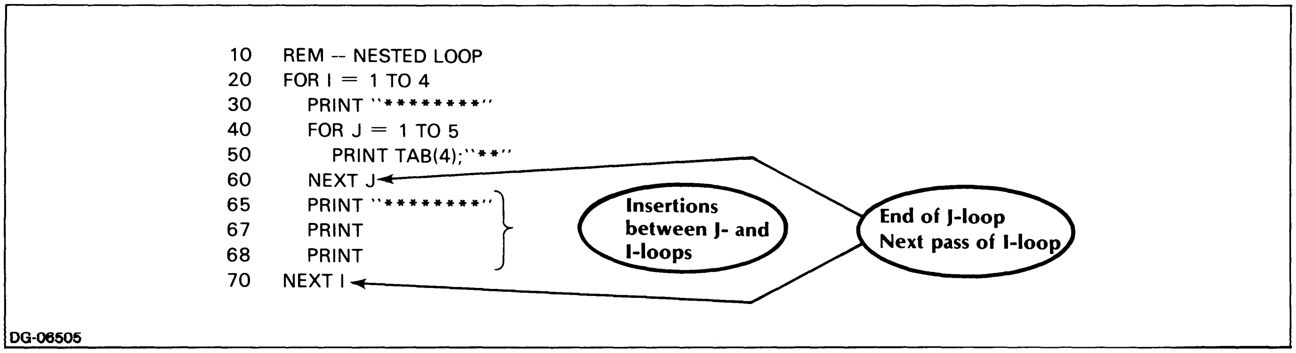
To get an I-shaped figure (see Figure 5-28), we instructed the computer to print one single horizontal line of asterisks, plus two blank lines at the end of each J-loop before proceeding to the next pass of the I-loop. Our program is shown in Figure 5-29.



```
********
   **
   **
   **
   **
   **
********

********
   **
   **
   **
   **
   **
********

********
   **
   **
   **
   **
   **
********

********
   **
   **
   **
   **
   **
********
```

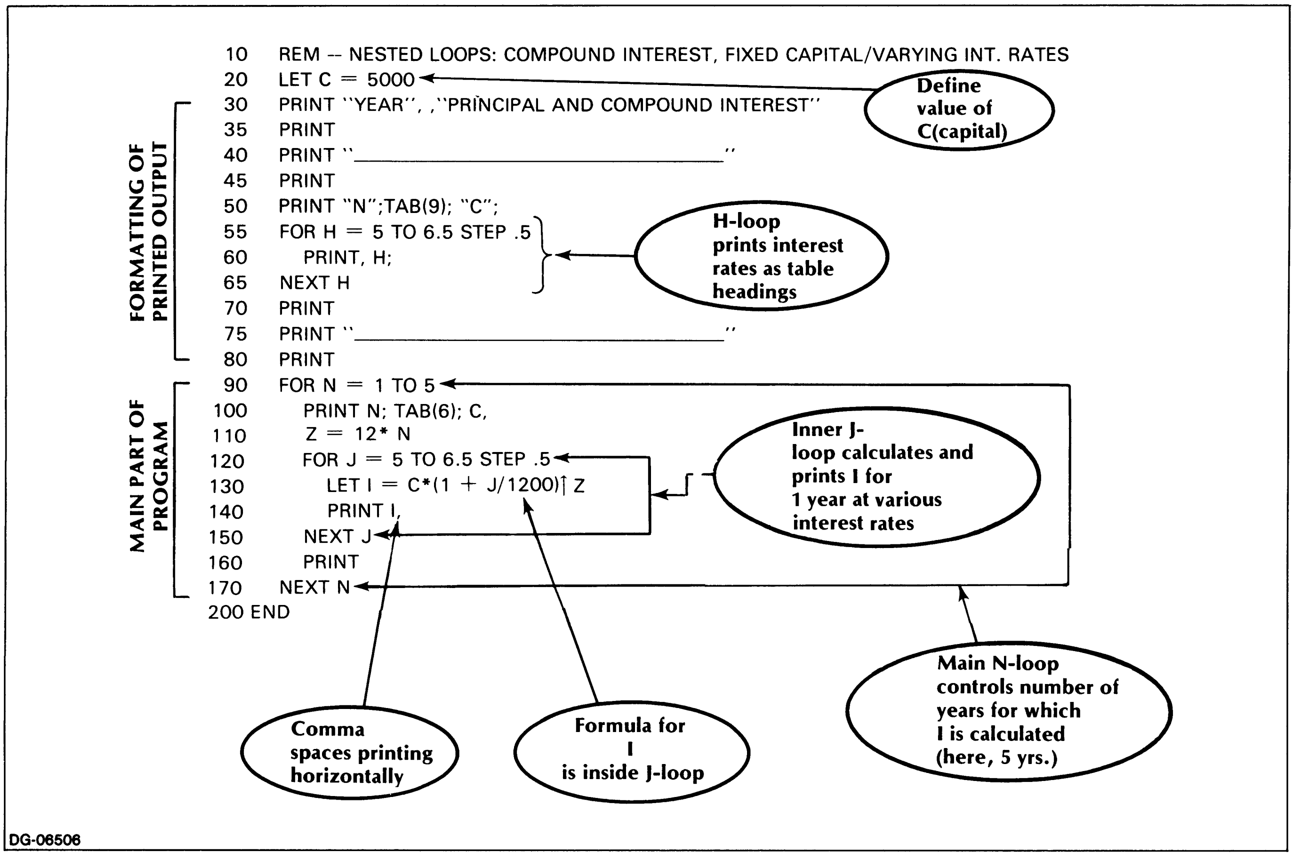
Figure 5-28. Output of Program in Figure 5-29

You can insert any number of commands within as well as between your nested loops; the main consideration is to keep your outer and inner loops straight.



```
10  REM -- NESTED LOOP
20  FOR I = 1 TO 4
30    PRINT "********"
40    FOR J = 1 TO 5
50      PRINT TAB(4);"**"
60    NEXT J
65    PRINT "********"
67    PRINT
68    PRINT
70  NEXT I
```

Figure 5-29. Modification of Loop Program to Produce I-shapes



```
 10  REM -- NESTED LOOPS: COMPOUND INTEREST, FIXED CAPITAL/VARYING INT. RATES
 20  LET C = 5000
 30  PRINT "YEAR", ,"PRINCIPAL AND COMPOUND INTEREST"
 35  PRINT
 40  PRINT "___________________________________"
 45  PRINT
 50  PRINT "N";TAB(9); "C";
 55  FOR H = 5 TO 6.5 STEP .5
 60    PRINT, H;
 65  NEXT H
 70  PRINT
 75  PRINT "___________________________________"
 80  PRINT
 90  FOR N = 1 TO 5
100    PRINT N; TAB(6); C,
110    Z = 12* N
120    FOR J = 5 TO 6.5 STEP .5
130      LET I = C*(1 + J/1200)↑ Z
140      PRINT I,
150    NEXT J
160    PRINT
170  NEXT N
200 END
```

FORMATTING OF PRINTED OUTPUT
MAIN PART OF PROGRAM
Define value of C(capital)
H-loop prints interest rates as table headings
Inner J-loop calculates and prints I for 1 year at various interest rates
Main N-loop controls number of years for which I is calculated (here, 5 yrs.)
Comma spaces printing horizontally
Formula for I is inside J-loop
DG-06506

Figure 5-30. Program to Calculate Interest at Different Rates

A Practical Application

In Chapter 4 we wrote a program to find out the income on a capital of $5,000 invested at different rates of interest.

We can apply the concept of nested loops to enhance the usefulness of that program. Specifically, we can now calculate principal and interest at different interest rates over a period of several years. We will assume interest rates of from 5% to 6.5% at intervals, and a time period of from 1 to 5 years.

The formula for compounding interest monthly is

I = C * (1 + J/1200) ^ (12 *N)

where

I = principal plus interest

C = capital amount (here 5000)

J = interest rate (here 5% to 6.5%)

N = number of years invested for (here 1 to 5)

To increase the efficiency of our program, the above formula can be rewritten as follows:

I = C * (1 + J/1200)^Z

where

Z = 12 * N

This part of the formula computes the total number of months for which the capital is invested. (The reason for this modification will shortly become apparent.) We can now write the program shown in Figure 5-30.

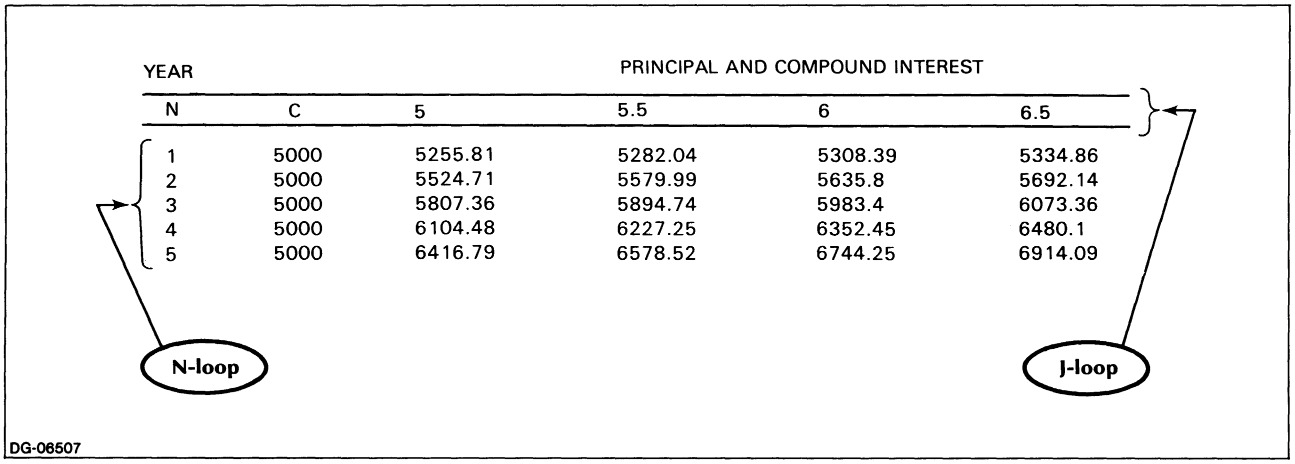
Up to line 90, the program prints and underlines the various headings we need for a clear output. (Note that in lines 55–65 we use a loop to print out the interest rate headings.) The working part of the program actually consists of its last nine lines (90–170), which contain two nested loops.

The N-loop is our main, or outer loop. It moves from 1 to 5, which corresponds to the number of years the capital is invested for. Here we compute the quantity

Z = 12 * N

converting the investment period from years into months.

There are two reasons for performing this calculation within the N-loop rather than within the J-loop, where the rest of the compound interest formula is calculated:



```
YEAR                             PRINCIPAL AND COMPOUND INTEREST
N        C          5              5.5             6              6.5
1        5000       5255.81        5282.04         5308.39        5334.86
2        5000       5524.71        5579.99         5635.8         5692.14
3        5000       5807.36        5894.74         5983.4         6073.36
4        5000       6104.48        6227.25         6352.45        6480.1
5        5000       6416.79        6578.52         6744.25        6914.09
```

Figure 5-31. Output from Program in Figure 5-30

- The product 12 * N is independent of the other variables in the compound interest formula.
- Since we are calculating compound interest for five years (N = 1 to 5), we need to compute Z = 12 * N no more than five times, i.e., once for each changing value of N. Hence, the N-loop is the logical place for this segment of the compound interest formula.

In the course of program execution, then, for each year, starting with N = 1:

- N is printed out, indicating what year we are working with.
- C is printed out as a reminder of what our original capital is. (Later on, when we vary capital amounts as well as interest rates, printing the value of C will be necessary anyway.)

At this point, we encounter the J-loop, or inner loop. This loop represents the various amounts of possible interest rates, from 5% to 6.5%, at one half of one percent intervals.

For each of these steps, the J-loop

- calculates I (Principal + Interest).
- prints I consecutively along the same line under appropriate interest rate headings.

In other words, all the interest rates specified by the J-loop are calculated for one year before the N-loop resumes and proceeds to the following year. (The entire J-loop is executed prior to the next pass of the N-loop.)

Year 1 (N = 1):

J = 5

J = 5.5

J = 6

J = 6.5

Year 2 (N = 2):

J = 5

J = 5.5

J = 6

J = 6.5

and so on.

Each time we move to a new value of N, we move through the entire J-loop again, to recalculate the various values of I at different values of J (and, naturally, of I and N).

Therefore,

- the N-loop will go through 5 passes (N=1 TO 5);
- while the J-loop will go through 20 passes (J=1 TO 4 for each of 5 N-loop passes).

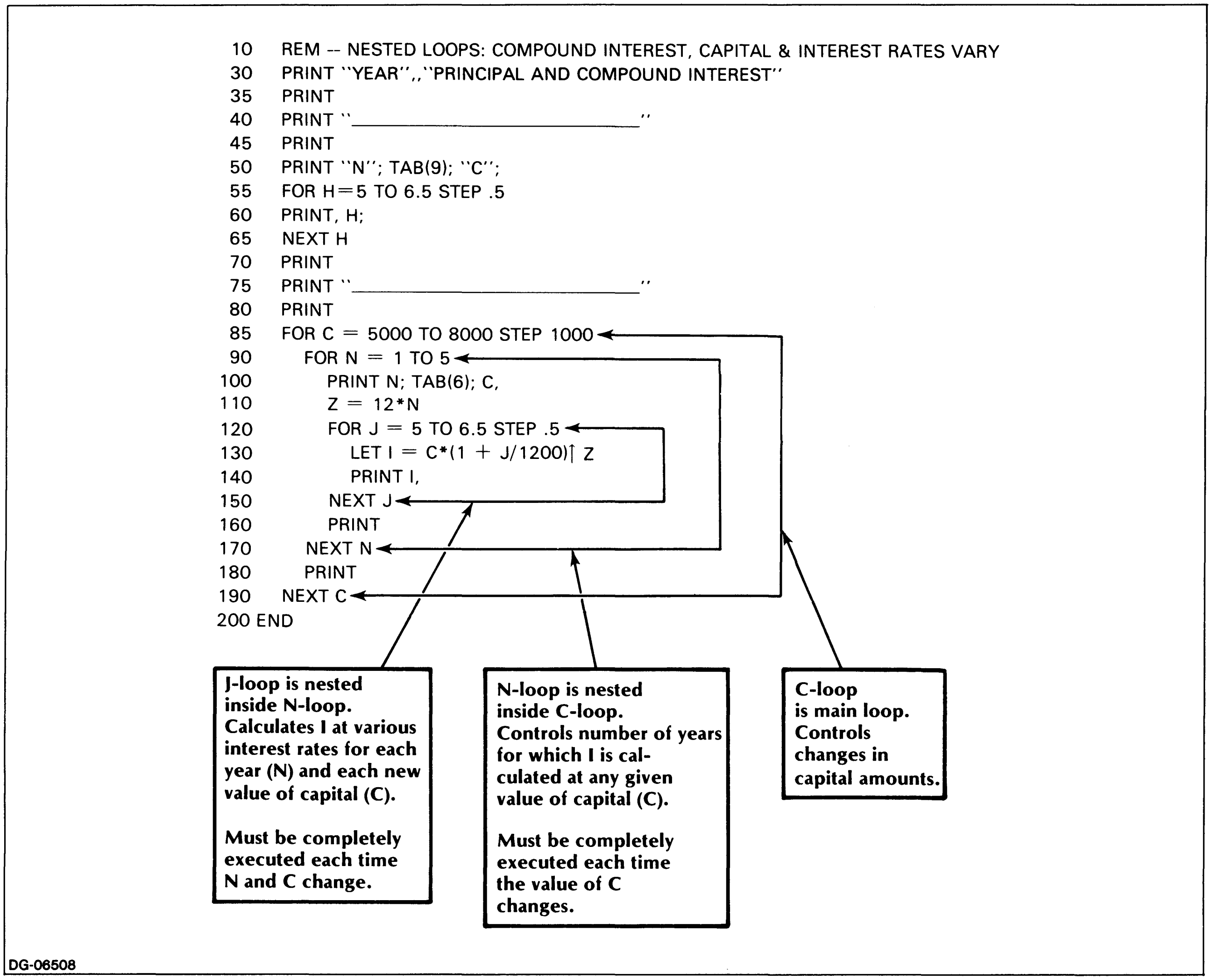
Had we left the calculation of 12 * N within the J-loop, this operation would have been repeated 20 times instead of the necessary five times, a redundancy both inefficient and costly.

This example stresses yet again the need for careful sifting of loop operations to ensure repeated execution only when such repetition is meaningful.

If we run the program, it will look like Figure 5-31.

Compound Interest Varying Capital, Time, and Interest Rates

As we've said before, you can have several loops nested within each other. A quick addition to our compound interest program will highlight the usefulness of this feature:



```
10   REM -- NESTED LOOPS: COMPOUND INTEREST, CAPITAL & INTEREST RATES VARY
30   PRINT "YEAR",,"PRINCIPAL AND COMPOUND INTEREST"
35   PRINT
40   PRINT "______________________________"
45   PRINT
50   PRINT "N"; TAB(9); "C";
55   FOR H=5 TO 6.5 STEP .5
60   PRINT, H;
65   NEXT H
70   PRINT
75   PRINT "______________________________"
80   PRINT
85   FOR C = 5000 TO 8000 STEP 1000
90     FOR N = 1 TO 5
100      PRINT N; TAB(6); C,
110      Z = 12*N
120      FOR J = 5 TO 6.5 STEP .5
130        LET I = C*(1 + J/1200)↑ Z
140        PRINT I,
150      NEXT J
160      PRINT
170    NEXT N
180    PRINT
190  NEXT C
200 END
```

J-loop is nested inside N-loop. Calculates I at various interest rates for each year (N) and each new value of capital (C).

Must be completely executed each time N and C change.

N-loop is nested inside C-loop. Controls number of years for which I is calculated at any given value of capital (C).

Must be completely executed each time the value of C changes.

C-loop is main loop. Controls changes in capital amounts.

DG-06508

Figure 5-32. Program to Calculate Interest, with Various Interest Rates and Principal Amounts

So far we have calculated the principal plus interest for a fixed capital of $5,000 invested at varying rates of interest for five years (i.e., the capital amount remained the same and only the interest rates changed).

Now we would like to change the capital amounts as well as the interest rates. That is, we want to create a table to show compound interest earned over a given period by various capital amounts invested at a number of different interest rates. Would this involve major modifications in our program?

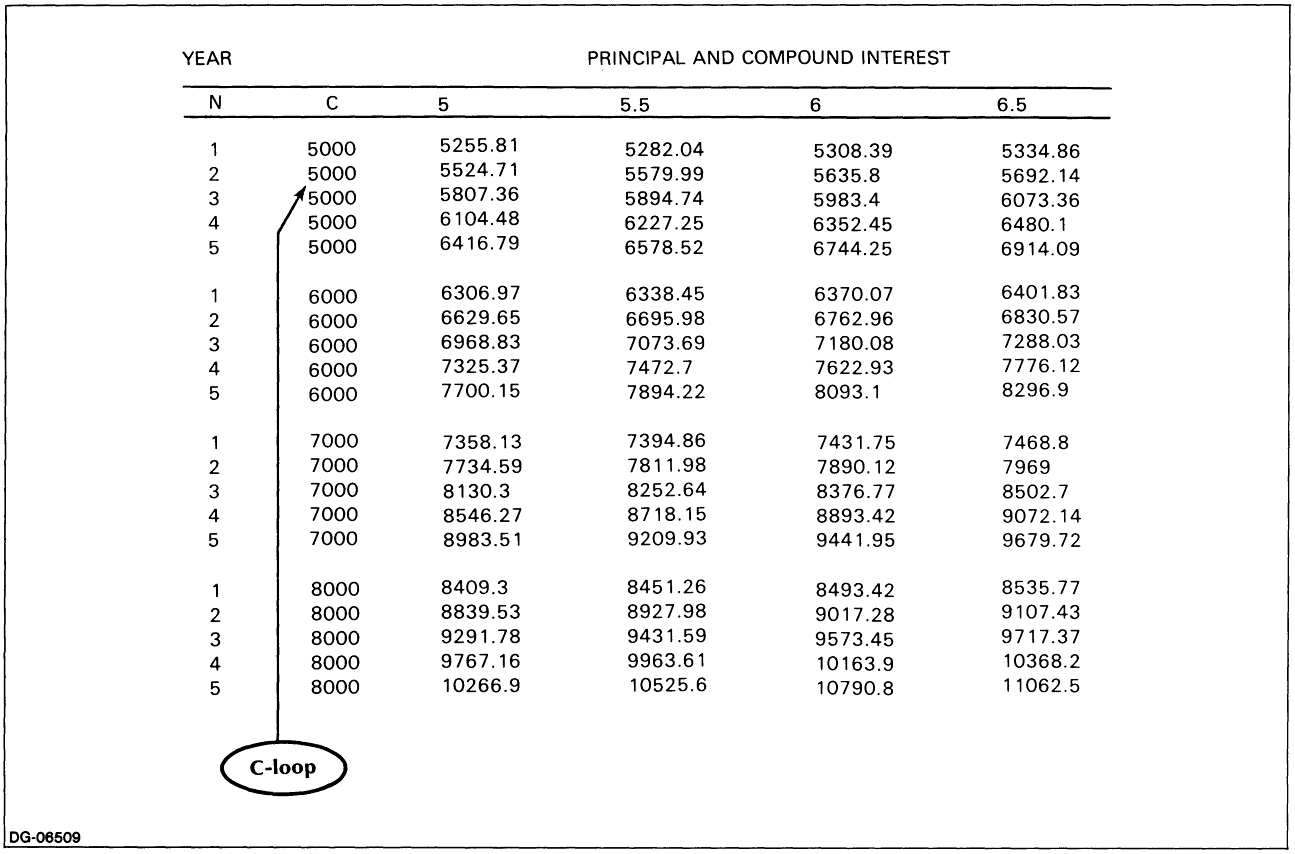
We can create such a table by the mere addition of two lines that will constitute a new outer loop, the C-loop; this loop will specify the capital amcunts for which we want the compound interest calculated. Let's use amounts from $5,000 to $8,000 at $1,000 increments.

Our new outer loop would look like this:

```
85 FOR C = 5000 TO 8000 STEP 1000
                  .
                  .
                  .
                  .
                  .
190 NEXT C
```

Our existing N- and J-loops produced values of I (principal and interest) for various values of J (J= 5 to 6.5 STEP .5) over a period of five years (N = 1 TO 5), for a single value of C (C=5000).

The new outer C-loop causes the N- and J-loops to be executed in their entirety for each new value of C (C = 5000 TO 8000 STEP 1000). In this way, we can easily obtain values for I, while varying both our capital (C) and our interest rates (J).



```
YEAR                     PRINCIPAL AND COMPOUND INTEREST
 N        C        5            5.5          6            6.5

 1       5000     5255.81      5282.04      5308.39      5334.86
 2       5000     5524.71      5579.99      5635.8       5692.14
 3       5000     5807.36      5894.74      5983.4       6073.36
 4       5000     6104.48      6227.25      6352.45      6480.1
 5       5000     6416.79      6578.52      6744.25      6914.09

 1       6000     6306.97      6338.45      6370.07      6401.83
 2       6000     6629.65      6695.98      6762.96      6830.57
 3       6000     6968.83      7073.69      7180.08      7288.03
 4       6000     7325.37      7472.7       7622.93      7776.12
 5       6000     7700.15      7894.22      8093.1       8296.9

 1       7000     7358.13      7394.86      7431.75      7468.8
 2       7000     7734.59      7811.98      7890.12      7969
 3       7000     8130.3       8252.64      8376.77      8502.7
 4       7000     8546.27      8718.15      8893.42      9072.14
 5       7000     8983.51      9209.93      9441.95      9679.72

 1       8000     8409.3       8451.26      8493.42      8535.77
 2       8000     8839.53      8927.98      9017.28      9107.43
 3       8000     9291.78      9431.59      9573.45      9717.37
 4       8000     9767.16      9963.61      10163.9      10368.2
 5       8000     10266.9      10525.6      10790.8      11062.5
```

Figure 5-33. Output of Program in Figure 5-32

The J-loop is still nested inside the N-loop, but now the N-loop itself is nested inside the C-loop.

In other words,

```
When C = 5000
   N = 1
      J = 5
      J = 5.5
      J = 6
      J = 6.5
   N = 2
      J = 5
      J = 5.5
      J = 6
      J = 6.5
   N = 3
      J = 5, etc.
   N = 4
      J = 5, etc.
   N = 5
      J = 5, etc.
```

With every change in the value of C, the above process is repeated. In total,

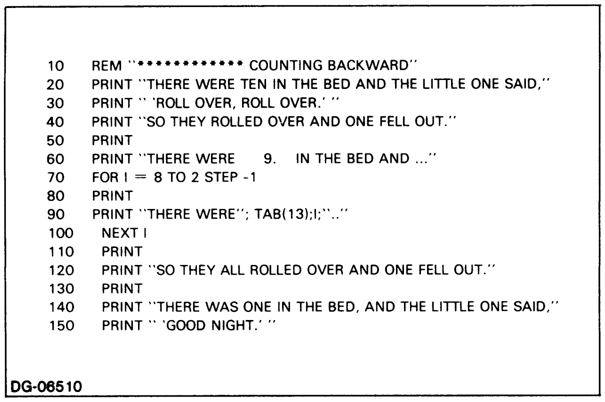
- the C-loop will go through 4 passes;
- the N-loop will go through 20 passes;
- the J-loop will go through 80 passes.

Figure 5-32 displays our program. As you see, it is identical to the program in Figure 5-30, with the exception of lines 85 and 190 (and line 180, which causes a blank line to be printed between the N- and the C- loops, for the sake of visual clarity).

Our compound interest table appears in Figure 5-33; here we show compound interest for capital amounts ranging from $5000 to $8000 at interest rates ranging from 5% to 6.5% for a period of from 1 to 5 years.

Roll over, Roll over...

In Figure 5-20 we illustrated the output resulting from a STEP command with a negative increment. You've doubtless experienced no difficulty in reconstructing the program which generated that output. Here is our version (Figure 5-34).



```
10    REM “************ COUNTING BACKWARD”
20    PRINT “THERE WERE TEN IN THE BED AND THE LITTLE ONE SAID,”
30    PRINT “ ‘ROLL OVER, ROLL OVER.’ ”
40    PRINT “SO THEY ROLLED OVER AND ONE FELL OUT.”
50    PRINT
60    PRINT “THERE WERE     9.    IN THE BED AND ...”
70    FOR I = 8 TO 2 STEP -1
80    PRINT
90    PRINT “THERE WERE”; TAB(13);I;“..”
100    NEXT I
110    PRINT
120    PRINT “SO THEY ALL ROLLED OVER AND ONE FELL OUT.”
130    PRINT
140    PRINT “THERE WAS ONE IN THE BED, AND THE LITTLE ONE SAID,”
150    PRINT “ ‘GOOD NIGHT.’ ”
```

DG-06510

Figure 5-34. Program to Produce Output in Figure 5-20

Keywords in Chapter 5

Table 5-2 lists the keywords introduced in Chapter 5.

Table 5-2. Keywords in Chapter 5

Keyword	**Can be used in** **Program Statement**	 **Immediate Mode**
FOR...NEXT	Yes	No
GOTO...	Yes	No
IF...THEN...ELSE	Yes	No
ON...GOTO...ELSE	Yes	No

End of Chapter

Chapter 6
Subroutines

In our previous chapter we used a Music Quiz program (Figure 5-10) to demonstrate the ON...GOTO... statement. As you may recall, the user was given a choice of three questions ranging from easy to advanced, and was asked to choose a question by typing a number from 1 to 3.

Thereafter, the statement

```
20 ON N THEN GOTO 30, 120, 200 ELSE GOTO 25
```

transferred the user to the appropriate section in the program where the chosen question was printed out.

As a glance at the flowchart of Figure 5-11 will show, a precisely identical sequence of events then took place within each of the three question groups, regardless of their type, namely:

- User inputs answer, A$.
- A$ is compared against the correct answer.
- If A$ was incorrect, inform user accordingly and transfer to the end of the program.
- If A$ was correct, branch to a line where congratulatory message is printed and transfer to the end of the program.

In our program we had to type in this entire sequence three times, once for each of our three questions. This created quite a bit of unnecessary repetition, but at the time we had no other alternative.

In such situations it is often more practical to treat a distinct sequence of reiterated program steps as a subroutine. That is, rather than retyping the sequence each time we need it in our program, we can isolate all of it as a semi-independent unit within the main program.

We create a subroutine by assigning to the sequence a set of line numbers outside the range of those of the rest of the program. In addition, we must provide appropriate commands to allow for the transfer of control to and from this code.

Whenever necessary, we can then branch from any location within the main program to the line number denoting the beginning of this subroutine, work through all its steps, and when we have finished, transfer control back to the main program.

The concept of subroutines is thus an advanced application of the control statements we have discussed in Chapter 5.

Creating a subroutine enables us to type a given sequence of program steps only once, while allowing us to use that sequence an endless number of times from different parts of the program.

GOSUB... and RETURN

To reach a subroutine from a program (this is described as calling a subroutine), you use the GOSUB command followed by a line number. For instance:

```
60 GOSUB 300
```

This command, in essence an extension of the GOTO... statement, tells the program to go to the subroutine beginning on line 300.

The program then branches to line 300, where it begins executing, in order, all the statements it finds on that line and following lines until it encounters a statement directing it to exit the subroutine. This is the RETURN statement, which we write as follows:

```
350 RETURN
```

The return command brings execution back to the main program. But, rather than returning to the line from which we left, namely

```
60 GOSUB 300
```

the RETURN always brings us back to the line directly below it (see Figure 6-1).

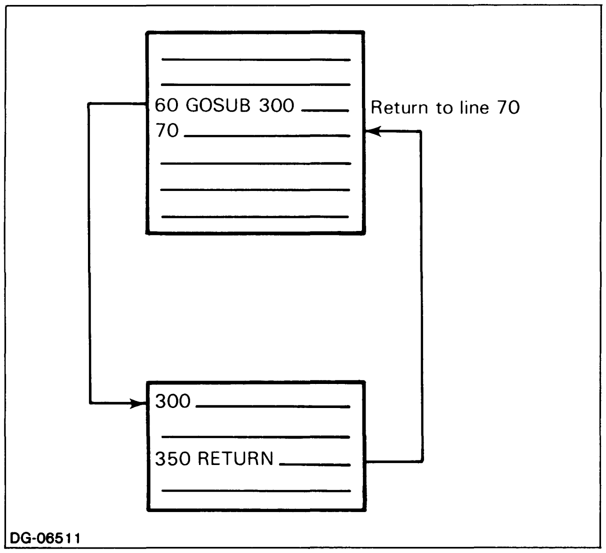


Figure 6-1. Flow of Control To and From a Subroutine

After the return, the program continues sequential execution unless it encounters other control statements.

Your program can contain any number of subroutines. In fact, as we shall see later in the chapter, you can also nest subroutines; that is, a subroutine can call a number of other subroutines, which, in their turn, may call more subroutines. (See Figure 6-5.)

Unlike the GOSUB... command, the RETURN command is not followed by a line number. The reason for this is readily apparent. Since a subroutine will likely be used several times within a program, the destination of the return command will necessarily vary, depending on the location of the latest GOSUB... command from which the program is exiting.

BASIC itself supplies the destination for each particular return. Each time it branches to a subroutine (from the main program or from within another subroutine), it keeps track of its departure point. When instructed to return, it always brings us back to the line below the last GOSUB... statement it just executed, ignoring any previous or subsequent GOSUB... commands.

This is the major distinction between a GOTO... and a GOSUB.... A GO...TO command is a major change of direction in a program: it is a a one-way trip making no provision for a return; and it does not remember its place of origin. The GOSUB... branch, on the other hand, being only a set of subsidiary operations rather than a change of routing, is a round trip with a guaranteed return to its place of origin.

This brings up a cautionary note. You must be sure that every single GOSUB... command in your program is equipped with its matching RETURN command.

A Program Example

We will modify our former Music Quiz program (Figure 5-10) to take advantage of the efficiency provided by subroutines.

Instead of giving the user a choice of questions, let's ask him this time to answer all three questions; let's also use counters to keep track of how many questions the user answers correctly and how many incorrectly.

We'll need the following variables:

C$ = correct answer to a given question;

A$ = user's answer to a given question;

W = counter for number of wrong answers;

C = counter for number of correct answers.

We suggest that you consult the diagrams in Figures 6-2 and 6-3 while you read the explanations below.

The main part of the program now prints out each question in turn and records in C$ the correct answer against which it will compare the user's answer.

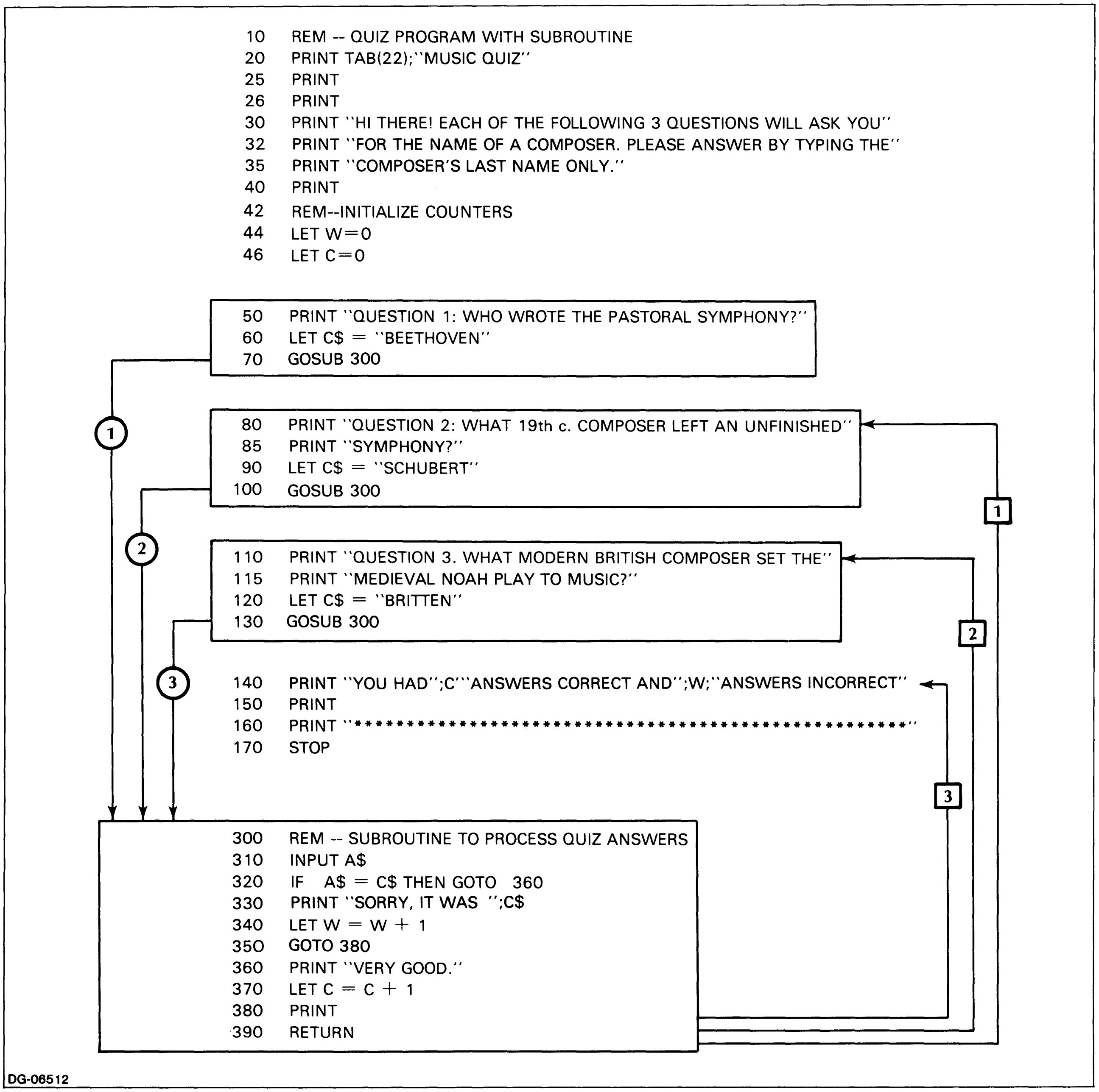
After each question is printed, the program branches to a subroutine set off from the rest of the program by having a totally different line sequence; i.e., 300–390. (The main program lines are numbered 10–170.)

Follow the arrows on the left side of Figure 6-2 to identify the occurrences of the GOSUB 300 command.

The subroutine performs the following steps (illustrated in detail in Figure 6-3).

- User inputs answer (A$);
- Compare A$ to correct answer stored in C$;
- If A$ was incorrect, tell the user and supply the correct answer, increment the counter W (wrong answer), branch to the penultimate line of subroutine which will print a blank line to separate program parts and then lead to a RETURN;
- If A$ was correct, branch to a line congratulating the user, increment the counter, C, which keeps track of the number of correct answers, and RETURN to the main program.

As Figure 6-2 shows, we repeat this sequence three times, each time entering the subroutine from a different part of the main program. Follow the arrows on the left side of Figure 6-2 to trace the subroutine entries.



```
10   REM -- QUIZ PROGRAM WITH SUBROUTINE
20   PRINT TAB(22);"MUSIC QUIZ"
25   PRINT
26   PRINT
30   PRINT "HI THERE! EACH OF THE FOLLOWING 3 QUESTIONS WILL ASK YOU"
32   PRINT "FOR THE NAME OF A COMPOSER. PLEASE ANSWER BY TYPING THE"
35   PRINT "COMPOSER'S LAST NAME ONLY."
40   PRINT
42   REM--INITIALIZE COUNTERS
44   LET W=0
46   LET C=0

50   PRINT "QUESTION 1: WHO WROTE THE PASTORAL SYMPHONY?"
60   LET C$ = "BEETHOVEN"
70   GOSUB 300

80   PRINT "QUESTION 2: WHAT 19th c. COMPOSER LEFT AN UNFINISHED"
85   PRINT "SYMPHONY?"
90   LET C$ = "SCHUBERT"
100  GOSUB 300

110  PRINT "QUESTION 3. WHAT MODERN BRITISH COMPOSER SET THE"
115  PRINT "MEDIEVAL NOAH PLAY TO MUSIC?"
120  LET C$ = "BRITTEN"
130  GOSUB 300

140  PRINT "YOU HAD";C'"ANSWERS CORRECT AND";W;"ANSWERS INCORRECT"
150  PRINT
160  PRINT "*****************************************************"
170  STOP

300  REM -- SUBROUTINE TO PROCESS QUIZ ANSWERS
310  INPUT A$
320  IF  A$ = C$ THEN GOTO  360
330  PRINT "SORRY, IT WAS ";C$
340  LET W = W + 1
350  GOTO 380
360  PRINT "VERY GOOD."
370  LET C = C + 1
380  PRINT
390  RETURN
```

Figure 6-2. Music Quiz Program Using a Subroutine

We GOSUB 300 from

line 70 after Question 1,

line 100 after Question 2,

line 130 after Question 3.

Similarly, we execute three returns from the subroutine, each time re-entering the program in a different place, one line below the specific command line which sent us to the subroutine. (Follow the arrows on the right side of Figure 6-2 to trace the returns.)

From 70 GOSUB 300 RETURN to line 80 where Question 2 is printed.

From 100 GOSUB 300 RETURN to line 110 where Question 3 is printed.

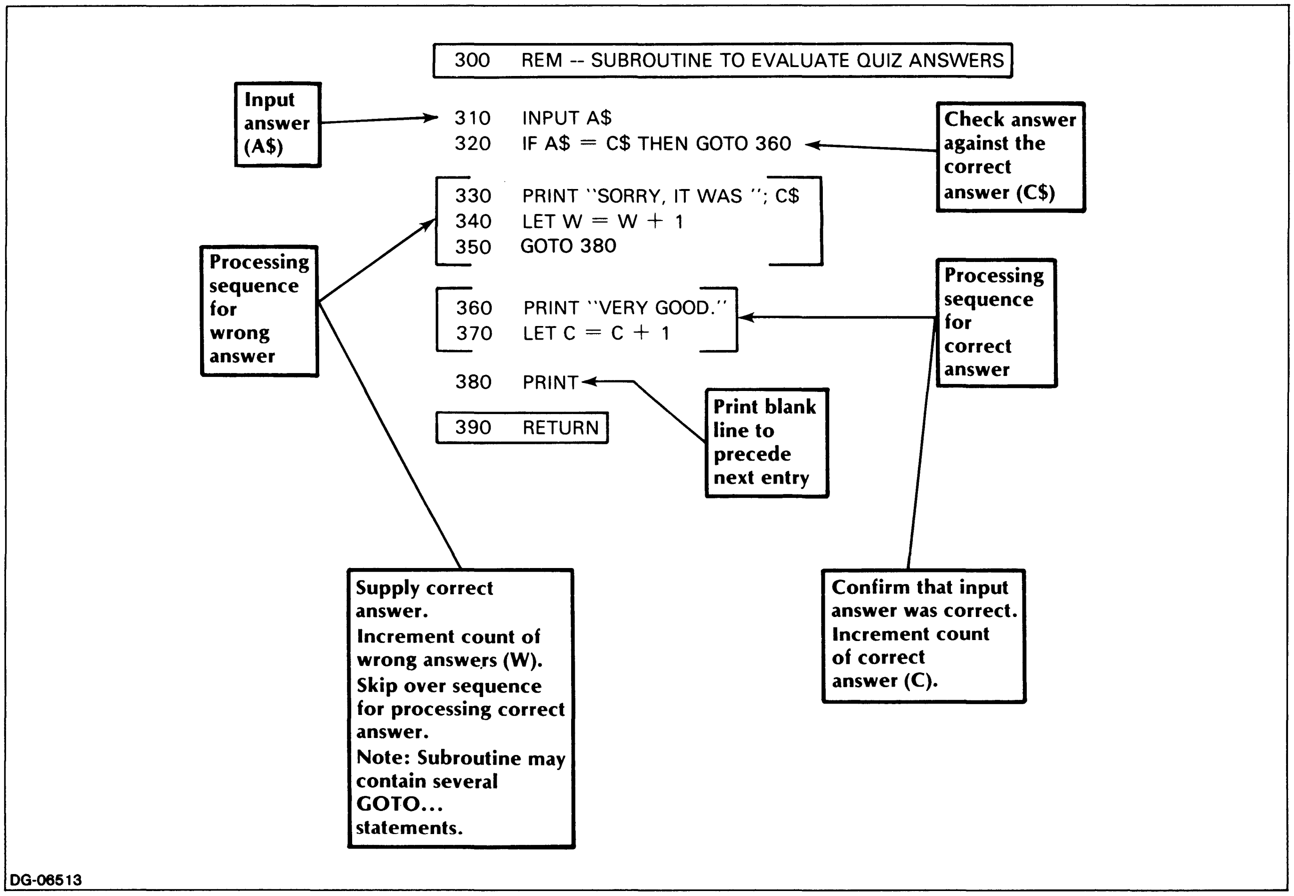
From 130 GOSUB 300 RETURN to line 140 where user is given the total of correct and wrong answers and the program is terminated.

Looking at the last part of the program, you may have noticed the STOP statement on line 170. We have placed this command there as a preventive measure. As it moves down in direct sequence, the program will, in due time, automatically reach line 300 where the subroutine begins, and thus enter the subroutine to no purpose. The STOP command effectively guards the subroutine from such unwarranted intrusion by the program.

Now take a quick look at the subroutine detail in Figure 6-3. Short as it is, our subroutine contains two branching statements:

- At line 320, where we branch to sequence for processing correct answers;
- At line 350, where we branch to line 380 (print blank line and return).

The point to emphasize is that subroutines may contain any number of control statements, in the same way that they can contain other subroutines.



```
300  REM -- SUBROUTINE TO EVALUATE QUIZ ANSWERS
310  INPUT A$
320  IF A$ = C$ THEN GOTO 360
330  PRINT "SORRY, IT WAS "; C$
340  LET W = W + 1
350  GOTO 380
360  PRINT "VERY GOOD."
370  LET C = C + 1
380  PRINT
390  RETURN
```

Input answer (A$)

Check answer against the correct answer (C$)

Processing sequence for wrong answer

Processing sequence for correct answer

Print blank line to precede next entry

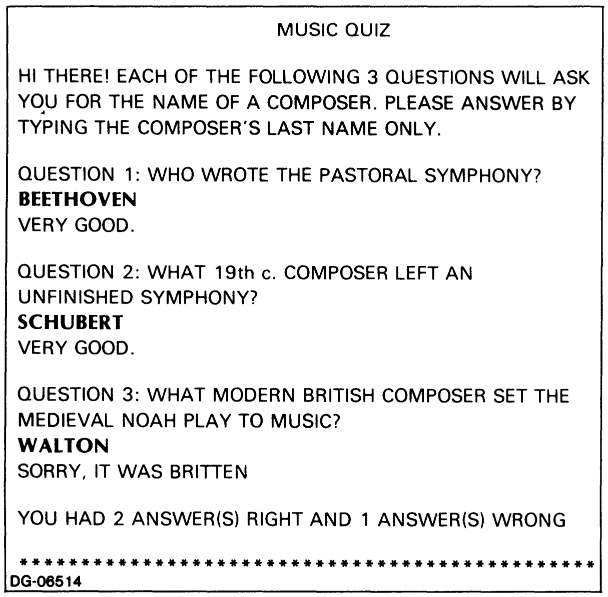
Supply correct answer.
Increment count of wrong answers (W).
Skip over sequence for processing correct answer.
Note: Subroutine may contain several GOTO... statements.

Confirm that input answer was correct.
Increment count of correct answer (C).

DG-06513

Figure 6-3. Subroutine in Music Quiz Program of Figure 6-2

Figure 6-4 displays a run of our Music Quiz program.



```
                     MUSIC QUIZ

HI THERE! EACH OF THE FOLLOWING 3 QUESTIONS WILL ASK
YOU FOR THE NAME OF A COMPOSER. PLEASE ANSWER BY
TYPING THE COMPOSER'S LAST NAME ONLY.

QUESTION 1: WHO WROTE THE PASTORAL SYMPHONY?
BEETHOVEN
VERY GOOD.

QUESTION 2: WHAT 19th c. COMPOSER LEFT AN
UNFINISHED SYMPHONY?
SCHUBERT
VERY GOOD.

QUESTION 3: WHAT MODERN BRITISH COMPOSER SET THE
MEDIEVAL NOAH PLAY TO MUSIC?
WALTON
SORRY, IT WAS BRITTEN

YOU HAD 2 ANSWER(S) RIGHT AND 1 ANSWER(S) WRONG

**********************************************
```

DG-06514

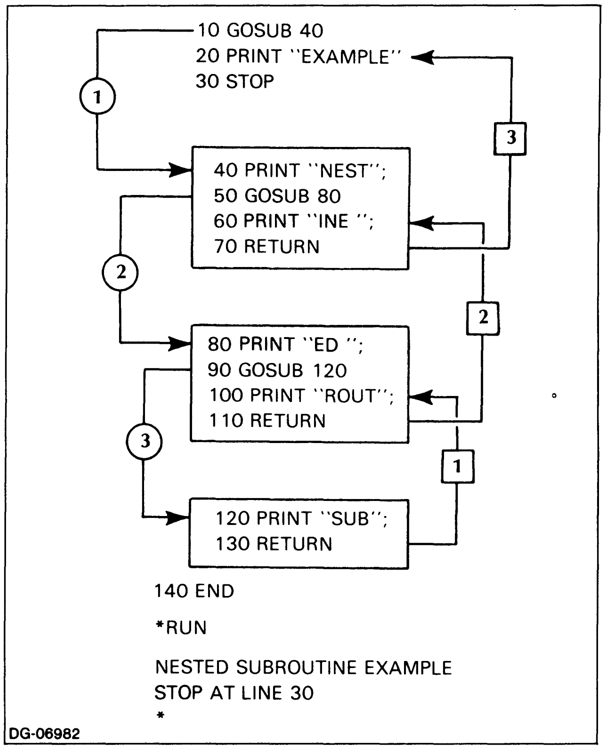
Figure 6-4. Sample Run of Music Quiz Program

When subroutines are nested, a GOSUB... originating from a subroutine will return to the appropriate line within the parent subroutine.

Note that only nine GOSUB statements may be executed without executing a RETURN. An error will be generated when the tenth is attempted. This, in effect, limits the number of nested subroutines to nine.

Nesting is illustrated in Figure 6-5 with two levels of nesting. We have deliberately selected a simple, somewhat contrived example, so as to dramatize the flow of control. As in Figure 6-2, the arrows on the left indicate GOSUB commands, those on the right RETURN commands.

A program can also contain several independent subroutines, identified by the line number which follows the GOSUB command, and returning directly to the main program.



```
10 GOSUB 40
20 PRINT "EXAMPLE"
30 STOP
40 PRINT "NEST";
50 GOSUB 80
60 PRINT "INE ";
70 RETURN
80 PRINT "ED ";
90 GOSUB 120
100 PRINT "ROUT";
110 RETURN
120 PRINT "SUB";
130 RETURN
140 END
*RUN
NESTED SUBROUTINE EXAMPLE
STOP AT LINE 30
*
```

DG-06982

Figure 6-5. Nested Subroutines

The ON...GOSUB...ELSE Statement

As with GOTO, there may be occasions when you want to execute one of several subroutines based on a value or variable. This is accomplished with the ON...GOSUB...ELSE statement. This statement functions exactly like ON...GOTO...ELSE except that program execution returns to the next statement when the subroutine has finished executing.

Again, we will modify our Music Quiz program to use ON...GOSUB...ELSE. We will again give the user a choice of questions. The program is shown in Figure 6-6. We describe the program here.

In this example, the main body of the program is contained in lines 10 through 95. Line 40 accepts the user's selection, and line 50 sends control to one of these subroutines (or back to the question if a number other than 1, 2, or 3 was entered).

Each subroutine prints the appropriate question and assigns the correct answer to the variable C$. When the subroutine is finished, program execution continues with statement 60. There, we input the user's response. If the answer is correct, we print a congratulatory message. If the answer is wrong, we tell the user the correct answer. The user is then given the option to select another question. If the user chooses to select another question, the process is repeated. Otherwise, the program stops.

```
10   REM "QUIZ PROGRAM WITH ON...GOSUB..."
20   PRINT "TO SELECT YOUR QUESTION, TYPE 1 FOR EASY, 2 FOR"
30   PRINT "INTERMEDIATE, OR 3 FOR ADVANCED:"
40   INPUT N
50   ON N GOSUB 100,200,300 ELSE GOTO 20
60   INPUT A$
70   IF A$=C$ THEN PRINT "CORRECT-VERY GOOD" ELSE PRINT "SORRY-IT WAS ";C$
80   PRINT "WOULD YOU LIKE TO SELECT ANOTHER QUESTION? TYPE 'Y' OR 'N':"
90   INPUT S$
95   IF S$="Y" THEN GOTO 20 ELSE STOP
100  PRINT "QUESTION 1: WHO WROTE THE PASTORAL SYMPHONY?"
110  LET C$="BEETHOVEN"
120  RETURN
200  PRINT "QUESTION 2: WHAT 19TH C. COMPOSER LEFT AN"
205  PRINT "UNFINISHED SYMPHONY?"
210  LET C$="SCHUBERT"
220  RETURN
300  PRINT "QUESTION 3: WHAT MODERN BRITISH COMPOSER SET THE MEDIEVAL"
310  PRINT "NOAH PLAY TO MUSIC?"
320  LET C$="BRITTEN"
330  RETURN
```

Figure 6-6. Music Quiz Program with ON...GOSUB...ELSE Statement

Keywords in Chapter 6

Table 6-1 lists the keywords introduced in Chapter 6.

Keywords in Chapter 6

Keyword	Can be used in	
	Program Statement	**Immediate Mode**
GOSUB...	Yes	No
RETURN	Yes	No
ON...GOSUB...ELSE	Yes	No

End of Chapter

Chapter 7
Arrays and Subscripted Variables

Assume that you are selling real estate and that among your listings you have nine Cape-style houses ranging in price from $43,500 to $138,900. You would like to store this information in your computer.

You begin by selecting nine variable names, one for each of your Capes, thus:

C = Cape #1
D = Cape #2
.
.
.
K = Cape #9

Next you need to write nine separate assignment lines to associate the price of each house with its corresponding variable:

```
100 LET C = 119000
110 LET D = 43500
.
.
.
180 LET K = 96800
```

This process, which causes you to select and use up a large number of variable names and to repeatedly type single assignment statements for each variable, is unreasonably cumbersome. (Imagine what it would mean if you wanted to record information on 60 Capes instead of only on 9.)

There is a better way of dealing with a large number of variables of the same type.

It is possible to take all nine of your variables and group them together under one single variable name which will serve as a designation for the whole group. At the same time, this single variable will also be capable of storing each of your separate components individually, so that you can have direct access to them.

Such a group of variables is known as an array. In this chapter we will introduce you to two kinds of arrays: one-dimensional and two-dimensional.

One-Dimensional Arrays

Subscripts

We will use the letter C (for Cape) as our array name (that is, this variable will now denote your whole group of nine Capes). Within C we can think of each specific Cape by the numbers 1 through 9 (for example, Cape 1, Cape 2, etc.).

Now we can represent the price of house I (where I is any number between 1 and 9), by the notation C(I).

The variable in parentheses (I) immediately after the variable name is known as the subscript.

We can now describe C as a subscripted variable. And since the array has only one subscript, (I), we refer to it as a one-dimensional array. See Figure 7-1.

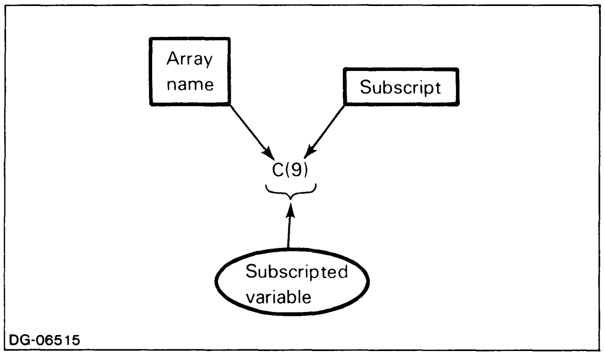


Figure 7-1. Subscripted Variable (Element of an Array

Each subscript represents one single array member. In our case, each subscript denotes one of the nine members of array C. It is much less confusing to refer to each of your Capes as C(1)...C(9), rather than as C, D, E, or whatever.

When pronouncing the names of individual elements of an array, e.g., C(1), C(2),...,C(I), etc., we often say "C sub 1", "C sub 2", or "C sub I", etc., as a sort of verbal shorthand to identify the subscripted elements.

Numeric and String Arrays

There are two types of arrays in MP/BASIC: numeric arrays and string arrays. As you might guess, numeric arrays are composed of elements that are numbers, while string arrays are composed of elements that are character strings. You cannot mix character strings and numbers within the same array. (There are ways, however, to indirectly accomplish this mixture, one of which is illustrated in the example at the end of this chapter.)

Naming Arrays

As we have mentioned, a subscript always appears within parentheses following a variable name. The presence of the parentheses distinguishes an ordinary variable from a subscripted variable. Thus, C1 is an ordinary variable name, whereas C(1) denotes the first member or element of array C, and C1(1) denotes the first element of array C1.

You may not use the same name as both a simple variable name and as an array name within the same program. In our example, you can use the letter C as your array name only if you have not already used it earlier as an ordinary variable.

The same rules apply for array names as for variable names of the same type. That is:

- the name of a numeric array consists of an alphabetic character followed by any combination of alphabetic characters, digits, and the underscore character;
- the name of a string array consists of an alphabetic character followed by any combination of alphabetic characters, digits, and the underscore character, with a dollar sign as the terminating character.

(This is an enhancement to the ANSI standard, which allows array names to be composed only of a single letter for numeric arrays, and a single letter followed by an optional digit and the dollar sign for string arrays.)

Figure 7-2 is a visual representation of array C, which holds nine elements.

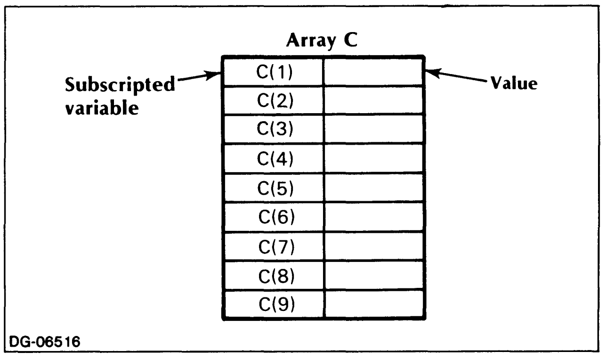
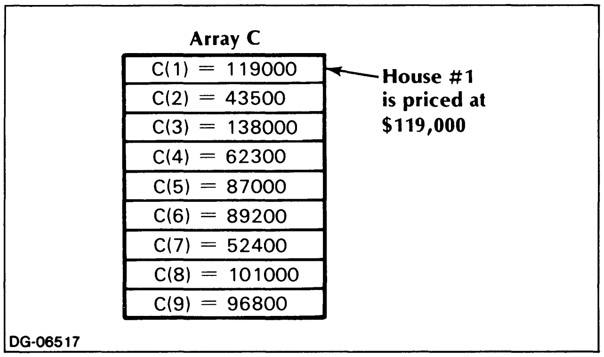


Figure 7-2. Subscripted Variable Names in Array C

Assigning Values to Members of an Array

Figure 7-2 showed us the elements of array C, but these elements have not as yet been assigned any values, that is, no sales price is as yet associated with each of the nine houses represented by C(1) through C(9). We would like the array to hold the values shown in Figure 7-3.



C(1) = 119000
C(2) = 43500
C(3) = 138000
C(4) = 62300
C(5) = 87000
C(6) = 89200
C(7) = 52400
C(8) = 101000
C(9) = 96800

Figure 7-3. Values for Subscripted Variables in Array C

Assigning a value to each member of an array is very easy.

Since we have a single variable name for all the Cape houses, and since each house is designated by a single subscript, we need only create a loop that will go from 1 to 9, read the price of each house from data lines, and store that information in the appropriate array element. We call this *reading values into an array.*

Figure 7-4 shows a program that reads our values (Cape prices) into array C, using only four instruction lines (60–90) instead of the original nine we would need to write separate LET statements for each of the nine variables. Moreover, this kind of simple loop will serve to read values into any array, regardless of its size.

NOTE: In the program in Figure 7-4, we were able to create array C simply by referring to it in line 70. This was possible due to special conditions (to be explained later) pertaining to the array's small size. Normally, however, you would create an array by using a special declaration statement. (See DIM and OPTION BASE statements, below.)

Up to now we have been using the letter I as a subscript for array C. In the program in Figure 7-4 we also use I as our loop counter.

As you know, you can use any letter you wish as your loop counter but, as long as you are within the loop, you must be sure to use that same letter as your subscript. If, for instance, your loop statement had read

```
60 FOR T = 1 TO 9
```

you would designate your array elements as C(T):

```
70 READ C(T)
80 PRINT C(T)
```

This is because the value of your subscript at any given time is keyed to the value of your loop counter; without such a correspondence, the subscript would have no specific value attached to it.

It is thus conceivable that your array subscripts would change several times within a program, depending on the letters you use as loop counters. You might, for example, designate your array elements as C(I) and subsequently as C(J), and C(Z).

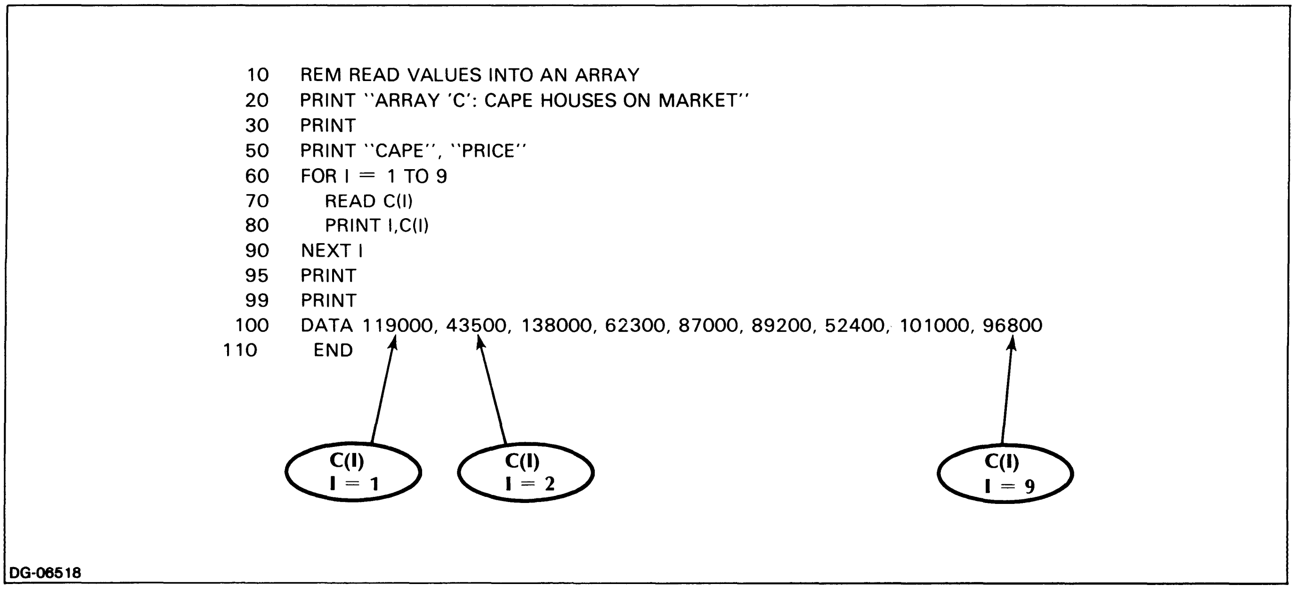
But no matter how many different letters you use as subscripts, the order and contents of your array elements remain unaffected. C(3) is always the third element of array C, regardless of whether you call it C(T), C(Z), or whatever (as long as the subscript corresponds to the loop counter or has been assigned the proper value outside of a loop).

The program in Figure 7-4 will execute as follows.

On its first pass, it will read the first value on the data line, namely 119000 (the sale price of Cape number 1), and store it in location C(I) which corresponds to location C(1) since I=1.

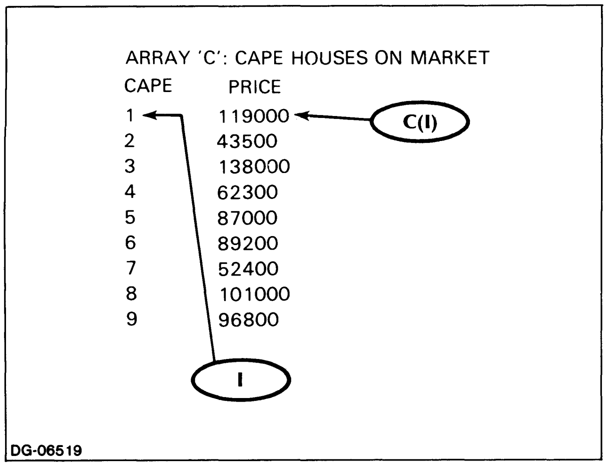
On the second pass, C(I) will now correspond to location C(2), since I = 2; the program will store in this subscript the second value on the data line. This continues until the program has read and stored all the data.

At the end of the loop, our array C will contain nine different values, each tucked away in the location corresponding to its individual element number. The resulting printed output will look like Figure 7-5.



```
 10   REM READ VALUES INTO AN ARRAY
 20   PRINT "ARRAY 'C': CAPE HOUSES ON MARKET"
 30   PRINT
 50   PRINT "CAPE", "PRICE"
 60   FOR I = 1 TO 9
 70     READ C(I)
 80     PRINT I,C(I)
 90   NEXT I
 95   PRINT
 99   PRINT
100   DATA 119000, 43500, 138000, 62300, 87000, 89200, 52400, 101000, 96800
110    END
```

Figure 7-4. Program to Read Values (Cape Prices) into Array C



```
ARRAY 'C': CAPE HOUSES ON MARKET
CAPE     PRICE
1        119000
2        43500
3        138000
4        62300
5        87000
6        89200
7        52400
8        101000
9        96800
```

Figure 7-5. Output from Array (Cape House) Program

Referencing Members of an Array

Referencing (sometimes also described as addressing) means referring to a specific data item in your program. (When you type, for example, PRINT Z, you are referencing that variable, and the computer will print out its current value.)

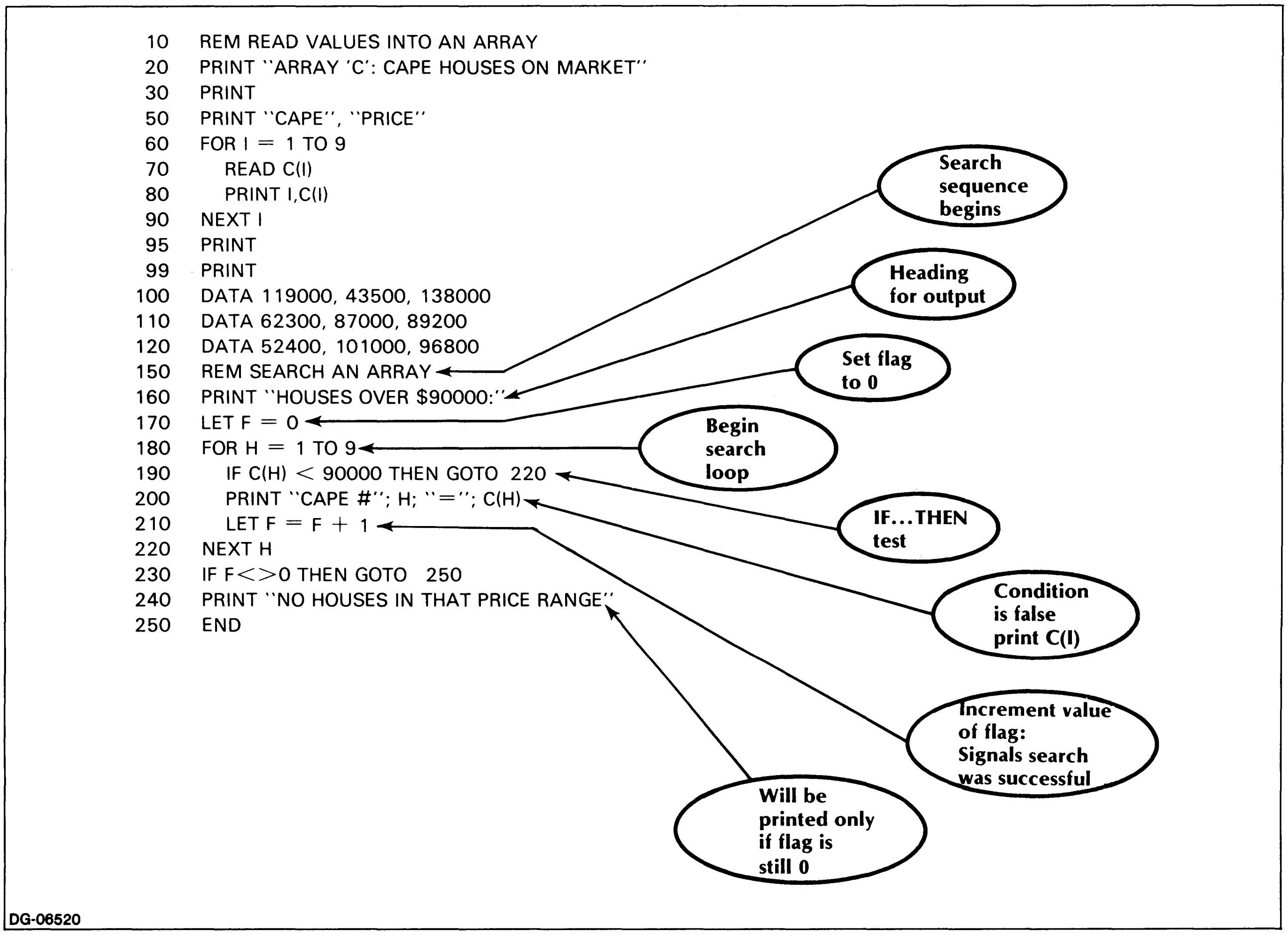
You can retrieve any member of your array by referring to its subscript, which is its individual address within the array. When added to the program, a line such as

```
105 PRINT "C(4) = "; C(4)
```

will produce the following output:

C(4) = 62300

Change line 105 a few more times to reference other members of array C.



```
 10  REM READ VALUES INTO AN ARRAY
 20  PRINT "ARRAY 'C': CAPE HOUSES ON MARKET"
 30  PRINT
 50  PRINT "CAPE", "PRICE"
 60  FOR I = 1 TO 9
 70    READ C(I)
 80    PRINT I,C(I)
 90  NEXT I
 95  PRINT
 99  PRINT
100  DATA 119000, 43500, 138000
110  DATA 62300, 87000, 89200
120  DATA 52400, 101000, 96800
150  REM SEARCH AN ARRAY
160  PRINT "HOUSES OVER $90000:"
170  LET F = 0
180  FOR H = 1 TO 9
190    IF C(H) < 90000 THEN GOTO 220
200    PRINT "CAPE #"; H; "="; C(H)
210    LET F = F + 1
220  NEXT H
230  IF F<>0 THEN GOTO  250
240  PRINT "NO HOUSES IN THAT PRICE RANGE"
250  END
```

Search sequence begins
Heading for output
Set flag to 0
Begin search loop
IF...THEN test
Condition is false print C(I)
Increment value of flag: Signals search was successful
Will be printed only if flag is still 0
DG-06520

Figure 7-6. Array Search Added to Cape House Program

Your subscripts can also occasionally be expressions, and you can reference them as such. For instance, assume that you are writing a loop to compare the price difference between each of your nine houses and the one preceding it, beginning with house number 2.

On each pass of the loop, the current house will then be designated as C(I) and its predecessor as C(I-1).

Searching an Array

Often, especially in working with large arrays, you may not know which element contains the data you are looking for. Suppose, for example, that you want a list of all Capes priced over $90,000 without knowing their location within the array.

You would reference these elements by having the computer search the entire array and apply an IF...THEN... test to retain only the desired elements.

That is, you write a loop which will look at each single element, C(I) (where I = 1 to 9), test whether it falls in your price range, and print out only those elements for which the test is true.

Figure 7-6 illustrates this addition to the program. (Lines 10–120 correspond to the original program in Figure 7-4.)

You will find this type of array search extremely useful any time you are faced with the need of retrieving a set of data from an array.

Flags

In lines 170, 210, and 230 of the program in Figure 7-6, we performed several operations with variable F without explaining them first.

The idea behind these operations was to build into the program a signal to tell us whether the array search was successful or not.

Had we been able to use a visual signal such as a flag, we might agree that "flag down" would mean "nothing was found", whereas "flag up" would mean that the search turned up at least one item.

We can use variable F to function internally as such a flag: if the value of F is zero, this will mean that nothing was found; any value of F greater than zero will signal that some houses were found and will tell us how many. In this way, F can serve simultaneously as a flag and as a counter.

We set F to 0 before searching the array (line 170). When the program encounters a house priced over $90,000, it will reach line 210, where it will increment the value of F by 1. (If the array contains no houses over $90,000, the value of F will remain 0, since the program will never reach line 210.)

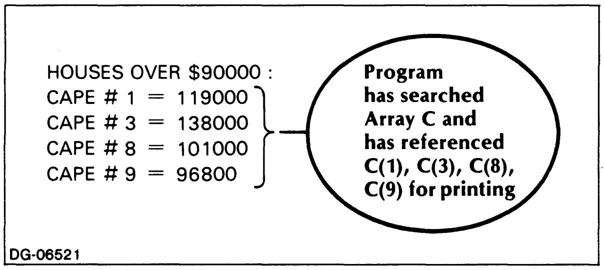
Subsequent passes of the loop will alter the value of F to reflect the number of additional houses found.

At the end of the program, we check the value of F by means of an IF...THEN... statement. If that value is not equal to zero, this means that at least one house in the desired price range was found and listed for the user and, therefore, the program can terminate. If, however, the value of F has remained at zero, this means that no houses were found. In that case, the program will print out a message such as the one in line 240:

NO HOUSES IN THAT PRICE RANGE

In searches involving large numbers of items it may also be useful to have the program print a message giving the value of F (i.e., total number of items found).

Figure 7-7 illustrates the result of the array search, wherein four of the Cape houses have been referenced and listed as being in the right price range.



```
HOUSES OVER $90000 :
CAPE # 1  =  119000
CAPE # 3  =  138000
CAPE # 8  =  101000
CAPE # 9  =  96800
```

Program has searched Array C and has referenced C(1), C(3), C(8), C(9) for printing

Figure 7-7. Sample Run of Program of Cape House (Array Search Version)

Try running the program and changing line 190 so as to look for houses in a nonexistent price range. For example, test for houses under $40,000 (change the wording of the heading in line 160 as well). In this way you will see how the flag operates to signal negative results.

Declaring an Array: The DIM Statement

An array declaration is a statement that tells the program how many elements your array is to contain. Then the appropriate number of spaces is reserved for that array and made available to it. In a moment you will see why this was not necessary in our previous example.

To declare an array, you use a DIM statement. (We have previously encountered this statement in Chapter 3 in connection with declaring string variable length.) A DIM statement for arrays lists the name of your array and its last subscript number, thus:

```
5 DIM D(12)
```

This is interpreted as meaning that you are creating array D which is to contain up to 13 elements, with subscripts ranging from 0 to 12. The element count always starts with 0 unless otherwise specified. (See OPTION BASE, below.) The program will then allow you to reference up to 13 elements in array D.

The ANSI standard specifies that your DIM statement must appear on a lower-numbered line than your first reference to this array. MP/BASIC, however, requires only that program execution pass through the DIM statement before the array is referenced; thus, your dimensioning declaration may appear anywhere in the program, provided the appropriate control statements cause it to be reached before you reference the array.

Once you have declared an array, you cannot revise its dimensions. That is, having explicitly dimensioned array D for 13 elements by means of a DIM statement, you cannot later in the program use a second DIM statement to redimension the array to some other number of elements. An array can be dimensioned only once in a program.

The maximum size of one-dimensional arrays is 32,767, although the actual size usable is somewhat less than this maximum.

If your program has several arrays, each array must be dimensioned separately, although you can combine several declarations on a single line. For example:

```
5 DIM A(30), D(80), X(15)
```

You need not actually use up all the elements you have reserved in your DIM statement. Thus, you might choose to assign values to only 20 out of the 31 members of array A above. But the program will allow you to address any element between 0 and 31. The function of the dimension statement, then, is to set the maximum size of any particular array. Should you, on the other hand, attempt to refer to a 35th element, A(35), in an array dimensioned for only 31 elements, you will get an error message.

Default Values of DIM

When the program finds a reference to an array that has not been preceded by a dimensioning statement, it automatically reserves 11 spaces for that array, with subscripts ranging from 0 through 10. This is described as a default value (assigned in default of any other declaration on your part).

It follows that you must explicitly dimension all arrays exceeding 11 elements.

Now the reason for our undeclared array in Figure 7-4 becomes clear. Array C, which we used there, referenced only nine elements; hence the default size allocated to that array in the absence of an explicit DIM statement was sufficient for our purposes.

Thus, if your array contains fewer than 11 elements, a dimensioning declaration is optional. Yet omitting that declaration results in wasted memory space, since the default value may allocate you more space than you need or want available.

Declaring an Array: The OPTION BASE Statement

As we noted earlier, the first element in any array corresponds to, i.e., I = 0 where I is the subscript of the array. Our loop to read and print out array C (the list of nine houses and their sale prices) in Figure 7-4 could thus have been written

```
60 FOR I = 0 TO 8
```

rather than

```
60 FOR I = 1 TO 9
```

But since we were also printing out the value of the subscript, this would have caused the printed list of Capes to begin with Cape number 0, a bizarre number for a house priced at $119,000!

As a result of such aesthetic considerations, element 0 of array C has remained unused (as has default element 10 at the array's upper bound).

If we want to avoid wasting that extra 0th element, we can assign a different lower bound to our arrays; that is to say, we can cause the beginning of an array to be set at either I = 1 or at I = 0.

This is done through an OPTION BASE declaration.

Thus,

```
5 OPTION BASE 1
10 DIM D(7), W(29)
```

means that arrays D and W are to contain 7 and 29 members, numbered from 1 to 7 and from 1 to 29, respectively.

In the absence of an OPTION BASE statement, the default value of the array's lower bound would be 0.

The OPTION BASE statement establishes the lower bound for all arrays in a program. It should be used only once in a program, and should be executed before the first DIM statement.

In summary:

- A DIM statement declares the upper bound of an array subscript;
- An OPTION BASE statement declares the lower bound of all array subscripts in your program.

A Program Example

In this example, we are helping the Audubon Society with a bird survey. They have divided the country into three regions, and they would like a program that will record the number of bird species sighted in each region during each of the year's four seasons. The program should also compute the average number of species sighted in each region per season.

We begin by creating three one-dimensional arrays, N1, N2, N3, one for each of the three regions. Each of these arrays will contain four elements indicating the number of species sighted in each of the four seasons in region one, two, and three. (The program will read the values for these array elements from data lines.)

Figure 7-8 illustrates the three arrays N1, N2, and N3.

Reading along Figure 7-8 horizontally, we see that

N1(1), N2(1), N3(1) = number of species sighted during season 1 (spring) in regions 1, 2, and 3;

N1(2), N2(2), N3(2) = number of species sighted during season 2 (summer) in regions 1, 2, and 3;

and so on.

This is also the way in which our program will read the data, so we should enter our values in that same order; i.e., we should list all the data for a given season, I, for regions 1 through 3 before moving on to the next season.

Since it will be necessary to total the number of species sighted in each region regardless of season, 'we'll also need the formula

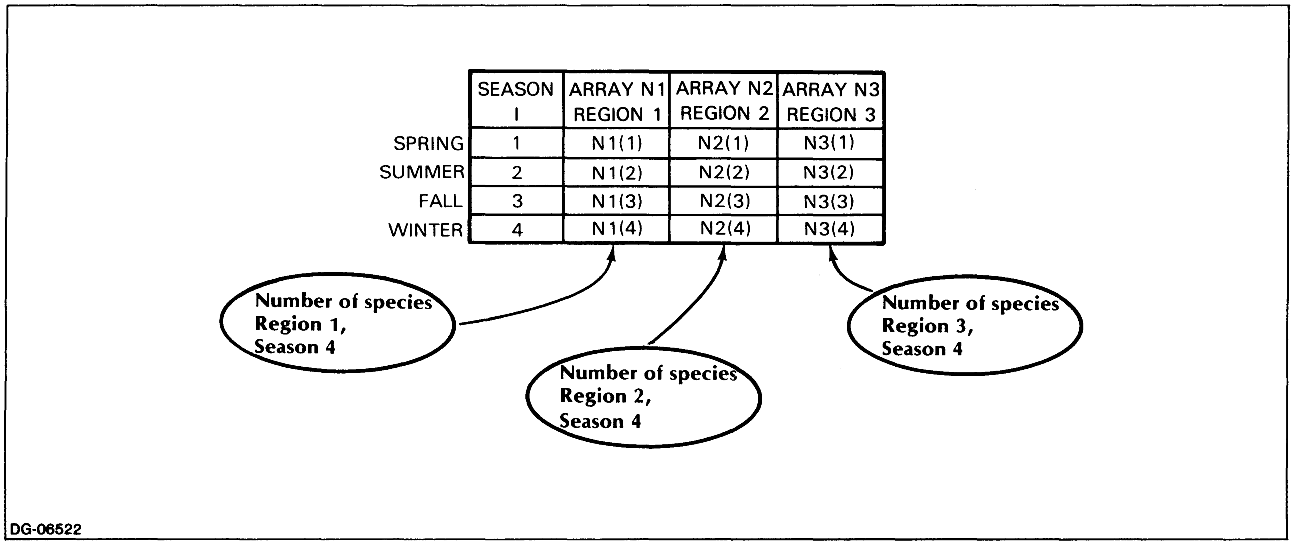
S1 = S1 + N1(I)
S2 = S2 + N2(I)
S3 = S3 + N3(I)

where

S1 = the sum total of species sighted in region 1
S2 = the sum total of species sighted in region 2
S3 = the sum total of species sighted in region 3

Lastly, to determine the average number of species per region, we simply divide the total for each region S1, S2 and S3 by 4 (the number of seasons):

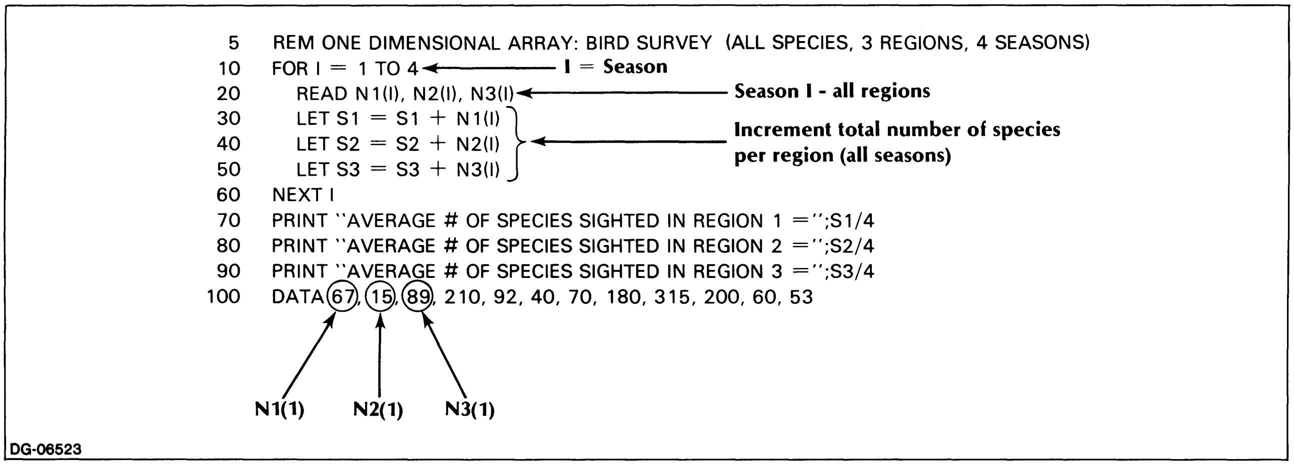
Average for region 1: S1/4
Average for region 2: S2/4
Average for region 3: S3/4



	SEASON I	ARRAY N1 REGION 1	ARRAY N2 REGION 2	ARRAY N3 REGION 3
SPRING	1	N1(1)	N2(1)	N3(1)
SUMMER	2	N1(2)	N2(2)	N3(2)
FALL	3	N1(3)	N2(3)	N3(3)
WINTER	4	N1(4)	N2(4)	N3(4)

Number of species Region 1, Season 4
Number of species Region 2, Season 4
Number of species Region 3, Season 4
DG-06522

Figure 7-8. Three Arrays for Bird Surveying Program



```
  5  REM ONE DIMENSIONAL ARRAY: BIRD SURVEY (ALL SPECIES, 3 REGIONS, 4 SEASONS)
 10  FOR I = 1 TO 4
 20    READ N1(I), N2(I), N3(I)
 30    LET S1 = S1 + N1(I)
 40    LET S2 = S2 + N2(I)
 50    LET S3 = S3 + N3(I)
 60  NEXT I
 70  PRINT "AVERAGE # OF SPECIES SIGHTED IN REGION 1 =";S1/4
 80  PRINT "AVERAGE # OF SPECIES SIGHTED IN REGION 2 =";S2/4
 90  PRINT "AVERAGE # OF SPECIES SIGHTED IN REGION 3 =";S3/4
100  DATA 67, 15, 89, 210, 92, 40, 70, 180, 315, 200, 60, 53
```

Figure 7-9. Bird Surveying Program (Using Three Arrays)

Our program (Figure 7-9), will read the data into the three arrays, then total and average them for each array.

This will produce the output shown in Figure 7-10.

```
AVERAGE # OF SPECIES SIGHTED IN REGION 1 = 136.75
AVERAGE # OF SPECIES SIGHTED IN REGION 2 = 86.75
AVERAGE # OF SPECIES SIGHTED IN REGION 3 = 124.25
```

DG-06524

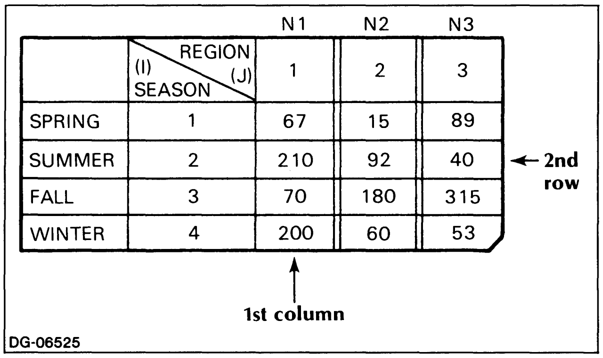
Figure 7-10. Output from Bird Surveying Program (Using Three Arrays)

Two-Dimensional Arrays (Double Subscripts)

The bird survey program in Figure 7-9 called for three separate one-dimensional arrays, N1, N2 and N3, one for each region. Each array contained four elements, indicating the number of species sighted in each of the four seasons.

Thus, arrays N1, N2, and N3 corresponded respectively to regions 1, 2 and 3, and the subscript I denoted the Ith season.

Since all the above arrays deal with similar data, it would certainly be helpful to combine them into a single array capable of distinguishing among regions as well as among seasons. Figure 7-11 illustrates such an arrangement.



		N1	N2	N3
	REGION (I) (J) SEASON	1	2	3
SPRING	1	67	15	89
SUMMER	2	210	92	40
FALL	3	70	180	315
WINTER	4	200	60	53

2nd row
1st column
DG-06525

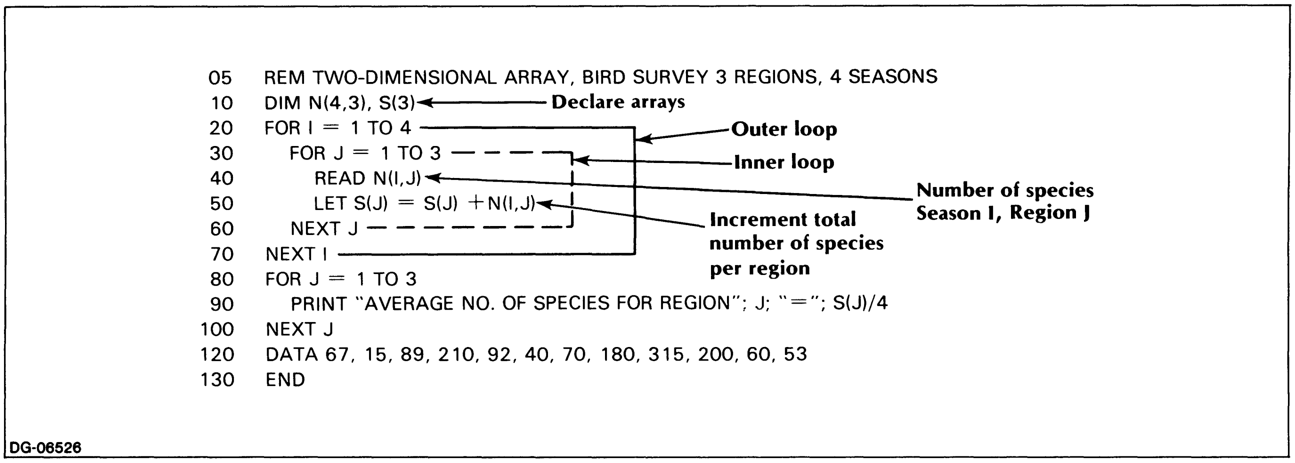
Figure 7-11. Matrix References both Regions and Seasons

By placing our previous arrays N1, N2 and N3 side by side, as before, we obtain a two-dimensional arrangement of elements. Such an arrangement is called a matrix.

As you look at this matrix, you notice rows of elements (horizontal), and columns of elements (vertical). Figure 7-11 has four rows, each corresponding to one of the seasons, and three columns, one for each region.

If we use the subscript I to denote rows (e.g., I=1 corresponds to spring, I=2 to summer, etc.), and the subscript J to designate the columns (J=1 to 3 for regions 1 to 3), then any element in row I and column J represents the number of species sighted in season I and region J.

We can represent such a value by using an array with two subscripts, i.e., N(I,J)



```
 05  REM TWO-DIMENSIONAL ARRAY, BIRD SURVEY 3 REGIONS, 4 SEASONS
 10  DIM N(4,3), S(3)
 20  FOR I = 1 TO 4
 30    FOR J = 1 TO 3
 40      READ N(I,J)
 50      LET S(J) = S(J) +N(I,J)
 60    NEXT J
 70  NEXT I
 80  FOR J = 1 TO 3
 90    PRINT "AVERAGE NO. OF SPECIES FOR REGION"; J; "="; S(J)/4
100  NEXT J
120  DATA 67, 15, 89, 210, 92, 40, 70, 180, 315, 200, 60, 53
130  END
```

Declare arrays
Outer loop
Inner loop
Number of species Season I, Region J
Increment total number of species per region
DG-06526

Figure 7-12. Bird Surveying Program Using a Two-Dimensional Array

Declaring a Two-Dimensional Array

You declare a two-dimensional array by using a DIM declaration in which you list both of the array's subscripts.

The statement

5 DIM N(4,3)

means that you are reserving space for a 5 x 4 matrix, or a total of 20 elements, including the initial zero. This gives you five rows and four columns.

NOTE: You cannot use the same variable name for a one-dimensional and a two-dimensional array: C(I) cannot be reused in the same program as C(I,J).

Default Size of Two-Dimensional Arrays

In the absence of a DIM declaration, the default size assigned to a double-subscripted array is 121 elements, i.e., an 11 x 11 matrix, 0 to 10, in each dimension.

Referencing Elements of a Two-Dimensional Array

To address any member of a two-dimensional array, we must refer to both its subscripts. Thus, the number of species sighted in the fall in region 2 could be expressed as N(3,2)

Similarly, you could express the number of species sighted in the spring in region 1 as N(1,1) and so on.

Moving from left to right, the first subscript denotes rows (here, I), and the second denotes columns (here, J).

Program Example

We will now modify the bird survey program by replacing its original three arrays with a single two-dimensional array.

We will include a nested loop in the program to read the data. Since the program will read the data by rows, I will be the main, or outer loop.

For each value of I (going from 1 to 4) we will read 3 values of J (going from 1 to 3); i.e., for every season represented by I, we will read in the number of species sighted in that season in each of the three regions, J.

Hence, J will be the inner loop.

Since the regions are now represented by a separate subscript, J, the three variables, S1, S2, and S3 (which we needed to compute the totals and averages of species per region) can now be replaced by a single array S, in which the element S(J) denotes the total number of species sighted in region J.

Figure 7-12 shows our new bird survey program. Figure 7-13 shows how the nested loops will work during execution.

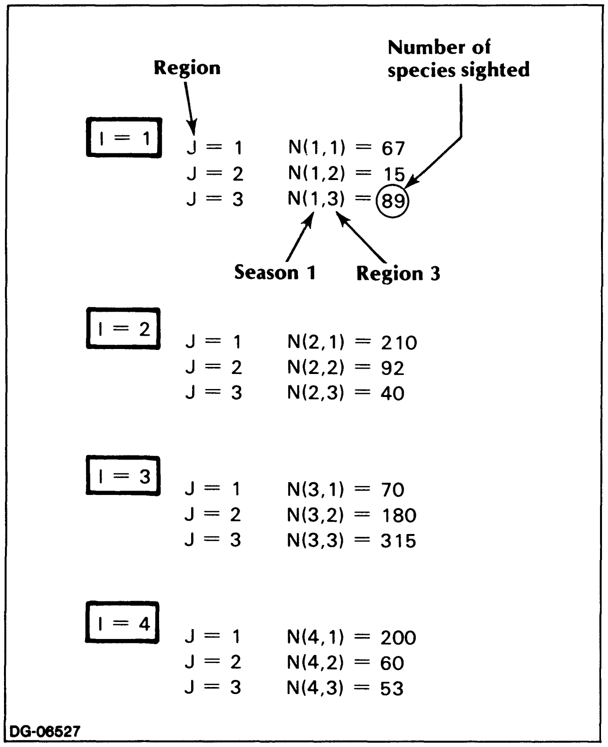


Figure 7-13. Nested Loops in Bird Surveying Program (Two-Dimensional Array Version)

The computer passes through the I loop 4 times; on each of these passes, the J loop is run 3 times. Thus, the J loop is run a total of 12 times.

Figure 7-14 displays the program's output.

```
AVERAGE NO. OF SPECIES FOR REGION 1 = 136.75
AVERAGE NO. OF SPECIES FOR REGION 2 = 86.75
AVERAGE NO. OF SPECIES FOR REGION 3 = 124.25
```

DG-06528

Figure 7-14. Output from Bird Surveying Program (Two-Dimensional Array Version)

Naturally, the results of this printout are exactly the same as those obtained with our earlier program (Figures 7-10 and 7-14 are identical). But we arrived at these results more efficiently when we used two-dimensional arrays.

If we had been dealing with 200 regions instead of only three and working with one-dimensional arrays, we would have needed 200 such arrays (N1...N200) as well as 200 program lines to increment two hundred variables (S1...S200) representing the totals for each region.

But by using two-dimensional arrays you can handle any amount of data by means of the same half dozen lines we used for our I and J loops in Figure 7-12.

String Arrays

So far, in this chapter, we have discussed numeric arrays. Arrays may also contain string data.

String arrays work just like numeric arrays. They can be one- or two- dimensional. Each individual element in a string array can accommodate a maximum of 18 characters.

Let's go back to our Cape house example. In addition to the price of a particular house, we might also want to save the street address. We accomplish this by using a two-dimensional array of 2 columns with 9 elements in each column. Column one will contain the address, column two will contain the price. Figure 7-15 illustrates this array.

```
ARRAY 'CAPES$':CAPE HOUSES ON MARKET

ADDRESS                 PRICE

14 MAIN ST.              119000
17436 FARMING ST.       43500
2 KINGSWAY CT.           138900
4638 JONES AVE.         62300
345 CENTER ST.          87000
26 CHURCH LN.           89200
1629 LAKE ST.           52400
100 BEACH LN.           101000
226 W. 26TH ST.         96800
```

Figure 7-15. A Two-Dimensional String Array

The program shown in Figure 7-16 creates, and then prints, this array.

```
 10 REM "CREATE AND PRINT A STRING ARRAY"
 20 DIM CAPES$(9,2)
 30 FOR I=1 TO 9
 40    FOR J=1 TO 2
 50       READ CAPES$(I,J)
 60    NEXT J
 70 NEXT I
 80 PRINT "ARRAY 'CAPES$':CAPE HOUSES ON MARKET"
 90 PRINT
100 PRINT "ADDRESS                    PRICE"
110 PRINT
120 FOR I=1 TO 9
130    FOR J=1 TO 2
140       PRINT CAPES$(I,J),
150    NEXT J
160    PRINT
170 NEXT I
180 STOP
190 DATA "14 MAIN STREET","119000","17436 FARMING ST."
200 DATA "43500","2 KINGSWAY COURT","138900"
210 DATA "4638 JONES AVE.","62300","345 CENTER ST."
220 DATA "87000","26 CHURCH LANE","89200"
230 DATA "1629 LAKE STREET","52400","100 BEACH LANE"
240 DATA "101000","226 W. 26TH ST.","96800"
```

Figure 7-16. Revision of Cape House Program to Create and Print a Two-Dimensional String Array

The array is created in statements 20 through 70, using the DATA statements in lines 190 through 240. The array is printed in lines 100 through 170.

Notice the use of the DIM statement in line 20. Without this statement, the program would still work, but we would be wasting memory space.

The data in one array must be all numeric or all string; you cannot have an array which contains a mixture of string and numeric data. Notice that we got around this restriction in the program shown by treating the prices as character strings. If you want to expand the program in Figure 7-16 to perform some arithmetic calculation on the prices (for example, calculate the average price), you can use the VAL function (described in Chapter 8) to extract the numeric value represented by the character strings in the second column of the arrray CAPE$.

Individual elements in a string array can accommodate a maximum of 18 characters.

Keywords in Chapter 7

Table 7-1 lists the keywords introduced in Chapter 7.

Table 7-1. Keywords in Chapter 7

Keyword	Can be used in	
	Program Statement	**Immediate Mode**
DIM	Yes	Yes
OPTION BASE	Yes	Yes

End of Chapter

Chapter 8
Functions

Functions are independent programs stored in the computer that perform specific mathematical or string operations. A user program can call a function program whenever it needs to have any of these operations executed.

BASIC, like other systems, makes available to you a repertoire of functions, which you can use throughout your programs at will. Such a collection saves you, as a programmer, a great deal of coding time and enables you to use a function without having to know the details of the code that produced it.

In this chapter we discuss two kinds of functions: implementation-defined functions made available to you by the system and user-defined functions, which you can create yourself.

Implementation-Defined Functions

There are two types of implementation-defined functions:

Mathematical functions

String functions

Parts of a Function

Figure 8-1 illustrates the parts of a function; these are common to mathematical and to string functions alike.

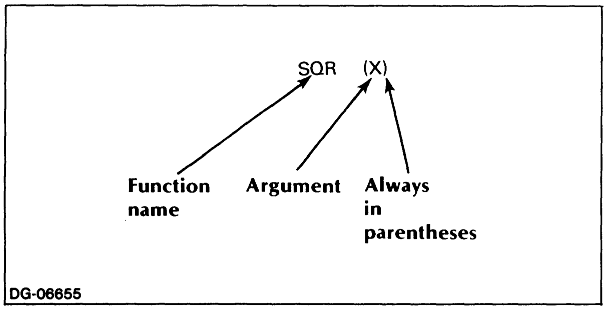


Figure 8-1. Parts of a Function

Let us consider each of these parts very briefly.

The function name is usually followed by an argument; this may be either a constant, a variable, or an expression which you supply at the time you call for the function, for example,

```
SQR(100)

SQR(A^2+ B^2)
```

Note that the argument must always be enclosed in parentheses.

You must ensure, of course, that your variables have been assigned specific values before the computer reaches the line on which you wish the function to be executed. The value of the function is the result that the function returns after it has been executed.

Calling a Function

Calling a function (also described as *invoking* or *referencing* a function) is the process whereby you cause the system to execute a function as part of your program.

To call a function, you name that function and list its argument within the program line where you wish it executed. For example,

```
100 LET X = SQR(100)
```

or

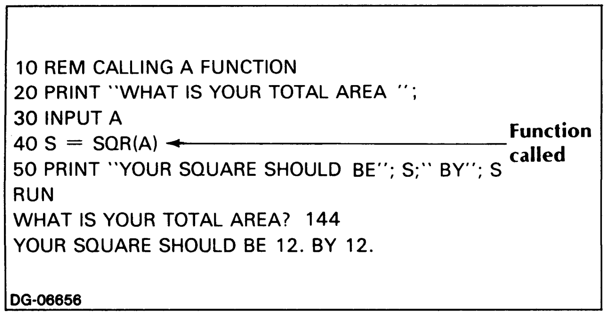
```
100 PRINT LEN(A$)
```

After you have invoked the function, the program will retrieve the function for execution and transfer its result (the value) into your program. Thereafter, it will resume ordinary execution according to normal line order.

We illustrate briefly with the SQR(X) function. This is a mathematical function, but the process is the same for string functions.

Our sample program in Figure 8-2 asks you to INPUT a total area, A, after which it calls the SQR(A) function to give you the dimension of one side of a square capable of containing that area. We show the output immediately following the program.

You can call a function any number of times within a program.



```
10 REM CALLING A FUNCTION
20 PRINT "WHAT IS YOUR TOTAL AREA ";
30 INPUT A
40 S = SQR(A)
50 PRINT "YOUR SQUARE SHOULD BE"; S;" BY"; S
RUN
WHAT IS YOUR TOTAL AREA? 144
YOUR SQUARE SHOULD BE 12. BY 12.
```

Function called
DG-06656

Figure 8-2. Calling a Function

Order of Execution

The parts of a function are executed in the following order:

1. Operations within the argument are taken care of.
2. The function is evaluated.
3. All other arithmetic operations in the statement are done in their normal order of precedence. (See Chapter 2.)

Table 8-1. Predefined Mathematical Functions

Function	Function Value
ABS(X)	The absolute value of X.
ATN(X)	The arctangent of X in radians, that is, the angle whose tangent is X. The range of the function is -(pi/2)<ATN(X)<(pi/2) where pi is the ratio of the circumference of a circle to its diameter.
COS(X)	The cosine of X, where X is in radians.
*DATE	The current date in decimal form YYDDD, where YY represents the last two digits of the year and DDD is the number of days elapsed in the year; for example, the value of DATE on May 9, 1977 was 77129. See also the string function DATE$.
*DEG(X)	The number of degrees in X radians.
EXP(X)	The exponential of X, that is, the value of the base of natural logarithms (e=2.71828...) raised to the power X; if EXP(X) is less than machine infinitesimal, then its value shall be replaced by zero.
*FP(X)	The fractional part of X: X - IP(X); for example, X = 1.345, FP(X) = .345.
INT(X)	The largest integer not greater than X; for example, INT(1.3) = 1 and INT(-1.3) = -2.
*IP(X)	The integer part of X: X - FP(X); for example, X = 1.345, IP(X) = 1.
LOG(X)	The natural logarithm of X; X must be greater than zero.
*MOD(X,Y)	Modulo function. X-Y * INT(X/Y) if Y is nonzero; generates an error message if Y is zero.
*PI	The constant 3.14159... which is the ratio of the circumference of a circle to its diameter.
*RAD(X)	The number of radians in X degrees.
*REM(X,Y)	Remainder of the division X/Y. X-Y * IP(X/Y) if Y is nonzero; generates an error message if Y is zero.
RND	The next pseudorandom number in an implementation-supplied sequence of pseudorandom numbers uniformly distributed in the range 0<=RND<1.
SGN(X)	The algebraic "sign" of X:-1 if X<0, 0 if X=0, and +1 if X>0.
SIN(X)	The sine of X, where X is in radians.
SQR(X)	The nonnegative square root of X; X must be nonnegative.
TAN(X)	The tangent of X, where X is in radians.
*TIME	The time elpased since the previous midnight, expressed in seconds; for example, the value of TIME at 11:15 AM is 40500. See also the string function TIME$.

*MP/BASIC extensions to ANSI standard functions.

Mathematical Functions

Table 8-1 summarizes the mathematical functions available in MP/BASIC.

Note that the variable X appearing in this table as an argument for each function is a "dummy" variable. That is, when you invoke any particular function, you may replace variable X by any other variable, constant, or expression (as previously illustrated) provided always that you have already defined these in your program.

Most of the functions listed in Table 8.1 are well known and well defined, except for the RND function. Further explanation of this function is supplied in the next two sections.

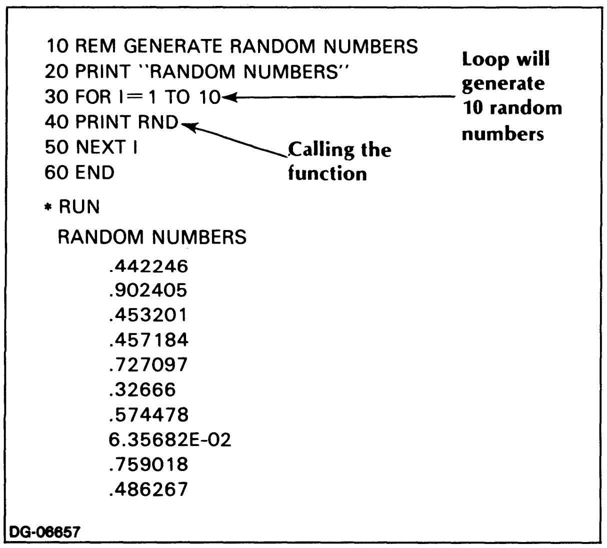
RND: The Random Number Function:

Each time it is invoked, this function produces a random number greater than or equal to zero, but smaller than one.

Observe that this function does not take an argument.

The RND function will produce an identical sequence of random numbers each time you run a given program. But this predictability can be eliminated by means of the RANDOMIZE command. (See the following section.)

As Figure 8-3 shows, the RND function (which we invoked within a loop) generated 10 random decimals.



```
10 REM GENERATE RANDOM NUMBERS
20 PRINT "RANDOM NUMBERS"
30 FOR I=1 TO 10
40 PRINT RND
50 NEXT I
60 END
* RUN
 RANDOM NUMBERS
     .442246
     .902405
     .453201
     .457184
     .727097
     .32666
     .574478
     6.35682E-02
     .759018
     .486267
```

Loop will generate 10 random numbers
Calling the function
DG-06657

Figure 8-3. The RND Function

You are, however, by no means limited to that particular range; you can obtain random numbers within any range of your choice.

Thus, assuming you wish a random number in the range A to B, the general formula you would use is

(B - A) * RND + A

Hence, if A = 1 and B = 6, you would write

(5 * RND + 1)

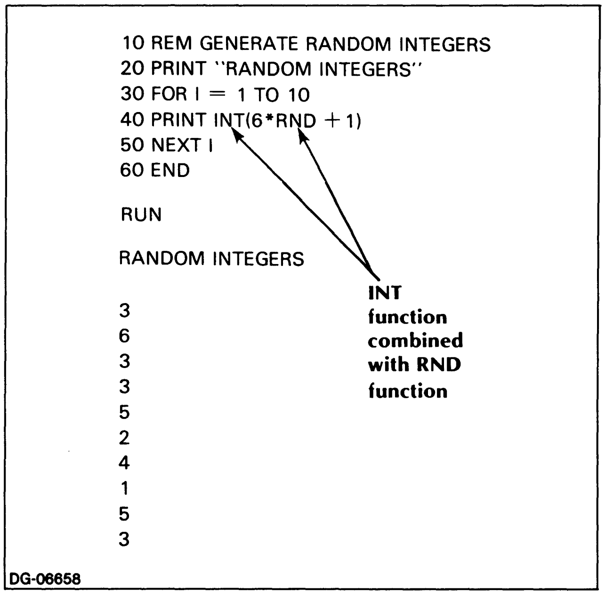
to get random numbers from 1 to (but not including) 6.

You have equal latitude in terms of determining whether your random numbers should be decimals or integers.

Should you wish random integers, you would combine the RND function with the INT function. Thus, in order to generate a random integer between A and B, use the formula:

INT((B - A + 1) * RND) + A

To illustrate, the program in Figure 8-4 below generates 10 random integers ranging from 1 through 6.



```
10 REM GENERATE RANDOM INTEGERS
20 PRINT "RANDOM INTEGERS"
30 FOR I = 1 TO 10
40 PRINT INT(6*RND +1)
50 NEXT I
60 END

RUN

RANDOM INTEGERS

3
6
3
3
5
2
4
1
5
3
```

INT function combined with RND function
DG-06658

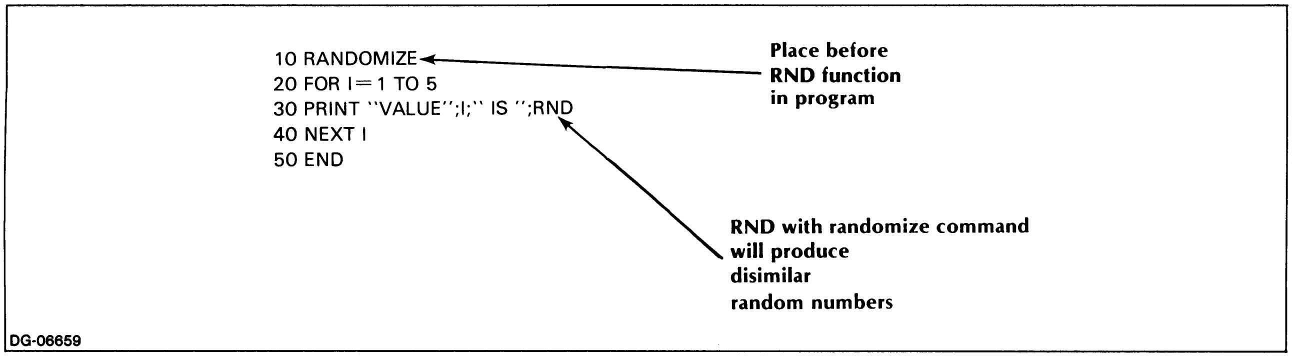
Figure 8-4. Generating Random Integers

RANDOMIZE

As we mentioned earlier, the RND function will produce an identical sequence of numbers every time you run a given program.

This predictability is an advantage while you are trying to debug (correct errors in) your programs, but it can be a distinct liability otherwise.

The RANDOMIZE statement allows you to bypass the predefined sequence of random numbers and to generate new and unpredictable sequences each time you execute a given program.



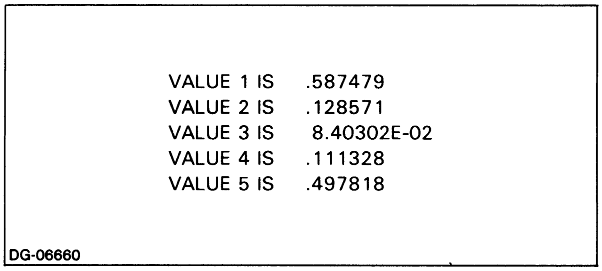
```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT "VALUE";I;" IS ";RND
40 NEXT I
50 END
```

Figure 8-5. Effect of RANDOMIZE Command

To use this facility, you merely type the command RANDOMIZE (preceded, of course, by a line number if you want it executed as a statement in your program). Place this command so that it precedes your first use of the RND function in the program.

The program in Figure 8-5 illustrates the effect of the RANDOMIZE command.

Figures 8-6 and 8-7 represent two separate runs of the above program. Note that each run produced a different sequence of random numbers.



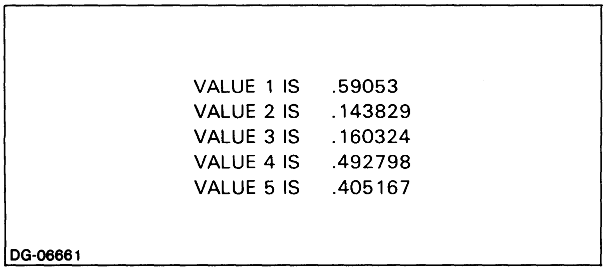
```
VALUE 1 IS   .587479
VALUE 2 IS   .128571
VALUE 3 IS    8.40302E-02
VALUE 4 IS   .111328
VALUE 5 IS   .497818
```

Figure 8-6. One Run of Random Number Program Using RANDOMIZE

Table 8-2. Predefined String Functions

Function	Function Value
*BSTR$(V,R)	The string representing the value of the binary number V in base (radix) R. Complements BVAL(B$,R).
*BVAL(B$,R)	The 16-bit unsigned binary value (in base R) of the number represented by the string B$.
*CHR$(M)	The string character for which M is the ASCII numeric representation. Complements ORD(A$).
*DATE$	The date, expressed as the string yy/mm/dd. See also the numeric function DATE.
*LEN(A$)	The number of characters in the string A$.
*ORD(A$)	The decimal number which is the ASCII code for the single character A$. Complements CHR$(M).
*POS(A$,B$)	The position of the first
*POS(A$,B$,M)	occurrence (if any) of string B$ within string A$. If B$ does not occur within A$, returns 0. If M is specified, the search begins at position M in A$ (characters in positions 1 through M-1 are ignored).
*STR$(X)	The string representation of the decimal value of expression x. Complements VAL(A$).
*TIME$	The time of day, as the string NN:MM:SS. See also the numeric function TIME.
*VAL(A$)	The value of the first block of numeric characters (if any) within the string A$. Complements STR$(X).

*MP/BASIC extensions to ANSI standard functions.



```
VALUE 1 IS   .59053
VALUE 2 IS   .143829
VALUE 3 IS   .160324
VALUE 4 IS   .492798
VALUE 5 IS   .405167
```

DG-06661

Figure 8-7. Another Run of Random Number program Using RANDOMIZE

String Functions

Table 8-2 summarizes the string functions available in MP/BASIC. These functions are described in the following sections. All MP/BASIC string functions are extensions to the ANSI standard.

*BSTR$(V,R)

This function returns a string which is the representation (in radix, or base, R) of the value of the 16-bit unsigned binary integer V. The binary number V is expressed in the function in its decimal representation; it is rounded to the nearest integer before the conversion is made. R can be any value from 2 through 16.

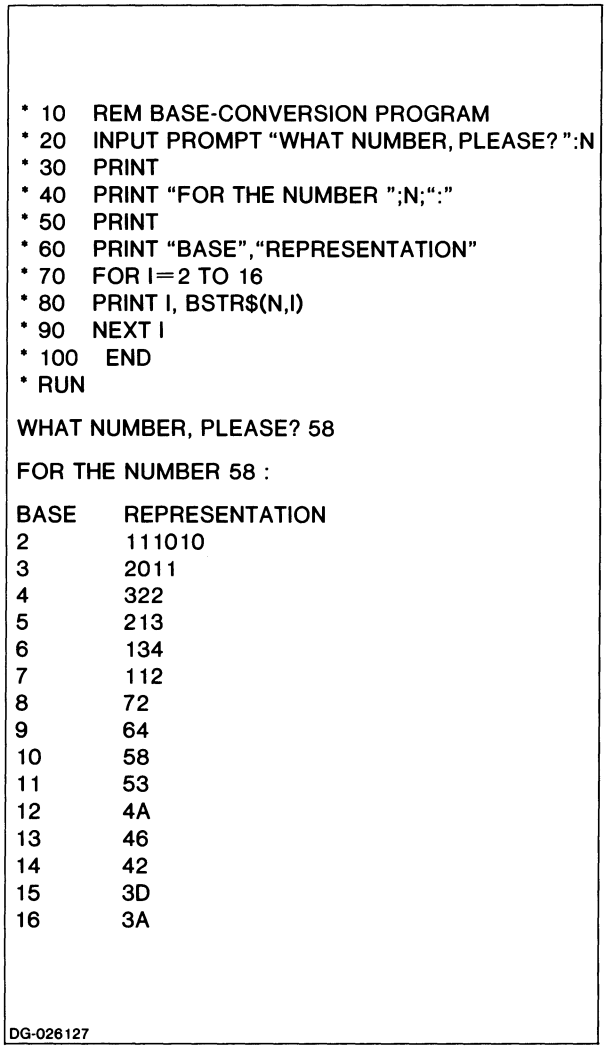
For example:

```
BSTR$(7,2) = "111"
BSTR$(12,6) = "15"
BSTR$(12.4,6) = "15"
```

This function complements the $BVAL function.

One application of the BSTR$ function is in manipulating bit-string patterns.

Another application of the BSTR$ function is in the conversion of numbers from decimal to some other base. Figure 8-8 shows a program which lists a decimal number's string representation in bases 2 (binary) through 16 (hexadecimal), along with a sample run.



```
* 10   REM BASE-CONVERSION PROGRAM
* 20   INPUT PROMPT "WHAT NUMBER, PLEASE? ":N
* 30   PRINT
* 40   PRINT "FOR THE NUMBER ";N;":"
* 50   PRINT
* 60   PRINT "BASE","REPRESENTATION"
* 70   FOR I=2 TO 16
* 80   PRINT I, BSTR$(N,I)
* 90   NEXT I
* 100  END
* RUN
WHAT NUMBER, PLEASE? 58

FOR THE NUMBER 58 :

BASE    REPRESENTATION
2       111010
3       2011
4       322
5       213
6       134
7       112
8       72
9       64
10      58
11      53
12      4A
13      46
14      42
15      3D
16      3A
```

DG-026127

Figure 8-8. BSTR$ Function

*BVAL(B$,R)

This function complements the BSTR$(V,R) function. B$ is the string representation of a number in radix (base) R. BVAL returns the 16-bit unsigned binary integer value of that number.

For example:

```
BVAL("110",2) = 110
BVAL("5B",16) = 0101 1011
BVAL("10",10) = 1010
```

NOTE: if you PRINT the value of a result of BVAL, it will look like a decimal number. This is because the PRINT command treats the binary number as a decimal INTEGER. For example:

```
PRINT BVAL("5B",16)
91
```

because 91 is the decimal equivalent of the binary number 0101 1011.

R can be any number from 2 through 16, and B$ must consist of the valid characters used in string representations of numbers in the base R. For example, if R=2 (binary numbers), B$ must consist of the characters 0 and 1; if R=8 (octal numbers), B$ must consist of the characters 0,1,....7; if R=16 (hexadecimal numbers), B$ must consist of the characters 0,1,...9,A,B,...F.

*CHR$(M)

Returns the string character for which M is the ASCII (decimal) numeric representation (see Figure 1-1). M must be a decimal number in the range 1 to 255. The ASCII decimal representations range from 0 through 127; if M is in the range 128 through 255, the character returned is the one represented by MOD(M,128).

For example:

```
CHR$(53) = "5"

CHR$(65) = "A"

CHR$(193) = "A"
```

The above function is complemented by the ORD(A$) function.

*DATE$

Indicates the date in the string representation YY/MM/DD. For example, the value of DATE$ on May 9, 1977 was 77/05/09.

*LEN(A$)

This function returns the length of string A$, i.e., the number of characters currently in that string. For example,

```
10 LET CCHAR$ = "CRAB"
.
.
.
.
80 PRINT LEN(CCHAR$)
90 END
RUN
4
```

Note the use of variable CCHAR$, with which we replaced the "dummy" variable A$ used in the section heading.

The LEN function will include in its count of string length non-alphabetic characters, such as spaces, punctuation, or any of the other characters defined as character strings in Table 3-1. Thus, LEN C$ would have resulted in a value of 5 if we had defined C$ as "CRAB?".

Among other things, the LEN(A$) function makes it possible for you to compare strings to each other in terms of their respective lengths.

Our sample program in Figure 8-9 uses this function, in combination with an IF...THEN... statement, to identify and print out a list of words of more than six characters from a sentence in the United States Constitution. The output appears immediately following the program.

*ORD(A$)

This function complements the CHR$(M) function. It converts a one-character string, A$, into its numeric decimal ASCII representation (see Figure 1-1). A$ may be any single ASCII character. For example:

```
ORD("A") = 65
ORD("a") = 97
ORD("5") = 53
```

*POS(A$,B$)

This function compares two strings. If it finds string B$ contained within string A$, it returns the position in A$ where B$ begins, i.e., the position in A$ of the first character of B$. If A$ does not contain B$, the function returns 0. For example:

```
A$ = "PANCAKE"
B$ = "CAKE"
POS(A$, B$) = 4
```

Only the first occurrence of B$ within A$ is returned; later occurrences are ignored. For example:

```
A$ = "BOBOLINK"
B$ = "BO"
POS(A$,B$) = 1
```

If B$ is a null string, the function returns a value of 1. For example:

```
A$ = "SUPER"
B$ = " "
POS(A$,B$) = 1
```

*POS(A$,B$,M)

This function searches for string B$ within A$, beginning from position M in A$. It returns zero if it does not find B$ in A$. For example:

```
A$ = "GRANDSTANDING"
B$ = "AND"
POS(A$,B$,5) = 8
POS(A$,B$,1) = 3
POS(A$,B$,9) = 0
```

If M is out of range of the length of A$, the function will return 0.

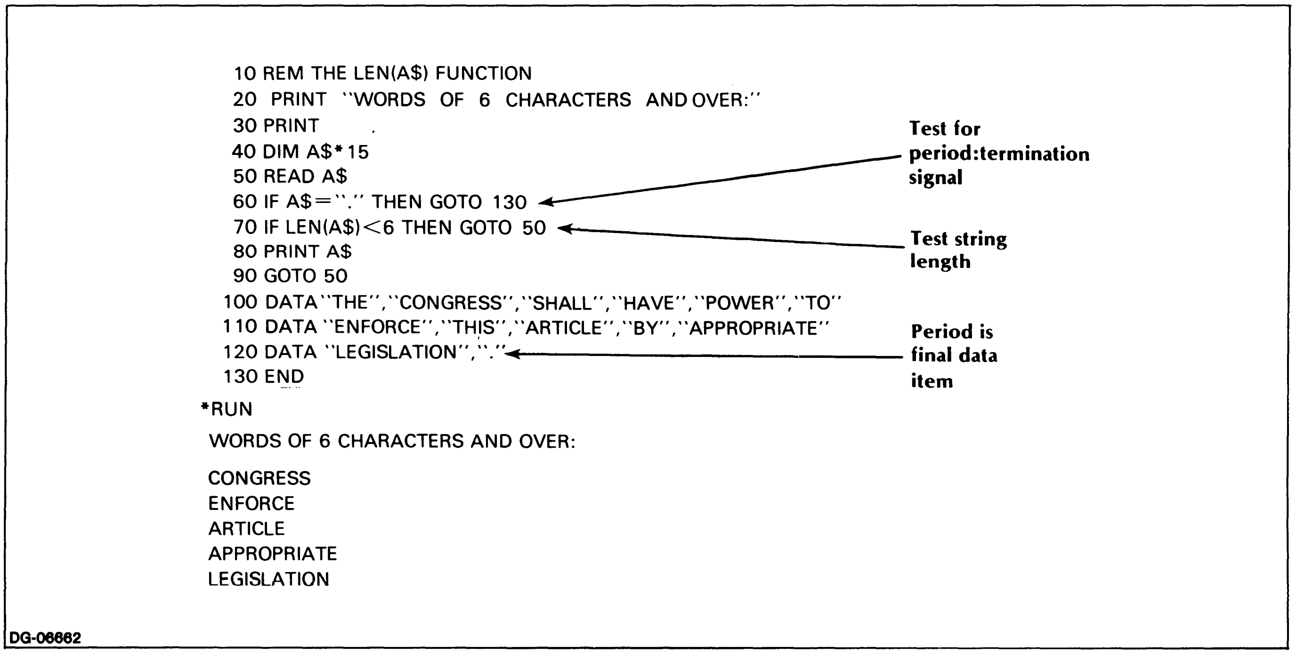
*STR$(X)

This function complements the VAL(A$) function: it converts the numeric value of an expression to a string. For example,

```
STR$(5+9) = "14"
```

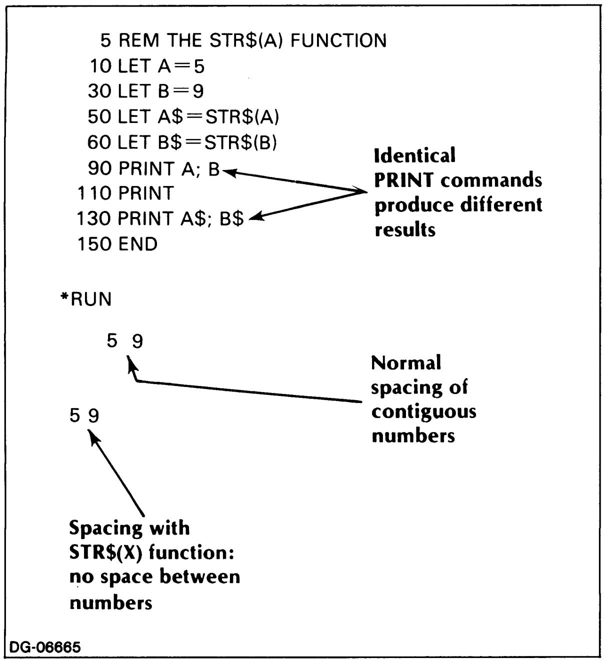
Ordinarily, BASIC leaves a space before and after each number it writes out. These spaces are not included when you invoke the STR$(X) function, as we see in the program and accompanying output of Figure 8-10.

This function is thus particularly useful for generating tightly formatted output.



```
 10 REM THE LEN(A$) FUNCTION
 20 PRINT "WORDS OF 6 CHARACTERS AND OVER:"
 30 PRINT
 40 DIM A$*15
 50 READ A$
 60 IF A$="." THEN GOTO 130
 70 IF LEN(A$)<6 THEN GOTO 50
 80 PRINT A$
 90 GOTO 50
100 DATA"THE","CONGRESS","SHALL","HAVE","POWER","TO"
110 DATA "ENFORCE","THIS","ARTICLE","BY","APPROPRIATE"
120 DATA "LEGISLATION","."
130 END
*RUN
 WORDS OF 6 CHARACTERS AND OVER:

 CONGRESS
 ENFORCE
 ARTICLE
 APPROPRIATE
 LEGISLATION
```

Figure 8-9. LEN(AS) Function



```
   5 REM THE STR$(A) FUNCTION
  10 LET A=5
  30 LET B=9
  50 LET A$=STR$(A)
  60 LET B$=STR$(B)
  90 PRINT A; B
 110 PRINT
 130 PRINT A$; B$
 150 END

*RUN

   5  9

 5 9
```

Identical PRINT commands produce different results
Normal spacing of contiguous numbers
Spacing with STR$(X) function: no space between numbers
DG-06665

Figure 8-10. STR$(X) Function

*TIME$

Returns the time of day in twenty-four hour notation, in the representation HH:MM:SS. For example, the value of TIME$ at 11:15 AM is 11:15:00.

*VAL(A$)

You can use this function to extract the numeric portion of a character string and convert it to numeric data.

Your string argument must begin with a number, which may include leading plus or minus signs, digits, decimal points, or the letter E for scientific notation.

Nonnumeric characters appearing after the number portion of the string are ignored by this function. For example, a string such as

"123ABC"

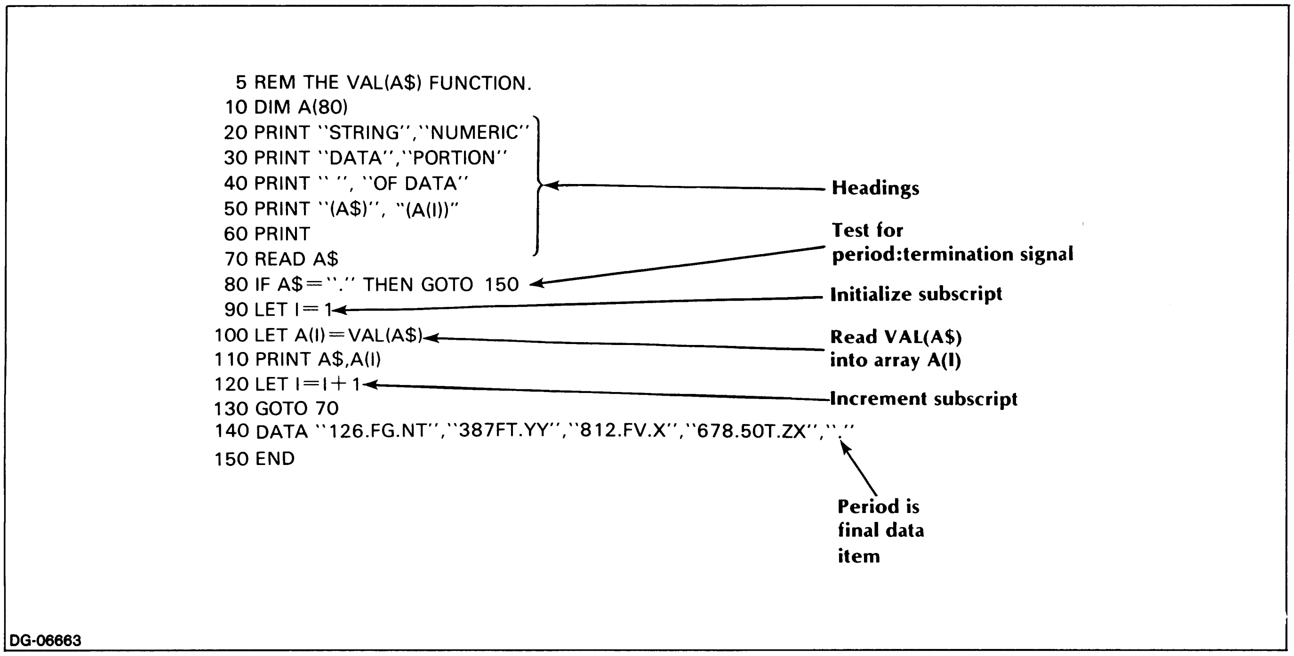
would be read as "123".

If the evaluation of the string's numeric portion results in an underflow (i.e., a number too small to be represented), the value returned will be 0. For example,

VAL("2.E-99")

is zero.

If A$ does not contain a numeric value, an error is generated. For example,



```
  5 REM THE VAL(A$) FUNCTION.
 10 DIM A(80)
 20 PRINT "STRING","NUMERIC"
 30 PRINT "DATA","PORTION"
 40 PRINT " ", "OF DATA"
 50 PRINT "(A$)", "(A(I))"
 60 PRINT
 70 READ A$
 80 IF A$="." THEN GOTO 150
 90 LET I=1
100 LET A(I)=VAL(A$)
110 PRINT A$,A(I)
120 LET I=I+1
130 GOTO 70
140 DATA "126.FG.NT","387FT.YY","812.FV.X","678.50T.ZX","."
150 END
```

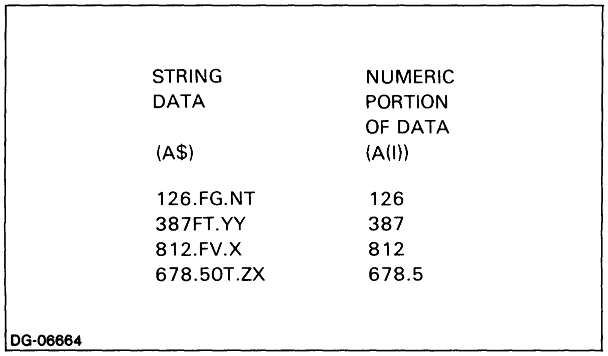
Headings
Test for period:termination signal
Initialize subscript
Read VAL(A$) into array A(I)
Increment subscript
Period is final data item
DG-06663

Figure 8-11. VAL(A$) Function

```
A$ = "CVS"
; VAL(A$)
```
Illegal data type.

VAL(A$) can be used in situations where INPUT data may be either alphabetic or numeric. These values may be assigned to a string variable the user can examine; if the string is a number, it can then be converted by means of the VAL(A$) function.

The sample program in Figure 8-11 scans a series of strings, A$, and stores only their numeric portion, which it extracts, converts to numeric data, and reads into array A(I). Figure 8-12 shows the output of this program.



```
STRING          NUMERIC
DATA            PORTION
                OF DATA
(A$)            (A(I))

126.FG.NT       126
387FT.YY        387
812.FV.X        812
678.50T.ZX      678.5
```

Figure 8-12. Output from the VAL(A$) Function

The VAL (A$) function complements the STR$(X) function.

User-Defined Functions

In addition to providing the predefined functions we have discussed, BASIC also allows you to define and call your own functions within a program.

When creating a user-defined function, you can use only one statement written on a single line. Functions of this type return a single value.

Parts of a User-Defined Function

Figure 8-13 demonstrates the parts of a user-defined function.

DEF

This command indicates that the statement following it contains a function definition.

Since a function must be defined before it can be called for execution, ANSI requires the DEF statement to appear in your program on a lower-numbered line than any statements calling the function it defines. In MP/BASIC, however, the only constraint is that program execution should pass through the DEF statement before you call the function.

Once you have defined a function, you can redefine it later in your program.

When the program reaches a line containing a DEF statement, it does not execute the function just defined but proceeds instead to the next line without any further effect. The function is executed only when you call it.

In your function definition you may refer to other functions, provided they have already been defined; you may not, however, refer to the function you are currently defining.

For example,

```
10 DEF FNF(K) = K*FNF(K-1)
```

is not permissible in BASIC.

FNx

The name of the function comes immediately after the word DEF. All function names for user-defined functions must begin with the letters FN, followed by a single letter.

It follows that you can define a maximum of 26 functions (FNa...FNz) in your BASIC programs.

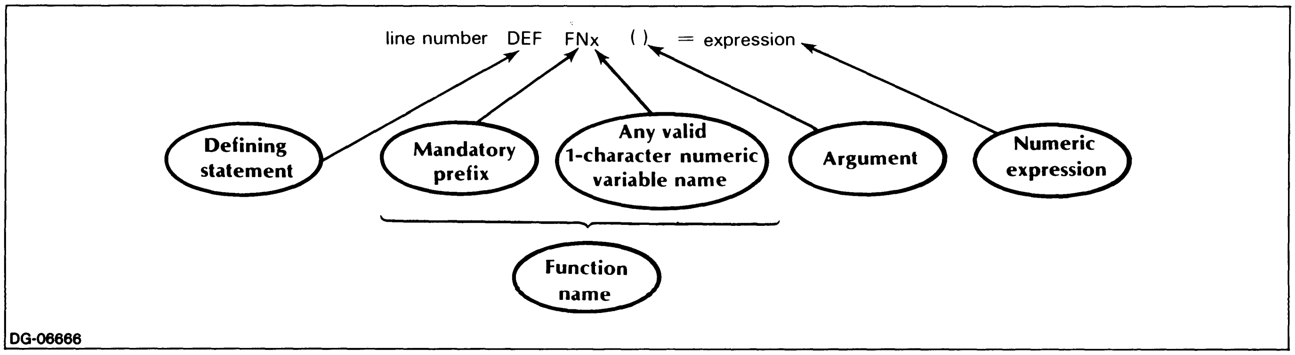


Figure 8-13. Parts of a User-Defined Function

Argument

User-defined functions are limited to a single optional argument.

If used, the argument follows the function name and is always enclosed in parentheses ( ).

Your function call must correspond to your function definition with respect to argument use: don't omit the argument from your call if you used it in the DEF statement, and don't call a function with an argument if your definition did not use one.

The variable in the argument is, once again, a dummy variable; when you call the function, it will be evaluated with the value of the argument you list in the call, rather than with the argument listed in the DEF statement.

In the example below, we define FNA with dummy argument X, but in line 30 we call the function with the actual argument we want to use, namely, K.

```
10 DEF FNA(X) = (40 - X)/(10 - X)
20 FOR K = 0 TO 5
30 PRINT FNA(K)
40 NEXT K
50 END
```

```
RUN
4
4.33333
4.75
5.28571
6
7
```

You should also keep in mind the fact that the variable you use as an argument in your DEF statement is local to that definition; that is, the variable X we just used in our example has no relationship whatsoever to any variable X we might have used elsewhere in the body of our program.

Expression

The arithmetic expression following the equal sign defines the calculation to be performed by the function we have defined. It may contain the argument and, if it does, the value of the argument is local to the function, as we have just explained.

The expression may also contain variables other than the argument. These variables are global; that is, they are identical to program variables of the same name.

For example, the variables A and B in the DEF statement below are global values, identical to the program variables A and B defined in lines 20 and 30 of the same program.

```
10 DEF FNC(X) = (A-X)/(B-X)
20 LET A = 40
30 LET B = 10
40 FOR K = 0 TO 5
50 PRINT FNC(K)
60 NEXT K
70 END
```

```
RUN
4
4.33333
4.75
5.28571
6
7
```

Referencing a User-Defined Function

Once you have defined a function, you can call it as often as you wish throughout the program, in the same way that you would call system-defined functions.

The statement in which you call a user-defined function can contain that function name combined with numbers, variables, other functions, or mathematical expressions. See, for example, line 30 below.

```
10 DEF FNA(X) = X/(1-X)
20 FOR N = 2 TO 5
30 Z = 20*FNA(1/N)*EXP(-N/5) 40 PRINT Z
45 NEXT N
50 END
```

```
RUN
13.4064
5.48812
2.99553
1.8394
```

Keywords in Chapter 8

Table 8-1 lists the keywords introduced in Chapter 8.

Table 8-3. Keywords in Chapter 8

Keyword	Can be used in	
	Program Statement	Immediate Mode
DEF	Yes	No
RANDOM-IZE	Yes	Yes

End of Chapter

Chapter 9
File Input and Output

Up to this time, you have been saving your programs, but you have been unable to save the data you processed in these programs.

If, for example, you wanted to retrieve the elements of an array you created in an earlier session, you first had to recreate that array by re-executing your original program. You could save the program, but you were unable to save the array itself.

Computer files enable you to save data and retrieve them as needed. File capability is, therefore, one of the significant special resources made available by MP/BASIC as an extension of ANSI Minimal BASIC.

A file is a collection of related data kept in one place and treated as a unit; it is like a drawer subdivided into compartments, each of which is a storage location for information. MP/BASIC files automatically accommodate their length to the amount of data we want to store in them, expanding or telescoping themselves as needed.

The operations you can perform with a file include creating it, writing into it, updating, reading, editing, printing, and deleting it.

While working with files is not difficult, it does involve a number of new interrelated concepts, such as records, modes, and random versus sequential file access.

Records are special ways of grouping and accessing file data. Record structures constitute one of the most powerful aspects of files, and they are our first topic in this chapter.

Next we take up the subject of file modes, which determine what operations can be performed with a given file.

Following that, we discuss file commands, presenting them within the context of program examples. This constitutes the major part of the chapter.

Records

A record is a single data item or several data items that together constitute a separate unit within a file.

For example, one record in a file containing a list of a company's employees might be composed of a given employee's name, address, telephone number, social security number, and name of next of kin (five different data items identifiable as a single unit).

Or, as in our example in Figure 9-1, a single record in a file of yearly fuel expenses might contain the name of the month, number of gallons of fuel used, and the price per gallon (three different data items).

You can treat any number of data as a single record, depending on the way you define record size.

And since each record has a distinct location within the file (analogous to the first, second, third,... compartment inside a drawer), you can easily reach a particular record by reference to its location. We will elaborate on this later, in our discussion of random access files.

(The experienced user can redefine the composition of his or her file records at will by using diverse record declarations in processing a given file. We will, however, confine our discussion to less complex levels of record use.)

Please note that the first record location in a file is 0. That is, we store our first record of data in file compartment number 0 rather than in file compartment number 1. (See Figure 9-1.)

Figure 9-1 is an example of a hypothetical fuel file containing two records, one stored in record location #0 and the other stored in record location #1. Each record consists of three pieces of information, namely, month, gallons of fuel, and price per gallon.

Notice that our record contains a mixture of character strings and numbers. These are dissimilar data types ordinarily requiring separate storage. An array, for example, can be composed only of either numbers or strings (but not a mixture of both); similarly, it would be incorrect to use a numeric variable for processing a string variable, and vice versa. Records thus offer great flexibility by permitting meaningful data combinations and storage, regardless of data type.

There are two major types of file records:

1. Fixed-length records
2. Variable-length records

That is, the size of the file subdivisions to which your records correspond may be of either uniform (fixed) length, much like the uniform drawers in a file cabinet, or variable length, something like the diverse sizes of compartments in a tool box.

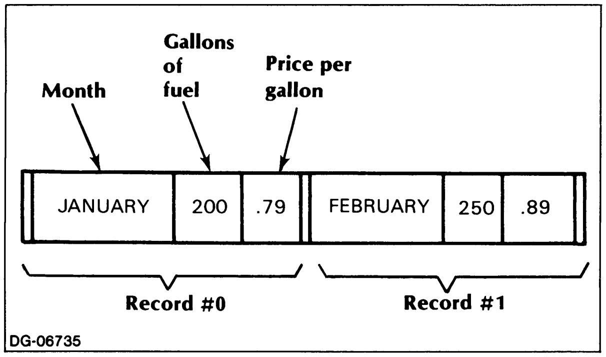


Figure 9-1. Fuel File with Two Records

Your choice of record type is determined by your anticipated use of the file. (We will elaborate on this later.) In any case, as we have said, record types do not remain set over the life of the file, since it is possible to process a file assuming different record characteristics for it each time around.

When we speak of a record's length, or of the length of data within a record, we mean the amount of space being occupied by that record inside the file. We measure this amount of space in terms of the number of bytes used. (A byte is a unit of information occupying space in computer memory).

Each record in a file can be from 0 to 32,767 bytes long. See "Calculating Record Length" for further details.

Fixed-Length Records

Figure 9-2 shows the same two records we used in Figure 9-1.

Looking at these records, we notice that there seem to be empty spaces at the end of each, and that these empty spaces have been padded with nulls (four and three nulls, respectively).

As a result of this padding, each of our two records occupies exactly the same amount of space (20 bytes) in the file, despite the fact that the actual data in them varies in length (16 and 17 bytes, respectively). This is what is meant by fixed-length records.

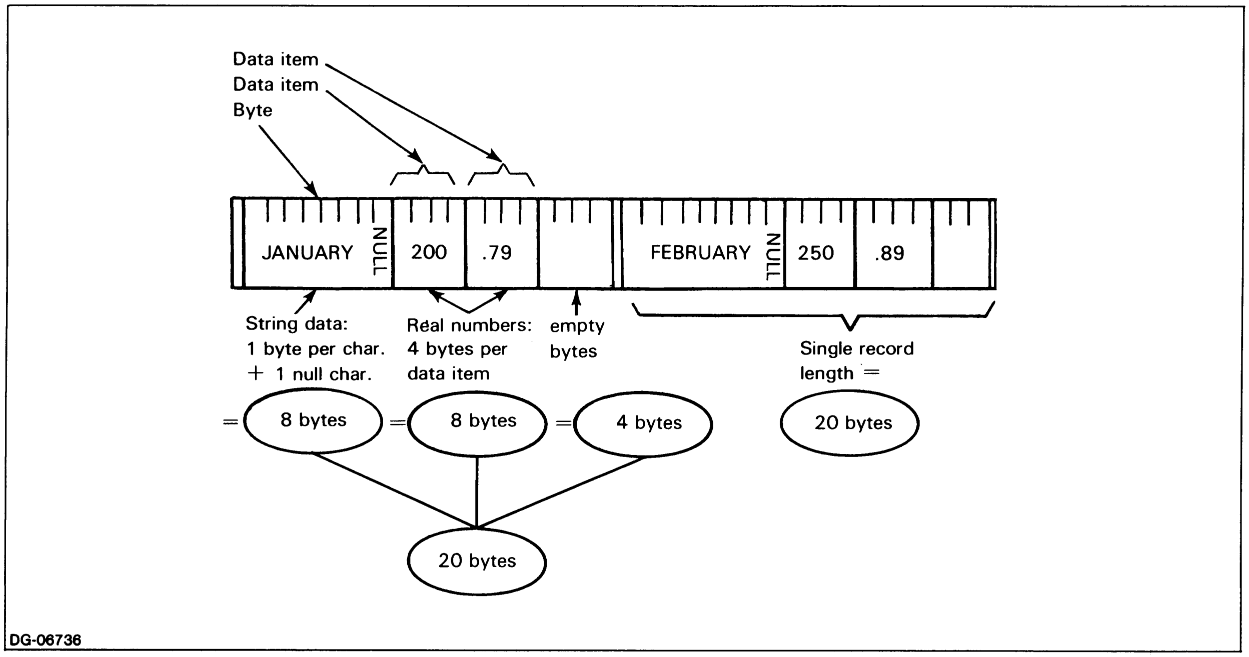


Figure 9-2. Fixed-Length Records

Calculating Record Length

Figure 9-3 illustrates how we determine the number of bytes a given record requires.

- String data occupy one byte per character, plus one byte for a null character serving as a delimiter at the end of the string.

  Hence, our first data item (January) will occupy a total of eight bytes (seven for the seven characters in the string, plus one for the null delimiter).

- Integers occupy two bytes per data item. For now, we are working only with real numbers. (See Chapter 2 for converting integers to real numbers and vice versa.)

- Real numbers occupy either four bytes per data item (for single-precision numbers) or eight bytes per data item (for double-precision numbers).

  Recall that real numbers differ from integers in having fractional parts.

  In the absence of any declaration to the contrary, MP/BASIC treats all numbers as though they were single-precision real rather than integer or double-precision real numbers. 200 (number of gallons), and .79 (price per gallon) in our example are both treated as single-precision real numbers occupying four bytes each.

Each element of an array occupies the number of bytes that corresponds to the type of data in the array. For example, an array of integers occupies two bytes per element. In calculating the space occupied by an entire array, recall that arrays almost always begin with element 0, unless the OPTION BASE statement has been specified.

Applying these byte requirements, we can now determine the size of the first record in Figure 9-3 (Record # 0) as follows:

January	8 bytes
200	4 bytes
.79	4 bytes

Total bytes in Record # 0:   16 bytes

If we define the fixed length of our records to be 20 bytes, there will be four bytes not used up by data in Record number 0.

Try calculating the number of bytes needed for the next record (1) in Figure 9-3, and see if your figures correspond to ours.

Creating a Fixed-Length Record

You declare a file as containing fixed-length records by specifying the length of your record in your OPEN FILE statement (to be discussed later). The number indicating record length is the last item in that statement. For example:

```
10 OPEN FILE (1,1), "FUEL", 20
```

You have now created a file number 1 named "FUEL," with a fixed record length of 20 bytes. The records illustrated in Figure 9-2 would have been created by means of a statement similar to the one above.

To use the fixed-length record capability, you must write or read your data by reference to a specific record number. This will indicate to BASIC at what exact byte number in the file it should begin writing or reading. (See "Accessing Records.")

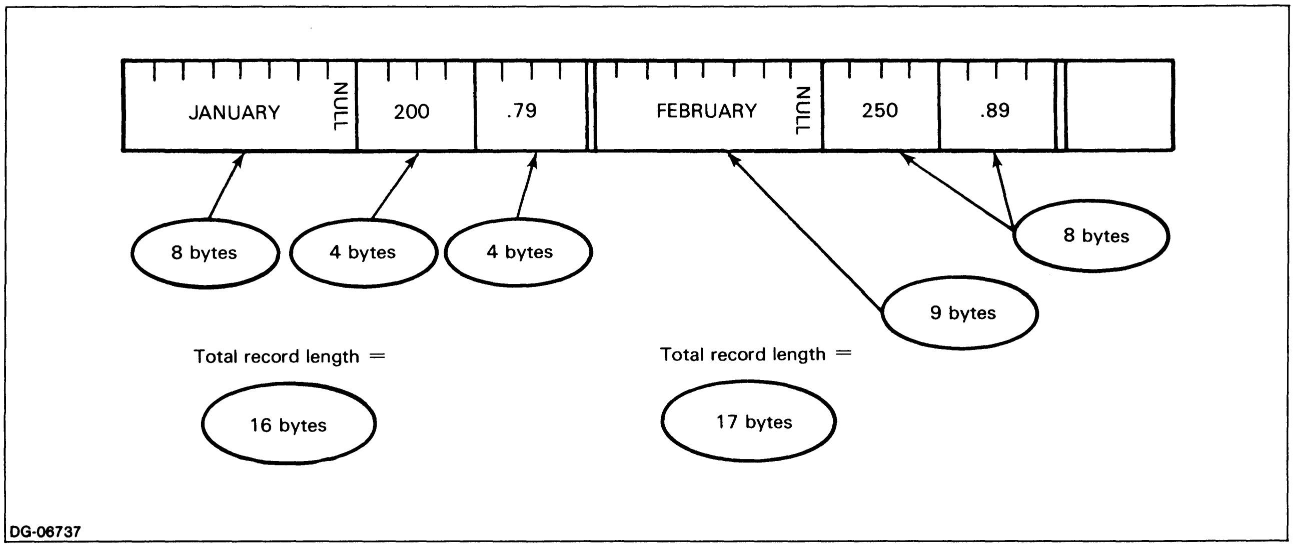


Figure 9-3. Calculating Record Length

If your processing instructions (such as WRITE or READ) don't specify a record number, the system will disregard your fixed-length record declaration and process the data sequentially in variable-length records.

Variable-Length Records

In Figure 9-3, we used the same two records as in Figure 9-2. But in Figure 9-3 the records differed in size, since they contained only their respective data items without any empty spaces left over.

Thus, in variable-length records, the space allocated for each record corresponds exactly to the amount of space needed for the data that particular record holds.

It follows that records of variable length offer a more efficient use of space. On the other hand, a file containing this type of record can only be accessed sequentially: to reach any record within it, you must move past all preceding records starting from the first (record 0), until you reach the place you want. (See "Accessing Records.")

Figures 9-5 through 9-11 illustrate variable-length, sequential-access file operations.

Accessing Records: Random versus Sequential Access

As already stated, the great advantage of working with fixed-length records is that they allow you go directly to any record within the file.

Sequential access is a somewhat more cumbersome process if you wish to reach only specific records rather than the entire contents of a file. To understand how this works, we need to explain the operation of a device called the file pointer.

The File Pointer

Although we do not see it, the file pointer is a mobile device that indicates our current location within a file, much as one would use a finger to keep track of one's place on a page.

When you open a file, the pointer is at the first compartment, or subdivision, of the file; as soon as you begin moving into other compartments for purposes of writing into the file or reading from it, the pointer moves right along with you to the compartment you are currently working on. When you finish working and close your file, the pointer ceases to exist.

When you specify a record number for reading or writing, the file pointer moves directly to the appropriate byte at the start of that record. For example, if you've declared record length as 10 bytes, and you instruct BASIC to read record number five, then pointer would move directly to byte number 50 in the file and read bytes 50 to 59. If your next instruction specifies no record number, reading will proceed sequentially from the current location of the pointer at byte 60, and so on, until you specify another record number.

In this way, record numbers create pointer addresses, enabling us to move back and forth to specific file locations (random access).

You should be careful not to write a record that is longer than the space allotted for it in the file. Consider for example the following case:

- You declare a record length of 10 bytes.
- You write into record number 3 (i.e., beginning at byte 30) a data item occupying 15 bytes.
- Bytes 30–44 are now filled; your pointer is at the end of byte 44.
- If you now instruct BASIC to write a 10-byte item into record number 4 (i.e., beginning at byte 40) the pointer will move to that location and will overwrite the data presently in bytes 40–44.

The best safeguard here is to make adequate provision for the maximum number of bytes your records are likely to contain, and to monitor your processing instructions so that you know where you have placed your data.

Processing a file sequentially rather than by reference to specific record locations requires using program instructions to move the file pointer in sequence along all records in the file from record number 0 to record number 2, and so on, until it reaches the location you want.

Hence, in spite of an inevitable waste of space (which you can, however, minimize by accurate calculation of your record length), fixed-length records are your best choice if you anticipate many random-access operations, and if you know the exact location of your data in the file.

See Figures 9-12 through 9-14 and accompanying discussion for further illustrations of the concept of random-access files.

File Modes

We must declare a file mode as part of the statement creating a file.

There are four possible file modes, and they are expressed by numbers ranging from 0 to 3. Each of these modes serves to identify the specific function of a file. All four file modes permit random access, provided a fixed record length has been declared in the OPEN FILE statement.

Table 9-1. summarizes the modes and their functions.

File Mode	Operations Possible		
	Write	Write/ Append	Read
0	Yes	No	Yes
1	Yes	No	No
2	No	Yes	No
3	No	No	Yes

Mode 0

As Table 9-1 shows, Mode 0 is the most versatile of the four modes. When you open a file in Mode 0, you can write into it or read from it, whereas in Modes 1, 2, and 3 you are limited to only one of these operations. (Modes 1 and 2 allow only writing, while Mode 3 allows only reading.)

When you open a file in Mode 0, the system checks whether the file specified already exists. If the file does not exist the system creates it.

Mode 1

Mode 1 is, as we have said, limited to writing operations.

When you open a file in Mode 1, the system checks whether this file already exists. If it finds the file, it deletes it, and then creates a new (empty) file to be written into. You should therefore use Mode 1 only to create a new file, never to reopen an old one.

Mode 2: The Write/Append Facility

Mode 2 is also limited to writing operations. This mode is normally used to reopen a file you have previously created, since its main purpose is to allow you to add to existing data. If, however, the file you are opening does not already exist, a new file will be created.

When you open or reopen a file in Mode 2, the file pointer moves automatically to the end of the file, thus enabling you to append to existing data. This Write/Append feature is unique to Mode 2. Without it, you would have to go through the process of moving the file pointer to the first empty location before writing additional data.

Mode 3

Mode 3 files can be used only for reading; consequently, this mode will only allow you to open a previously created file. If you try to create a new file for reading, you will generate an error message stating

* *File does not exist in line ...*

where *line* ... is the line number of your erroneous OPEN FILE command.

It is entirely possible to open a file in Mode 3 using a record type different from the record type of the original file; that is, you can write a file with one record type, and read it with another record type. But the inexperienced user is advised to be consistent in the use of record types: if your original file in Modes 0, 1 or 2 processed the data by reference to fixed-length record locations, your Mode 3 file should be opened declaring the same record length, and read by means of references to specific record locations. If you reopen the file with a different record length, the individual record locations and boundaries will not match up, and you will not be able to read the data correctly.

File Commands

File commands are the program instructions that enable you to create new files and to write to and to read from your files. The eight file commands available in MP/BASIC and covered in this chapter are (in order of discussion) OPEN FILE, WRITE FILE, CLOSE FILE, READ FILE, DELETE , PRINT FILE, INPUT FILE, and LINPUT FILE.

Before we discuss these commands, we must introduce two file attributes that allow the system to identify your files. These are file number and filename.

File Number

Whenever you open a file, you assign it an identifying number. This number may range from 0 to 7. You can have no more than eight files open at any given time.

In an OPEN FILE statement, the file number is always the first number inside the parentheses following the word FILE. (The second number refers to the file mode.) For example,

```
10 OPEN FILE (1,0),...
```

See also Figure 9-4.

The format of all file commands requires that a file number be included in any subsequent references to an opened file, so the system can identify it correctly.

Filename

For identifying purposes, each of the files you create must have a name as well as a number.

Filename Format

As you see from Figure 9-4, the filename appears within quotation marks immediately after the file number and the file mode.

The filename is preceded by a comma. In our example, the filename is followed by another item (a number indicating record size), and thus has a comma after it as well as preceding it. When the filename is the last item on the OPEN FILE command line, however, no comma follows it. For example,

```
10 OPEN FILE (1,1), "INVENTORY"
```

Filename Length

Maximum filename length depends on the operating system. Filenames may be a maximum of 15 characters on MP/OS or MP/AOS, and a maximum of 31 characters on AOS.

Legal Filenames

What follows is a quick review of material discussed in Chapter 1.

You can create filenames from any combination of the alphanumeric characters (character strings and numbers) listed below:

- A through Z
- a through z (converted to uppercase by the operating system)
- 0 through 9
- . (period)
- $ (dollar sign)
- _ (underscore)

Filename Extensions

Since BASIC does not recognize any special alphanumeric extensions (suffixes), you can create filename extensions to suit your own needs. For example, later on we will create some files of fuel expenses for 1978 and 1979. We will call each file FUEL and use the extension .78 or .79 to indicate the year it contains.

"FUEL.78" (Fuel file for 1978)

"FUEL.79" (Fuel file for 1979)

In this way you can use a file extension as a quick index to file content.

The following are examples of legal filenames:

FILE_NAME.NEW

FILE$.SR

LIFE.JAN.5

The file we have created in Figure 9-4 has been assigned the number 1 and named LIBRARY. OPEN FILE

The OPEN FILE command is the first step in any file transaction. By correlating a filename with a file number, this command creates new files, or reopens previously created files for further processing.

Figure 9-4 illustrates the format of the OPEN FILE command.

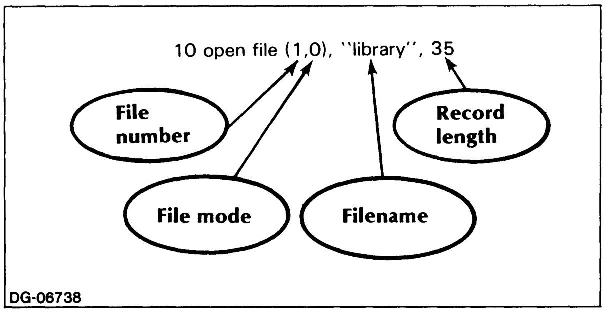


Figure 9-4. OPEN FILE Command

Observe that four items follow the words OPEN FILE. Each of these items is a specific file attribute, serving either to identify the file (file number and filename), or to define its type and function (record length and file mode).

The order in which each item appears on the command line is extremely important, i.e.:

file number

file mode

filename

record length (optional)

Being already familiar with all four of these file attributes, you will be able to interpret Figure 9-4 as follows:

File number =	1
File mode =	0 (You may write into the file, or read from it.)
Filename =	"Library"
Record type =	Fixed-length record of 35 bytes; random access permitted.

The format of the OPEN FILE command is the same whether you are creating a new file or reopening an old one. Recall, however, that if you use Mode 1 to reopen an existing file for writing, you will lose its contents, since the system will delete the file before recreating it. We will present the WRITE FILE, CLOSE FILE, and READ FILE commands within the framework of a program example dealing with variable-length records.

Later program examples will demonstrate these commands in the context of fixed-length records. Assume you want to create a short file to contain a list of your magazine subscriptions with their expiration dates. You expect to use this file for several purposes:

1. To provide an instant, up-to-date listing of all your current subscriptions.
2. To search for the status of any given magazine, or group of magazines.
3. To search for upcoming magazine expirations.
4. To group subscriptions by category (e.g., by expiration date, alphabetically by magazine title, by subject matter).

Each of the records in your file is to contain information on a single magazine consisting of the magazine's name and its exact expiration date (month, day, and year). We need four variables to hold these four values in each record:

M$ = magazine name

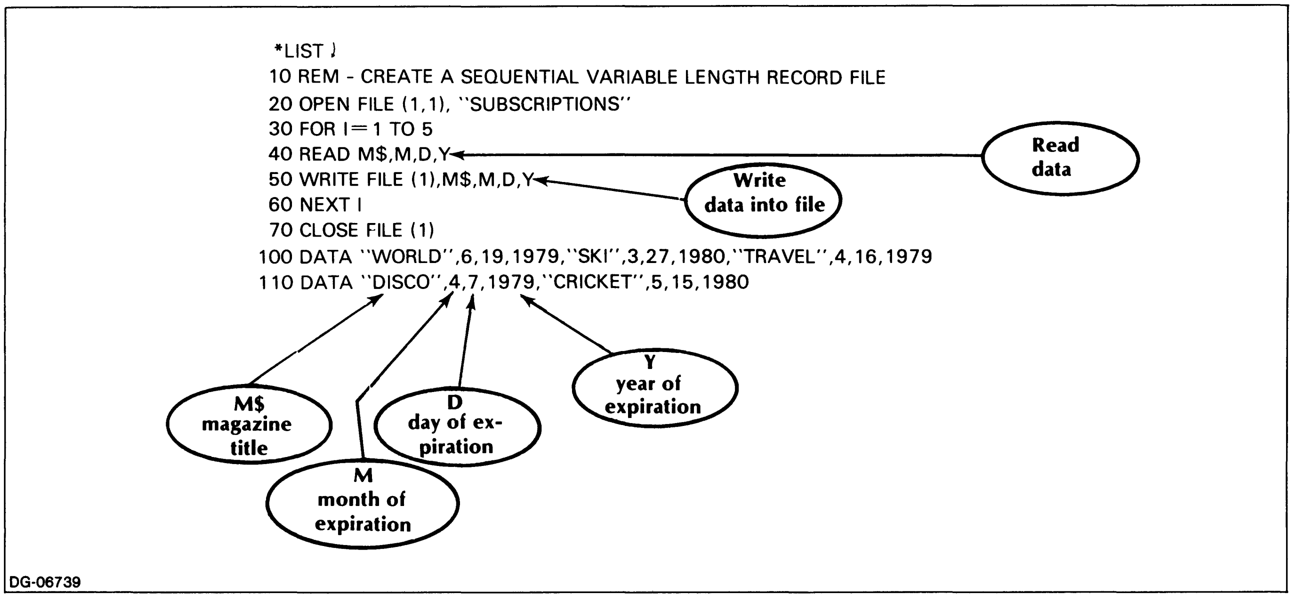
M = month of expiration

D = day of expiration

Y = year of expiration

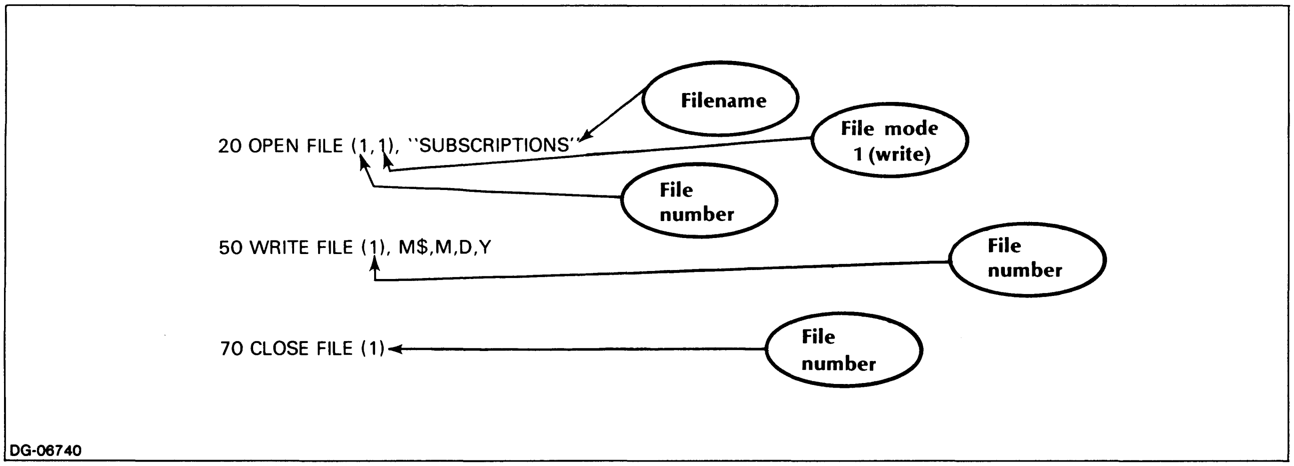
Next we decide to work with a sequential file and variable-length records. This will save us the work of calculating record length. (As you recall, in variable-length records, the size of each record corresponds automatically to the amount of space, in bytes, actually occupied by the data.)

We will begin with five magazines and with no particular order in mind for their sequence within the file. This being the case, we simply enter our four values for each magazine (M$, M, D, and Y) into data lines and write a loop that will read them and write them into the file. Figures 9-5 and 9-6 show the program and an extract from it highlighting the file commands we've used.



```
*LIST ↵
 10 REM - CREATE A SEQUENTIAL VARIABLE LENGTH RECORD FILE
 20 OPEN FILE (1,1), "SUBSCRIPTIONS"
 30 FOR I=1 TO 5
 40 READ M$,M,D,Y
 50 WRITE FILE (1),M$,M,D,Y
 60 NEXT I
 70 CLOSE FILE (1)
100 DATA "WORLD",6,19,1979,"SKI",3,27,1980,"TRAVEL",4,16,1979
110 DATA "DISCO",4,7,1979,"CRICKET",5,15,1980
```

Figure 9-5. Magazine Subscription Program



```
20 OPEN FILE (1,1), "SUBSCRIPTIONS"

50 WRITE FILE (1), M$,M,D,Y

70 CLOSE FILE (1)
```

Figure 9-6. Excerpt from Magazine Subscription Program

WRITE FILE

This command (line 50 in our program) places data into a file. Its format is simple: the words WRITE FILE need only be followed by the file number within parentheses, so that the program, which may have up to eight files open simultaneously, will identify the correct file to be written into.

The file number is immediately followed by a list of the data you want written into the file.

Our loop (lines 30 - 60) reads the data pertaining to a single record (line 40) and writes them into the file.

At the same time, the file pointer moves from its previous position to the end of the record just filled.

In the next pass of the loop, the program will write the new set of data it has just read into the next empty location indicated by the pointer. In this way, data are entered in sequential order, and the moving pointer ensures that new material is written next to and not on top of older material.

After execution of the loop, the pointer is set at the end of the last record written into, and all the data are permanently stored in the file.

The format of WRITE FILE is slightly different when you are writing into fixed-length records. You must specify the location of the record you are writing to. We discuss this later in this chapter.

CLOSE FILE

This command disables the relationship between a filename and a file number, indicating that the file in question will no longer be processed. When a file is closed, that file's pointer also ceases to exist.

You must close all your open files at the end of every program.

You can execute a CLOSE FILE whenever you need to return to the beginning of a sequential file.

You might, for example, want to read a file you have just written using Mode 1. To do so, you first CLOSE the file. Then you reopen it in Mode 0 or 3, which places the pointer in front of Record 0. Now you can read the file from its beginning.

If the file you want to reread contains fixed-length records, you need not close it in order to reset the pointer; ask the program to read record number 0, and the pointer will automatically return to the beginning of the file.

Format

Like the WRITE FILE command, the CLOSE FILE command requires only the file number in parentheses immediately after the words CLOSE FILE. The file number identifies the specific file you want closed, in case you happen to have several files open.

You can, however, simply use the keyword

```
CLOSE
```

by itself, without the word FILE or a file number. This version of the command will automatically close all files opened so far during the execution of your program.

Note that CLOSE FILE and CLOSE may be used in immediate mode as well as within a program statement. The same is true of all file commands, except DELETE, which may be used only in immediate mode.

The keyword BYE at the end of a BASIC session also has the effect of closing all files opened during the current session.

Appending Data to an Existing File: WRITE FILE, Mode 2

Assume you have just enlarged your subscription list by four additional magazines, which will necessitate updating your SUBSCRIPTIONS file by appending the new data to it.

You do this by reopening your file, SUBSCRIPTIONS in Mode 2, WRITE/APPEND. This will preserve the contents of your original file (reopening in Mode 1 would delete them) and will position the pointer at the end of the last filled record, so that writing may continue from that point on, either sequentially or to specified record numbers, as the case may be.

The procedure for reading the new data and writing them into the end of our SUBSCRIPTION file is precisely the same as the procedure we used to create the original file, with the single exception of the OPEN FILE statement, which now reads:

```
20 OPEN FILE (1,2), "SUBSCRIPTIONS"
```

This indicates that we are reopening an already existing file number 1 named SUBSCRIPTIONS and that we are going to use Mode 2 (WRITE/APPEND). The complete program appears in Figure 9-7.

Since we have four new magazines, our loop for reading the data from data lines and writing them into the file will go from 1 to 4.

Within this loop, we again read the values for variables M$, M, D, and Y from data lines, and write them into the file, just as we did before.

At the end of the program we again close the file.

After you have run this program, your file should contain a total of nine records, namely, the original five, and the four you have just appended to them.

If you have just tried to run this program, you may have been disconcerted to note that no output appeared on your screen or printer. This is just as it should be: you have instructed the program to place its output into the special file you have created, rather than into a device (screen or printer) that has no storage capabilities.

To see what is in your file, you must open it and read its contents.

READ FILE

We will now write a program to reopen your SUBSCRIPTIONS file, read its contents, and print them out under appropriate headings. Figure 9-8 displays that program.

A quick reference to the mode summary in Table 9-1 will remind you that, in order to read a file, you must open it in either Mode 0 or Mode 3. (Modes 1 and 2 permit writing into a file but not reading from it).

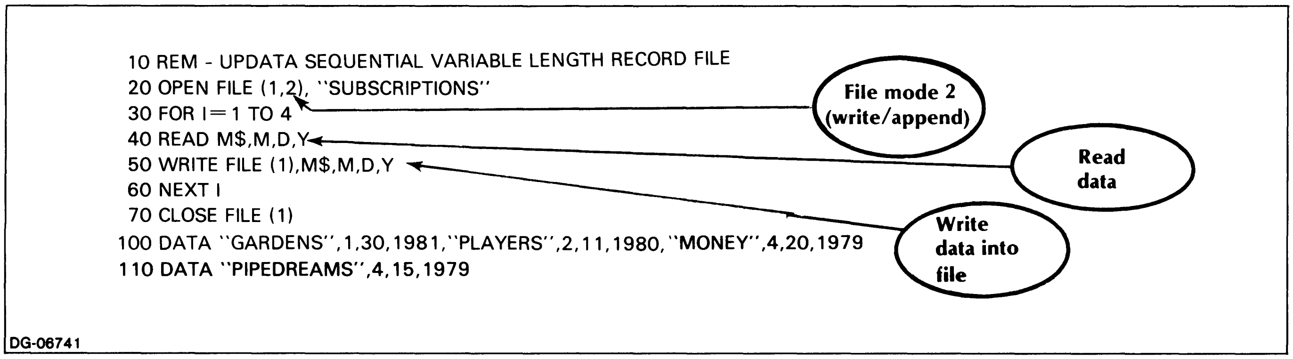
The OPEN FILE statement in line 20 specifies Mode 3 as the file mode. This is the only change we made in that statement.

Line 60 is the line executing the READ FILE command:

```
60 READ FILE (1), M$, M, D, Y
```

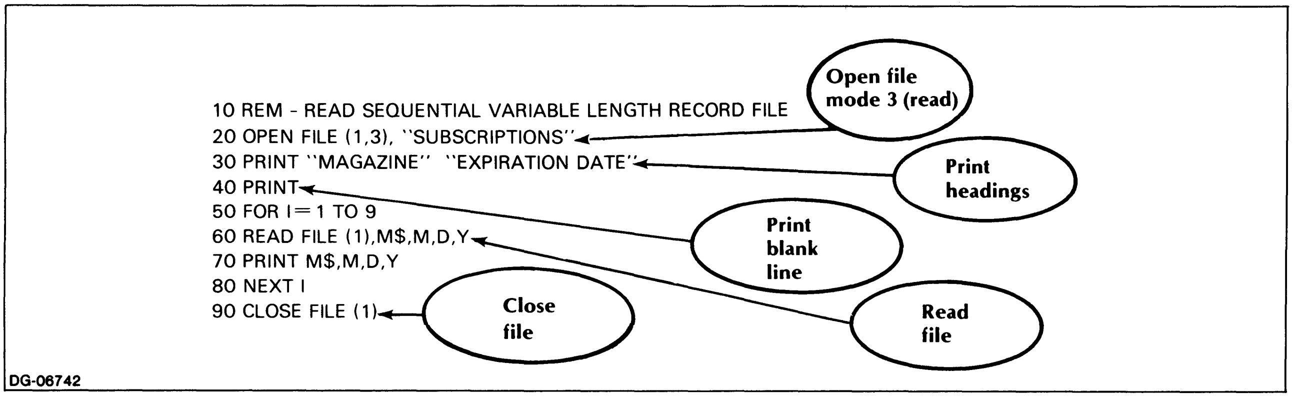
As in the case of the WRITE FILE command, we need only follow the word FILE with the file number we wish to read, enclosed in parentheses. (The slightly different format for random-access files will be discussed later.) We follow the file number with a list of the variables that constitute a single record to be read.

As each record in the file is read, the pointer moves along it, just as it does when you write. Thus, it will at any given time indicate the last file location we read; we can return the pointer to the beginning of the file by closing the file and then reopening it.



```
 10 REM - UPDATA SEQUENTIAL VARIABLE LENGTH RECORD FILE
 20 OPEN FILE (1,2), "SUBSCRIPTIONS"
 30 FOR I=1 TO 4
 40 READ M$,M,D,Y
 50 WRITE FILE (1),M$,M,D,Y
 60 NEXT I
 70 CLOSE FILE (1)
100 DATA "GARDENS",1,30,1981,"PLAYERS",2,11,1980,"MONEY",4,20,1979
110 DATA "PIPEDREAMS",4,15,1979
```

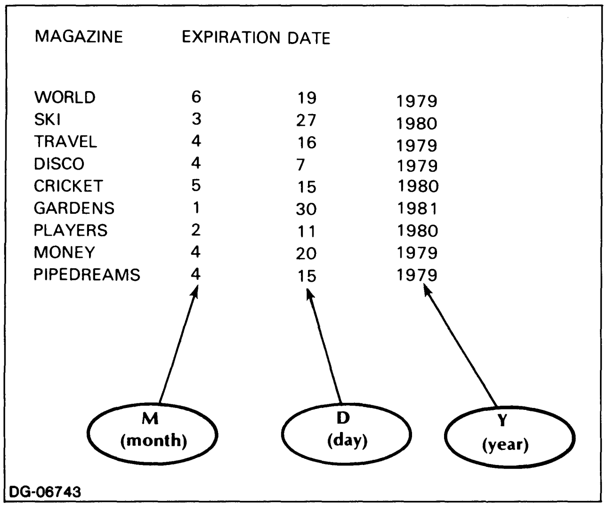
Figure 9-7. Program to Update SUBSCRIPTIONS File



```
10 REM - READ SEQUENTIAL VARIABLE LENGTH RECORD FILE
20 OPEN FILE (1,3), "SUBSCRIPTIONS"
30 PRINT "MAGAZINE" "EXPIRATION DATE"
40 PRINT
50 FOR I=1 TO 9
60 READ FILE (1),M$,M,D,Y
70 PRINT M$,M,D,Y
80 NEXT I
90 CLOSE FILE (1)
```

Figure 9-8. Program to Reopen, Read, and Print SUBSCRIPTIONS File

Figure 9-9 displays a listing of the contents of SUBSCRIPTIONS; the records you placed into that file in the course of two separate sessions are all present and accounted for here.



```
MAGAZINE       EXPIRATION DATE

WORLD           6        19        1979
SKI             3        27        1980
TRAVEL          4        16        1979
DISCO           4        7         1979
CRICKET         5        15        1980
GARDENS         1        30        1981
PLAYERS         2        11        1980
MONEY           4        20        1979
PIPEDREAMS      4        15        1979
```

Figure 9-9. Contents of the SUBSCRIPTIONS File

Working with a sequential, variable-length record file, you have so far done the following:

- Created a new file for writing (OPEN FILE MODE 1);
- Placed data in the file (WRITE FILE);
- Reopened your file for update (OPEN FILE MODE 2);
- Updated the file (WRITE FILE);
- Reopened your file for reading (OPEN FILE MODE 3);
- Read the file (READ FILE);
- Closed the file (CLOSE FILE).

Application: Access and Search a Sequential File

You can now apply these file access concepts to search for and process material in your file in a variety of ways, some of which we suggested a few pages earlier. We will illustrate one file application as an example.

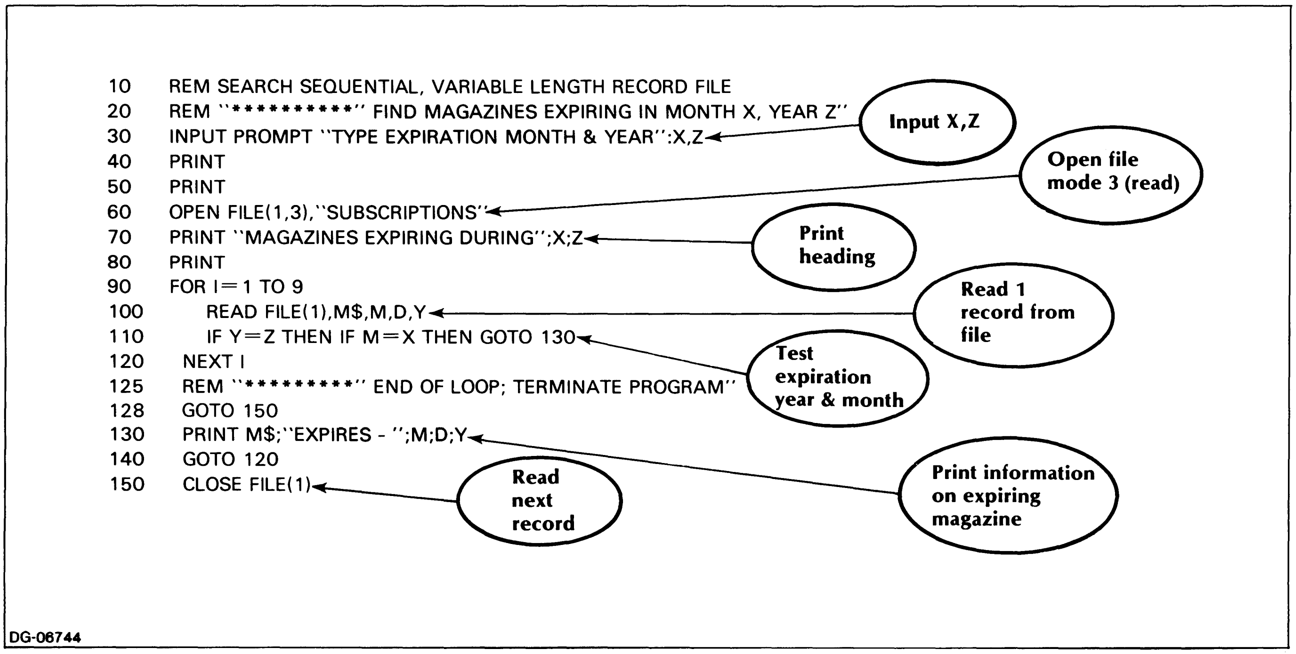
You want to write a program that will search your file and print out a list of all magazines expiring on a given date.

We need two new variables:

X—expiration month

Z—expiration year

These variables denote the expiration month and year for which you will be searching the file. (We will ignore the expiration day.)



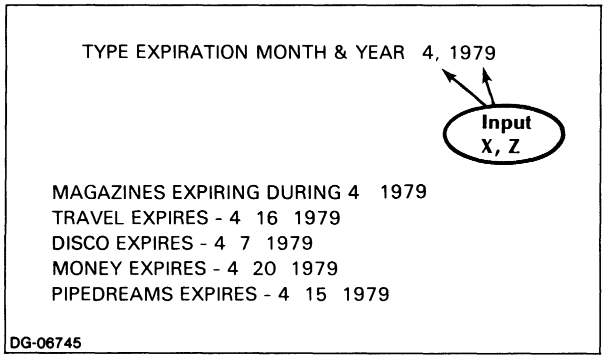
```
10    REM SEARCH SEQUENTIAL, VARIABLE LENGTH RECORD FILE
20    REM "**********" FIND MAGAZINES EXPIRING IN MONTH X, YEAR Z"
30    INPUT PROMPT "TYPE EXPIRATION MONTH & YEAR":X,Z
40    PRINT
50    PRINT
60    OPEN FILE(1,3),"SUBSCRIPTIONS"
70    PRINT "MAGAZINES EXPIRING DURING";X;Z
80    PRINT
90    FOR I=1 TO 9
100      READ FILE(1),M$,M,D,Y
110      IF Y=Z THEN IF M=X THEN GOTO 130
120    NEXT I
125    REM "*********" END OF LOOP; TERMINATE PROGRAM"
128    GOTO 150
130    PRINT M$;"EXPIRES - ";M;D;Y
140    GOTO 120
150    CLOSE FILE(1)
```

Figure 9-10. Program to Ccheck Expiration Dates in SUBSCRIPTIONS File

The program will ask you to input the values for X and Z, after which it will read each record in the subscription file in sequence, compare its expiration year and month (Y and M) against those you have specified in X and Z, and print out those dates that match yours. See Figure 9-10.

In addition to the INPUT statement (line 30), the program uses one loop (lines 90–120), one IF...THEN... statement (line 110), and two GOTO... statements (lines 110, 140) to control program flow.

When you run the program in figure 9-10, the output appears as in Figure 9-11.



```
TYPE EXPIRATION MONTH & YEAR  4, 1979

MAGAZINES EXPIRING DURING 4   1979
TRAVEL EXPIRES - 4  16  1979
DISCO EXPIRES - 4  7  1979
MONEY EXPIRES - 4  20  1979
PIPEDREAMS EXPIRES - 4  15  1979
```

Figure 9-11. Output from Program in Figure 9-10

The detailed sequence of events is as follows.

1. An INPUT PROMPT line (30) asks you to input your expiration month and year, in that order.
2. You type 4, 1979, i.e., April, 1979, in response.
3. File SUBSCRIPTIONS is reopened in Mode 3 (READ, line 60).
4. A heading followed by a blank line is printed (lines 70–80). Note that the heading appears outside the loop.
5. A loop reads each record in the file starting from the first. In each pass, an IF...THEN... control statement compares Y, the expiration year in the record, with Z, the expiration year you have input.

   If the years do not match, i.e., if Y <> Z, that record is passed over, and the program reads the next record. If the years match (Y = Z), you have found a magazine expiring during the year you want (here, 1979).
6. Now the program must determine whether the expiration month for that magazine is April, i.e., whether M = X.

   The second IF clause of the IF...THEN statement performs this comparison. In our example, this test will look for a value of 4 for both M and X.

   If the expiration months do not match, the program will branch back to the loop and read the next record.

   If the values of M and X do match, the program branches to line 130, which prints out the name of the magazine, along with its expiration date.
7. Thereafter, the program will branch back into the loop and repeat the process of reading and testing each record, until every record in the file has been read, compared to the information you have input, and either printed out, or passed over.

   (Building in control statements to return the program to the loop at the right junctures is one of the prerequisites for correct program flow here.)
8. File SUBSCRIPTIONS is closed before execution is terminated.

Try running this program with a variety of other expiration dates. You might also try building in a test sequence that will print out an appropriate message, if no magazines expire on the date you have specified.

DELETE

We discussed the DELETE command in Chapter 1 in connection with deleting program statements as well as entire saved programs.

When you no longer need a file, you erase it by typing the command DELETE followed by the name of the file. Thus, you could delete file SUBSCRIPTIONS by typing

```
* DELETE "SUBSCRIPTIONS"
```

Note that deletion of a program affects only the program, but not the files processed by that program; any such files remain intact until you explicitly delete them.

Now that you know how to use sequential files, we will illustrate how you work with fixed-length records and how you take advantage of the random-access features of files containing such records.

OPEN FILE, Fixed-Length, Random-Access File

In Figures 9-2 and 9-3 we used a hypothetical fuel file as an example of records. We would now like to create such a file with records of fuel expenses for the year 1978.

Naturally, we will need to have a complete set of data for each month, but it will save a great deal of time if we don't need to sort the months chronologically.

Our program should read the fuel data for the entire year in random order, then sort them out and write them into records in the proper sequence of months, moving from January through December.

With a random-access file, this is a simple matter.

Each file record will consist of three data items, as follows:

M$	name of month
G	gallons of fuel used that month
P	price per gallon of fuel used.

Since we want to use the random-access features of our file, we will have to work with fixed-length records. So our next step is to determine their size.

G and P are real numbers; therefore, they will require four bytes each.

M$ will require a different number of bytes each month, depending on the number of characters in a given month's name. (Recall that each string character requires one byte, plus one extra byte at the end of the string for the null character serving as a delimiter.)

The maximum number of bytes we will need for M$ will thus correspond to the number of characters in the longest month of the year. That month happens to be September, which has nine characters; we, therefore, need a maximum of ten bytes for variable M$ (up to nine bytes for the characters, plus one byte for the null character).

The total number of bytes we will need for each record is thus:

M$	10 bytes
G	4 bytes (numeric data item)
P	4 bytes (numeric data item)
	18 bytes total

Now we can create our fuel file by means of the statement,

```
30 OPEN FILE (1,0), "FUEL.78", 18
```

where

1	is the file number.
0	is the file mode (fixed length, WRITE/READ).
FUEL.78	is the filename (with extension designating file for 1978).
18	is the maximum record length (in bytes).

Determining Record Location:

WRITE FILE, Random Access – Next we turn to the problem of ordering the records within the file. As we have said, we want to enter the data randomly and let the program sort them out so that the record for January should appear in the file before the record for February, June should appear before July, and so on.

We can achieve this by assigning each of our records an index number corresponding to the location in which that record should appear in the file, i.e.:

```
January = 0
February = 1
.
.
.
December = 11
```

We will use the variable J to hold these index values.

Next, we'll list the value of J at the beginning of each record as we enter it in the data line. For example,

```
150 DATA 2, MARCH, 170, .55, 9, OCTOBER, 150, .56
```

and so on, where the number 2 preceding MARCH means that March belongs in the record number 2, and the number 9 preceding OCTOBER means that this record belongs in record number 10 (beginning our record location count from 0).

The program will read J along with M$, G, and P and, if so instructed, it will write M$, G, and P directly into the file location indicated by the value of J. (J itself will not appear anywhere, since it serves only as an index.)

When the WRITE FILE statement includes a record number, that number appears in parentheses immediately after the file number. For example,

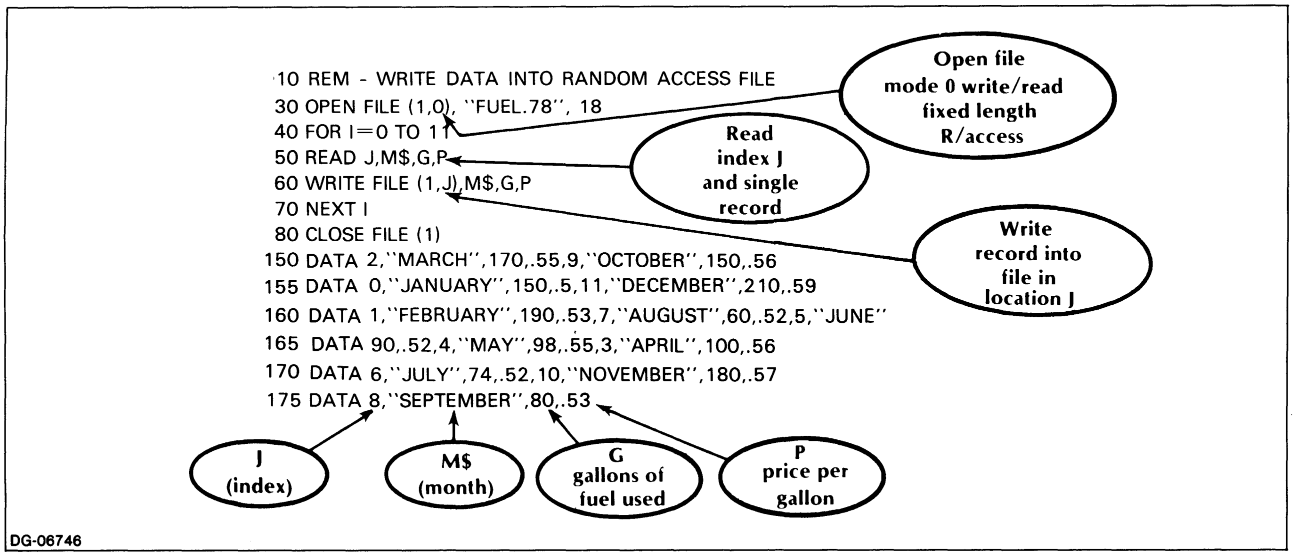
```
60 WRITE FILE (1, J), M$, G, P
```

where 1 is the file number and J is the index specifying the record number within the file.

You can, of course, also refer to records directly, without using an index. The format is the same; for example,

```
* 150 WRITE FILE (1,7), X
```

where 7 is the number of the record you want to process.



```
10 REM - WRITE DATA INTO RANDOM ACCESS FILE
 30 OPEN FILE (1,0), ''FUEL.78'', 18
 40 FOR I=0 TO 11
 50 READ J,M$,G,P
 60 WRITE FILE (1,J),M$,G,P
 70 NEXT I
 80 CLOSE FILE (1)
150 DATA 2,''MARCH'',170,.55,9,''OCTOBER'',150,.56
155 DATA 0,''JANUARY'',150,.5,11,''DECEMBER'',210,.59
160 DATA 1,''FEBRUARY'',190,.53,7,''AUGUST'',60,.52,5,''JUNE''
165 DATA 90,.52,4,''MAY'',98,.55,3,''APRIL'',100,.56
170 DATA 6,''JULY'',74,.52,10,''NOVEMBER'',180,.57
175 DATA 8,''SEPTEMBER'',80,.53
```

Figure 9-12. Fuel Program, Illustrating Random Access

In an ordinary WRITE FILE operation, the pointer, you recall, moves in sequence, processing data as though you were dealing with records of variable length.

In random-access execution, however, the pointer will move back and forth from one record location designated by J to the next. In our example, the pointer will move to the beginning of record number 2 (byte 36) where the data for March will be written; from there it will move to the beginning of record number 9 (byte 162), where the October data will be entered, returning from there to the beginning of record 0 (byte 0) for the January data, and so on.

The program appears in Figure 9-12.

To test whether all 12 records for the year appear in correct order, you can write a short program to reopen your FUEL.78 file in Mode 0 or Mode 3 (we chose the former), read its contents sequentially in a loop, and print them out under an appropriate heading.

To have your records read correctly, you should reopen the FUEL.78 file with the same record-length declaration you used in creating the file. Moreover, even though you are reading sequentially, your READ FILE statement should specify the record location of the data, so that the reading begins at the byte number corresponding to the beginning of your data.

The format for indicating record location in the READ FILE statement is identical to that of the WRITE FILE statement. For example,

```
* 60 READ FILE (0, I), M$, G, P
```

where I is a location index. If you refer directly to a record location, that number appears in the same place as the index would.

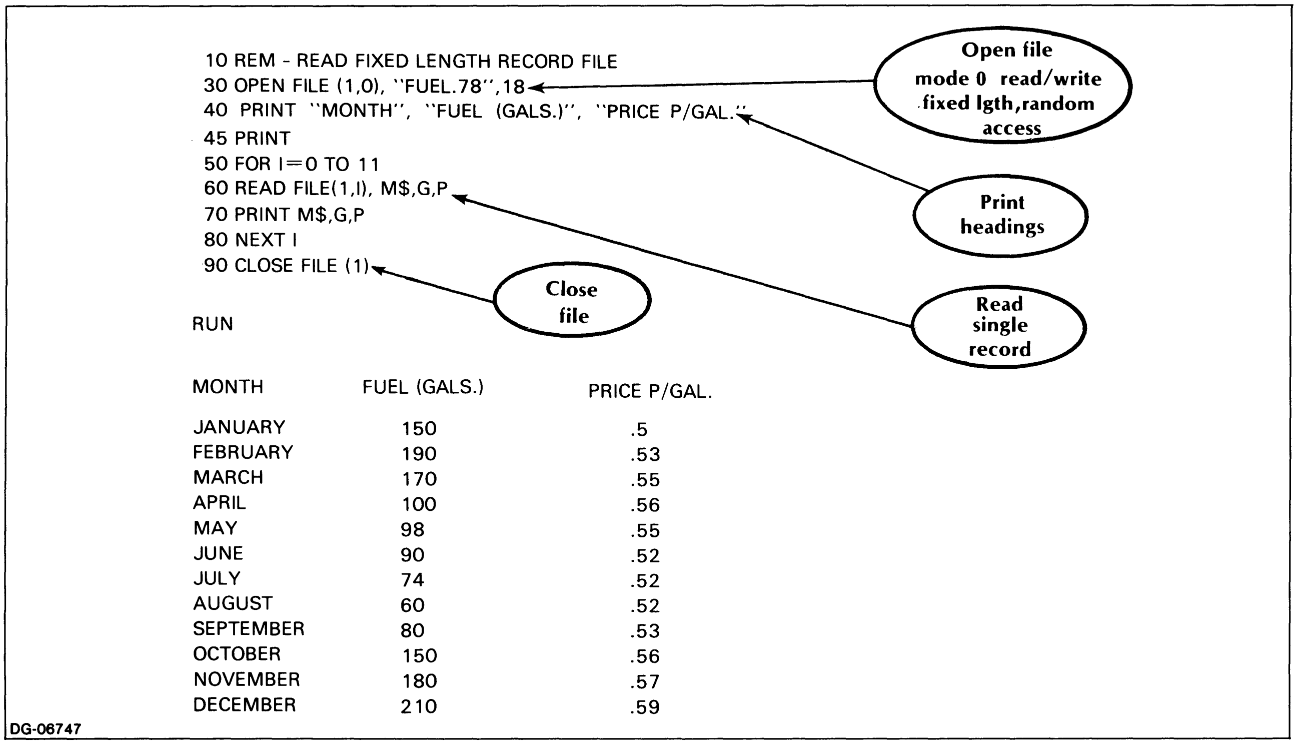
Since you are reading the data in a loop, you can use your loop counter, I, as an index to specify record location. This will cause sequential reading of your first, second, third, through twelfth records in FUEL.78.

Figure 9-13 lists a program to read and print out the contents of FUEL.78 and shows the resulting output.

Extracting Data from a Random Access File: READ FILE with Location Index

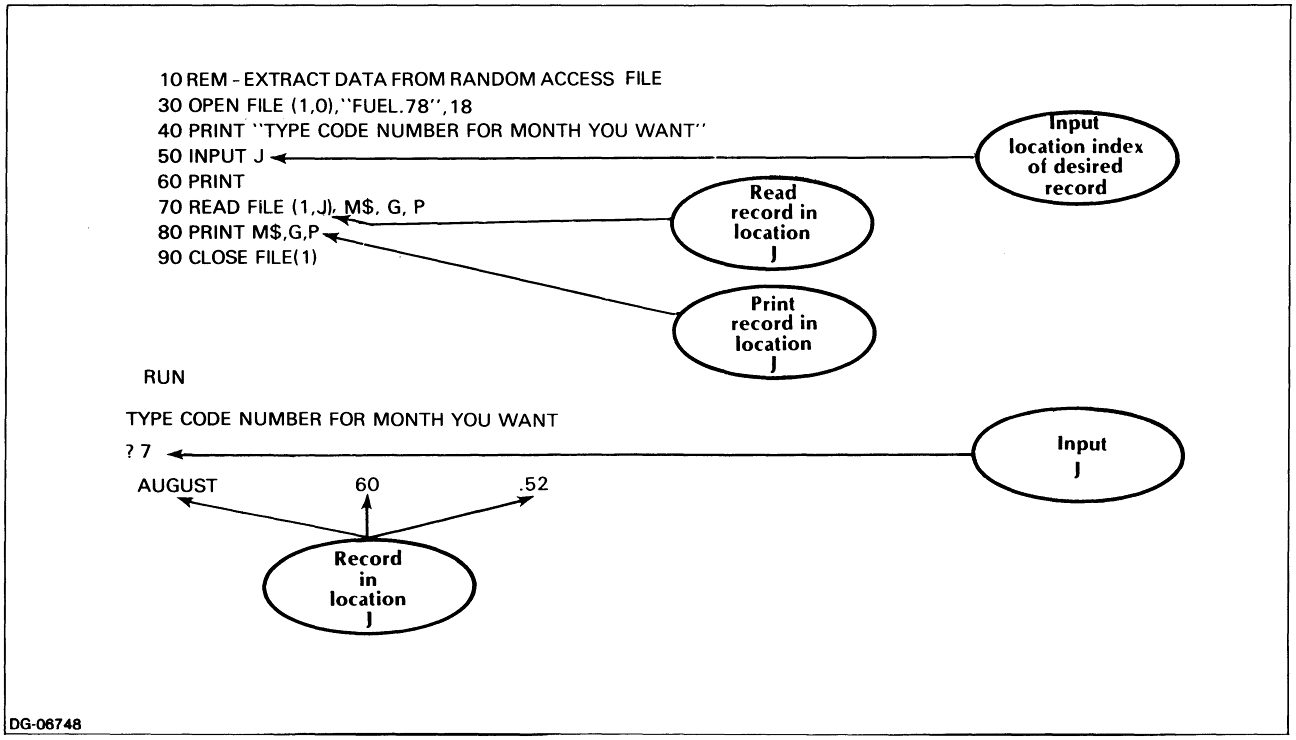
Addressing any individual record (or part of it) is a simple matter, once you know its location. Figure 9-14 illustrates a program to read any given item from a random-access file. The user inputs the location index for the data he wishes to see; the system responds by extracting and displaying the data at the specified byte number. We are still using the FUEL.78 file, and J as the location index to records in that file; but, as we've already stated, you could also refer to records directly by their number, if you have that information.

In a sequential file you would have to instruct the computer to read the contents of your file up to the item you want, so as to move the file pointer sequentially to the desired location. In a random-access file such as the one we are working with, the file pointer is automatically moved to the location you specify, saving you several coding instructions.



```
10 REM - READ FIXED LENGTH RECORD FILE
30 OPEN FILE (1,0), "FUEL.78",18
40 PRINT "MONTH", "FUEL (GALS.)", "PRICE P/GAL."
45 PRINT
50 FOR I=0 TO 11
60 READ FILE(1,I), M$,G,P
70 PRINT M$,G,P
80 NEXT I
90 CLOSE FILE (1)

RUN

MONTH          FUEL (GALS.)        PRICE P/GAL.

JANUARY           150                 .5
FEBRUARY          190                 .53
MARCH             170                 .55
APRIL             100                 .56
MAY               98                  .55
JUNE              90                  .52
JULY              74                  .52
AUGUST            60                  .52
SEPTEMBER         80                  .53
OCTOBER           150                 .56
NOVEMBER          180                 .57
DECEMBER          210                 .59
```

Figure 9-13. Program to Read and Print Out Table Generated by Program in Figure 9-12



```
10 REM - EXTRACT DATA FROM RANDOM ACCESS FILE
30 OPEN FILE (1,0),"FUEL.78",18
40 PRINT "TYPE CODE NUMBER FOR MONTH YOU WANT"
50 INPUT J
60 PRINT
70 READ FILE (1,J), M$, G, P
80 PRINT M$,G,P
90 CLOSE FILE(1)

RUN

TYPE CODE NUMBER FOR MONTH YOU WANT
? 7
 AUGUST              60              .52
```

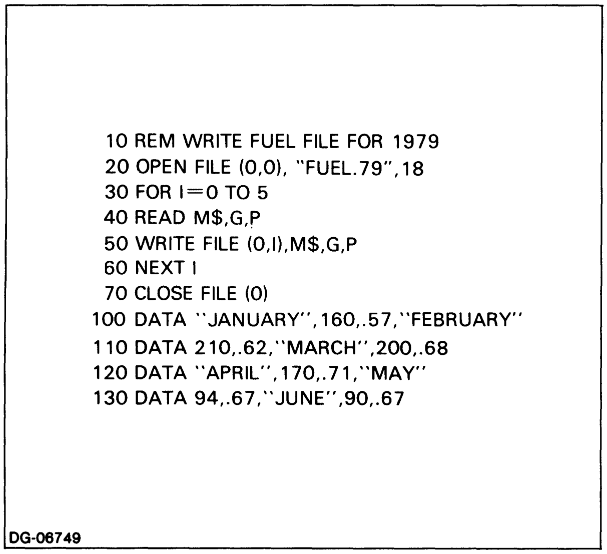
Input location index of desired record
Read record in location J
Print record in location J
Input J
Record in location J
DG-06748

Figure 9-14. Program to Read Data froma Random Access File

Application: Comparing Data from Two Files

Assuming you have kept separate fuel files over a number of years, you could use them to obtain comparative figures on amounts of fuel used and prices paid.

To suggest the range of possibilities inherent in file comparisons, we create a FUEL.79 file for the first six months of 1979. See Figure 9-15.



```
 10 REM WRITE FUEL FILE FOR 1979
 20 OPEN FILE (0,0), "FUEL.79",18
 30 FOR I=0 TO 5
 40 READ M$,G,P
 50 WRITE FILE (0,I),M$,G,P
 60 NEXT I
 70 CLOSE FILE (0)
100 DATA "JANUARY",160,.57,"FEBRUARY"
110 DATA 210,.62,"MARCH",200,.68
120 DATA "APRIL",170,.71,"MAY"
130 DATA 94,.67,"JUNE",90,.67
```

DG-06749

Figure 9-15. Program to Create a File for Comparison

This program is essentially the same as the one we used to create the FUEL.78 file (for expediency, however, we have entered the data in order by month and have dispensed with the ordering index J). We retain the "FUEL" part of the filename, adding a new extension to indicate the year 1979.

Now we can write a program to open both files, read them, and print out a table comparing gallons of fuel used and price per gallon for January through June of 1978 and 1979, along with totals and an average price per gallon for each of these periods. This program appears in Figure 9-16.

Note that each file is reopened in a separate statement (lines 20, 30).

We then print headings for our comparative table (lines 40, 50).

We write a loop (lines 60-160), to read the first six records in one file, once more using the loop counter as a location index. The loop totals the number of gallons used (G) and the prices paid for them (P), calculates an average price per gallon paid during the first six months of 1978, and then repeats these operations with the first six records of the next file.

The formulas below supply our total and average figures:

$$T = T + G$$

$$S = S + P$$

$$A = S/6$$

where

T = Cumulative total, 1978 file: number of gallons used for first six months of year.

S = Cumulative total, 1978 file: price per gallon paid during first six months of year.

A = Average price per gallon, 1978 file: total price paid, divided by number of months.

For the FUEL.79 file, we use the following variables to designate the same cumulative operations as above:

$$T2 = T2 + G2$$

$$S2 = S2 + P2$$

$$A2 = S2/6$$

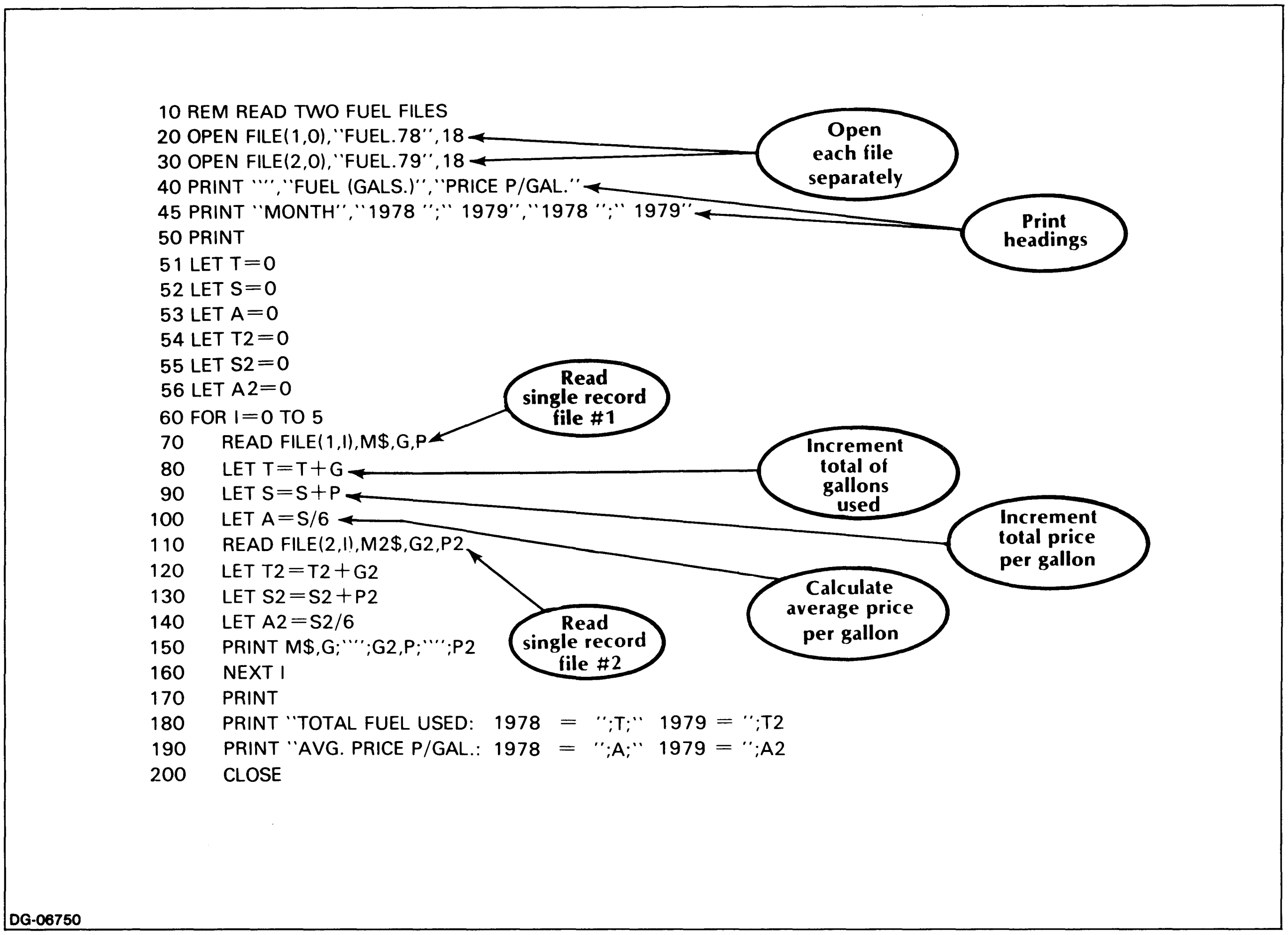
where

T2 = Cumulative total, 1979 file: number of gallons used for first six months of year.

S2 = Cumulative total, 1979 file: price per gallon paid during first six months of year.

A2 = Average price per gallon, 1979 file: total paid, divided by number of months.

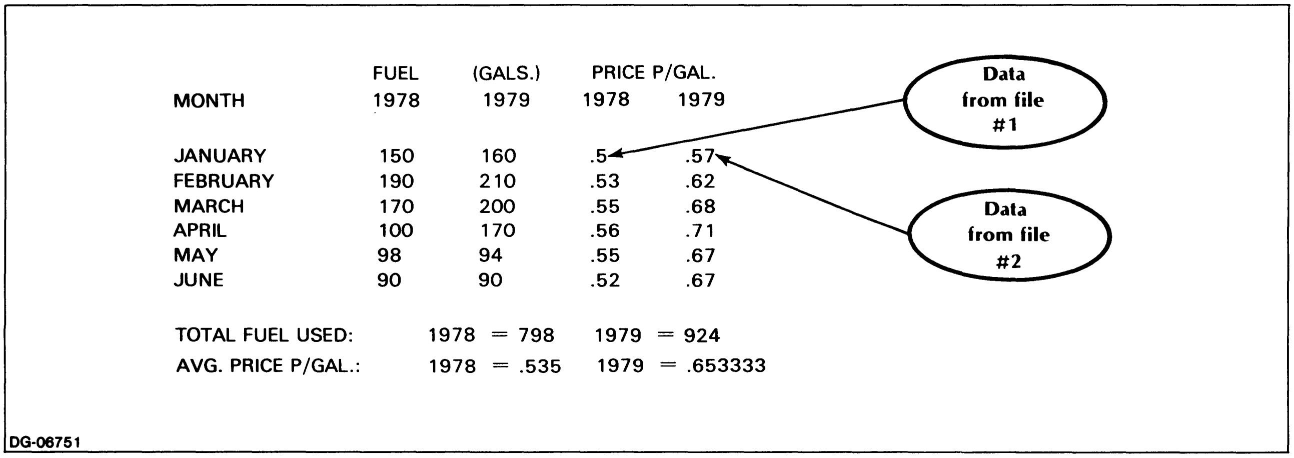
Note that the output from this program appears in Figure 9-17.



```
10 REM READ TWO FUEL FILES
20 OPEN FILE(1,0),"FUEL.78",18
30 OPEN FILE(2,0),"FUEL.79",18
40 PRINT "","FUEL (GALS.)","PRICE P/GAL."
45 PRINT "MONTH","1978 ";" 1979","1978 ";" 1979"
50 PRINT
51 LET T=0
52 LET S=0
53 LET A=0
54 LET T2=0
55 LET S2=0
56 LET A2=0
60 FOR I=0 TO 5
70      READ FILE(1,I),M$,G,P
80      LET T=T+G
90      LET S=S+P
100     LET A=S/6
110     READ FILE(2,I),M2$,G2,P2
120     LET T2=T2+G2
130     LET S2=S2+P2
140     LET A2=S2/6
150     PRINT M$,G;"";G2,P;"";P2
160     NEXT I
170     PRINT
180     PRINT "TOTAL FUEL USED:  1978  =  ";T;"  1979 = ";T2
190     PRINT "AVG. PRICE P/GAL.: 1978  =  ";A;"  1979 = ";A2
200     CLOSE
```

Open each file separately
Print headings
Read single record file #1
Increment total of gallons used
Increment total price per gallon
Calculate average price per gallon
Read single record file #2
DG-06750

Figure 9-16. Fule Comparison Program: Two Files



```
                 FUEL     (GALS.)    PRICE P/GAL.
MONTH            1978      1979     1978     1979

JANUARY          150       160      .5       .57
FEBRUARY         190       210      .53      .62
MARCH            170       200      .55      .68
APRIL            100       170      .56      .71
MAY              98        94       .55      .67
JUNE             90        90       .52      .67

TOTAL FUEL USED:      1978 = 798    1979 = 924
AVG. PRICE P/GAL.:    1978 = .535   1979 = .653333
```

Figure 9-17. Output from Fuel Comparison Program

Table 9-2. File Input and Output Commands

Function	Instruction for a binary file	Instruction for an ASCII file
Move data from your work area into a file.	WRITE FILE	PRINT FILE
Move data from a file into your work area.	READ FILE	INPUT FILE or LINPUT FILE

Data Formats

We have now covered the most common file concepts and commands, as well as some basic file uses. The data transfer statements we have covered so far in this chapter (READ FILE and WRITE FILE) read and write data from and to files in the format illustrated in figures 9-2 and 9-3. This is known as binary format. We can also read and write from and to files in ASCII format, using the three related commands PRINT FILE, INPUT FILE, and LINPUT FILE. These commands are the same as the PRINT, INPUT, and LINPUT commands, except that they input or output from or to a file rather than from or to your terminal.

The relation between these commands is summarized in Table 9-2.

Numerical data is more efficiently stored in binary format than in ASCII format. In ASCII format, all data is stored as ASCII strings, one byte per character plus a terminator. For example, the number 123.456 would be stored in ASCII as the string "123.456", which would require 7 bytes (at one byte per character) plus 1 byte for the terminator, for a total of 8 bytes. The same number, stored as a single-precision real number in binary format, would require only 4 bytes.

Also, programs using binary files usually run faster than programs using ASCII files.

On the other hand, data stored in binary format is not directly accessible to programs that require all information to be stored in ASCII format. For example, you cannot use a CLI command to output a binary-formatted file to the line printer.

Data written into a file in binary format cannot be read back into your work area by an instruction for reading ASCII data, and vice versa. When you want to store your information in a file for later use, you should choose the type of data format based upon your anticipated use of the data. As you become a more sophisticated programmer, you will possibly consider several criteria in this selection, but for now, the following guidelines may be helpful:

- If you expect to only be reading the data back into your work area for use in an MP/BASIC program, then use the binary format (i.e., use the WRITE FILE and the READ FILE statements).
- If you intend to use a CLI command to print the data on the line printer, or on your terminal, as well as to read the data back into your work area for use in an MP/BASIC program, then use the ASCII format (i.e., use the PRINT FILE and the INPUT FILE or LINPUT FILE statements).

The important thing to remember is that a file created with the PRINT FILE statement cannot be read with the READ FILE statement, and a file created with the WRITE FILE statement cannot be read with the INPUT FILE or the LINPUT FILE statements.

PRINT FILE

This command writes data into a file in ASCII format, rather than in binary. For example, the program statement 20 PRINT FILE(1), A writes the current numeric value of the variable A into file 1 as an ASCII character string, just as the statement 20 PRINT A outputs the current value of A to your screen as an ASCII character string.

Data written into a file with the PRINT FILE command may be output to a line printer or to a disk file for later printing.

As with the PRINT statement described in Chapter 1, you can add the USING keyword to the PRINT FILE statement, to achieve formatted output. For example:

```
PRINT FILE(1), USING "###":010
```

INPUT FILE

A file created with the PRINT FILE command, cannot be read with the READ FILE command. Instead, use the INPUT FILE or LINPUT FILE commands.

The format for the INPUT FILE, PRINT FILE, and LINPUT FILE commands is the same as for other file commands: the command is followed by the file number in parentheses.

For example:

```
INPUT FILE(1), X,Z
```

This command will read the first two numeric variables in the file (provided, of course, that the file has been properly opened).

If you wish to read a given record in the file, specify its number immediately after the file number, just as you do with random-access READ FILE or WRITE FILE commands. (Remember to specify fixed record length when opening a file for random access.)

For example:

```
INPUT FILE(1,5), X,Z
```

This command will read record number five in the file.

As we mentioned above, the INPUT FILE statement is the same as the INPUT statement, except for the source from which the data is input (from a file for INPUT FILE, from your keyboard for INPUT). Thus, when you create a file to be read by the INPUT FILE statement, you must make sure that the content of the file is the same as if you had given an onscreen response to the INPUT statement. This means that, if your INPUT FILE command requests several variables, then the data items in the file must be separated by literal commas which you enter within quotes when creating the file. For example, assume you create a file, "S.OUT," into which you enter three numeric values. You would code the instructions so that a comma follows each of the values entered into the file, as in line 50 below:

```
* 10 OPEN FILE(1,0), "S.OUT"
* 20 LET X = 5
* 30 LET Y = 10
* 40 LET Z = 20
* 50 PRINT FILE(1), X;",";Y;",";Z
* 60 CLOSE FILE(1)
```

The file created in this fashion can now be read by the INPUT FILE statement, as follows:

```
* 70 OPEN FILE(1,0), "S.OUT"
* 80 INPUT FILE(1), R,S,T
* 90 PRINT R,S,T
* 100 CLOSE FILE (1)
* RUN
5 10 20
*
```

LINPUT FILE

The LINPUT FILE command functions exactly as the LINPUT command described in Chapter 4. Its format corresponds to that of the INPUT FILE command.

Change lines 80 and 90 in the above example to read as follows:

```
* 80 LINPUT FILE(1), A$
* 90 PRINT A$
```

The output would now consist of the string A$, as follows:

5, 10, 20

Since the LINPUT FILE command accepts all data up to a NEW LINE, carriage return, form feed, or null character, the special considerations discussed with regard to the INPUT FILE command do not apply; you need not include literal commas as data separators when creating files to be read with LINPUT FILE.

Keywords in Chapter 9

Table 9-3 lists the keywords introduced in Chapter 9.

Table 9-3. Keywords in Chapter 9

Keyword	Can be used in: Program Statement	Immediate Mode
CLOSE	Yes	Yes
CLOSE FILE	Yes	Yes
DELETE	No	Yes
INPUT FILE	Yes	Yes
LINPUT FILE	Yes	Yes
OPEN FILE	Yes	Yes
PRINT FILE	Yes	Yes
READ FILE	Yes	Yes
WRITE FILE	Yes	Yes

End of Chapter

Chapter 10
Program Segmentation

You may need to write programs that are too large to fit into memory. You can divide such programs into segments, and then load and execute each program segment in sequence to perform the desired overall programming task.

This is sometimes called the chaining facility.

MP/BASIC provides four commands for program segmentation: SAVE, CHAIN, LOAD, and SWAP.

Suppose you are writing a large program that will process masses of data to generate a report, and then print that report. Because of the size of the program, you decide to break it into two parts:

- a main program segment to process the data and generate the information for the report;
- a second program segment to print the report.

We are not concerned here with the details of the actual processing of the data or the printing of the report, but just with how the two program segments interrelate. Therefore, we will present only those parts of each segment that set up the segmentation.

The first step is to write and type in each segment and store it in a file. Let us store the main program segment in file MAINPRG and the second program segment in file PRNTREP.

Saving Program Segments: The SAVE Command

You can type in and store the segments in any order; in this example, let us start by typing in the second program segment:

```
10 REM THIS IS THE SECOND PROGRAM SEGMENT
20 PRINT "PRNTREP IS STARTING"
30 REM
40 REM
50 REM Statements to
60 REM actually prepare
70 REM and print the
80 REM report would
90 REM go here.
100 REM
110 REM
120 PRINT "PRNTREP IS FINISHING"
130 END
```

We store the second program segment for later execution by using the command

```
SAVE "PRNTREP"
```

This saves our program, along with its data area, in a special format, in file "PRNTREP".

The format of the file made by the SAVE command is different from the format of a file made by the LIST command (discussed in Chapter 1). The file made by LIST (called a *source file*) consists of just the program statements, character by character, in ASCII format. The file made by SAVE saves a data area (containing, for example, the current values of any variables and other data required by MP/BASIC) as well as the statements.

The segmentation commands CHAIN, LOAD, and SWAP all work with programs that have been stored by the SAVE command.

Once we have saved the second segment, we type in the main program segment. The SAVE "PRNTREP" command does not delete the second program segment from working memory, so we should use the NEW command to clear our memory area before typing in the main program segment.

We type in the main program segment as follows:

```
10 REM THIS IS THE MAIN PROGRAM SEGMENT
20 PRINT "MAIN PROGRAM SEGMENT IS STARTING"
30 REM
40 REM
50 REM Statements to actually process the
60 REM data and prepare the information
70 REM for the report would go here.
80 REM
90 REM PRINT "LINK TO SECOND SEGMENT"
100 CHAIN "PRNTREP"
```

Next, we save this segment in file "MAINPRG" by using the command

```
SAVE "MAINPRG"
```

Linking to the Next Segment: The CHAIN Command

When the main program segment is executed later on, the statement

```
100 CHAIN "PRNTREP"
```

deletes the main program segment from working memory (as if a NEW had been executed), loads the second segment from its SAVE file and begins executing the second segment at its first line.

Notice that both the main program segment and the second segment use some of the same line numbers. Since only one segment is in memory at a time, there is complete independence of statement numbers between the segments (as we will discuss in more detail below, there is also independence of variables between segments).

Loading a Program Segment: The LOAD Command

We can now try running our program, as follows:

```
* NEW
* LOAD "MAINPRG"
* RUN
MAIN PROGRAM SEGMENT IS STARTING
LINK TO SECOND SEGMENT
PRNTREP IS STARTING
PRNTREP IS FINISHING
```

The command LOAD "MAINPRG" brings the main program segment (along with its stored data area) into memory. The RUN command executes the main segment, beginning at its first line. At the end of the main segment, the statement

```
100 CHAIN "PRNTREP"
```

clears the main program from memory, and calls in and executes the second program segment. The program terminates at the END statement in the second segment.

After the program terminates, the second segment remains in your memory area until you delete it or replace it with something else.

Swapping the Current Segment: the SWAP Statement

Suppose we wanted our main program segment to perform other operations after the completion of the second segment. For example, after printing the report we might want to store the date that the source data was processed and printed. We can return control back to the main program segment by using a SWAP statement instead of a CHAIN statement in line 100 of the main program, and including additional statements as follows:

```
100 SWAP "PRNTREP"
110 PRINT "MAIN PROGRAM SEGMENT IS CONTINU-
ING"
120 REM additional processing
130 REM statements could
140 REM go here
150 PRINT "MAIN PROGRAM SEGMENT IS FINISHED"
160 END
```

When executed, the SWAP statement on line 100 would SAVE the main segment, and load and execute the second segment. When the second segment terminates with a STOP or END, then the main segment (along with its SAVEd data area) will be LOADed back into memory and execution would continue at the line in the main program segment following the SWAP statment (in this case, line 110).

If we now run, we will get

```
*LOAD "MAINPRG"
*RUN
MAIN PROGRAM SEGMENT IS STARTING
PRNTREP IS STARTING
PRNTREP IS FINISHING
MAIN PROGRAM SEGMENT IS CONTINUING
MAIN PROGRAM SEGMENT IS FINISHED
```

Data Independence

Notice that the SWAP statement in the example just above saves the data area of the main program segment, and loads this area back into memory upon completion of the second segment. Therefore, any values of variables assigned during the execution of the main segment prior to the SWAP are retained after the completion of the SWAP.

The important thing to remember is that each program segment has a data area that is saved by the SAVE command and the SWAP statement; whenever the data area is loaded back into memory, the variables will retain their former values.

Because only one program segment (and its data area) is in memory at one time, there is independence between the names and current values of variables used in linked segments. Thus, changing the values of variables in one segment will not have any effect upon the values of variables in another linked segment, even if two variables in different segments have the same name.

This data independence also means that no data is passed directly between segments linked by CHAIN or SWAP.

Transferring Data Between Segments

You can transfer data between linked segments through files. In the example above, the main program segment must pass the information for the report to the second segment. This can be done by having the main program segment write the necessary information into a file that the second segment reads.

CHAIN and SWAP do not close any open files before deleting the calling program segment, nor do they reset any file pointers. This means that the file pointer will remain where it was set by the previous segment.

Running Under Different Versions of MP/BASIC

In order to permit future enhancement of MP/BASIC, files generated by SAVE under one version of MP/BASIC will not be compatible with a later version of MP/BASIC. The SAVE file format contains an MP/BASIC revision number so that incompatable SAVEd files can be detected.

You can, however, update your SAVE format files for compatibility, quite simply. If you have a source file (generated by the LIST command) of your program segments, then, operating under the new version of MP/BASIC, use ENTER to load the source file into memory. Then use the SAVE command to create a new file, which will be compatible with the new version. If you do not have a source file, then you can create one from the SAVEd file. Under the old version of MP/BASIC, LOAD in the program segment and then use the LIST command to create the source file. Source files are compatible between versions of MP/BASIC.

Debugging

One of the advantages of segmenting is that you can write and debug each program segment relatively independently. Once you have SAVEd a segment, you can use the LOAD command to call it back into memory. This is especially useful when you need to modify a segment, but the source code is not readily available.

Keywords in Chapter 10

Table 10-1 lists the keywords introduced in Chapter 10.

Table 10-1. Keywords in Chapter 10

Keyword	Can be used in	
	Program Statement	**Immediate Mode**
CHAIN	YES	No
LOAD	NO	YES
SAVE	YES	YES
SWAP	YES	NO

End of Chapter

Chapter 11
Exception Handling

Suppose you need to find out how many records there are in a sequential data file consisting of people's names. One way of obtaining this information would be through the following program fragment:

```
.
.
.
320 OPEN FILE(1,3),"NAMES"
340 LET COUNT=0
350 REM READ A RECORD FROM THE FILE
360 READ FILE(1),LAST$,FIRST$
380 LET COUNT=COUNT+1
400 GOTO 350
.
.
.
```

The statements on lines 350 through 400 form a loop, in which a counter is incremented once for each record in the file. When the end-of-file marker is reached, the value of COUNT will be equal to the number of records in the file.

However, when the end-of-file marker is reached, MP/BASIC will terminate your program and print an error message. You probably needed the number of records in the file in order to carry out another task, so you do not want your program terminated immediately after the counting task.

What you would like to do in this situation is to temporarily circumvent MP/BASIC's normal error handling code and to substitute your own instructions for regaining control after the end-of-file. You can do this by using the MP/BASIC exception handling facilities (errors are sometimes called exceptions).

The commands provided to handle exceptions are ENABLE HANDLER, DISABLE HANDLER, HANDLER, END HANDLER, RESUME, CONTINUE, RETRY, and CAUSE. MP/BASIC also provides the functions EXLINE and EXTYPE.

Implementing Exception Handling

You need to do two things to implement the MP/BASIC exception handling facilities. First, you need to tell the computer what you want done if an error occurs; then you need to temporarily divert control from MP/BASIC's normal error-handling code to your own error-handling code.

You accomplish the first task by writing an exception handler, which is a block of statements in your program that tells the computer what you want done if an exception occurs.

Defining an Exception Handler: the HANDLER and END HANDLER Statements

The block of statements that is the exception handler is set off from the rest of your program by the statements HANDLER and END HANDLER.

An exception handler for our example above is:

```
600 HANDLER HNDEOF
620 IF EXTYPE<>16399 THEN GOTO 680
640 PRINT "REACHED END OF FILE.";COUNT;"
NAMES IN FILE"
660 CONTINUE
680 PRINT "FILE ERROR: NOT EOF"
700 CONTINUE
720 END HANDLER
```

Let us temporarily disregard the statements on lines 620 through 700, which make up the body of the handler. We will discuss them below. We are concerned here with statements 600 and 720.

The HANDLER statement on line 600 marks the beginning of the exception handler, and also indicates the name (in this case HNDEOF) by which the handler can be referenced from the rest of the program. The name of the handler must begin with an alphabetic character, which can be followed by any combination of alphabetic and numeric characters, to a maximum of six characters.

The END HANDLER statement marks the end of the error handling code.

An exception handler can be placed anywhere in your program; the block of statements beginning with HANDLER and ending with END HANDLER will not be executed if they are encountered in the normal numerical sequence of your program; they are intended to be executed only when an exception occurs. You should not transfer control into a handler by a GOTO statement from somewhere outside the handler.

Enabling and Disabling the Handler: The ENABLE HANDLER and DISABLE HANDLER Statements

The second requirement in implementing an exception handler, as we mentioned above, is to temporarily divert control from the default MP/BASIC error-handling procedures to your exception handling code. The statement

```
355 ENABLE HANDLER HNDEOF
```

activates the handler named HNDEOF. Whenever an exception occurs, then control will be transferred to that handler. If another handler is already enabled, it is disabled and the one mentioned in the new ENABLE statement becomes effective.

The statement

```
365 DISABLE HANDLER
```

deactivates the current exception handler, if any, and enables MP/BASIC's default exception handling code. Notice that the name of the current handler is not included as part of the DISBLE HANDLER statement.

If we add the ENABLE HANDLER and DISABLE HANDLER statements to our opening example, then all errors occuring between statements 355 and 365 are directed to the exception handler named HNDEOF. We should emphasize here that control will be passed to HNDEOF in *all* cases of error—not just in the end-of-file case we are specifically looking for here. This means that our exception handler must be written to handle all possible error conditions, a requirement that we discuss in more detail below. We need to keep this requirement in mind, however, when enabling the handler, and to leave the handler enabled only for the statement(s) in which we expect our particular exception to occur.

You can have several different exception handlers in your program, each with a different name. You can also enable a single handler at more than one point in your program. Hovever, only one handler may be in effect at a time. Also, you cannot nest handlers (that is, you cannot have an ENABLE HANDLER statement within a HANDLER...END HANDLER block).

When your program is not running, MP/BASIC does all the exception handling.

Determining the Type of Error: The EXTYPE Function

Now let us consider the statements within handler HNDEOF in the example above. Statement 620 determines if the error is an end-of-file error. It does this by examining the error type, which is a code corresponding to the kind of error that occurred. (Error types are summarized in Appendix A.) The function EXTYPE returns as a value the error type. End-of-file is a system error, with corresponding error code 16399. Thus, the statement on line 620 tests to see if the exception was something other than an end-of-file; if so, then it sends control to line 680. If the exception is an end-of-file, then statement 640 prints an appropriate message, and passes control to line 660. The CONTINUE statement returns control to the main program, at the next line after the one in which the exception occurred.

Exiting from the Handler

There are four ways to exit from an error handler, and each has a different effect on where execution continues after the handler is done.

The first option is in lines 660 and 700 of the example above. The CONTINUE statement continues execution at the line following the one that caused the exception (in this case, the exception occurred at line 360, so control would pass to line 365).

In some situations, you might want to reattempt the statement that caused the exception. In these cases, you would use a statement

```
660 RETRY
```

This second method of exiting from an error handler returns control to the main program at the same line in which the exception occurred.

A third exit option is to use the RESUME statement, which causes execution of the main program to continue at a line which is determined when you enable the handler. Using RESUME requires an extended form of the ENABLE HANDLER statement. Consider, for example, the statement

```
555 ENABLE HANDLER CH104, RESUME AT 800
```

in the main program and the statement

```
940 RESUME
```

within the exception handler named CH104. Execution of line 940 would pass control to line 800.

The final exit option is the END HANDLER statement. We have already seen that this statement marks the last line of the handler's instructions. If the END HANDLER statement is encountered in the normal course of execution, then the handler is disabled and control goes to the next statement in numerical sequence after the END HANDLER statment.

Determining the Location of an Error: the EXLINE Function

When an exception occurs, the function EXLINE has as a value the line number of the line in which the exception occurred. For example, executing the statement

```
650 PRINT "ERROR OCCURRED ON LINE";EXLINE
```

within the exception handler HNDEOF will print the following message upon the occurrence of end-of-file:

ERROR OCCURRED ON LINE 360

Generating an Exception: The CAUSE Command

The statement

```
315 CAUSE 16399
```

would generate an end-of-file condition. Likewise, using any exception code listed in Appendix A in place of the 16399 would generate the corresponding error condition. Thus, CAUSE is useful in debugging a program that contains exception handlers.

As we mentioned above, an exception handler must provide a way of processing all possible errors—not just the one or two conditions you are specifically looking for. Suppose, for instance, that the handler HNDEOF was reached because one of the names in file NAMES was too long for the variable LAST$, instead of because of an end-of-file. In this case, the program would transfer control to HNDEOF, which would print the message *FILE ERROR: NOT EOF* and return control to line 380.

Another way of handling this situation is to refer all errors other than the one(s) we are specifically interested in back to the default MP/BASIC error handler. We can do this by using the following statements:

```
670 PRINT "ERROR ON LINE";EXLINE
680 DISABLE HANDLER
690 CAUSE EXTYPE
```

By deactivating our exception handler, and using the CAUSE statement to simulate the same error condition which originally passed control to HNDEOF, we can send all error situations other than end-of-file back to the default handler.

The statement on line 670 above prints the line on which the original error occurred, since the default handler will return the line number as 690.

Handling ^C^A Interrupts

Exceptions generated by the CTRL-C, CTRL-A sequence (or the ESC key, if the CHARACTERISTICS/ON/ESC command was given prior to entering MP/BASIC) are handled separately from the other exceptions.

The statement

```
220 ENABLE KEY HANDLER HNDESC
```

activates the handler named HNDESC to respond to a CTRL-C, CTRL-A sequence. The statement

```
250 DISABLE KEY HANDLER
```

deactivates the key handler.

```
30 INPUT PROMPT "TYPE S TO START, E TO END:":S$
40 IF S$="S" THEN GOTO 80
50 IF S$="E" THEN GOTO 190
60 PRINT "PLEASE ENTER ONLY S OR E"
70 GOTO 30
80 ENABLE KEY HANDLER HNDKEY,RESUME AT 30
90 LET COUNT1=0
100 LET COUNT1=COUNT1+1
110 LET COUNT2=0
120 FOR I=1 TO COUNT1
130 LET COUNT2=COUNT2+1
140 NEXT I
150 GOTO 100
190 STOP
300 HANDLER HNDKEY
305 PRINT
310                PRINT               "^C^A:
COUNT1=";COUNT1;"COUNT2=";COUNT2
320 PRINT "RETURNING TO BEGINNING"
330 DISABLE KEY HANDLER
340 RESUME
350 END HANDLER
360 END
```

The key handler operates independently from any other handlers in your program. That is, you can have both a key handler and a (regular) handler enabled at the same time.

Keywords in Chapter 11

Table 11-1 lists the keywords introduced in Chapter 11.

Table 11-1. Keywords in Chapter 11

Keyword	Can be used in	
	Program Statement	**Immediate Mode**
CAUSE	YES	No
CONTINUE	YES	NO
DISABLE HANDLER	YES	NO
DISABLE KEY HANDLER	YES	NO
ENABLE HANDLER	YES	NO
ENABLE KEY HANDLER	YES	NO
END HANDLER	YES	NO
EXLINE	YES	YES
EXTYPE	YES	YES
HANDLER	YES	NO
RESUME	YES	NO
RETRY	YES	NO

End of Chapter

Chapter 12
Using Assembly Language Subroutines with MP/BASIC

This chapter describes how to call assembly language subroutines from an MP/BASIC program. In order to use this feature of MP/BASIC, you must be an experienced assembly language programmer.

You may want to call an assembly language subroutine directly from an MP/BASIC program. For example, you might have a special device (such as an alarm or a temperature sensor) on your system, for which the device driver exists only as an assembly language subroutine. Or, you might need to do a mathematical or logical operation that could be more efficiently performed in assembly language.

Calling an Assembly Language Subroutine: The SUMMON Command

Suppose you have an assembly language program named CRINKL. You can call that program from MP/BASIC by using the statement

```
100 SUMMON "CRINKL"
```

SUMMON can be used either directly from the terminal (in immediate mode) or as a program statement.

You can also pass parameters to the assembly language subroutine, for example:

```
100 SUMMON "BUNGLE" [12, FADDR$, FDATE]
```

This statement calls the assembly language routine BUNGLE with three parameters:

- the numerical constant 12
- the string variable FADDR$
- the numerical variable FDATE.

Note the use of square brackets here. In our notation in this manual, we ordinarily use square brackets to indicate an optional part of a statement. Parameters are, indeed, optional in a SUMMON statement; however, if they are included they are enclosed in square brackets rather than in parentheses.

You can pass as parameters any legal MP/BASIC expression. You can also pass an array as a parameter. To pass an array as a parameter, use the syntax for *formal arrays*. A formal array is simply a means of referring to an entire array with a single construct; the format is the array name followed by adjacent paired parentheses. For example:

```
40 DIM LIST$(5,10)
50 SUMMON "FILL_LIST$" [A$,LIST$(),7]
```

Writing a Subroutine to be Called from MP/BASIC

The assembler processes a source file (containing your assembly language instructions) and produces an object file (containing absolute, or relocatable, code). Then the binder processes the object files of all the assembly language subroutines that are to be executed together, and produces a program file that the computer can execute.

We will first describe how you write an assembly language subroutine so that it can be called from MP/BASIC, and then tell how to include it in the BIND line.

We will use an example of an assembly language routine that can be called from MP/BASIC to illustrate some points you should know when writing a subroutine of your own.

Figure 12-1 shows a routine that performs a bit-by-bit logical AND on two integers. This operation is not directly available within MP/BASIC (the MP/BASIC operation AND returns a value of 0 or 1 based on each of the two integers as wholes, whereas the assembly language instruction AND produces an integer, each bit of which is the logical AND of the corresponding bits in each of the two operands).

```
        .TITLE  LANDER  ;Logically AND two integers

        .ENT    ?SUM

        .NREL

;This routine will take two integers passed as arguments, AND them
;together, and return the result in a third integer variable.
;It is called from MP/BASIC with the statement
;               100 SUMMON "ANDER" [A,B,C]
;where A, B, and C are all integer variables. A and B are the integers
;to be ANDed, and the result is returned in C.



;Subroutine table
?SUM:   ?SUMMON ANDER
        -1              ;end of subroutine table

ANDER:  SAVE    0       ;save the accumulators and the return address

;Get the two arguments passed from MP/BASIC

        LDA     0,@?AR01,3      ;AC0 = "A"
        LDA     1,@?AR02,3      ;AC1 = "B"

;Perform the AND and return the result to MP/BASIC

        AND     1,0             ;AC0 = "A" AND "B"

        STA     0,@?AR03,3      ;return the result in "C"
        RTN                     ;return to MP/BASIC
```

Figure 12-1. An Assembly Language Subroutine that Can Be Called from an MP/BASIC Program

In order to set up the connections between the assembly language subroutine and your MP/BASIC program, you will need to do three things when you write the subroutine:

- provide a way for your program to find the subroutine;
- provide a way to pass parameters between the subroutine and your program;
- save and restore the accumulators.

The macro definition and frame pointer offsets necessary for the first two tasks are contained in the parameter file BASIC_USER.SR, which is supplied with MP/BASIC (you include BASIC_USER.SR along with your source code when you assemble your subroutine). The definitions and offsets in BASIC_USER.SR are shown in Figure 12-2. You can obtain additional information about the usage of BASIC_USER.SR by listing the file.

Finding the Subroutine: the Subroutine Table

You provide a way for your program to find the subroutine by creating a subroutine table (also called a ?SUM table). This is an identifying table that associates the name in the SUMMON statement with the entry point of the subroutine. A ?SUM table appears at the beginning of the example in Figure 12-1.

The ?SUM table uses the ?SUMMON macro. ?SUMMON is defined in the parameter file BASIC_USER.SR

The subroutine table for the example in Figure 12-1 is included with the source code for the subroutine. If you want, you can write and assemble a ?SUM table separately and include it in the BIND line.

You can have only one subroutine table referenced in a BIND line, but it can include several subroutines, as shown in figure 12-3. In this example, the title MUSH is the name which you will later supply to the binder (if you are assembling the table separately).

```
        .TITLE  BASIC_USER ;BASIC user subroutine parameters



;The ?SUMMON macro is used to create the subroutine name table.
;Remember that the table must be terminated with a -1.

        .MACRO  ?SUMMON
**      .NOCON  1
**      .PUSH   .NOCON
        .+3*2   ;BA(ROUTINE NAME)
        L$      ;A(NEXT ENTRY)
**      .DO '^2'==''
        ^1      ;A(SUBROUTINE)
**      .ENDC
**      .DO '^2'<>''
        ^2      ;A(SUBROUTINE)
**      .ENDC
        .TXT    /^1/
L$:
**      .NOCON  .POP
%

;The addresses of the arguments passed to the subroutine called by
;the SUMMON statement are all placed on the hardware stack, and can
;be found by using the following symbols with the frame pointer.


.DUSR   ?AR01=  -5      ;Offset to the first argument
.DUSR   ?AR02=  ?AR01-1 ;Offset to the second argument
.DUSR   ?AR03=  ?AR02-1
.DUSR   ?AR04=  ?AR03-1
.DUSR   ?AR05=  ?AR04-1
.DUSR   ?AR06=  ?AR05-1
.DUSR   ?AR07=  ?AR06-1
.DUSR   ?AR08=  ?AR07-1
.DUSR   ?AR09=  ?AR08-1
.DUSR   ?AR10=  ?AR09-1

;The address of argument eleven is found by subtracting one from
;?AR10, twelve is found by subtracting one from argument eleven and so on.
```

Figure 12-2. Offsets and Macro Definitions from BASIC_USER.SR (continues)

```
;These parameters define the area pointed to by AC2 on the
;successful completion of a ?FVAR call or by the addresses in the list
;passed by the SUMMON statement. (See above)

;Definition of the area for string variables.

.DUSR   ?DTCL=  0       ;offset to current length
.DUSR   ?DTML=  1       ;offset to maximum length


;Definition of the area for array variables.

.DUSR   ?DTD1=  0       ;offset to dimension 1
.DUSR   ?DTD2=  1       ;offset to dimension 2


;This parameter defines the offset to the start of the data for
;both strings and arrays.

.DUSR   ?DTAD=  2       ;offset to the array or string data
```

Figure 12-2. Offsets and Macro Definitions from BASIC_USER.SR (concluded)

```
                        .TITL MUSH

        .ENT ?SUM     ;Table name must be ?SUM

 ?SUM:  ?SUMMON CRINKL      ;This is the entry point for the routine CRINKL

        ?SUMMON ROUT2       ;This is the entry point for the routine ROUT2

        ?SUMMON ROUT3       ;This is the entry point for the routine ROUT3

        -1                  ;The table must be terminated with a -1.
```

Figure 12-3. A ?SUM Table for Three Subroutines

Passing Parameters

Notice the offsets used in ANDER (?AR01, ?AR02, and ?AR03) to retrieve the addresses of the passed parameters. These frame pointer offsets are necessary for your subroutine to retrieve the addresses of the passed parameters; they are defined in the parameter file BASIC_USER.SR (see Figure 12-2).

The hardware stack pointer points to the address of argument one, the value of the stack pointer minus one points to the address of argument two, and so forth. The address of the argument depends on the type of data in the argument (for more information on the internal representation of data, see Appendix C). For a REAL or INTEGER number, the address is the address of the data. For a string, the address is the address of the two-word string descriptor that precedes the string. For an array, the address is the address of the two-word array header that precedes the array data.

Constants and the results of expressions can be passed provided that there is a sufficient amount of memory space available that is unused by either the program or data sections. Numeric results require two words, and the address passed to the user is the address of the data. String constants require two words plus enough words to contain the string characters, and the address passed is similar to that for string variables (except that the offset that normally contains the maximum string length will contain a 0). Substrings cannot be passed to subroutines.

Saving and Restoring the Accumulators

When control is transferred to the subroutine, the accumulators will contain the following:

AC0 0

AC1 undefined

AC2 the address of a word containing the number of parameters in the SUMMON statement; if there are no parameters, then this word will contain 0

AC3 the return address

The first instruction in your subroutine should save the contents of the accumulators (use SAV for MP/OS, or SAVE for MP/AOS, MP/AOS-SU, AOS, and AOS/VS). Likewise, the last instruction at each exit of your subroutine should reload the accumulators (use RET for MP/OS, RTN for MP/AOS, MP/AOS-SU, AOS, and AOS/VS).

Returning an Exception Code

In addition to the three primary tasks mentioned above, you might also want to provide a way for your subroutine to indicate that an error or exceptional condition has occurred.

One way to do this is for your subroutine to pass the name of a variable in the parameter list. The subroutine can then place the exception code in the variable, and the MP/BASIC program can check it when the subroutine returns. This means that you must include an error-parameter in the SUMMON statement in your program. This method of passing an exception code is completely controlled by the user.

An alternate method of returning an exception code takes advantage of the fact that MP/BASIC has special conventions for the use of AC0 upon entering and leaving a subroutine. Recall that, when control passes from MP/BASIC to the assembly subroutine, AC0 has been set to 0. When the subroutine is finished, if AC0 contains anything other than a 0, MP/BASIC will act as if an error occurred in the SUMMON statement. This error can then be handled by the MP/BASIC exception handling facilities.

Also, recall that the RET (or RTN) instruction will reload the contents of the accumulators that were saved by the SAV (or SAVE) instruction. In order for the subroutine to pass back a value in AC0, you must use a code sequence similar to the following (assume AC1 contains the exception code):

```
LDA    3,41         ;Load AC3 with the frame pointer
STA    1,?OAC0,3    ;Store exception code for return to
RTN                 BASIC
                    ;Return to MP/BASIC
```

?OAC0 is defined in MPARU.SR, so when the subroutines are assembled, you should include MPARU.SR/S before BASIC_USER.SR in the command line to the assembler. For example:

```
X MASM MPARU.SR/S BASIC_USER.SR/S <user
routine>
```

Creating the Program File

After you have assembled your subroutine(s) and table, you can create your BASIC.PR file by an EXECUTE MBIND command to the CLI. The exact format of this command will depend upon the operating system, whether you want a version with or without overlays, whether or not your computer has hardware/firmware floating point arithmetic and whether or not you want a run-only version.

The command lines for each situation are presented below. The file names must be listed in the same order as shown in each command. The notation <user subroutines> in each command is where you should insert the list of names of the subroutine(s) (.OB's) to be bound into MP/BASIC (be sure to include the name of the file containing the subroutine table).

The commands below all assume hardware/firmware floating point. For software floating point, substitute SWBASIC wherever HWBASIC occurs.

MP/OS Overlays Not Run-only

```
EXECUTE MBIND/N/REV=3.00/L=BASIC.MP&
/P=BASIC,HWBASIC1.LB, <user subroutines>,&
!* HWBASIC2.LB ! HWBASIC3.LB *! HWBASIC4.LB,&
MSL.LB,INIT
```

MP/OS No Overlays Run-only

```
EXECUTE MBIND/N/REV=3.00/L=BASIC.MP&
/P=BASIC,HWBASIC1.LB, <user subroutines>,&
HWBASIC3.LB,ROBASIC,HWBASIC4.LB,&
MSL.LB,ROINIT
```

MP/OS No Overlays Not Run-only

```
EXECUTE MBIND/N/REV=3.00/L=BASIC.MP&
/P=BASIC,HWBASIC1.LB, <user subroutines>,&
HWBASIC2.LB,HWBASIC3.LB,HWBASIC4.LB,&
MSL.LB,INIT
```

AOS Overlays Not Run-only

```
EXECUTE MBIND/N/REV=3.00/AOS&
/L=MBASIC.MP/P=MBASIC,MICREM.OB,&
HWBASIC1.LB,<user subroutines>,&
!*HWBASIC2.LB!HWBASIC3.LB*!&
HWBASIC4.LB,MMSL.LB,URT.LB,INIT
```

AOS No Overlays Run-only

```
EXECUTE MBIND/N/REV=3.00/AOS&
/L=MBASIC.MP/P=MBASIC,MICREM.OB,&
HWBASIC1.LB,<user subroutines>,&
HWBASIC3.LB,ROBASIC, HWBASIC4.LB,&
MMSL.LB,URT.LB,ROINIT
```

MP/AOS Overlays Not Run-only

```
EXECUTE MBIND/N/REV=3.00/MPAOS&
/L=OBASIC.MP/P=OBASIC,HWBASIC1.LB,&
<user subroutines>,!* HWBASIC2.LB !&
HWBASIC3.LB *! HWBASIC4.LB,&
OSL.LB,INIT
```

MP/AOS No Overlays Run-only

```
EXECUTE MBIND/N/REV=3.00/MPAOS&
/L=OBASIC.MP/P=OBASIC,HWBASIC1.LB,&
<user subroutines>, HWBASIC2.LB,ROBASIC,&
HWBASIC4.LB, OSL.LB,ROINIT
```

Using Longer Names for Subroutines

Subroutine entry points are limited to six alphabetic characters. However, an option in the ?SUMMON macro allows you to call an assembly language routine by a name other than its entry point. Using this option, you can use any legal string constants as subroutine names in your MP/BASIC program. To use this option, add a second argument that specifies the entry point name if it is different from the call name. For example, the macro call

```
?SUMMON GET_REMOTE_TEMP, RMOT
```

in the subroutine table causes the MP/BASIC statement

```
100 SUMMON "GET_REMOTE_TEMP" [3, BCOUNT]
```

to transfer control to the assembly language routine whose entry point is RMOT.

Calling Routines Written in Other Languages

The SUMMON statement passes arguments using the Data General Common Language Runtime Environment (CLRE) format. This means that you can use routines written in other Data General programming languages that support the CLRE format.

For example, you can call SP/PASCAL routines from MP/BASIC. You must declare the SP/PASCAL routine to be CLRE entrant (see the *SP/PASCAL Programmer's Reference manual* for more information on external CLRE declaration). The statement

```
EXTERNAL CLRE PROCEDURE MOLE;
```

declares the SP/PASCAL routine MOLE to be CLRE entrant.

End of Chapter

Chapter 13
Summary

This chapter summarizes the key points in Chapters 1 through 12. Together with the information contained in Chapter 14, "Dictionary of Statements, Commands, and Functions" provide the experienced BASIC programmer with a complete overview of MP/BASIC.

Fundamentals of Programming (Chapter 1)

Characters

Figure 1-1 contains the ASCII character code with decimal, octal, and hexadecimal notation.

Line Numbers

Line numbers may range from 1 to 65534.

Logging On and Off

To enter MP/BASIC from MP/OS, type

```
BASIC ↵
```

The commands for reaching MP/BASIC from other operating systems are listed in Chapter 1.

An asterisk (*) prompt will appear at the beginning of the line upon completion of the log-on procedure and throughout the MP/BASIC session to indicate that the system is ready to accept instructions.

Log off by typing BYE.

Clearing Memory Area

To clear memory area, type the command NEW before typing new programs or recalling old ones. To obtain current information about the number of memory words used and the number still available, type the command MEMORY.

Program Organization

Use the REM statement for explanatory comments.

Typing a line number followed by NEW LINE will cause that line to be deleted. The DELETE command, followed by a line number, will also cause that line to be deleted. To delete several lines, use the DELETE command with specific line numbers, and with the related keywords FIRST, TO, and LAST as needed.

Current program lines may be deleted by using the CTRL-C, CTRL-A keys (displayed on the console as ^C^A). MP/BASIC will respond with a prompt.

To use the ESC key instead of CTRL-C, CTRL-A, type the following before logging on to BASIC:

```
CHARACTERISTICS/ON/ESC
```

You can use the DEL key to back up to a typing error on the current line, and then retype the line from the correction to the end.

Program lines may be renumbered by means of the RENUMBER command and its related keywords, AT and STEP.

Default values are 100 for the first line number and 10 for the STEP increment.

The END statement is mandatory in ANSI Minimal BASIC, but optional in MP/BASIC.

Naming a Program

See "Legal Filenames" in Chapter 9 for a description of legal filenames.

Saving and Retrieving a Program

To save a program, type

```
LIST "filename"
```

where "filename" is the name of your program.

To retrieve a program, type

```
ENTER "filename"
```

where "filename" is the name of your program.

Interrupting Program Execution

Interruptions to program execution may be built into the program by means of STOP statements. The command CON allows program execution to continue from the point of interruption.

Program execution may be stopped at any time by striking the CTRL-C, CTRL-A keys. The program may then be re-excuted by means of the RUN command, or execution may be resumed from the point of interruption by using the CON command.

The Printed Output

Numbers are printed with leading spaces reserved for the sign. Negative signs are always printed; positive signs are not.

Unless otherwise stated (see DECLARE INTEGER in Chapter 14), all numbers are treated as REAL numbers.

There are five print zones of 14 characters each.

The spacing of printed output is determined by the punctuation separating items on a print line: a comma sends the output to the next available print zone, while a semicolon results in adjacent printing of items.

The TAB function allows spacing of output anywhere along the line.

No carriage return and line feed are output if a print list ends with a comma or semicolon, unless this would bring the output beyond the allowable line width.

A semicolon (;) may be used instead of the word PRINT in a PRINT command. This is an extension to the ANSI standard.

PRINT may be used in immediate mode for dynamic debugging, as well as for performing calculations unrelated to any specific program.

PRINT USING enables you to print numerical data in a formatted output (left- or right-justified; with inserted dollar signs, asterisks, commas and decimal points; suppression or printing of leading zeros; indication of positive and negative numbers; control of spacing).

Error Messages

Error messages are followed by a display of the erroneous line, beneath which a caret (^) localizes the area where the mistake occurred.

Numeric Expressions: Variables, Constants, and Operators (Chapter 2)

Numeric Variables

Numeric variables can be simple or subscripted.

Naming Numeric Variables

Numeric variable names may be composed of alphabetic characters, digits, and the underscore character; the first character must be alphabetic. For compatability with AOS/VS BASIC, we suggest a maximum of 32 characters in a variable name.

Table 2-1 illustrates valid and invalid numeric variable names.

Assigning a Value to Numeric Variables

This is done by means of the LET statement, (Chapter 2), the READ/DATA or INPUT statements (Chapter 4), or the READ FILE statement (Chapter 9).

Initializing Numeric Variables

Do not reference numeric variables before assigning them a value or initializing them to zero.

Numeric Constants

Numeric constants can be positive or negative. The positive plus (+) sign is optional, whereas the minus (−) sign must always be included.

The range of numeric constants is −7.237E+75 to +7.237E+75. In entering constants in scientific notation, you can use exponents as high as +99 or as low as −99 as long as the limits of +7.237E + 75 or −7.237E + 75 are not exceeded.

Writing Numeric Constants

Numeric constants may be written in decimal notation or in scientific notation.

In decimal notation, the value of numeric constants is truncated to 15 decimal digits.

Numeric Data Types

MP/BASIC supports numeric data of type INTEGER, in addition to those of type REAL supported by ANSI. (Default type is REAL.) The range of integer values is −32,768 to +32,767, inclusive. You may specify part or all of your data to be of either type by using one of the following statements:

line no. DECLARE INTEGER *list of data*
line no. DECLARE REAL *list of data*

where *list of data* may be numeric variables, dimensioned strings, or arrays. These statements may also serve as dimensioning statements for strings and arrays.

If you wish all your data to be of a given type, use one of the following statements:

line no. DECLARE ALL INTEGER
line no. DECLARE ALL REAL

These statements may not be followed by a data list.

REAL numbers can be single- or double-precision. The default is single-precision. You can specify the precision of REAL numbers by using the following forms of the DECLARE statements:

line no. DECLARE REAL*n *list of data*
line no. DECLARE ALL REAL*n

where *n* is 4 for single-precision and 8 for double-precision.

Working with integers results in faster arithmetic operations and loop executions and saves memory space.

Only the variables following the DECLARE... statements are affected by them.

Arithmetic Operators

The standard arithmetic operators and the rules of operator precedence are summarized in Tables 2-2 and 2-3, respectively.

Character Strings (Chapter 3)

Character strings in the form of string variables and string constants may be used as data.

A character string may consist of any of the following, in any combination:

- uppercase alphabet characters, A through Z;
- lowercase alphabet characters, a through z;
- digits 0 through 9;
- one or more of the characters in Table 3-1.

String Variables

Naming String Variables

String variable names may be composed of alphabetic characters, digits, and the underscore character, and must be terminated by a dollar sign; the first character must be alphabetic. This is an MP/BASIC extension to ANSI Minimal BASIC, which only permits use of the 26 letters of the alphabet followed by a dollar sign. For compatability with AOS/VS BASIC, we suggest a maximum of 32 characters in a variable name.

Assigning Values to String Variables

Values are assigned to string variables by means of the LET, INPUT, LINPUT, and READ/DATA statements.

Length of String Variables

String variables are automatically initialized to null, i.e., to a length of 0.

ANSI Minimal BASIC specifies 18 characters as the maximum length of a string variable. MP/BASIC allows you to create string variables of indefinite length.

MP/BASIC default length of string variables is 18. Longer string variables must be dimensioned, and shorter ones may be dimensioned, using the DIM statement. For example,

```
DIM A$ * 25, B$ * 8
```

Referencing Substrings

MP/BASIC allows you to reference portions of a string variable. A statement such as

```
20 PRINT A$(3:6)
```

references a portion of A$ beginning at the third character within A$, and ending with the sixth.

If you request a substring longer than your string, the excess length will be ignored, and the substring will terminate at the last character in your string. For example:

```
60 A$ = "SUPERSTAR"
70 ; A$(6:11)
```

will produce a printout reading

STAR

If the beginning of the substring is set at character 0 of the string, BASIC displays a substring beginning at the first character of the string. Using the previous example, if we type

```
70 ; A$(0:5)
```

BASIC will display

SUPER

You can also assign a value to a selected portion of a string. For example:

```
LET ITEM$ = "JELLYROLL"
LET ITEM$(6:9) = "JAR "
PRINT ITEM$
```

will produce the output

JELLYJAR

If the beginning of your substring is the last character of the string, or if the starting and ending character positions are reversed, a null string is returned.

String Constants

The length of a string constant must not exceed the length of a program line, i.e., 156 characters.

Writing Character Strings: The Use of Quotation Marks

Quotation marks around character strings are mandatory in some cases, optional in others. Table 13-1 summarizes these conventions. See also Table 4-2, which lists all quoted characters.

Table 13-1. Use of Quotation Marks with String Characters

Quotes Optional	Quotes Mandatory
A period (.) A plus sign (+) A minus sign (-) REM statements	When string contains comma (,) or other special characters; see Table 3-2 for complete list To preserve leading or trailing spaces When string contains embedded quotes

String Operations

The ampersand (&) is the concatenation operator; for example:

```
CSTR$ = ASTR$ & BSTR$
```

This is an MP/BASIC extension to the ANSI standard.

INPUT and READ DATA (Chapter 4)

INPUT

This statement can take several numeric and string variables, separated by a comma. You must enter the corresponding values in the same order in which they are requested by the INPUT line.

Excess values as well as insufficient values will generate an error message. In that event, a new question mark will appear, and the entire list of values must be re-entered.

The following variations of the INPUT statement are extensions of the ANSI standard.

The INPUT PROMPT statement allows the inclusion of a prompt message in the INPUT statement. Upon execution, the prompt message is printed before the input is accepted. The prompt message can be any string expression.

The LINPUT statement takes only string arguments, assigning them to a string variable. It accepts a line of data literally, including characters normally receiving special treatment, e.g., comma, embedded quotes, or quoted characters. The NEW LINE, carriage return, form feed, and null characters are recognized as delimiters.

INPUT and LINPUT may also be used with file operations. (See Chapter 9.)

READ and DATA

Variables appearing on a data line are separated by commas; the order of their appearance must match the order in which they are to be read.

It is permissible to split data among several data lines, provided the order of enumeration matches the order of reading.

Data lines should be placed close to their corresponding read statements.

Insufficient data will generate an error message; excess data are ignored.

RESTORE

Used without arguments, this keyword resets the pointer to the beginning of the first DATA statement in the program. If followed by a program line number, for example,

```
*200 RESTORE 120
```

this keyword will reset the pointer to the beginning of the data line whose number you have specified. Successive reads can thus begin anywhere you wish. This feature is an extension to ANSI Minimal BASIC.

Control Statements; Flowcharts (Chapter 5)

Unconditional Branching

This type of branching results from the use of the GOTO... statement.

Conditional Branching

The IF...THEN...ELSE statement makes branching contingent on the result of relational operations performed on numeric and string expressions.

The ELSE clause (here as well as in the ON...GOSUB...ELSE and ON...GOTO...ELSE statements) is an MP/BASIC extension to the ANSI standard.

Relational Operators

Table 5-1. summarizes the six relational operators, all of which may be applied to numeric variables.

MP/BASIC extends the ANSI standard by accepting the formats

=> for >=

=< for <=

To be considered equal, two strings must contain an identical sequence of characters, including spaces. String comparisons are sensitive to the difference between upper- and lowercase alphabetic characters.

Nested IF...THEN...ELSE statements are permitted.

Multiple Branching

The ON...GOTO...ELSE... statement permits the selection of the appropriate branch from among several choices.

Loops

Looping can be controlled by means of a counter or by means of the FOR and NEXT statements.

Nested loops as well as simple loops are permissible.

The optional STEP keyword specifies the size of the FOR...NEXT... increment (the default increment is 1). The increment specified by STEP can be positive (for an ascending FOR...NEXT sequence) or negative (for a descending FOR...NEXT sequence).

Declaring the initial value, the counter variable, the increment, and the limit of a FOR...NEXT statement as integers (see Chapters 2 and 14), speeds execution of the loop.

Flowchart Symbols

These are summarized in Figure 5-9.

Logical Operators

The logical operators in MP/BASIC are AND, OR, and NOT.

Subroutines (Chapter 6)

Branching to and returning from a subroutine are controlled by the GOSUB... and RETURN statements, respectively. The GOSUB may be part of an ON...GOSUB...ELSE statement.

Subroutines can be nested to nine levels.

Subscripted Variables (Chapter 7)

One- and two-dimensional arrays may be created with numeric or string variables. You cannot combine numbers and strings within a single array.

Naming Arrays

MP/BASIC expands the ANSI standard by allowing as an array name any name which is valid as the name of a string or numeric variable.

You may not use the same name for both a simple variable and an array within the same program. Neither can you use the same name for both a one-dimensional array and a two-dimensional array within the same program.

Array Size

The DIM statement determines the upper bound of an array. When you declare a two-dimensional array, both subscripts must be included in the DIM statement. You need not use all the elements you have reserved in the dimensioning declaration.

A one-dimensional array has a default size of 11 elements with subscripts ranging from 0 through 10. A two-dimensional array has a default size of 121 elements. Dimensioning declarations are optional for arrays that do not exceed default size.

Each array must be dimensioned separately, but several arrays may be dimensioned in the same statement. The ANSI standard requires DIM statements to appear on a lower numbered program line than the reference to the array. MP/BASIC expands on the standard by specifying only that program execution pass through the DIM statement before the array is referenced.

An array can be dimensioned only once in a program.

The OPTION BASE statement determines the lower bound of an array as 0 or 1; the default lower bound is 0.

Functions (Chapter 8)

MP/BASIC offers the user implementation-defined functions as well as the option of creating user-defined functions.

Implementation-Defined Functions

This group includes mathematical and string functions.

Mathematical Functions

Table 8-1 summarizes the implementation-defined mathematical functions. Starred functions represent the MP/BASIC expansions of the ANSI Minimal BASIC standard.

Mathematical functions are executed in the following order: operations within the argument of a function are done first; next the function is evaluated; thereafter, all other arithmetic operations in the statement are done in their normal order of precedence.

String Functions

Table 8-2 summarizes the implementation-defined string functions, all of which are extensions of the ANSI standard.

User-Defined Functions

A user-defined function can return only a single value. A user-defined function definition is limited to one statement typed on a single line.

Defining a Function

The DEF statement declares and defines a function created by the user. ANSI requires that a function be defined on a lower-numbered line than its reference; MP/BASIC specifies only that program execution pass through the DEF statement before the function is referenced.

User-defined functions may be redefined in a program.

A user-defined function definitions may refer to other functions but not to the function currently being defined.

Naming the Function

A user-defined function name consists of the characters FN followed by a single alphabetic character.

A maximum of 26 functions, (FNa...FNz) may be defined by the user.

Arguments

User-defined functions may contain a single, optional argument.

File Input and Output (Chapter 9)

MP/BASIC file operations are an extension of the ANSI standard. File operations include creating a new file, writing into it, appending new data, reading the file, and printing its contents. The user may have up to eight files open at any one time.

File attributes such as filename and filenumber are described in the OPEN FILE statement in Chapter 14.

File Modes

The mode in which a file is opened determines the kind of operation that can be performed with that file as well as the type of access permitted to it. Some files may be accessed sequentially, while others allow random as well as sequential access.

Four possible file modes are expressed by numbers ranging from 0 to 3. See Table 9-1 and the OPEN FILE statement in Chapter 14.

File Records

MP/BASIC files can contain two types of records: fixed-length records and variable-length records. These types may not be mixed within a single file.

Records may contain alphanumeric data.

All data in a single record must be processed in a single write or read statement. The first record location within a file is record number 0.

Record length is expressed in terms of bytes and may range from 0 to 32,767 bytes.

When record size is declared and data are written into a specific record location, any empty bytes in that record are left vacant.

Examples of random access are provided in Chapter 14 in connection with the WRITE FILE and READ FILE statements.

Data Formats of Files

MP/BASIC supports two formats for data in files: binary format and ASCII format.

READ FILE and WRITE FILE read and write data from and to files in binary format. INPUT FILE, LINPUT FILE, and PRINT FILE read and write data from and to files in ASCII format.

File Data in ASCII Format

To write data into a file in ASCII format, use the PRINT FILE command. Files of this type can be read with the INPUT FILE and the LINPUT FILE statements, but not with the READ FILE statement.

INPUT FILE and LINPUT FILE are forms of (and therefore function like) the INPUT and LINPUT commands described in Chapter 4. (INPUT and LINPUT read data from your terminal; INPUT FILE and LINPUT FILE read data from a file).

To be read with the INPUT FILE, the file must match the appearance of a user-entered INPUT statement: literal commas must be entered between data items when the file is created, if the INPUT FILE statement requests several variables.

LINPUT FILE accepts only strings as arguments.

Program Segmentation (Chapter 10)

Program segmentation is an MP/BASIC extension to the ANSI standard.

Program segments are stored (by the SAVE statement) in a special format which includes a data area. This format is compatable with all the segmentation commands.

You can load the first segment by using the LOAD command.

You can chain from segment to segment by the CHAIN statement (which passes control to the next segment with no explicit provision for returning to the current segment) or the SWAP statement (which returns control to the current segment upon completion of the SWAPped segment).

The SAVEd data area is loaded back into your work area each time you LOAD, CHAIN, or SWAP a segment; thus, you can preserve values of variables between occurrences of a segment.

There is independence between variables used in linked segments. You transfer data between linked segments through files. CHAIN and SWAP do not close any open files or reset any file pointers when chaining between segments.

Exception Handling (Chapter 11)

User-defined exception handlers are an MP/BASIC extension to the ANSI standard.

The block of statements comprising an exception handler is set off by the HANDLER and END HANDLER statements.

Exceptions that can be handled by user-defined handlers are in two categories: the KEY exception, which is the ^C^A interrupt and all other exceptions.

Only one handler can be enabled at one time to handle each category. If no user-defined handler is enabled for a category, then the MP/BASIC default error handler is enabled.

The ENABLE [KEY] HANDLER and DISABLE [KEY] HANDLER statements enable and disable a handler, respectively.

Handlers cannot be nested.

Control is returned from an exception handler by a RETRY, RESUME, or CONTINUE statement.

The CAUSE statement simulates an error condition.

The functions EXTYPE and EXLINE return, respectively, the exception code (listed in Appendix A) and the line number of the statement where the exception occurred.

Using Assembly Language Subroutines with MP/BASIC (Chapter 12)

Calling assembly language subroutines from MP/BASIC is an extension to the ANSI standard.

The SUMMON statement enables you to call an assembly language subroutine directly from an MP/BASIC program.

You can, optionally, pass parameters (including arrays) to and from a subroutine.

To be called from an MP/BASIC program, an assembly language subroutine uses a subroutine table, which must be included with the subroutine in the BIND line.

The subroutine table macro definition and the frame pointer offsets for passing parameters are included in file BASIC_USER.SR, which is supplied with MP/BASIC.

You can call a subroutine from MP/BASIC by a name that is longer than the six-character upper limit on assembly language subroutine entry points. To do this, use an option for the ?SUMMON macro when you construct the subroutine table.

This facility can also be used to call routines written in other Data General programming languages that support the CLRE format.

End of Chapter

Chapter 14
Dictionary of Statements, Commands, and Functions

This chapter lists in alphabetical order all statements, commands, and functions. In each case, the purpose is stated and the format is given, with an explanation of the arguments used. Where appropriate, examples are supplied.

Starred functions and commands (i.e., *DEG) represent extensions to ANSI Minimal BASIC (note that the * is not a part of the statement or function).

We use the following conventions in the format portion of each description:

KEYWORD required , (choice1 | choice2) *[,(option1 | option2 | option3)]...*

where — means

KEYWORD — You must type the KEYWORD exactly as shown.

required — You must enter some argument here.

[option1 | option2 | option3] — You have the option of entering the material enclosed in the brackets. Don't enter the brackets; they only set off what is optional (the exceptional cases where the brackets are to be entered are noted in the individual descriptions).

You enter one element among the two or more choices joined by the vertical line. In the example above, you must enter a value for either choice1 or choice2, and you may enter a value for either *option1, option2, or option3.*

(choice1 | choice2) — The enclosed material constitutes an element of a statement. In some cases the parentheses should be entered as a part of the command or function. In other cases the parentheses set off an element of the command or function and should not be entered. The "Remarks" and "Examples" sections will show the appropriate use in each case.

... — You may repeat the preceding entry or entries. The "remarks" and "Examples" sections will tell you exactly what you may repeat.

In the examples, an asterisk (*) at the beginning of a line indicates the MP/BASIC prompt and a single right parenthesis ( )) indicates the CLI prompt.

ABS(X)

Format

ABS (expr)

Returns the absolute (positive) value of expr.

Arguments

expr is a numeric expression.

Examples

```
*LIST
10 PRINT ABS(−30)
*RUN
30
*
```

*ALL

Always used in conjunction with other keywords. See DECLARE INTEGER, DECLARE REAL.

ATN(X)

Format

ATN (expr)

Calculates the angle (in radians) whose tangent is expr. Range of the function is $-pi/2 < ATN(X) < pi/2$.

Arguments

expr is a numeric expression.

Examples

```
*LIST
10 REM-CALCULATE ANGLE WHOSE TAN=2
20 PRINT ATN(2)
*RUN
1.10715
*
```

*BSTR$(V,R)

Format

BSTR$(expr,base-expr)

Returns the string representation of the value of the binary number V in base (radix) R.

Arguments

expr is a numeric expression that evaluates to a 16-bit unsigned integer.

base-expr is a numeric expression that evaluates to an integer value in the range 2 through 16.

Remarks

BSTR$ complements the BVAL(A$,R) function.

Examples

```
BSTR$(8,2)="1000"
```

*BVAL(A$,R)

Format

BVAL(str-expr,base-expr)

Returns the 16-bit unsigned binary value (in base R) of the number represented by the string B$.

Arguments

str-expr is a string expression that evaluates to the string representation of a number in base (radix) base-expr.

base-expr is a numeric expression that evaluates to an integer value in the range 2 through 16.

Remarks

BVAL complements the BSTR$ function.

When evaluated, str-expr can consist only of the characters that are valid as numbers in base R. For example, if R=2 (binary) then the string can consist of characters 0 and 1, if R=16 (hexadecimal) then the string can consist of characters 0 through 9 and A through F.

If you PRINT the value of BVAL, it will look like a decimal number. This is because the PRINT command treats the binary value returned by BVAL as a decimal INTEGER. Thus, BVAL("5B",16) has a value "01011011", but would be PRINTed as the decimal number 91.

Examples

```
BVAL("3A",16)=00111010
```

*BYE

Format

BYE

Signs off from MP/BASIC and returns to the CLI.

Remarks

When used at the end of a BASIC session, BYE closes all open files.

Examples

```
*BYE
)
```

*CAUSE

Format

CAUSE expr

Generates an exception.

Arguments

expr is a numeric expression that evaluates to one of the error codes listed in Appendix A.

Remarks

CAUSE is useful in debugging a program containing an exception handler.

CAUSE can be used only as a statement in a program; it cannot be used as a command in immediate mode.

Examples

```
200 CAUSE 28
```

This statement simulates a file not open error.

*CHAIN

Format

CHAIN "seg-file-name"

Loads and begins execution of a program segment.

Arguments

seg-file-name is the name of the file, expressed as a string literal, in which the program segment is stored.

Remarks

CHAIN clears your work area, loads the segment from file *"seg-file-name"*, and begins execution at the segment's first line.

The program segment must have been previously stored in file "seg-file-name" by a SAVE command.

CHAIN does not close any open files or reset any file pointers when switching between segments.

CHAIN can be used only as a statement in a program; it cannot be used as a command in immediate mode.

Examples

```
CHAIN "CKWRIT"
CHAIN "SEGTWO"
```

For more extensive examples, see Chapter 10.

*CHR$(M)

Format

CHR$ (expr)

Returns the string character for which expr is the ASCII decimal representation.

Arguments

expr is a numeric expression that evaluates to a positive integer in the range of 1 to 255.

Remarks

If expr is in the range from 128 to 255, the character returned is the character corresponding to the ASCII code MOD(expr,128).

CHR$(M) complements the ORD(A$) function.

Examples

```
* LIST
10 REM CONVERT NUMERIC DATA TO STRING
20 FOR M = 65 TO 68
30 PRINT CHR$(M)
40 NEXT M
* RUN
A
B
C
D
*
```

*CLOSE FILE

Format

CLOSE *[FILE (file)]*

Disables the relationship between a filename and a file number so that the file can no longer be referred to.

Arguments

file is a numeric expression that evaluates to a number in the range from 0 to 7 and that corresponds to the number associated with a filename in the OPEN FILE statement.

Remarks

When a file is closed, the file pointer no longer exists.

You can use the CLOSE FILE statement to close a file and then reOPEN it with a new mode argument.

The CLOSE form of the statement closes all open files.

Examples

```
*150 CLOSE FILE(1)
*300 CLOSE
```

*CON

Format

CON

Continues the execution of a program after a STOP statement in the program has been executed, the CTRL-C CTRL-A keys have been struck, or an error has occurred.

Remarks

The CON command causes the continuation of the program from the point where it stopped.

If a run-time error is encountered within the program, you may correct the error and issue the CON command to begin execution from the statement following the one in which the error occurred.

Examples

```
*LIST
10 PRINT "PRINCIPAL INT(%)";
20 PRINT "TERM(YRS) TOTAL"
30 READ P,I,T
35 IF T=0 THEN GOTO 80
40 LET A=P*(1+I/100)^T
50 PRINT P;TAB(12);I;
55 PRINT TAB(21);T;TAB(32);A
60 GOTO 30
70 DATA 1000,5,10,0,0,0
80 PRINT
90 PRINT "CHANGE DATA AT LINE 70"
100 STOP
105 RESTORE
110 GOTO 10
*RUN

PRINCIPAL INT(%)TERM(YRS)TOTAL
1000 5 10 1628.89
CHANGE DATA AT LINE 70
Stop at line 100
*70 DATA 2500,3,10,1459,6,12,0,0,0
*CON
PRINCIPALINT(%)TERM(YRS)TOTAL
2500  3  10  3359.79
1459  6  12  2935.79
CHANGE DATA AT LINE 70
Stop at line 100
*
```

*CONTINUE

Format

CONTINUE

Returns control from an exception handler to the statement following the one in which the exception occurred.

COS(X)

Format

COS (expr)

Calculates the cosine of an angle expressed in radians.

Arguments

expr is a numeric expression specified in radians.

Examples

```
* LIST
10 REM PRINT COSINE OF 30 DEGREES
20 LET P = PI/180
30 PRINT COS(30*P)
* RUN
.866025
*
```

DATA

Format

DATA (val | "str lit")*[,(val | "str lit")]...*

Provides values for variables specified in a READ statement.

Arguments

val and str lit are elements that can form a list of numeric constants and string literals.

Remarks

You can use more than one DATA statement in a program.

The DATA statement is a nonexecutable statement. The values appearing in a DATA statement or statements form a single list. The first element of this list is the first item in the lowest numbered DATA statement. The last item in this list is the last item in the highest numbered DATA statement.

Both numbers and string literals may appear in a DATA statement and each value in the DATA statement list must be separated from the next value by a comma.

Quotation marks are required around str lit only if it contains one or more of the characters listed in Table 4-2. Quotation marks are optional in all other cases.

Examples

```
100 DATA 1, 17, "AB,CD", −1.3E−13
```

See the READ statement for usage and additional examples.

*DATE

Format

DATE

Returns the current date.

Remarks

The current date is returned in the decimal form YYDDD, where YY is the last two digits of the year and DDD represents the number of days elapsed in the year.

See also the string function DATE$.

Examples

The value of DATE on May 9, 1977 was 77129.

*DATE$

Format

DATE$ Returns the current date in the string representation YY/MM/DD.

Remarks

See also the numeric function DATE.

Examples

DATE$ for May 9, 1977 was 77/05/09.

*DECLARE INTEGER

Format

DECLARE ALL INTEGER DECLARE INTEGER (var |
array(m)[*n] |
array(row,col) [*n] | svar*n) [(, var | array(m)[*n] |
array(row,col) [*n] | svar*n)] ...

Specifies the numeric variables and arrays in the data list as being 16-bit integer variables or arrays, and dimensions the string variables and arrays in the data list.

Arguments

var is a numeric variable name.

array is a BASIC numeric variable name.

m is an expression for the number of elements in a one-dimensional array.

row is an expression for the number of rows in a two-dimensional array.

col is an expression for the number of columns in a two-dimensional array.

svar is a string variable.

n is an expression for the maximum number of characters in a dimensioned string variable.

Remarks

In the absence of a declaration, MP/BASIC treats all numeric data as single-precision real.

The declaration of numeric data type may serve as a dimensioning statement for arrays and string variables (see also DIM).

Examples

```
*10 DECLARE ALL INTEGER
```

Declares all numeric variables used in the program as being integers.

```
* 10 DECLARE INTEGER X,Y,A(40),B(20,20), A$ * 26
```

Declares the numeric variables X and Y and the numeric arrays A and B as INTEGERs; dimensions the numeric a A and B and the string A$.

*DECLARE REAL

Format

DECLARE ALL REAL*[*p]*

DECLARE REAL*[*p]* (var | array(m)*[*n]* | array(row,col) *[*n]* | svar*n) *[(, var | array(m)[*n] | array(row,col) [*n] | svar*n)] ...*

Specifies the numeric variables and arrays in the data list as being 32-bit or 64-bit floating point values.

Dimensions the string variables and arrays in the data list.

Arguments

p is the number 4 or 8, indicating single or double precision, respectively.

var is a numeric variable name.

array is a BASIC numeric variable name.

m is an expression for the number of elements in a one-dimensional array.

row is an expression for the number of rows in a two-dimensional array.

col is an expression for the number of columns in a two-dimensional array.

svar is a string variable.

n is an expression for the maximum number of characters in a dimensioned string variable.

Remarks

In the absence of a declaration otherwise, MP/BASIC treats all numeric data as single-precision real.

The declaration of numeric data type may serve as a dimensioning statement for arrays (see also DIM).

Examples

```
* 10 DECLARE ALL REAL
```

Declares all numeric variables used in the program as being of type real.

```
* 10 DECLARE REAL*8 X,Y,A(40),B(20,20),NAMES$(10,30)
```

Declares the numeric variables X and Y as double-precision real numbers, the numeric arrays A and B as containing double-precision real numbers; dimensions the numeric arrays A and B and dimensions the string array NAMES$.

DEF FNa(d)

Format

DEF FNa*[(d)]* = expr

Permits you to define as many as 26 different functions that can be repeatedly referred to throughout a program.

Arguments

a is a single letter from A to Z.

d is a dummy arithmetic variable that may appear in expr.

expr is an arithmetic expression that may contain the dummy variable *d*.

Remarks

Each function returns a numeric value.

BASIC does not relate the dummy variable named in the DEF statement to variables in the program with the same name; the DEF statement simply defines the function and does not cause any calculation to be carried out.

In the function definition, expr can be any legal arithmetic expression and may include other user-defined functions.

Function definition is limited to a single-line DEF statement. Complex functions requiring more than one program statement should be constructed as subroutines.

Once you have defined a function, you can redefine it later in your program.

Examples

```
* LIST
10 DEF FNE(J)=(J^2)+2*J+1
20 LET Y=FNE(5)
30 PRINT Y
* RUN
36
*
```

In line 10 the FNE function is defined. In line 20 the FNE function is referred to and evaluated with numeric argument 5.

The following example illustrates the nesting of user-defined functions.

```
* LIST
10 DEF FNR(X)=X*PI/180
20 DEF FNS(X)=SIN(FNR(X))
30 DEF FNC(X)=COS(FNR(X))
40 FOR X=0 TO 45 STEP 5
50 PRINT X,FNS(X),FNC(X)
60 NEXT X
* RUN

 0   0.               1.
 5   8.715578E-02     .996195
10    .173648         .984808
15    .258819         .965926
20    .34202          .939693
25    .422618         .906308
30    .5              .866026
35    .573577         .819152
40    .642788         .766045
45    .707107         .707107
 *
```

*DEG(X)

Format

DEG (expr)

Converts expr to degrees.

Arguments

expr is a numeric expression specified in radians.

Examples

```
* LIST
10 REM CONVERT RADIANS INTO DEGREES
20 PRINT "DEG(X) = "; DEG(PI/2)
* RUN
DEG(X) = 90
*
```

DELETE [FIRST TO LAST]

Format

```
         ┌ "filename"      ┐
         │ n1              │
         │ FIRST           │
DELETE   │ FIRST TO LAST   │
         │ FIRST to n2     │
         │ n1 to LAST      │
         └ n1 to n2        ┘
```

Removes a file from your directory; deletes a line or a group of lines from your current program. *FIRST* and *LAST* refer to the first and last program lines, respectively.

Arguments

"filename" is a file in your directory, expressed as a string literal or string variable, that is not protected.

n1 is the first program line you wish to delete.

n2 is the last program line you wish to delete.

Variations

DELETE n1

Delete the entire line numbered n1.

DELETE FIRST TO n2

Delete all lines from the first program line through n2.

DELETE n1 TO n2

Delete from line number n1 through line number n2.

DELETE n1 TO LAST

Delete lines from n1 through the last program line.

DELETE FIRST TO LAST

Delete the entire program.

Remarks

After the DELETE "filename" command, BASIC searches your directory and deletes the file named "filename."

An error message is returned if the file cannot be found, or is not in your directory.

Examples

```
*DELETE "TEST.SR"
*
```

BASIC removes file TEST.SR from your directory and frees the disk blocks which it occupied.

```
*DELETE 10
```

Deletes line 10 from your current program.

```
*DELETE 10 TO 90
```

Deletes line numbers 10 through 90 from your current program.

```
* DELETE FIRST TO 90
* DELETE 30 TO LAST
* DELETE FIRST TO LAST
```

DIM

Format

DIM array(m)*[*n]* | array(row,col)*[*n]* | svar * n
*[,array(m)[*n] | ,array(row,col)[*n] | ,svar * n] ...*

Defines the size of one or more arrays; defines the limits to a string variable.

Arguments

array is an MP/BASIC numeric or string variable name.

m is an expression for the number of elements in a one-dimensional array.

row is an expression for the number of rows in a two-dimensional array.

col is an expression for the number of columns in a two-dimensional array.

svar is a string variable name.

n is the maximum length of a string variable.

Remarks

The concept of arrays is described in Chapter 7. The DIM statement can declare the upper bound (highest-numbered element) other than the default (10) for each dimension. For example,

```
•10 DIM A(13), B(7,7), C(20,5)
```

Any variable or expression that you use for a subscript must have a value ranging from 0 or 1 on the lower bound (see OPTION BASE) up to the value specified in the DIM statement. For example,

```
•1 DIM A(5,5)
•5 X=2
•10 PRINT A(1,X^2)
```

If the variable or expression subscript does not evaluate to an integer, BASIC will round it to an integer.

If a subscript evaluates to a larger integer than the upper bound of the dimension for the array , a subscript error condition occurs.

An array can be dimensioned only once within a program. (Redimensioning within a program will generate a run-time error.)

Dimensioning Strings

The DIM statement can declare the size of a string variable to be different from the default size of 18 characters.

A string variable may be dimensioned to any number of characters. This is an extension to ANSI Minimal BASIC, which specifies a maximum of 18 characters.

Example

```
10 DIM A$ * 3, B$ * 25
```

*DISABLE [KEY] HANDLER

Format

DISABLE *[KEY]* HANDLER

Disables an exception handler.

Remarks

Deactivates the current exception handler, if any, and enables MP/BASIC's default error handling code.

If the KEY option is specified, then the exception handler enabled by the most recently executed ENABLE KEY HANDLER statement is disabled from handling ^C^A interrupts. The handling of all other exceptions will not be affected.

If the KEY option is not specified, then the exception handler enabled by the most recently executed ENABLE HANDLER statement is disabled from handling exceptions other than ^C^A interrupts. The handling of ^C^A interrupts will not be affected.

Examples

See Chapter 11 for examples of context.

*ELSE

Always used in conjunction with other keywords. See IF...THEN...ELSE, ON...GOTO...ELSE, and ON...GOSUB...ELSE.

*ENABLE [KEY] HANDLER

Format

ENABLE *[KEY]* HANDLER handler-name *[,RESUME AT line-no]*

Enables an exception handler.

Arguments

handler-name is the name of an exception handler.

line-no is the line number of a statement at which control is to be returned by any RESUME statment(s) in the exception handler.

Remarks

Deactivates MP/BASIC's default error handling code and enables the named exception handler.

If the KEY option is specified, then the named exception handler will be executed in the event of a ^C^A interrupt. The handling of all other exceptions will not be affected.

If the KEY option is not specified, then the named exception handler will be executed in the event of an exception other than a ^C^A interrupt. The handling of ^C^A interrupts will not be affected.

If another handler is enabled when this statment is executed, the first handler will be disabled and the handler named in the current ENABLE [KEY] HANDLER statement will be enabled.

Only one KEY handler and one non-KEY handler can be active at any one time.

Examples

```
130 ENABLE HANDLER OVRFLO
180 ENABLE KEY HANDLER, RESUME AT 200
```

END

Format

END

Terminates execution of the program and returns to interactive mode.

Remarks

ANSI Minimal BASIC requires the use of END to terminate each program. This keyword is optional in MP/BASIC.

*END HANDLER

Format

END HANDLER

Identifies the end of the block of code comprising an exception handler.

Remarks

If the END HANDLER statement is encountered in the normal course of execution, then the handler will be disabled and control will go to the next statement in numerical seauence following the END HANDLER statement.

Examples

See Chapter 11.

*ENTER

Format

ENTER "filename"

Transfers and merges BASIC statement lines from filename into your current program storage area.

Arguments

"filename" is a string expression representing a device or a disk file.

Remarks

If "filename" is a disk file, BASIC first searches for it in your directory.

When a statement line from a "filename" entered in this way has the same statement number as a line in the current program, the entered statement replaces the current program statement.

Examples

```
*NEW
*ENTER "TEST1.SR"
*ENTER "TEST2.SR"
*LIST "FINAL.SR"
```

Your storage area is cleared and source programs TEST1.SR and TEST2.SR are ENTERed and merged. The resultant program is LISTed to your directory as FINAL.SR.

*EXLINE

Format

EXLINE

Returns the line number of the line in which an exception occurred.

Examples

```
320 PRINT "ERROR ON LINE "; EXLINE
```

EXP(X)

Format

EXP (expr)

Calculates the value of E (2.71828) to the power of expr.

Arguments

expr is a numeric expression from approximately −178 through 175.

Examples

```
* LIST
10 REM-CALCULATE VALUE OF E^1.5
20 PRINT EXP(1.5)
* RUN
4.48169
*
```

*EXTYPE

Format

EXTYPE Returns the exception code.

Remarks

EXTYPE returns one of the error codes listed in Appendix A.

Examples

```
200 PRINT "ERRORCODE "; EXTYPE; "ON LINE ";
EXLINE
```

*FIRST

Refers to the first line in a program. Always used in conjunction with other keywords. See DELETE, LIST.

FOR and NEXT

Format

```
FOR control var = expr1 TO expr2
[STEP expr3]
  .
  .
  . Block of statements
  .
  .
NEXT control var
```

Execute a block of statements a specified number of times.

Arguments

control var is a nonsubscripted numeric variable.

expr1 is a numeric expression that defines the initial (first) value of control var.

expr2 is a numeric expression that defines the limiting value of control var.

expr3 is a numeric expression that defines the increment or decrement added to control var each time the loop is executed.

The block of statements consists of any statements which may also contain FOR...NEXT loops.

Remarks

A program loop begins with a FOR statement providing the specifications for repetition, a block of statements executed by BASIC during each repetition of the program loop, and a NEXT statement denoting the end of the loop.

The initial, limiting, and incremental values for control var determine the number of times the statements contained in a FOR...NEXT loop are to be executed. The loop is repeated until the value of the control var meets the termination condition.

If none of the initial, limiting, and incremental values for control var involve fractional parts, then it is a good idea to declare control var to be of data type INTEGER; this can speed up the execution of a loop.

The FOR statement performs an initial termination-condition test. All subsequent incrementing and testing are done at the NEXT statement (the FOR statement is executed only once).

Rules

Every FOR or NEXT statement should have a matching NEXT or FOR statement. If a FOR statement doesn't have a matching NEXT, then MP/BASIC will assume everything following the FOR is part of a loop, and will execute each line in sequence through the end of the program, and then stops. A NEXT without a matching FOR will be accepted when you type in your program, but a run-time error will occur if a NEXT statement is executed without the corresponding FOR statement.

control var must not be subscripted.

expr1, expr2, and *expr3* may have positive or negative values; *expr3* should not be zero. If you omit *STEP expr3* from the FOR statement, then *expr3* is assumed to be 1.

The termination condition for a FOR...NEXT loop depends on the values of expr1 and *expr3*. The loop terminates if

- *expr3* is positive and the next value of control var is greater than expr2;
- *expr3* is negative and the next value of control var is less than expr2.

If the value of expr1 (the initial value) meets the termination condition, then the loop is not performed even once. (See "Example 3.")

When the termination condition is met, the loop is exited; control var equals the first value not used in the loop.

Program Loop Operation

1. The expressions expr1, expr2, and *expr3* are evaluated. If you omit *expr3*, it is assumed to be 1.
2. The control var is set to the value of expr1.
3. If *expr3* is positive and control var is greater than expr2, then the termination condition is satisfied and control passes to the statement following the corresponding NEXT statement. The value of control var then equals the first value not used in the loop, i.e., control var + *expr3*.

   If *expr3* is negative and control var is less than expr2, then the termination condition is satisfied and control passes to the statement following the corresponding NEXT statement. The value of control var then equals the first value not used in the loop; i.e., control var + *expr3*.

   Otherwise, the system performs the following steps.
4. BASIC executes the statements in the FOR...NEXT block.

5. When the corresponding NEXT statement is executed, control var is set to the value of control var + *expr3.*
6. Repeat step 3 (control passes to FOR statement).

Nesting Loops

You can nest FOR...NEXT loops. The FOR statement and its terminating NEXT statement must be completely contained within the loop in which it is nested.

```
Legal Nesting        Illegal Nesting

FOR X= ...           FOR X= ...
  FOR Y= ...           FOR Y= ...
    FOR Z= ...       NEXT X
    NEXT Z             NEXT Y
  NEXT Y
NEXT X
```

Example 1

```
* LIST
10 FOR I=1 TO 9
20 NEXT I
30 PRINT I
* RUN
10
*
```

I (control var) equals first value not used in the loop.

Example 2

```
* LIST
40 FOR J=1 TO 9 STEP 3
50 NEXT J
60 PRINT J
* RUN
10
*
```

Final value of J when terminating value (expr2) was exceeded.

Example 3

```
* LIST
10 FOR I=1 TO 3 STEP -1
20 PRINT "SHOULD NOT ENTER HERE"
30 NEXT I
40 PRINT I
* RUN
1
*
```

*FP(X)

Format

FP (expr)

Returns the fractional part of expr, where expr is a real number.

Arguments

expr is a numeric expression.

Examples

```
*LIST
10 REM PRINT FRACTIONAL PART OF A NUMBER
20 LET X = 1.345
30 PRINT "FP(X) = "; FP(X)
*RUN FP(X) = .345
*
```

GOSUB and RETURN

Format

GOSUB line no.
RETURN

GOSUB directs program control to the first statement of a subroutine. RETURN exits the subroutine and returns program control to the next statement following the GOSUB statement.

Arguments

line no. is a program line number.

Remarks

A subroutine is a group of program statements entered via the GOSUB statement and exited via the RETURN statement. Instead of repeating the statements each time they are required, you write the statements into the program only once and access them by GOSUB statements. The RETURN statement returns control to the statement following the last executed GOSUB statement. In this manner, the program continues at the appropriate place after the subroutine has been executed.

A subroutine must always be entered by using a GOSUB statement. Otherwise, the error message,

RETURN with no GOSUB in line xxx

is printed when the RETURN statement is executed.

You may use more than one RETURN statement in a subroutine if program logic requires the subroutine to terminate at one of a number of different places.

Although the subroutine may appear anywhere in a program, it is good practice to place it so that it is distinctly separate from the main program. To prevent inadvertent entry to the subroutine by other than a GOSUB statement, the subroutine should be preceded by a STOP or GOTO statement directing control to a line number following the subroutine.

Subroutines may be nested. Nesting occurs when a subroutine is called during the execution of a subroutine. Upon execution of the first RETURN statement, control passes to the statement immediately following the last executed GOSUB statement. The next RETURN statement causes control to pass to the next to last executed GOSUB statement, and so on. Subroutines may be nested to a level of 9.

Example 1

```
* LIST
10 LET A=6
20 GOSUB 100
30 LET A=10
40 GOSUB 100
50 STOP
100 FOR I=1 TO A STEP 2
110 PRINT I,
120 NEXT I
130 PRINT
140 RETURN
* RUN
1  3  5
1  3  5  7  9
STOP AT LINE 50.
*
```

Example 2

```
* LIST
10 GOSUB 40
20 PRINT "EXAMPLE"
30 STOP
40 PRINT "NEST";
50 GOSUB 80
60 PRINT "INE ";
70 RETURN
80 PRINT "ED ";
90 GOSUB 120
100 PRINT "ROUT";
110 RETURN
120 PRINT "SUB";
130 RETURN
* RUN
NESTED SUBROUTINE EXAMPLE
STOP AT LINE 30
*
```

GOTO

Format

GOTO line no.

Unconditionally transfers control to the statement with the specified line number.

Arguments

line no. is a program statement line number.

Remarks

If control passes to an executable statement, that statement and those following it are executed.

If control passes to a nonexecutable statement (e.g., DATA), program execution continues at the first executable statement following the nonexecutable statement.

If line no. is not a line number in the program, an error will occur.

Examples

```
* LIST
10 READ X
20 ;X
30 GOTO 10
40 DATA 1,2,3,4,5
50 DATA 20,21,23
* RUN
1
2
3
4
5
20
21
23
Incorrect number of data items in line 50.
Stop at line 50.
*
```

*HANDLER

Format

HANDLER handler-name

Identifies the beginning of the block of code comprising an exception handler.

Arguments

handler-name is the name of the handler. It must be from one to six alphanumeric characters, the first of which must be alphabetic.

Remarks

The HANDLER statement cannot occur within a HANDLER...END HANDLER block. This means that exception handlers cannot be nested.

Examples

```
600 HANDLER OVRFLO
```

IF...THEN...ELSE

Format

IF (log-expr | expr) THEN statement
[ELSE statement]

Executes a statement on the basis of whether a logical expression is True or False, or whether a numeric expression is zero or nonzero.

Arguments

log-expr is a logical expression as defined in Chapter 5.

expr is a numeric expression.

statement is any MP/BASIC statement (This is an extension of the ANSI standard.)

Remarks

MP/BASIC evaluates the logical expression, log-expr. If it is True, then MP/BASIC executes statement following the THEN. If the expression is False, the THEN clause is ignored, and program execution continues at the statement following the ELSE; if there is no ELSE, then program execution continues at the statement following the IF...THEN... statement.

You can use a numeric expression (expr) instead of a logical expression (log-expr). The numeric expression is considered False if it has a value of 0 and True if it has a nonzero value.

The ELSE clause is an addition to the ANSI standard.

Example 1

```
* LIST
5 IF A = B THEN 100
10 IF A = B THEN GOTO 100
20 IF A-B <= 5 THEN C = 0
30 IF A*B < 50 THEN GOSUB 300
*
```

Lines 5 and 10 are equivalent variations of the IF...THEN... statement.

Example 2

```
* LIST
5 REM - START
10 LET N = 10
20 INPUT PROMPT "X= ": X
30 IF X THEN GOTO 50
40 GOTO 100
50 IF X>=N THEN GOTO 80
60 PRINT X, "X IS LESS THAN 10"
70 GOTO 20
80 PRINT X, "X IS GREATER OR EQUAL TO 10"
90 GOTO 20
100 PRINT X, "X=0"
*
*RUN
X= 5
5 X IS LESS THAN 10
X= 7
7 X IS LESS THAN 10
X= 12
12 X IS GREATER THAN OR EQUAL TO 10
X= 10
10 X IS GREATER THAN OR EQUAL TO 10
X= 0
0 X = 0
*
```

Lines 30 and 40 in this example can be combined into one equivalent statement:

```
30 IF X THEN GOTO 50 ELSE GOTO 100
```

Example 3

Note the nested IF statement in the following example.

```
* LIST
10 LET X = 5
20 LET Z = 6
30 IF X=5 THEN IF X * Z = 30 THEN PRINT Z
*
* RUN
6
*
```

INPUT, INPUT PROMPT, INPUT FILE

Format

INPUT (svar | var) *[,(svar | var)]...*
INPUT PROMPT string expression: (svar | var) *[,(svar | var)]...*
INPUT FILE (file |file,record), (svar | var) *[,(svar | var)]...*

Requests data from your terminal or from a device and assigns the values you supply to a list of variables.

Arguments

string expression is the prompt message.

var and svar are numeric and string variables.

file is a numeric expression that evaluates to the number of a file opened for sequential or random access.

record is a numeric expression that evaluates to the number of a record in a file opened for random access.

Remarks

You can use the INPUT statement to enter numeric data, string data, or both to a program. Strings including characters other than letters, digits, minus sign, plus sign, period, and embedded spaces require quotation marks around the string.

svar and var can be array elements.

Numeric data may include digits, plus and minus signs, decimal points, and the letter E (exponential notation).

When an INPUT statement is executed, a question mark is output as an initial prompt. If the INPUT statement is preceded by a PRINT line ending with a semicolon, the question mark appears immediately at the end of the printed message (Example 1).

If your INPUT statement incorporates a prompt message INPUT PROMPT, no question mark appears. The prompt message is displayed before the response to the INPUT statement is accepted. (Example 2.)

You respond by typing a list of data, separating each data item from the next by a comma. End the list with a carriage return.

If you type a carriage return before supplying a value for each variable in the INPUT statement, an error message will be generated, followed by a new question mark. You must then re-input the entire data list. (Example 3).

The data that you input in response to the prompt must be of the same type (numeric or string) as the variable in the INPUT statement list for which the data is being supplied. Variables in the INPUT statement list may be subscripted array elements or strings.

If the data you input from the terminal does not match the type of a variable in the INPUT statement list, an error message is displayed, followed by a new question mark. Your entire data list must then be re-input. (Example 4).

If you supply more items than there are variables in the INPUT list, an error condition occurs.

The INPUT FILE statement reads files created with the PRINT FILE statement. (Example 5.) To be read as an input such files must be formatted in the same way as a user-entered response to an INPUT statement, with the data items separated by commas. (See PRINT FILE)

The INPUT PROMPT and the INPUT FILE statements are extensions of the ANSI standard.

Example 1

```
* LIST
10 PRINT "YEAR, MONTH, DAY ";
20 INPUT Y, M, D
30 PRINT Y, M, D
*
* RUN
YEAR, MONTH, DAY? 1979, 10, 5
1979 10 5
*
```

In this example, the question mark is on the same line as the printed message.

Example 2

```
* LIST
10 INPUT PROMPT "ENTER YEAR, MONTH, DAY"
: Y, M, D
20 ; Y, M, D
*
* RUN
ENTER YEAR, MONTH, DAY 1979, 10, 5
1979 10 5
*
```

This example illustrates the use of the INPUT PROMPT statement.

INPUT, INPUT PROMPT, INPUT FILE (continued)

Example 3

```
* LIST
10 INPUT A,B,C,D,E
20 PRINT A+B, C+D, D+E
*
* RUN
? 1,2
Wrong number of items in input reply in line 10.
? 1,2,3,4,5
3 7 9
*
```

User entered fewer data items than requested in the INPUT statement. An error message was displayed, and the entire list of data had to be input from the start.

Example 4

```
*LIST
10 INPUT A$, B, C
20 PRINT A$, B, C
*RUN
? 75, 3, "fred"
Illegal data type in line 10.
? "fred", 3, 124
fred 3 124
```

The user mistakenly input a string variable instead of the numeric variable specified in the INPUT data list. After the error message was displayed, the entire data list was re-input.

Example 5

```
*LIST
10 INPUT FILE (1), A$, B, C
20 PRINT A$, B, C
* RUN
fred 3 124
```

Rather than being typed in by the user, the values requested were read from a file in response to the INPUT FILE statement.

INT(X)

Format

INT (expr)

Returns the value of the largest integer less than or equal to expr.

Arguments

expr is a numeric expression.

Remarks

The INT(X) function returns the largest integer which is less than or equal to the argument.

Examples

```
*LIST
10 PRINT "INT(15.8) ="; INT(15.8)
20 PRINT "INT(-15.8) ="; INT(-15.8)
30 PRINT "INT(15.8+.5) ="; INT(15.8+.5)
*RUN
INT(15.8) = 15
INT(-15.8) = −16
INT(15.8+.5) = 16
*
```

Line 30 of this example demonstrates a technique for rounding real numbers to the nearest integer.

*IP(X)

Format

IP (expr)

Calculates the integer part of a real number.

Arguments

expr is a numeric expression.

Examples

```
* LIST
10 REM PRINT INTEGER PART OF NUMBER
20 X = −123.456
30 PRINT "IP(X) = "; IP(X)?
* RUN
IP(X) = -123
*
```

*LAST

Refers to the highest numbered line in a program.

Always used in conjunction with other keywords. See DELETE and LIST.

*LEN(A$)

Format

LEN (str-expr)

Returns a value equal to the number of characters in the current value of the string expression str-expr.

Arguments

str-expr is a string expression.

Remarks

Nonalphabetic characters such as spaces and punctuation are included in the count of LEN(A$).

Examples

```
* LIST
5 DIM A$*80, B$*80
10 INPUT A$, B$
20 LET B=LEN(A$)
40 IF B>LEN(B$) THEN GOTO 60
50 GOTO 100
60 PRINT "LENGTH OF A$ ="; LEN(A$)
70 PRINT "LENGTH OF B$ ="; LEN(B$)
80 PRINT "A$ IS LONGER THAN B$"
90 GOTO 110
100 PRINT "B$ IS SAME OR LONGER THAN A$"
*RUN
? CHEESE , CAKE
LENGTH OF A$ = 6
LENGTH OF B$ = 4
A$ IS LONGER THAN B$
*
* RUN
? S P A C E , SPACE
LENGTH OF A$ = 9
LENGTH OF B$ = 5
A$ IS LONGER THAN B$
*
```

LET

Format

LET (var | svar) = expr

Evaluates expr and assigns the resultant value to var or svar.

Arguments

var	is a numeric variable name, or array reference.
svar	is a string variable name.
expr	is an arithmetic or string expression.

Remarks

The variables var and svar may be subscripted.

String expressions may be assigned to string variables.

Example 1

```
* 10 LET C = 0
```

Variable C is initialized to the value of zero.

Example 2

```
*10 LET A = A + 1
```

Variable A is assigned a value one greater than it was before.

Example 3

```
*20 LET A(2,1) = B^2+10
```

The element in row 2, column 1 of array A is assigned the value of expression B^2+10.

Example 4

```
*10 LET C$ = A$ & B$
```

String variable C$ is assigned the value resulting from the concatenation of string variables A$ and B$.

*LINPUT, LINPUT PROMPT, LINPUT FILE

Format

LINPUT svar *[,svar]...*
LINPUT PROMPT string expression: svar *[, svar]...*
LINPUT FILE (file |file,record), svar *[, svar]...*

Requests string data from your terminal or from a device, and assigns the values received to a string variable.

Arguments

string expression	is the prompt message.
file	is a numeric expression that evaluates to the number of a file opened for sequential or random access.
record	is a numeric expression that evaluates to the number of a record in a file opened for random access.
svar	is a string variable.

Remarks

Only string data may be used as arguments.

LINPUT and LINPUT FILE read a line of string data including characters normally receiving special treatment. The entire line as it stands is assigned to a string variable.

The NEW LINE, carriage return, line feed, and null characters are recognized as delimiters.

A prompt message may be combined with the input request by using the LINPUT PROMPT command.

The LINPUT, LINPUT PROMPT, and LINPUT FILE statements are extensions to the ANSI standard.

Example 1

```
*LIST
10 DIM A$*80
20 OPEN FILE(1,0), "APHORISMS"
30 FOR I = 1 TO 3
40 LINPUT A$
50 PRINT FILE(1), A$
60 NEXT I
70 CLOSE
*
* RUN
? 'SOAP AND EDUCATION ARE NOT AS SUDDEN
? AS A MASSACRE, BUT THEY ARE MORE DEADLY
? IN THE LONG RUN.' [TWAIN]
```

In this example, a loop requests the user to enter three lines of text into the terminal, and the text so input is then printed to a file.

Example 2

```
*LIST
10 DIM A$*80
20 OPEN FILE(1,0), "APHORISMS"
30 FOR I = 1 TO 3
40 LINPUT FILE(1), A$
50 PRINT A$
60 NEXT I
70 CLOSE
*
* RUN
'SOAP AND EDUCATION ARE NOT AS SUDDEN
AS A MASSACRE, BUT THEY ARE MORE DEADLY
IN THE LONG RUN.' [TWAIN]
```

Here the program reads text lines from the file and displays them on the screen. Single quotes and a comma were preserved as part of the string.

*LIST [FIRST TO LAST]

Format

LIST "filename"
LIST n1 *["filename"]*
LIST n1 TO n2 *["filename"]*
LIST n1 TO LAST *["filename"]*
LIST FIRST *["filename"]*
LIST FIRST TO n2 *["filename"]*
LIST FIRST to LAST *["filename"]*

Outputs part or all of your current program in ASCII to the disk file or device specified by *"filename"* or to your terminal if *"filename"* is not specified.

FIRST and LAST refer to the first and last program lines, respectively.

Arguments

"filename" is a disk file or device expressed as a string literal.

n1 is the line number of the first or only statement to be listed.

n2 is the line number of the first or only statement to be listed.

Variations

You can use the LIST command in the following ways:

LIST

List the entire program from the lowest numbered statement.

LIST n1

List only the single statement at line number n1.

LIST FIRST TO n2

List from the lowest numbered line through line number n2.

LIST n1 TO n2

List from line number n1 through line number n2.

LIST n1 TO LAST

List from line number n1 through the highest numbered line in the program.

LIST "filename" Write lines to disk file or device *"filename."*

*LIST [FIRST TO LAST] (continued)

Remarks

When you include the *"filename"* argument, the LIST command writes the specified lines to the disk file or device called *"filename"* in ASCII format.

If *"filename"* is a disk file that already exists in your directory, BASIC will print the message:

File already exists.

In that case, you must delete the old *"filename"* before you save the new one.

The file created by the LIST command can be read back into the program storage area by the ENTER command.

Examples

```
*LIST "TEST.SR"
```

Outputs your current program in ASCII to your directory with the filename, TEST.SR (provided TEST.SR does not duplicate another file with that name).

```
*LIST
```

Lists your current program on your terminal.

```
*LIST 20
```

Lists line number 20 on your terminal.

```
*LIST 700 TO 9999
```

Lists line numbers 700 through 9999 at the terminal.

```
*LIST 100 TO 200 "TEMP"
```

Lists line numbers 100 through 200 to disk file TEMP (providing this filename does not already exist).

```
* LIST FIRST TO 90
```

Lists program lines from the first line through line 90.

```
* LIST 100 TO LAST
```

Lists line numbers 100 through the last line in the program.

*LOAD

Format

LOAD "seg-file-name"

Loads (but does not begin execution of) a program segment.

Arguments

seg-file-name is the name of the file, expressed as a string literal, in which the program segment is stored.

Remarks

LOAD loads the segment from file "seg-file-name" into your work area. LOAD does not clear your work area before loading in the segment.

Use LOAD to load in the first segment of a chain of program segments.

LOAD is also useful in loading in a segment for debugging.

The program segment must have been previously stored in file "seg-file-name" by a SAVE command.

LOAD can be used only as a command in immediate mode; it cannot be used as a statement in a program.

Examples

```
*NEW
*LOAD "FRSTSEG"
*RUN
```

For more extensive examples, see Chapter 10.

LOG(X)

Format

LOG (expr)

Calculates the natural logarithm of expr.

Arguments

expr is a numeric expression.

Examples

```
*LIST
10 REM -CALCULATE THE LOG OF 959
20 PRINT LOG(959)
*RUN
6.86589
*
```

*MEMORY

Format

MEMORY

Prints the number of words used by the program and the total number of words still available.

Remarks

The number of words used is broken down into two groups:

- the number of words used for the program segment; the number of words used for the data segment.
- The total number of words left is reported.

Examples

```
MEMORY
Program area: 0 words.
Data area: 7 words.
Available area: 7630 words.
```

*MOD(X,Y)

Format

MOD (expr, expr)

Calculates X modulo Y.

Arguments

expr is a numeric expression.

Remarks

This function is calculated as X−Y * INT(X/Y) if Y is nonzero; an overflow error will be generated if Y is zero.

Examples

```
* LIST
10 REM CALCULATE 50 MODULO 8
20 LET X = 50
30 LET Y = 8
40 PRINT "MOD(X,Y) ="; MOD(X,Y)
* RUN
MOD(X,Y) = 2
*
```

*NEW

Format

NEW

Clears the program and variables currently stored in your program storage area and closes any open files.

Remarks

You must clear your storage area with a NEW command before entering a new program, to avoid intermixing lines from previous programs with it.

NEW cannot be used as a program statement, but functions only in immediate mode.

Examples

```
* NEW
* ENTER "SQUARE"
* LIST
10 LET A = 5
20 PRINT SQR(A^2+75)
*
```

NEXT

Always used in conjunction with other keywords. See FOR.

ON...GOSUB...ELSE

Format

ON expr GOSUB line no.*[,line no.]... [ELSE statement]*

Transfers control to one of several subroutines in a program depending on the value of an expression at the time the statement is executed.

Arguments

expr is a numeric expression evaluated to an integer.

line no. is a list of line numbers of first lines of subroutines in the current program, whose positions in the argument list are numbered from 1 through n.

statement is any MP/BASIC statement.

Remarks

This statement functions the same as the ON...GOTO...ELSE statement with the exception that the subroutine call GOSUB is performed instead of the GOTO statement.

The expression expr is evaluated and rounded to an integer, if it is not an integer.

The program transfers control to the subroutine whose position in the argument list corresponds to the computed value of expr.

If expr evaluates to an integer greater than the number of entries given in the argument list or less than or equal to zero, the program transfers control to the statement following ELSE; if ELSE is not specified, then an error is returned.

The ELSE clause is an extension to the ANSI standard.

ON...GOTO...ELSE

Format

ON expr GOTO line no.*[,line no.]...[ELSE statement]*

Transfers control to one of several lines in a program depending on the value of an expression at the time the statement is executed.

Arguments

expr is a numeric expression evaluated to an integer.

line no. is a list of line numbers in the current program whose positions in the argument list are numbered from 1 through n.

statement is any MP/BASIC statement.

Remarks

The expression expr is evaluated and rounded to an integer, if it is not an integer.

The program transfers control to the line number whose position in the argument list corresponds to the computed value of expr.

If expr evaluates to an integer greater than the number of entries given in the argument list or less than or equal to zero, then the program transfers control to the statement following the ELSE; if ELSE is not specified, then an error is returned.

The ELSE clause is an extension to the ANSI standard.

Example 1

```
* LIST
10 ON M-5 GOTO 500, 75, 1000
```

If M-5 evaluates to 1, 2, or 3 then control passes to statement 500, 75, or 1000, respectively. If M-5 evaluates to any other value, an error is returned.

Example 2

```
* LIST
10 ON (SGN(M-5)+2) GOTO 100,200,300
```

This statement is equivalent to the following three statements:

```
*10 IF M-5<0 THEN GOTO 100
*20 IF M-5=0 THEN GOTO 200
*30 IF M-5>0 THEN GOTO 300
```

*OPEN FILE

Format

OPEN FILE (file,mode), "filename" *[,record size]*

Assigns a file number and access mode to "filename" for future referencing in file I/O statements in your program.

Arguments

file is a numeric expression that evaluates to a number from 0 through 7. BASIC uses this number to simplify the reference to "filename" in other file I/O statements.

mode is a numeric expression that evaluates to a number from 0 to 3. This number specifies the access mode of the file. (The modes are described under "Remarks," below.)

"filename" is a string expression that evaluates to a filename.

record size is an optional numeric expression that evaluates to a fixed length (in bytes) for each record in a file. Record size may be any value from 1 to 32,767.

There are two types of records: fixed length and variable length.

If the output has more bytes than was specified for the record, then output continues until the end of the data.

Remarks

You can calculate record length as follows:

Integers:	2 bytes per data item.
Real numbers:	4 bytes per data item for single-precision, 8 bytes per data item for double-precision.
String data:	1 byte per character in string, plus 1 byte for string delimiter.

*OPEN FILE (continued)

Modes 0 to 3 are as follows:

Mode 0 Input and output (read and write). Only disk files may be opened in random mode for reading and writing. If the disk file named "filename" does not exist in your directory, BASIC creates it.

Mode 1 Output (creates a new file for writing). You can open either a disk file or an appropriate output device in Mode 1. Only writes are permitted. If "filename" already exists in your directory, the previous copy is deleted from the disk. In either case, a new file is created (initialized with 0 length).

Mode 2 Output (appends to an existing file). You can use this mode to open any file previously opened in Mode 1 or Mode 2. When an existing file is opened in Mode 2, the file pointer moves to the end of the file so that subsequent data written to the file will extend it. If the file does not exist in your directory, it will be created. Only writes are permitted.

Mode 3. Input (for reading only). You can open either a disk file or appropriate input device in Mode 3. If a disk file is opened in this mode, the file must already exist. Only reads are permitted from a file opened in Mode 3. If the file is not found in your directory, BASIC searches for it in your CLI searchlist (refer to the CLI manual).

Examples

```
*100 OPEN FILE (1,1), "NETSAK.JR"
```

This statement opens file 1, named NETSAK.JR, as an output file.

```
*100 OPEN FILE (2,0), "RESSEHC.TO", 20
```

This statement opens the file named RESSEHC.TO as file number 2 for either read or write. Records are 20 bytes long.

OPTION BASE

Format

```
OPTION BASE expr
```

Sets the lower bound of an array to 0 or 1.

Arguments

expr is a numeric expression, which can be 0 or 1 when evaluated.

Remarks

In the absence of an OPTION BASE statement, the default value of an array's lower bound is 0.

This is an extension to the ANSI standard, which does not allow the use of an expression.

Examples

```
10 OPTION BASE 1
20 DIM A(25)
```

*ORD(A$)

Format

ORD (str-expr)

Returns the ASCII decimal representation of str-expr.

Arguments

str-expr is a string expression

Remarks

When str-expr is evaluated, it can be any single ASCII character.

ORD(A$) complements the CHR$(M) function.

Examples

```
* LIST
10 REM CONVERT UPPER TO LOWER CASE WITH
ORD FUNCTION
20 FOR I = 1 TO 7
30 READ A$
40 PRINT A$, ORD(A$), CHR$(ORD(A$)+32)
50 NEXT I
60 DATA "C", "O", "N", "V", "E", "R", "T"
* RUN

          C    67    c
          O    79    o
          N    78    n
          V    86    v
          E    69    e
          R    82    r
          T    84    t
          *
```

```
* LIST
10 LET H$="BCADEF"
20 PRINT ORD(H$(3:3))
* RUN
65
```

*PI

Format

PI

Returns the constant 3.14159... which is the ratio of the circumference of a circle to its diameter.

Remarks

The value of PI will be accurate to 15 digits for hardware/firmware floating point systems, or to 6 digits for software floating point systems.

Examples

```
* LIST
10 REM CALCULATE THE AREA OF A CIRCLE OF
RADIUS R
20 LET R = 10
30 PRINT "AREA ="; PI * R ^ 2
* RUN
AREA = 314.159
*
```

*POS(A$,B$)

Format

POS (str-expr,str-expr)

Returns the position of the first character of B$ in A$.

Arguments

str-expr is a string expression.

Remarks

This function will return a value only if the entire string B$ is contained within string A$. Otherwise, it returns a value of 0.

Only the first occurrence of B$ within A$ is considered; later occurrences, if any, are ignored.

If B$ is a null string, the function returns a value of 1.

Examples

```
* LIST
10 LET A$ = “HEAVYWEIGHT”
20 LET B$ = “WEIGH”
30 PRINT POS(A$,B$)
* RUN
6
*
```

```
* LIST
10 LET P$ = “ULULATIONS”
20 LET G$ = “UL”
30 PRINT POS(P$,G$)
* RUN
1
*
```

```
* LIST
10 LET A$ = “BEARPAW”
20 LET B$ = “PEAR”
30 PRINT POS(A$,B$)
* RUN
0
*
```

*POS(A$,B$,M)

Format

POS (str-expr,str-expr,expr)

Searches A$ for B$, starting at position M and returns the position of the first character of B$ within A$.

Arguments

str-expr is a string expression

expr is a numeric expression.

Remarks

Returns 0 if B$ is not in A$ from position M onward.

If M is out of range of the length of A, the function will return 0.

Examples

```
* LIST
10 LET A$ = “PERSEVERANCE”
20 LET B$ = “SEVER”
30 ; “STARTING POINT OF SEARCH WITHIN
STRING”;
40 INPUT M
50 P = POS(A$,B$,M)
60 IF P > 0 THEN GOTO 120
70 IF M > LEN(A$) THEN GOTO 100
80 PRINT “B$ IS NOT IN A$”
90 GOTO 130
100 PRINT “SEARCH IS OUT OF RANGE”
110 GOTO 130
120 PRINT “B$ STARTS AT POSITION”; P; “ IN A$”
130 END
* RUN
STARTING POINT OF SEARCH WITHIN STRING?
3
B$ STARTS AT POSITION 4 IN A$
*
* RUN
STARTING POINT OF SEARCH WITHIN STRING?
7
B$ IS NOT IN A$
*
* RUN
STARTING POINT OF SEARCH WITHIN STRING?
13
SEARCH IS OUT OF RANGE
*
```

PRINT

Format

(PRINT | ;) *[expr] [(,|;)[expr]]...*

(PRINT | ;) USING svar:(expr)*[,[expr]]...*

Performs print operations at your terminal.

The semicolon (;) is a synonym for the keyword PRINT. (This is an extension of the ANSI Standard.)

Arguments

svar is a string variable whose value consists of image characters for formatted output (see Table 1-1).

expr is a numeric or string expression.

Remarks

This keyword may be used in immediate mode and as a program statement.

The following PRINT operations are possible:

1. Print the result of a computation.
2. Print verbatim the characters in a string literal or string variable.
3. Print a combination of operations 1 and 2.
4. Print a blank line (skip a line).

Zone Spacing of Output

The print line on a terminal is divided into five print zones of 14 characters each (see Figure 14-1). The first columm on a line is column one.

A comma (,) between items in the PRINT statement list causes the next item to be printed in the leftmost position of the next printing zone. If there are no more printing zones on the current line, printing continues in the first printing zone on the next line. If an item requires more than one print zone, the next item in the list is printed in the next free print zone. (See Example 1.)

Before each item is printed, its length is compared with the space remaining on the line. If insufficient space remains on the current line, the item is moved to the next line.

Compact Spacing of Output

A semicolon (;) between items in the PRINT statement list causes the next item to be printed at the next character position. Note that a space is reserved for the plus (+) sign even though it is not printed (Example 2), and there is always a trailing space.

Spacing to the Next Line

When the last item in a print list has been printed, BASIC outputs a carriage return and line feed unless the last item in the list is followed by a comma (,) or semicolon (;). In this case, the carriage return and line feed are not output and the next item is printed on the same line, according to the comma or semicolon punctuation. (See Example 3.)

If, however, the comma or semicolon would cause printing of the next item to occur beyond the allowable line width, a carriage return and line feed are output.

Printing Blank Lines

A PRINT statement with no list of print items or punctuation will output a carriage return and line feed. (See Example 4.)

Formatted Output

The USING option allows you to format numeric output. You specify the format using a string composed of image characters (see Table 1.1). The format can include left- and right-justification, the insertion of dollar signs, asterisks, commas, and decimal points, the suppression or printing of leading zeros, the indication of positive and negative numbers, and control of the spacing of the numeric display.

For details of the implementation of the USING option and examples, see Chapter 1.

The USING option is an enhancement to the ANSI standard.

For more printing versatility, see the TAB(X) function.

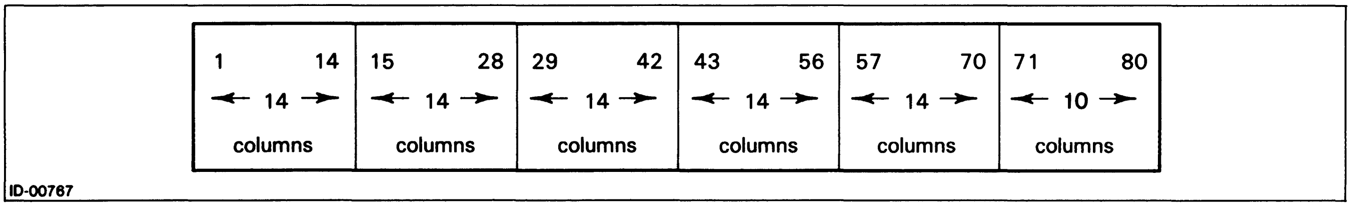


Figure 14-1. Zone Spacing of Output

PRINT (continued)

Example 1

```
* LIST
10 LET X=25
20 PRINT "SQUARE ROOT OF X IS: ",SQR(X)
* RUN
SQUARE ROOT OF X IS:        5
*
^             ^             ^
1             15            29
```

The column positions print out at intervals of 14 spaces.

Example 2

```
* LIST
10 LET X = 5
20 PRINT X;(X*2)^6;X*2;(X*2)^4;
30 PRINT X-25;(X*2)^8;X-100
* RUN
 5 1000000  10 10000 -20 1E+08
-95
*

 ^ ^        ^  ^     ^   ^
 2 4        13 16    22  26
```

The printout is displayed at positions 2, 4, 13, 16, 22, 26, on the first line, and position 1 on the second line.

Example 3

```
* LIST
10 LET X = 5
20 PRINT X,(X*2)^6,
30 ; X^4
40 ; "FIN"
* RUN
 5            1000000        625
FIN
*

 ^            ^              ^
 2            15             30
```

The printout appears in columns 2, 15, and 30.

Example 4

```
* LIST
10 LET X=5
20 PRINT X;(X*2)^6,X*2
30 PRINT X-25;(X*2)^8
40 ;  X-100
50 PRINT
60 ;  "DONE"
*RUN
 5  1000000    10
-20  1E+08
-95

DONE
*

 ^  ^          ^
 2  4          16
```

The output is printed in positions 2, 5, and 16 of the first line.

In line 20, the comma and semicolon spacing characters are both used. Line 50 outputs a blank line before printing "DONE".

*PRINT FILE

Format

(; | PRINT) FILE (file | file,record), *[expr][(,|;)[expr]]...*

(; | PRINT) FILE USING svar: (expr) *[,[expr]]...*

Writes data in ASCII format into a sequential- or random-access file.

Arguments

file is a numeric expression that evaluates to the number of a file opened for sequential or random access.

record is a numeric expression that evaluates to the number of a record in a file opened for random access.

expr is a numeric or string expression.

svar is a string variable whose value is composed of image characters for formatted output (see Table 1-1).

Remarks

This statement is intended for outputting to an ASCII device such as a line printer, or to a disk file for later off-line printing.

Each item in the expression list must be separated from the next by a comma, semicolon, or carriage return. Output formatting is identical to that discussed in Remarks under the PRINT statement.

The USING option functions in the same way for PRINT FILE as for PRINT.

The first record number in a random-access file is 0.

Files created with the PRINT FILE statement may be read with the INPUT FILE and the LINPUT FILE statement. (Example 3.)

If a list of variables is to be read with an INPUT FILE statement, the file contents must be formatted in the same way as a user-entered INPUT statement: literal commas must be entered in the file as data separators. (Example 4.)

Example 1

```
*10 OPEN FILE(3,1), "$LPT"
*100 PRINT FILE(3), "OUT6"
*200 PRINT FILE(3), "X ="; X, "X SQUARED="; X^2
*300 PRINT FILE(3), A; B; C
*400 CLOSE
```

Example 2

```
*10 OPEN FILE(3,1), "$LPT",80
*100 FOR I=1 TO 10000
*110 PRINT FILE(3),I;
*120 NEXT I
*130 CLOSE
```

Example 3

```
*LIST
10 OPEN FILE (3,0), "TEST", 4
20 FOR I = 0 TO 10
30 PRINT FILE(3,I), I
40 NEXT I
50 FOR I = 0 TO 10
60 INPUT FILE(3,I), Z
70 PRINT Z
80 NEXT I
90 CLOSE
* RUN
1
2
3
4
5
6
7
8
9
10
*
```

*PRINT FILE (continued)

Example 4

```
* LIST
10 OPEN FILE(6,0), “STATS”
20 LET H$ = “HEIGHT”
30 LET H = 6
40 LET W$ = “WEIGHT”
50 LET W = 250
60 PRINT FILE(6),H$;“,”;H;“,”;W$;“,”;W
70 CLOSE
80 OPEN FILE(6,0), “STATS”
90 INPUT FILE(6), H$, H, W$, W
100 PRINT H$; H, W$; W
110 CLOSE
* RUN
HEIGHT 6 WEIGHT 250
*
```

This example formats a file so that several variables may be read with a single INPUT statement.

*RAD(X)

Format

RAD (expr)

Converts X degrees to radians.

Arguments

expr is a numeric expression specified in degrees.

Examples

```
* LIST
10 REM CALCULATE NUMBER OF RADIANS IN X
DEGREES
20 LET X = 35
30 PRINT “RAD(X) = ”; RAD(X)
* RUN
RAD(X) = .610865
*
```

RANDOMIZE

Format

RANDOMIZE

Causes the random number generator to start at a different point in the sequence of random numbers generated by the RND function.

Remarks

The RANDOMIZE statement resets the random number generator based on the time of day, thereby producing different random numbers each time a program using the RND function is run.

Wihout RANDOMIZE, the same sequence of random numbers is generated by the RND function each time a program is run. This feature is useful for debugging programs. When the program runs successfully, the RANDOMIZE statement should be included in the program before the first occurrence of a RND function if you desire different starting points in the sequence.

Examples

This program will print different values each time it is run. I

```
*10 RANDOMIZE
*20 FOR I = 1 TO 3
*30 PRINT RND;
*40 NEXT I
*50 PRINT
*RUN
.619604 .298047 .698036

*RUN
.776468 7.84348E-02 .603916

*RUN
.784302 .117651 .800002

*RUN
.956853 .98038 9.41276E-02

*RUN
.10981 .760783 6.31819E-06

*
```

READ

Format

READ (var | svar) *[(, var | , svar)] ...*

Reads values from the DATA list (DATA statements) and assigns them to variables.

Arguments

var and svar are numeric and string variables, separated by commas.

Remarks

READ statements must always be used in conjunction with DATA statements.

The variables listed in the READ statement may be subscripted or nonsubscripted and may be numeric or string.

The order in which variables appear in the READ statement is the order in which values for the variables are retrieved from the DATA list.

A data element pointer is moved to the next available value in the DATA list as values are retrieved for variables in READ statements. If the number of variables in the READ statement exceeds the number of values in the DATA list, BASIC prints an error message reading, Incorrect number of data items.

The type of variable (numeric or string) in the READ statement must match the type of the corresponding DATA element value, or else BASIC prints an error message reading, *Illegal data type*

The RESTORE statement can be used to reset the data element pointer to the first item of the lowest numbered DATA statement or to the first item of a specific DATA statement.

READ (continued)

Examples

```
* LIST
10 READ A,B,C
20 READ D(1),D(2),D(3)
30 PRINT C^2,D(2)^2
40 READ E
50 ; E
60 READ F$
70 ; F$
80 DATA 1,2,3,4,5,6,7,"ABC"
90 END
*
* RUN
9 25
7
ABC
*
```

In this example the variables are assigned values as follows:

Variable	Value
A	1
B	2
C	3
D(1)	4
D(2)	5
D(3)	6
E	7
F$	ABC

*READ FILE

Format

READ FILE (file | file,record), (var | svar)
[,(var | svar)] ...

Reads data in binary format from a file sequentially or randomly accessed.

Arguments

file is a numeric expression that evaluates to the number of a file opened for random or sequential access.

record is a numeric expression that evaluates to the number of a record in a file opened for random access.

var and svar are numeric variables and string variables.

Remarks

The type of variable in the READ FILE variable list must correspond to the data type of the corresponding item being read from the record.

The number of the first record in a random-access file is 0.

In random-access files, records that have not been written into will contain all zeros when read. An attempt to read a record coming after the last record written will cause an end-of-file condition.

Example

```
* LIST
1 REM - READ FILE
10 OPEN FILE(1,0), "TESTFILE", 20
20 FOR I = 1 TO 12
30 READ FILE(1,I),B
45 PRINT B,
50 NEXT I
60 CLOSE
70 END
* RUN
36 33 30 27 24 21
18 15 12  9  6  3
*
```

This program uses the file TESTFILE created in the program example provided with the WRITE FILE statement.

REM

Format

REM *[message]*

Inserts explanatory remarks within a program.

Arguments

message is a text comment. It can contain any characters, incuding quotation marks.

Remarks

REM statements do not affect program execution. BASIC stores them with a program and outputs them with each program listing.

If control is transferred to a REM statement from a GOTO... or GOSUB... statement, then execution continues with the next executable statement. If no executable statement follows the REM statement, then the program will end and control will return to interactive mode.

Remarks can extend over more than one line. Each line must start with a line number and the keyword REM.

Example

```
*LIST
10 REM REMARKS IN A PROGRAM.
20 REM HELPS EXPLAIN THE PURPOSE OF
30 REM STATEMENTS. LINES 10, 20, 30
40 REM AND 40 AREN'T EXECUTED.
50 PRINT "END"
*RUN
END
*
```

*REM(X,Y)

Format

REM (expr,expr)

Returns the remainder of X/Y.

Arguments

expr is a numeric expression.

Remarks

This function is calculated as X−Y * IP(X/Y) if Y is nonzero. If Y is zero, an overflow error will be generated.

Examples

```
* LIST
10 REM CALCULATE THE REMAINDER OF X/Y
20 LET X = -50
30 LET Y = 8
40 PRINT "REM (X,Y) ="; REM(X,Y)
* RUN
REM(X,Y) = -2
*
```

*RENUMBER [AT STEP]

Format

RENUMBER *[AT n1 | AT n1 STEP n2 | STEP n2]*

Renumbers the statements in the current program.

Arguments

n1 is the initial line number for the current program.

n2 is the desired increment between line numbers.

Variations

The RENUMBER command has several variations, as follows:

*RENUMBER

Renumber the current program, assigning default line number 100 to the first line, and using a default increment of 10 between line numbers.

*RENUMBER AT n1

Renumber the current program, assigning the number specified in n1 to the first line and incrementing subsequent line numbers by 10.

*RENUMBER STEP n2

Renumber the current program, assigning default number 100 to the first line and incrementing line numbers by n2.

*RENUMBER AT n1 STEP n2
RENUMBER STEP n2 AT n1

Renumber the current program, assigning n1 to the first line and incrementing line numbers by n2.

Remarks

Line numbers are limited to five digits. If a RENUMBER command causes a line number to exceed 65,534, an error results.

The RENUMBER command modifies the line numbers specified in IF...THEN..., GOTO..., and GOSUB... statements to agree with the new line numbers.

Example

```
* LIST
10 LET I=1
13 READ A(I)
17 IF A(I)=0 THEN GOTO 25
19 I=I+1
20 GOTO 13
25 FOR J=1 TO I-1
27 PRINT J,A(J)
29 NEXT J
30 DATA 90,95,82,61,40,0
50 END
* RENUMBER AT 10 STEP 5
* LIST
10 LET I=1
15 READ A(I)
20 IF A(I)=0 THEN GOTO 35
25 I=I+1
30 GOTO 15
35 FOR J=1 TO I-1
40 PRINT J,A(J)
45 NEXT J
50 DATA 90,95,82,61,40,0
55 END
*
```

RESTORE

Format

RESTORE *[line no.]*

Resets the position of the data element pointer.

Arguments

line no. is a DATA statement line number.

Remarks

If you use the RESTORE statement without a line number argument, the data element pointer is reset to the beginning of the data list.

If you use the RESTORE statement with an argument giving the DATA statement line number, the data element pointer is moved to the first value in the specified DATA statement line. This feature is an extension of ANSI standard BASIC.

If line no. is not a DATA statement, the data element pointer will point to the first DATA statement following line no. If line no. does not exist in the program, an error message is displayed.

Example

```
*5 READ A,B,C
*10 READ D,E,F
*15 RESTORE 50
*20 READ G,H,I
*25 RESTORE
*30 READ J,K,L
*40 DATA 2,4,6
*50 DATA 8,10,12
```

In the above example the variables are assigned values as follows:

Variable	Values
A	2
B	4
C	6
D	8
E	10
F	12
G	8
H	10
I	12
J	2
K	4
L	6

*RESUME

Format

RESUME

Returns control from an exception handler to a predetermined line number in your program.

Remarks

RESUME returns control to the statement indicated in the RESUME option of the most recently executed ENABLE [KEY] HANDLER statement.

*RETRY

Format

RETRY Returns control from an exception handler to the statement which caused the error.

Remarks

RETRY causes your program to reattempt execution of the statement which caused the exception.

RETURN

Format

RETURN

Exits a subroutine and returns control to the next statement following the GOSUB statement.

Remarks

Always used in conjunction with GOSUB. See the GOSUB statement for discussion.

RND

Format

RND

Produces a pseudo-random number, n, such that 0 < = n < 1.

Remarks

Each time the RND function is called, it provides a pseudorandom number, n, which is greater than or equal to zero, and smaller than one.

Each occurrence of the RND function in a program yields the value of the next random number in the list.

Each time you issue a NEW or RUN command, BASIC returns to its original starting place in the sequence of random numbers. Because the sequence is fixed, and the starting place is the same for each run, the RND function will provide the same numbers each time you execute your program. The capability of reproducing the sequence can be a useful debugging aid.

To alter the starting point in the sequence, use the RANDOMIZE statement described in Chapter 8. That statement resets the starting place based on the time of day, thus providing a different sequence for each run.

For random numbers in a given range A to (but not including) B, use the formula

(B − A)* RND + A

Example 1

```
* LIST
10 FOR I=1 TO 4
20   PRINT RND
30 NEXT I
* RUN
.4442246
.902405
.453201
.457184
*
```

Running the above program a second time will produce the same five random numbers.

Example 2

```
* LIST
10 FOR R = 10 TO 15
20   PRINT (6*RND+1)
30 NEXT R
* RUN
3.65347
6.41443
3.71921
3.7431
5.36258
2.95996
*
```

The above program produces six random numbers in the range 1 to 7.

Example 3

```
* LIST
10 FOR J=1 TO 4
20   PRINT INT(10*RND)
30 NEXT J
* RUN
4
9
4
4
*
```

This program produces four random integers in the range 0 to 9.

*RUN

Format

RUN *[n]*

Executes a program from either the first line or a specified line. Arguments

n is the line in the current program from which execution is to begin.

Variations

You may use the RUN command with the following variations:

RUN

Clear all variables, undimension all arrays, do a RESTORE, initialize the random number generator, and then run the current program from the first line number.

RUN n

Run the program from line n. This form of the RUN command allows resumption of program execution retaining current values of all variables and parameters. It may be used after a STOP or after an error and will incorporate any alterations you make to the program after the STOP or error occurred.

Examples

```
* RUN

* RUN 250
```

*SAVE

Format

SAVE "seg-file-name"

Stores a program segment for later execution.

Arguments

seg-file-name is the name of the file, expressed as a string literal, in which the program segment is to be stored.

Remarks

SAVE stores whatever program is currently in your work area.

The program segment (along with a data area, which includes current values of any variables) is stored in a special format that can be accessed only by a CHAIN, LOAD, or SWAP statement.

SAVE does not delete the program segment from memory.

SAVE can be used either as a statement in a program or as a command in immediate mode.

Examples

```
SAVE "CKWRIT"

SAVE "SEGTWO"
```

For more extensive examples, see Chapter 10.

SGN(X)

Format

SGN (expr)

Returns a value representing the sign of an expression.

Arguments

expr is a numeric expression.

Remarks

The value returned is:

1	if positive
0	if 0
−1	if negative

Examples

```
*LIST
10 LET A = −3
20 LET B = 8
30 PRINT "SGN(";A;") =";SGN(A)
40 PRINT "SGN(";B;") =";SGN(B)
*RUN
SGN(−3) = −1
SGN(8) = 1
*
```

SIN

Format

SIN (expr)

Calculates the sine of an angle that is expressed in radians.

Arguments

expr   is a numeric expression specified in radians.

Examples

```
*LIST
10 REM - PRINT SINE OF 30 DEGREES
20 PRINT SIN(30*PI/180)
*RUN
.5
*
```

SQR(X)

Format

SQR (expr)

Computes the square root of expr.

Arguments

expr   is a nonnegative numeric expression.

Examples

```
*LIST
10 LET A=5
20 PRINT SQR(A^2+75)
*RUN
10
*
```

STEP

Controls counter increments when they are different from 1 (in the For.. .NEXT statement) or from 10 (in the RENUMBER statement). Always used in conjunction with other keywords. See FOR...NEXT and RENUMBER.

STOP

Format

STOP

Terminates execution of the current program and returns control to interactive mode.

Remarks

You can place STOP statements anywhere in the program to terminate execution. When STOP is encountered, BASIC prints the following message on your terminal:

```
Stop at line XXXX
*
```

where *XXXX* is the line number of the STOP statement.

After resumption of interactive mode, you can modify the program if you wish. To restart the program from the beginning, use RUN; to continue from the STOP statement, use CON or RUN line no.

If a program segment that was called by the SWAP statement executes a STOP, then MP/BASIC will return control to the program that executed the SWAP and continue running at the statement following the SWAP statement.

Examples

```
*LIST
10 REM -- TERMINATE PROGRAM BY STOP
20 INPUT A
30 IF A<0 THEN GOTO 50
40 GOTO 20
50 STOP
*RUN
? 1
? 3
? −5
STOP AT LINE 50
*
```

*STR$(X)

Format

STR$ (expr)

Converts the numeric value of an expression to string.

Arguments

expr   is a numeric expression.

Remarks

This function eliminates leading and trailing spaces. Hence, it allows the generation of tightly formatted numeric output.

Examples

```
* LIST
10 READ A
15 IF A=0 THEN STOP
20 LET A$=STR$(A)
30 IF A$(4:6)= “222” THEN GOTO 50
40 GOTO 70
50 PRINT A;“ -THIS IS MODEL 222”
60 GOTO 10
70 PRINT A;“ -THIS ISN'T OUR MODEL”
75 GOTO 10
80 DATA 111222,212222,123456,0
* RUN
111222. -THIS IS MODEL 222
212222. -THIS IS MODEL 222
123456. -THIS ISN'T OUR MODEL
*
```

*SUMMON

Format

SUMMON "sub-name" *[par [,par]...]*

Calls an assembly language subroutine from MP/BASIC.

Arguments

sub-name is the name of the subroutine being called.

par is the name of a parameter being passed to or from the subroutine.

Remarks

The parameter list is optional. If it is included, however, it must be contained in square brackets (see Examples below).

SUMMON can be used either as a statement in a program, or as a command in immediate mode.

You can pass any legal MP/BASIC expression as a parameter. You can also pass an array as a parameter by specifying it as a formal array (see Chapter 12).

The called subroutine must conform to certain specifications. Refer to Chapter 12 for details.

Examples

```
100 SUMMON "SHIFT_LFT" [STR1$]
120 SUMMON "GET_CHAR" [ITEM,CHAR$,LENG]
140 SUMMON "PUTTER"
```

*SWAP

Format

SWAP "seg-file-name"

Saves the current program segment, loads and executes a second segment, then restores and continues execution of the first segment.

Arguments

seg-file-name is the name of the file, expressed as a string literal, in which the second segment is stored.

Remarks

SWAP first performs a SAVE on the program segment currently in your work area. Then it loads and executes the program segment in file "seg-file-name". When this program segment terminates with a STOP or END, then SWAP clears the work area, LOADs the original program segment and continues execution at the line following the SWAP statement.

The program segment must have previously been stored in file "seg-file-name" by a SAVE command.

SWAP does not close any open files or reset any file pointers when switching between segments.

SWAP can be used as a statement in a program, but not as a command in immediate mode.

Examples

```
SWAP "CKWRIT"
SWAP "SEGTWO"
```

For more extensive examples, see Chapter 10.

TAB(X)

Format

TAB (expr)

Tabulates to column number specified by X.

Arguments

expr is a numeric expression rounded to the nearest integer.

Remarks

The TAB function can be used only in conjunction with the PRINT statement. It cannot be used with any other BASIC statement.

If expr evaluates to less than 1, it will be set to 1 (an error message will be printed, but the program will continue).

The first column on a line is column 1. The column number specified by expr is always relative to column 1.

The position at which BASIC prints an item in the print list depends on the value of expr and on the punctuation (; or ,) following the TAB(X) function.

If expr evaluates to a column number lower than that of the current cursor position, the cursor will be moved to the specified position on the line below.

If expr evaluates to a column number greater than the page width, then the cursor will wrap-around and print in column number MOD(expr-1,width-of-line) + 1 on the same line; if that column already contains a printed character, then printing will resume in the corresponding column on the next line.

More than one TAB(X) function may appear on a print line.

Examples

```
* LIST
10 REM SPACING WITH THE TAB FUNCTION
20 LET A = -6
30 LET B = 5
40 PRINT TAB(B); A; TAB(2*B); 2*A
* RUN
     -6   -12

      1   5    10
```

TAN

Format

TAN (expr)

Calculates the tangent of an angle that is expressed in radians.

Arguments

expr is a numeric expression specified in radians.

Examples

```
*LIST
10 REM - PRINT TANGENT OF X DEGREES
20 PRINT "X DEGREES";
30 INPUT X
40 LET P = PI/180
50 PRINT "TANGENT = ";TAN(X*P)
*RUN
X DEGREES? 45
TANGENT = 1
*
```

*TIME

Format

TIME

Returns the time elapsed since the previous midnight, expressed in seconds.

Remarks

See also the string function TIME$.

Examples

The value of TIME at 11:15 A.M. is 40500.

*TIME$

Format

TIME$

Returns the time of day in 24-hour notation in the representation HH:MM:SS.

Remarks

See also the numeric function TIME.

Examples

The value of TIME$ at 11:15 a.m. is 11:15:00.
The value of TIME$ at 3 p.m. is 15:00:00.

*TO

Always used in conjunction with other keywords. See FOR, DELETE, LIST.

*VAL(A$)

Format

VAL (str-expr)

Returns the numeric representation of a string value.

Arguments

str-expr is a string expression which, when evaluated, begins with a number.

Remarks

The value of the argument to the VAL function must begin with a number or else an error message will be output. The number may include digits, plus and minus signs, decimal points, and the letter E (scientific notation). Any nonnumeric characters appearing after the number portion of the string (as well as any numeric characters after the first nonnumeric character) are ignored. For example:

"+35.5E−03ABCD7N"

Substring "+35.5E-03" is returned as a numeric value and substring "ABCD7N" is ignored.

Misplaced signs terminate the input scan in a similar fashion:

"123 + 47 − 17"

Substring "123" is returned as a numeric value and "+47−17" is ignored.

A zero is returned if the evaluation of the string's numeric portion results in an underflow.

If A$ does not contain a numeric value, an error is generated.

Examples

```
* LIST
10 LET A$= "12345ABCD"
20 LET B= 54321
30 LET C=VAL(A$)
40 LET D=B+C
50 PRINT D
* RUN
66666
*
```

WRITE FILE

Format

WRITE FILE (file | file,record), (expr) *[,expr]...*

Writes a record of data in binary format into a sequential-access file or a random-access file.

Arguments

file is a numeric expression that evaluates to the number of a file opened for random or sequential access.

record is a numeric expression that evaluates to the number of a record in a file opened for random access.

expr is a numeric or string expression

Remarks

The first record number in a random-access file is 0.

Data files you created using WRITE FILE statements can be accessed by READ FILE statements.

Each expression in the list is evaluated and written as a separate record in the file.

If the output has more bytes than the number of bytes specified in the record, then output continues until the end of the data.

Example

```
* LIST
10 DIM A(3,4)
20 FOR I = 1 TO 3
30   FOR J = 1 TO 4
40   LET A(I,J) = ((I-1) * 4+J) * 3
45   PRINT A(I,J)
50   NEXT J
60 NEXT I
80 PRINT
90 OPEN FILE(1,0), "TESTFILE", 20
100 FOR I1=1 TO 3
110   LET I=4-I1
120   FOR J1=1 TO 4
130   LET J=5-J1
140   LET R=(3-I) * 4+(5-J)
150   WRITE FILE(1,R),A(I,J)
160   PRINT A(I,J),
170   NEXT J1
180   PRINT
190 NEXT I1
200 CLOSE
* RUN
3
6
9
12
15
18
21
24
27
30
33
36
36  33  30  27
24  21  18  15
12  9   6   3
```

End of Chapter

Appendix A
Run-Time Error Messages

MP/BASIC Error Mesages

Code	Text
2	Illegal data type
3	Number is too big
4	Incorrect number of data items
5	Number too large to convert to integer
7	RETURN with no GOSUB
8	Line number for CONTINUE lost. Use RUN to restart
11	I don't know what this is
12	(At least one) missing right parenthesis
13	Can't find subroutine
15	No such line number
16	Insufficient memory for program
17	Insufficient memory for data
18	Program statement is too long
20	Not a valid line number
21	Wrong number of items in input reply
22	Improper numeric expression
23	New line numbers will exceed maximum
24	GOSUB with no RETURN
25	The expression doesn't point to one of the line numbers
26	Expression can't be assigned a value
27	String is too large for variable
28	File is not open
30	End of string requires “
32	File number already in use
33	Improper string expression
34	Illegal data for operand or function
35	Variable or array is being improperly used
36	Illegal BASE value
37	Illegal FOR...NEXT variable
38	File was not OPENed for random access
39	Subscript value is out of range
40	User function calls itself
41	Invalid file mode
42	REAL loop values can't be used with INTEGER variable
43	File wasn't opened for this type of operation
44	Array is too large
45	Data type has already been declared or defaulted
46	Illegal data
47	NEXT with no FOR
48	Invalid file number
49	Negative numbers can only be raised to integer powers
50	LOG function can't have a negative or zero argument
51	User function has not been defined
53	Incompatible operands
54	Zero can't have a negative exponent
55	Line numbers aren't valid
56	Handler cannot be found
57	Incorrect number of parameters
59	File is not an MP/BASIC save file
60	Save file wasn't created with this version of MP/BASIC
61	A subscripted string can't be passed by the SUMMON statement
62	Program stored with SAVE command. Use LOAD to retrieve
63	Variable has not been assigned a value
64	Expression is too complicated or recursive
65	FATAL ERROR: Too many GOSUB statements have been executed
66	Keyword cannot be used as a variable name
67	Program limit of 255 variables has been exceeded
68	There are characters after the end of the statement
69	FILE reference is not correct
70	SQR can't have a negative argument
71	There's an illegal character in the USING string
72	There's an illegal format field in the USING string
73	is too long for the format field in the USING string
74	There isn't any format field in the USING string
75	Your version of MP/BASIC does not include this feature

System Error Messages

Code	Text
16385	Argument does not exist
16386	Buffer extends into system space
16387	Buffer too short
16388	Cannot delete permanent file
16389	Renaming error (file is open, cross device)
16390	Invalid device code
16391	Device is in use
16392	Fatal device error
16393	Device is off line
16394	Device read error
16395	Device write error
16396	Directory delete error
16397	Disk label does not match disk name
16398	Disk requires fixup
16399	End of file
16400	Extant user interrupt handler
16401	File already exists
16402	File does not exist
16403	File is in use
16404	File is attribute protected
16405	File name is too long
16406	Illegal file type
16407	Illegal option combination
16408	Invalid stack definition (too small, system space)
16409	Insufficient file space
16410	Invalid address
16411	Multiple waiters for single NPSC
16412	Invalid attributes
16413	Invalid channel number
16414	Invalid character in pathname
16415	Invalid characteristics
16416	Invalid event number (<076> ?EVMAX or <074> ?EVMIN)
16417	Invalid memory request
16418	Invalid operation for device
16419	Invalid priority
16420	Invalid starting address
16421	Invalid task identifier
16422	Line is too long
16423	No debugger present
16424	No free channels
16425	No free TCB available
16426	No such user interrupt service routine exists
16427	Non-directory entry in pathname
16428	Non-system name specified
16429	Pend timeout
16430	Range error
16431	Read access denied
16432	Searchlist overflow
16433	Switch not found
16434	Task in progress
16435	Write access denied
16436	Program internal error
16437	Illegal system call
16438	Internal error
16439	No available resource
16440	Console interrupt (^C^A)
16441	Son terminated via ^C^B
16442	Illegal packet type
16443	Call aborted due to program management call
16444	Program file format revision not supported
16445	Device not mounted
16446	Maximum link depth exceeded
16447	Invalid overlay descriptor
16449	Attempt to exceed maximum swap level
16450	No overlays defined for this program
16451	Specified overlay is not currently in use
16452	All tasks have died
16453	User and system debuggers can not coexist
16454	Not enough memory
16455	Son terminated via ^C^E
16456	Invalid element size
16457	Invalid file format
16458	User PC is equal to zero
16459	Condition already exists
16460	Illegal word count (<074> 2)
16461	Bad or runaway tape, or format error
16462	Uncorrectable tape data error
16463	Fatal tape hardware error
16464	Odd number of characters read from tape
16465	Tape drive is write-locked
16466	Illegal tape record number
16467	Illegal tape file number
16468	Logical end of tape encountered
16469	Tape drive not opened
16470	Physical end of tape encountered
16471	Unexpected beginning of tape encountered
16472	Too many records in tape file (<076> 65,536)
16473	Indecipherable dump format
16474	Comm. device break received
16475	Comm. device framing error
16476	Comm. device parity error
16477	Comm. device receiver overrun
16478	Device or line does not exist
16479	Invalid mailbox number
16480	No message waiting
16481	?XMT to a mailbox currently in use
16482	Attempt to read blank tape

End of Appendix

Appendix B
MP/BASIC with Overlays

MP/BASIC uses overlays to provide a significant increase in the amount of space available for user programs and data.

Once a program starts running with the version of MP/BASIC with overlays, it will run as fast as with a non-overlay version. Because of the way the overlays have been written, the part of MP/BASIC required to run a program is loaded into the computer when you type a RUN command and remains in memory until the program ends.

If your computer uses MP/BASIC from a floppy disk, you may notice a slight delay when you type an MP/BASIC interactive command such as

```
PRINT 2+2
```

This delay is because the execution overlay must be loaded from disk. If this delay is not acceptable, you may use the alternate version of MP/BASIC with no overlays called NO_OV_BASIC.PR (for MP/AOS-SU users, the non-overlay version is called ONO_OV_BASIC.PR). In this version, there are no delays between entering a command from the keyboard and its execution, but you will have less space available for your program and data.

The important points to remember are:

- Once a program starts running, it will run equally fast under either the overlay or non-overlay version of MP/BASIC.
- In executing certain commands in immediate mode, there may be slight delays using MP/BASIC with overlays. On the other hand, the version with overlays gives you more space for your programs and data. For most situations, you would probably want to use MP/BASIC with overlays.

End of Appendix

Appendix C
Data Formats

Integers

We represent an integer as a two's complement number in a 16-bit word. The format is shown in Figure C-1. The first bit positon (bit 0) contains the sign of the number (0 for positive, 1 for negative), and bits 1-15 contain the magnitude.

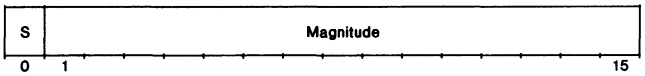


Figure C-1. Representation of Integers

We represent positive numbers in true binary form, with the sign bit set to zero. We represent negative integers in two's-complement notation, with the sign bit set to one.

The two's-complement form of a number is obtained by inverting each bit of the number and adding a one in the low-order bit position. Thus, the two's-complement form for the decimal number −34 is:

	S	Magnitude
representation of +34	0	000 0000 0010 0010
inverted	1	111 1111 1101 1101
Add 1		1
representation of −34	1	111 1111 1101 1110

Real Numbers

We represent a real number in a floating-point format, using four bytes (for single precision) or eight bytes (for double precision). The 4- or 8-byte aggregate contains 3 fields (see figure C-2):

- A fractional part called the mantissa, which, at the end of all floating-point operations, is always adjusted to be greater than or equal to 1/16 and less than 1 (this is called normalization);
- An exponent, which is adjusted to maintain the correct value of the number;
- A sign.

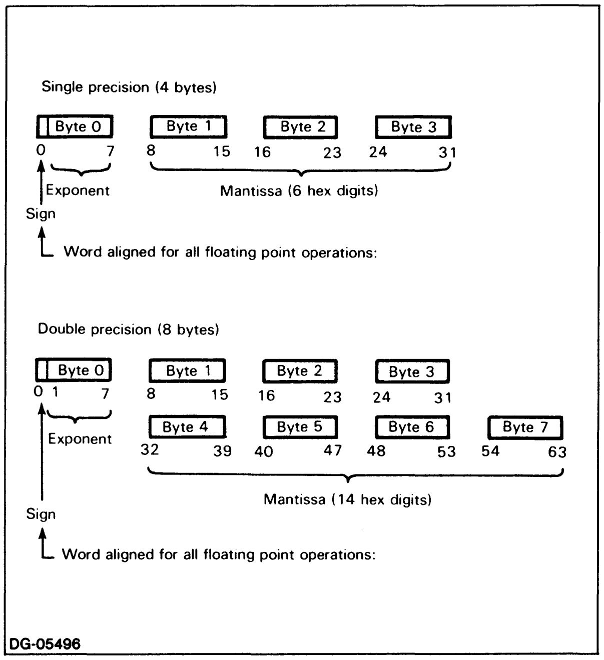


Figure C-2. Representation of Floating-Point Format

Operations on numbers in memory employing the floating-point arithmetic instructions require that the number be word-aligned, that is, bit 0 of the first byte of the number is bit 0 of the first word of a 2-word or 4-word area in memory.

The magnitude of a floating-point number is defined to be:

MANTISSA X $16^{\text{(true value of the exponent)}}$

The magnitude of a single- or double-precision number is thus on the range, approximately:

$-7.237 \times 10^{+75}$ to $7.237 \times 10^{+75}$

We represent zero in floating-point by a number with all bits zero, known as true zero. When a calculation results in a zero mantissa, the number is atomatically converted to a true zero.

Sign

Bit 0 of the first byte is the sign bit. If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative.

Exponent

The seven right-most bits of the first byte contain the exponent. We use excess 64 representation. For both positive and negative exponents, the value is 64 greater than the true value of the exponent. Table C-1 illustrates this.

Table C-1. Excess 64 Representation of Exponents

Exponent Field	True Value of Exponent
0 64 127	−64 0 63

Mantissa

Bytes 1–3 (single precision) or bytes 1–7 (double precision) contain the mantissa. By definition, the binary point lies between byte 0 and byte 1 of a floating-point number. In order to keep the mantissa in the range of 1/16 to 1, the results of each floating-point calculation are normalized by shifting it left one hex digit (4 bits) at a time, until the high-order four bits (the left-most four bits of byte 1) represent a nonzero quantity. For every hex digit shifted, the exponent is decreased by one.

Strings

We store a string in a series of 8-bit bytes, one character (in ASCII format) per byte. The bytes containing the characters in the string are preceeded by a two-word string descriptor (see figure C-3): the first word contains the current length (in number of characters) of the string, and the second word contains the maximum string length (this is the value that is set by the DIM command.

Arrays

The representation of an array in MP/BASIC begins with a two-word header (see figure C-4). The first word contains the first dimension of the array (this word is set to zero for a one-dimensional array), and the second word contains the second dimension of the array.

The header is followed by the array data. The array data is stored by row: the array elements appear sequentially in memory in the following order: A(0,0), A(0,1), A(0,2), ... , A(0,N), A(1,0), A(1,1), ..., A(1,N), ... , A(M,N).

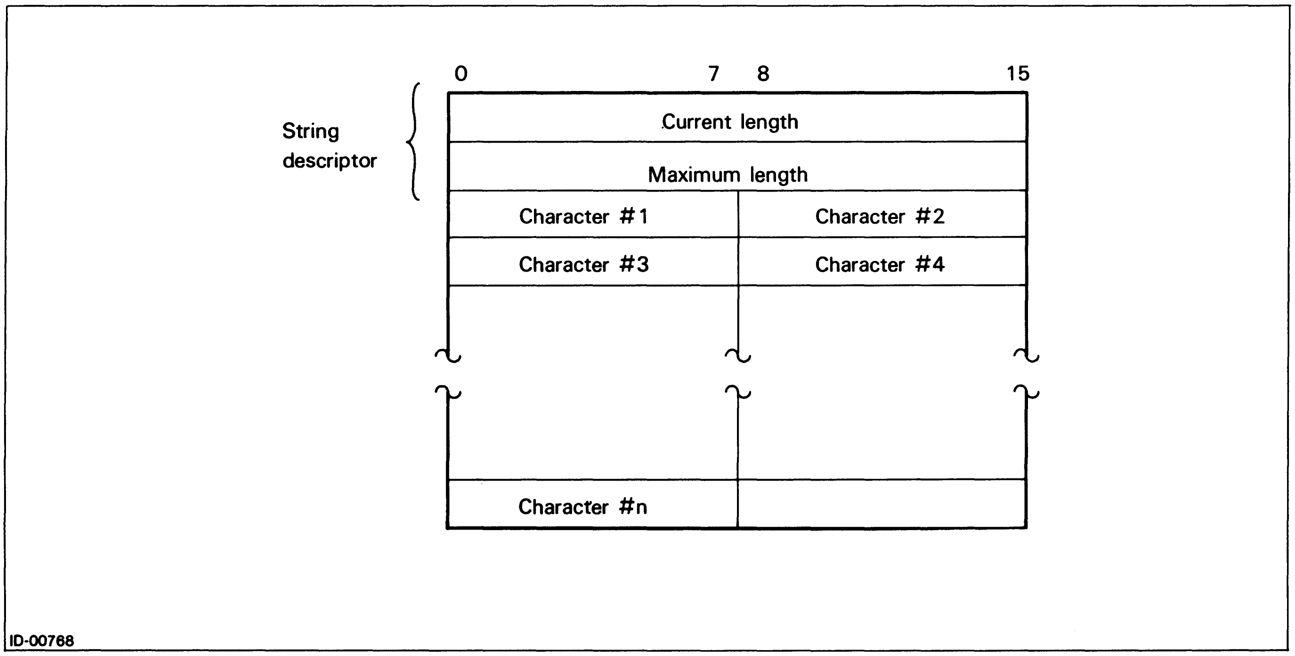


Figure C-3. Representation of Strings

The format of the array data depends on the kind of data in the array. Each data element is stored in the format described above for the corresponding data type. The number of words occupied by each data element in the data portion of the array representation is as follows:

- For integer data, each element is stored in one word in the data portion of the representation. If an element does not exist, its contents are undefined.
- For real data, each element is stored in two (for single precision) or four (for double precision) consecutive words in the data portion of the representation. If an element does not exist, its contents are undefined.
- For string data, the address of the first word of the string descriptor (see figure C.4) is stored in the corresponding element in the data portion of the array representation. If the element is vacant, the address is set to -1.

The representation for a one-dimension array is the same as for a two-dimension array; for a one-dimension array, the first header word (dimension one) is set to zero.

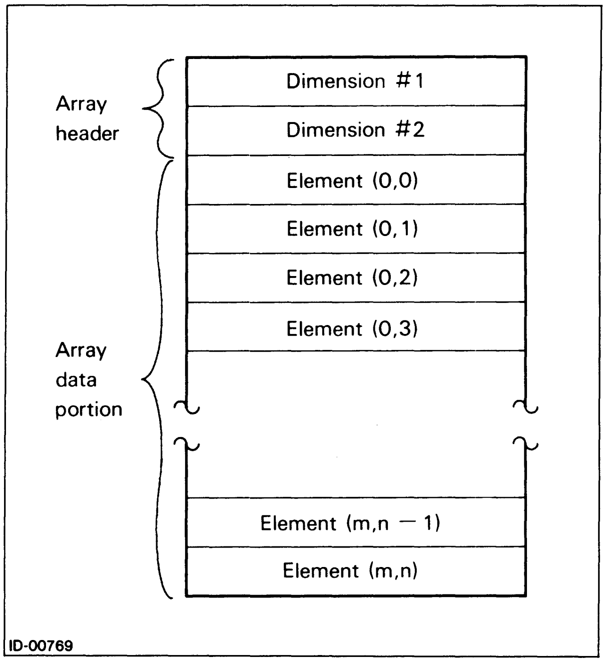


Figure C-4. Representation of Arrays

End of Appendix

Appendix D
Run-only MP/BASIC

There may be times when you would like to run an MP/BASIC program directly from the CLI.

For example, after you have finished debugging a program, there is no need to first call MP/BASIC from the CLI and then to run your program. Using a run-only version of MP/BASIC, you can call your MP/BASIC program directly from the CLI by using a single CLI command.

To use run-only MP/BASIC, you first write and debug your program, and store it using the SAVE command. Do all this using a regular (not run-only) version of MP/BASIC. Then, using a run-only version of MP/BASIC, you can use a single CLI command to run your program from the CLI.

For example, suppose you have written a program and SAVEd it in the file named PULVERIZE. You could then run that program directly from the CLI by typing the command

```
X ROBASIC PULVERIZE
```

(Note that the command to execute run-only MP/BASIC is slightly different depending upon the operating system; see Table D-1.)

Table D-1. CLI Calls to Execute Run-Only MP/BASIC

Operating System	CLI Command for Run-Only MP/BASIC
MP/OS	X ROBASIC <SAVE-file-name>
MP/AOS MP/AOS-SU	X OROBASIC <SAVE-file-name>
AOS	X MROBASIC <SAVE-file-name>
AOS/VS	X MVSROBASIC <SAVE-file-name>

When the above command is executed, the CLI calls in the run-only version of MP/BASIC, which will, in turn, call in the program in file PULVERIZE, and execute it. When the MP/BASIC program reaches a STOP or END statement, control returns to the CLI.

Because run-only MP/BASIC does not have any accessible immediate mode facilities, and because the program work area is destroyed after the MP/BASIC program finishes execution, the run-only version of MP/BASIC also prevents unauthorized tampering or examination of your MP/BASIC program.

Run-only MP/BASIC does not use an overlay, since it requires less space than the complete MP/BASIC; therefore, there is only a non-overlay version.

End of Appendix

Index

Within this index, "f" or "ff" after a page number means "and the following page" (or "pages"). Commands, calls, and acronyms are in uppercase letters (e.g., CREATE); all others are lowercase.

End of Index

CUT ALONG DOTTED LINE

Data General users group

Installation Membership Form

Name ____________________ Position ____________________ Date __________

Company, Organization or School ____

Address ____________________ City ______________ State __________ Zip __________

Telephone: Area Code __________ No. ______________ Ext. ______________

1. Account Category

□ OEM
□ End User
□ System House
□ Government

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
MV/Series ECLIPSE®		
Commercial ECLIPSE		
Scientific ECLIPSE		
Array Processors		
CS Series		
NOVA®4 Family		
Other NOVAs		
microNOVA® Family		
MPT Family		

Other __________
(Specify) __________
____

3. Software

□ AOS
□ AOS/VS
□ AOS/RT32
□ MP/OS
□ MP/AOS
□ RDOS
□ DOS
□ RTOS
□ Other

Specify __________

4. Languages

□ ALGOL
□ DG/L
□ COBOL
□ Interactive COBOL
□ PASCAL
□ Business BASIC
□ BASIC
□ Assembler
□ FORTRAN 77
□ FORTRAN 5
□ RPG II
□ PL/1
□ APL
□ Other

Specify __________
____

5. Mode of Operation

□ Batch (Central)
□ Batch (Via RJE)
□ On-Line Interactive

6. Communication

□ HASP
□ HASP II
□ RJE80
□ RCX 70
□ RSTCP
□ 4025
□ X.25
□ SAM
□ CAM
□ XODIAC™
□ DG/SNA
□ 3270
□ Other

Specify __________
____

7. Application Description

○ __________
____
____
____

8. Purchase

From whom was your machine(s) purchased?

□ **Data General Corp.**
□ Other
Specify __________
____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ __________
____
____

Data General

Data General Corporation, Westboro, Massachusetts 01580, (617) 366-8911

FOLD *FOLD*

TAPE *TAPE*

FOLD *FOLD*

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

DataGeneral

ATTN: Users Group Coordinator (C-228)
4400 Computer Drive
Westboro, MA 01581

Data General

TP____________

TIPS ORDER FORM
Technical Information & Publications Service

BILL TO:	SHIP TO: (if different)
COMPANY NAME________________	COMPANY NAME________________
ADDRESS ________________	ADDRESS ________________
CITY ________________	CITY ________________
STATE__________ ZIP __________	STATE__________ ZIP __________
ATTN: ________________	ATTN: ________________

QTY	MODEL #	DESCRIPTION	UNIT PRICE	LINE DISC	TOTAL PRICE

(Additional items can be included on second order form)

[Minimum order is $50.00]

Tax Exempt #__________
or Sales Tax (if applicable)

TOTAL	
Sales Tax	
Shipping	
TOTAL	

METHOD OF PAYMENT

□ Check or money order enclosed
For orders less than $100.00

□ Charge my □ Visa □ MasterCard
Acc't No.__________ Expiration Date__________

□ Purchase Order Number:__________________

SHIP VIA

□ DGC will select best way (U.P.S or Postal)

□ Other:
□ U.P.S. Blue Label
□ Air Freight
□ Other ____________________
____

NOTE: ORDERS LESS THAN $100, INCLUDE $5.00 FOR SHIPPING AND HANDLING.

Person to contact about this order ____________________ Phone ____________ Extension ________

Mail Orders to:

Data General Corporation
Attn: Educational Services/TIPS F019
4400 Computer Drive
Westboro, MA 01580
Tel. (617) 366-8911 ext. 4032

____
Buyer's Authorized Signature Date
(agrees to terms & conditions on reverse side)

____
Title

____
DGC Sales Representative (If Known) Badge #

**DISCOUNTS APPLY TO
MAIL ORDERS ONLY**

educational services

CUT ALONG DOTTED LINE

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form shown on the reverse hereof which is accepted by DGC.

1. **PRICES**
Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. **PAYMENT**
Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice.

3. **SHIPMENT**
Shipment will be made F.O.B. Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment.

4. **TERM**
Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. **CUSTOMER CERTIFICATION**
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

6. **DATA AND PROPRIETARY RIGHTS**
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

7. **DISCLAIMER OF WARRANTY**
DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS SUPPLIED HEREUNDER.

8. **LIMITATIONS OF LIABILITY**
IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. **GENERAL**
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer.

DISCOUNT SCHEDULES

DISCOUNTS APPLY TO MAIL ORDERS ONLY.

LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20%
15 or more manuals of the same part number - 30%

DISCOUNTS APPLY TO PRICES SHOWN IN THE CURRENT TIPS CATALOG ONLY.

User Documentation Remarks Form

Your Name ______________________ **Your Title** ______________

Company ______________________________________

Street ______________________________________

City ______________________ **State** __________ **Zip** __________

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title ______________________ **Manual No.** __________

Who are you? □**EDP Manager** □**Analyst/Programmer** □**Other** __________

□**Senior Systems Analyst** □**Operator** __________

What programming language(s) do you use? ______________________

How do you use this manual? *(List in order: 1 = Primary Use)* ______________________

___ **Introduction to the product** ___ **Tutorial Text** ___ **Other**

___ **Reference** ___ **Operating Guide** __________

		Yes	**Somewhat**	**No**
About the manual:	**Is it easy to read?**	□	□	□
	Is it easy to understand?	□	□	□
	Are the topics logically organized?	□	□	□
	Is the technical information accurate?	□	□	□
	Can you easily find what you want?	□	□	□
	Does it tell you everything you need to know	□	□	□
	Do the illustrations help you?	□	□	□

If you have any comments on the software itself, please contact Data General Systems Engineering.
If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Remarks:

Date

134-664

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

POSTAGE WILL BE PAID BY ADDRESSEE

DataGeneral

User Documentation, M.S. E-111
4400 Computer Drive
Westborough, Massachusetts 01581

Data General Corporation, Westboro, MA 01580