Honeywell

TIME-SHARING BASIC

SERIES 600/6000

SOFTWARE

Honeywell

TIME-SHARING BASIC

SERIES 600/6000

SUBJECT:

Introduction to Terminal Operation and the BASIC Language; Advanced BASIC Programming Techniques; Error Messages and Debugging.

SPECIAL INSTRUCTIONS:

This manual, Order Number BR36, Rev. 1, supersedes BR36, Rev. 0, and CPB-1510B, dated March 1970, and Addendum No. 1, dated February 1971. The new order number is assigned to be consistent with the overall Honeywell publications numbering system. New information and changes added to the manual since the previous edition are indicated by marginal change bars.

SOFTWARE SUPPORTED:

Series 600 Software Release 8.0
Series 6000 Software Release F

INCLUDES UPDATE PAGES ISSUED AS ADDENDUM A IN JUNE 1972; ADDENDUM B, OCTOBER 1972; AND ADDENDUM C, MAY 1973.

DATE:

September 1971

ORDER NUMBER:

BR36, Rev. 1 (Supersedes BR36, Rev. 0, and CPB-1510B)

PREFACE

This manual is intended to provide a reference source for users of BASIC. For the new user, a description of terminal operation is supplied and the elements of the language are listed. For the experienced user, methods for more advanced use of the language are provided. To facilitate the use of BASIC, possible error messages that may be encountered are listed and an example of program error location and correction (debugging) is given. Portions of this manual are based upon the BASIC language developed by Dartmouth College. Time-Sharing BASIC is a subsystem of the Series 600/6000 Time-Sharing System.

Series 600/6000 Time-Sharing BASIC is a coded program designed to extend the power of Series 600/6000 in the area of program preparation and maintenance. It is supported by comprehensive documentation and training; periodic program maintenance and, where feasible, improvements are furnished for the current version of the program, provided it is not modified by the user.

File No.: 1623, 1723

FUNCTIONAL LISTING OF PUBLICATIONS
for
SERIES 600 SYSTEM

FUNCTION	APPLICABLE REFERENCE MANUAL TITLE	FORMER PUB. NO.	ORDER NO.
	Series 600:		
Hardware reference:			
Series 600	System Manual	371	BM78
DATANET 355	DATANET 355 Systems Manual	1645	BS03
Operating system:			
Basic Operating System	General Comprehensive Operating Supervisor (GCOS)	1518	BR43
Control Card Formats	Control Cards Reference Manual	1688	BS19
System initialization:			
GCOS Startup	System Operating Techniques	DA10	DA10
Communications System	GRTS/355 Startup Procedures Reference Manual	1715	BJ70
Storage Subsystem Startup	DSS180 Disk Storage Subsystem Startup Procedures	DA11	DA11
Data management:			
File System	File Management Supervisor	DB54	DB54
Integrated Data Store (I-D-S)	Integrated Data Store	1565	BR69
File Processing	Indexed Sequential Processor	DA37	DA37
Multi-Access I-D-S	Multi-Access I-D-S Implementation Guide	DA80	DA80
File Input/Output	File and Record Control	1003	BN85
I-D-S Data Query System	I-D-S Data Query System Installation	DB57	DB57
I-D-S Data Query System	I-D-S Data Query System User's Guide	DB56	DB56
Program maintenance:			
Object Program	Source and Object Library Editor	1723	BJ71
System Editing	System Library Editor	1687	BS18
Test system:			
On-Line Peripheral testing	GCOS On-Line Peripheral Test System (OPTS-600)	1573	BR76
Total On-Line testing	Total On-Line Test System (TOLTS)	DA49	DA49
Language processors:			
Macro Assembly Language	Macro Assembler Program	1004	BN86
COBOL Language	COBOL Compiler	1652	BS08
COBOL Usage	COBOL User's Guide	1653	BS09
ALGOL Language	ALGOL	1657	BS11
JOVIAL Language	JOVIAL	1650	BS06
FORTRAN Language	FORTRAN	1686	BJ67
FORTRAN IV Language	FORTRAN IV	1006	BN88
DATANET 355	DATANET 355 Macro-Assembler Program	1660	BB98
Generators:			
Sorting	Sort/Merge Program	1005	BN87
Merging	Sort/Merge Program	1005	BN87
Simulators:			
DATANET 355 Simulation	DATANET 355 Simulator Reference Manual	1663	BW23

FUNCTION	APPLICABLE REFERENCE MANUAL TITLE	FORMER PUB. NO.	ORDER NO.
	Series 600:		
Remote terminal system:			
DATANET 30	NPS/30 Programming Reference Manual	1558	BR68
DATANET 30/305/355	GRTS Programming Reference	DA79	DA79
Service and utility routines:			
Loader	General Loader	1008	BN90
Utility Programs	Utility	1422	BQ66
Conversion	Bulk Media Conversion	1096	BP30
System Accounting	GCOS Accounting Summary Edit Programs	1651	BS07
FORTRAN	FORTRAN Subroutine Libraries Reference Manual	1620	BR95
Controller Loader	Relocatable Loader	DA12	DA12
Service Routines	Service Routines	DA97	DA97
Software Debugging	Trace and Debug Routines	DB20	DB20
Time-sharing systems:			
Operating System	GCOS Time-Sharing System General Information	1643	BS01
System Programming	GCOS Time-Sharing Terminal/Batch Interface Facility	1642	BR99
System Programming	GCOS Time-Sharing System Programmers' Reference Manual	1514	BR39
BASIC Language	Time-Sharing BASIC	1510	BR36
FORTRAN Language	Time-Sharing FORTRAN	1566	BR70
Text Editing	Time-Sharing Text Editor	1515	BR40
Transaction processing:			
User's Procedures	Transaction Processing System User's Guide	DA82	DA82
Handbooks:			
Console Messages	Console Typewriter Messages	1477	BR09
Index	Comprehensive Index	1499	BR28
Pocket guides:			
Time-Sharing Programming	GCOS Time-Sharing System	1661	BS12
Macro Assembly Language	GMAP	1673	BS16
COBOL Language	COBOL	1689	BJ68
Control Card Formats	GCOS Control Cards & Abort Codes	1691	BJ69
Software maintenance (SMD):			
Table Definitions	GCOS Introduction & System Tables SMD	1488	BR17
Startup program	Startup (INIT) SMD	1489	BR18
Input System	System Input SMD	1490	BR19
Peripheral Allocation	Dispatcher and Peripheral Allocation SMD	1491	BR20
Core Allocation/Rollcall	Rollcall, Core Allocation and Operator Interface SMD	1492	BR21
Fault Processing	Fault Processing SMD	1493	BR22
Channel Modules	I/O Supervisor (IOS) SMD	1494	BR23
Error Processing	GCOS Exception Processing SMD	1495	BR24
Output System	Termination and System Output SMD	1496	BR25
File System Modules	File System Maintenance SMD	1497	BR26
Utility Programs	GCOS Utility Routines SMD	1498	BR27
Time-Sharing System	Time-Sharing Executive SMD	1501	BR29

Rev. 7303

FUNCTIONAL LISTING OF PUBLICATIONS
for
SERIES 6000 SYSTEM

FUNCTION	APPLICABLE REFERENCE MANUAL TITLE	FORMER PUB. NO.	ORDER NO.
	Series 6000:		
Hardware reference:			
Series 6000	Series 6000 Summary Description	DA48	DA48
DATANET 355	DATANET 355 Systems Manual	1645	BS03
Operating system:			
Basic Operating System	General Comprehensive Operating Supervisor (GCOS)	1518	BR43
Control Card Formats	Control Cards Reference Manual	1688	BS19
System initialization:			
GCOS Startup	System Startup and Operation	DA06	DA06
Communications System	GRTS/355 Startup Procedures Reference Manual	1715	BJ70
Storage Subsystem Startup	DSS180 Disk Storage Subsystem Startup Procedures	DA11	DA11
Data management:			
File System	File Management Supervisor	DB54	DB54
Integrated Data Store (I-D-S)	Integrated Data Store	1565	BR69
File Processing	Indexed Sequential Processor	DA37	DA37
Multi-Access I-D-S	Multi-Access I-D-S Implementation Guide	DA80	DA80
File Input/Output	File and Record Control	1003	BN85
I-D-S Data Query System	I-D-S Data Query System Installation	DB57	DB57
I-D-S Data Query System	I-D-S Data Query System User's Guide	DB56	DB56
Program maintenance:			
Object Program	Source and Object Library Editor	1723	BJ71
System Editing	System Library Editor	1687	BS18
Test system:			
On-Line Peripheral Testing	GCOS On-Line Peripheral Test System (OPTS-600)	1573	BR76
Total On-Line Testing	Total On-Line Test System (TOLTS)	DA49	DA49
Error Analysis and Logging	Honeywell Error Analysis and Logging System	DB50	DB50
Language processors:			
Macro Assembly Language	Macro Assembler Program	1004	BN86
COBOL Language	COBOL Compiler	1652	BS08
COBOL Usage	COBOL User's Guide	1653	BS09
ALGOL Language	ALGOL	1657	BS11
JOVIAL Language	JOVIAL	1650	BS06
FORTRAN Language	FORTRAN	1686	BJ67
DATANET 355	DATANET 355 Macro-Assembler Program	1660	BB98
Generators:			
Sorting	Sort/Merge Program	1005	BN87
Merging	Sort/Merge Program	1005	BN87

FUNCTION	APPLICABLE REFERENCE MANUAL TITLE	FORMER PUB. NO.	ORDER NO.
	Series 6000:		
Simulators:			
DATANET 355 Simulation	DATANET 355 Simulator Reference Manual	1663	BW23
Service and utility routines:			
Loader	General Loader	1008	BN90
Utility Programs	Utility	1422	BQ66
Conversion	Bulk Media Conversion	1096	BP30
System Accounting	GCOS Accounting Summary Edit Programs	1651	BS07
FORTRAN	FORTRAN Subroutine Libraries Reference Manual	1620	BR95
Controller Loader	Relocatable Loader	DA12	DA12
Service Routines	Service Routines	DA97	DA97
Software Debugging	Trace and Debug Routines	DB20	DB20
Time-sharing systems:			
Operating System	GCOS Time-Sharing System General Information	1643	BS01
System Programming	GCOS Time-Sharing Terminal/Batch Interface Facility	1642	BR99
System Programming	GCOS Time-Sharing System Programmers' Reference Manual	1514	BR39
BASIC Language	Time-Sharing BASIC	1510	BR36
FORTRAN Language	FORTRAN	1686	BJ67
Text Editing	Time-Sharing Text Editor	1515	BR40
Remote terminal system:			
DATANET 30	NPS/30 Programming Reference	1558	BR68
DATANET 30/305/355	GRTS Programming Reference	DA79	DA79
Transaction processing:			
User's Procedures	Transaction Processing System User's Guide	DA82	DA82
Handbooks:			
Console Messages	Console Typewriter Messages	1477	BR09
Index	Comprehensive Index	1499	BR28
Pocket guides:			
Time-Sharing Programming	GCOS Time-Sharing System	1661	BS12
Macro Assembly Language	GMAP	1673	BS16
COBOL Language	COBOL	1689	BJ68
Control Card Formats	GCOS Control Cards and Abort Codes	1691	BJ69

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

SECTION I

INTRODUCTION

COMPUTER PROGRAMS

A computer program is a set of instructions that tells a computer how to accomplish a specific task. Each instruction is performed in the sequence specified by the program. In this way, the computer processes and produces information as directed by the program.

A program must meet two primary requirements before it can be run (have all instructions executed) on a computer.

- The program must be submitted to the computer in a language that the computer recognizes.
- All language instructions must be complete and be precisely stated.

PROGRAMMING LANGUAGES

Human languages are impractical for preparing computer programs because these languages contain many ambiguities and redundancies; the computer interprets language absolutely literally. By the same token, machine languages are also impractical because they are difficult for people to use. Most programming languages are compromises between human and machine languages.

BASIC PROGRAMMING LANGUAGE

BASIC (Beginner's All-Purpose Symbolic Instruction Code) is a problem-oriented, algebraic programming language that enables the user to present his program in ordinary mathematical notation, with simple and precise vocabulary and grammar. BASIC is intended to be used with a keyboard-type terminal tied into a time-sharing system.

TIME-SHARING SYSTEM

The time-sharing system uses a technique by which programs are handled in parallel. A supervisory program acts as a controller of these programs, controlling "stop" and "go" signals to inputs from terminals and preventing demands of one terminal from interfering with demands of other terminals. Thus, time-sharing permits a user to work directly with the computer, whether it is within his sight or thousands of miles away.

The user believes that he has exclusive use of the computer, even though many others at the same time share this illusion.

Time-sharing permits a dialogue between the computer and user, permitting the dialogue to begin immediately, without waiting for the computer to complete previous programs. Data is fed from the terminal directly to the computer and answers are received quickly at the same terminal.

If the program contains a mistake, the computer informs the user.

The program can be corrected or changed by the user as if he were conversing by phone, except in this case, the conversation is typed or displayed, dependent upon the type of terminal in use.

BASIC AND TIME-SHARING

Because BASIC is such a simple programming language and because time-sharing permits the correction and completion of most problems within minutes, BASIC in use in a time-sharing system provides a highly satisfactory computation environment for both the novice and experienced programmer.

Appendix A is a glossary of time-sharing terms which may be encountered by the user of BASIC.

SECTION II

BASIC LANGUAGE CHARACTERISTICS

BASIC PROGRAMS

A BASIC program is a sequence of instructions to the computer that results in the solution of a problem when the instructions are executed. These instructions are given in units called "statements". The computer executes the statements and produces the desired output.

Statements in a BASIC program may be used in elementary or sophisticated ways, depending upon the user's level of programming knowledge and skill. Information pertaining to the use of BASIC on an elementary level is contained in Section IV. Some advanced techniques in BASIC usage are contained in Section V.

BASIC STATEMENTS

The BASIC language allows the user seeking the solution to an algebraic problem to select words that, when formatted by the user into complete statements, result in powerful computer operations. When placed in a meaningful sequence, these statements constitute a BASIC program.

BASIC-language statements may be grouped into functional categories as follows:

Functional Category	When used in BASIC programs, these statements
arithmetic	- perform arithmetic operations
specification	- specify input data values and sizes of lists and tables
input/output	- direct input/output operations
loop and subroutine	- provide direction and control for use of loops and subroutines contained in the programs
logic	- modify the processing sequence when certain conditions occur
utility	- provide service routines
documentation	- allow remarks to be inserted into the program.

BASIC WORDS

BASIC words are short, distinctive, easily recognizable words that are either valid words or abbreviations of words. When formatted into a statement, a word becomes an explicit instruction to the computer to perform some operation. Some statements can be made by the use of a BASIC word alone; other statements require other information in addition to the BASIC word.

BASIC words may be grouped by type of statements in which they occur. The words and their associated functional statement categories are as follows:

Arithmetic Statements

BASIC Word	When formatted into a statement
DEF	- defines a repeatedly used function
LET	- requests a computation or manipulation upon an arithmetic variable
MAT	- requests a computation or manipulation upon a matrix

Specification Statements

BASIC WORD	When formatted into a statement
CHANGE	- converts string characters to numerical code or vice versa
DATA	- specifies numeric values for variables listed in a READ statement
DIM	- reserves space for list or table

Input/Output Statements

BASIC Word	When formatted into a statement
INPUT	- delays input of values to variables until program is in execution; program will request input of data by terminal user or a user's file when statement is executed
PRINT	- prints computed results; prints text - prints computed results and text - skips lines - formats output data

PRINT
USING - formats output line

READ - reads values from a DATA statement or user's file and assigns them to designated variables

RESTORE - restores previously processed blocks of input data from DATA statements

Loop and Subroutine Statements

BASIC Word	When formatted into a statement
CALL	- directs processing sequence to a subroutine previously saved
FOR	- is first statement of a loop and sets conditions of loop
NEXT	- is last statement of loop
GOSUB	- directs processing sequence to a subroutine
RETURN	- returns processing sequence from a subroutine

Logic Statements

BASIC Word	When formatted into a statement
GOTO	- unconditionally transfers the processing sequence to a designated statement
IF-----THEN or IF---GOTO	- conditionally transfers the processing sequence to a designated statement
ON-----THEN or ON---GOTO	- conditionally transfers the processing sequence to designated statements
STOP	- stops the execution of the program
END	- indicates end of program

Utility Statements

BASIC Word	When formatted into a statement
CHAIN	- compiles and executes series of programs
TRACE ON TRACE OFF	- prints line numbers of statements between TRACE ON/TRACE OFF statements

Documentation Statement

BASIC Word	When formatted into a statement
REM	- inserts a remark into the statement sequence

CREATING A BASIC PROGRAM

The essentials for forming statements and creating a BASIC program are as follows:

- Each statement includes or is composed of a BASIC word.[1]
- Each statement is prefixed by a line number to specify the order in which the statement is to be executed.
- Line numbers for statements are limited to a maximum of eight digits (a range of 1 to 99999999). If a line number contains two or more digits, no spaces are permitted between digits of the number. The number itself must be followed by one or more spaces.
- Each statement of a program must be completed within one line.[2]
- Statements may be entered out of numerical order. (Before the program is executed, the computer sorts and edits the program so that the statements are sequenced in the order specified by their line numbers.) If a line number has been duplicated, only the last statement identified by that number will be retained.

  The choice of line numbers for statements is arbitrary. The user may wish to assign a 1, 2, 3, ...order to a sequence of statements, a 10, 20, 30, ...order, or even an order which has no pattern (e.g., 8, 19, 27,....). Any numbering order with intervals between numbers would be a better choice than the first because it permits the insertion of statements after the initial program entry.
- Except within line numbers, omission or insertion of spaces will not affect the execution of a statement. Spacing within a statement to make it more readable is optional.

An example of a statement is as follows:

```
10 READ A,B,C,D
```

The line is identified as statement 10, READ is the BASIC word, and A,B,C,D are variables.

A second example is:

```
40 END
```

The line is identified as statement 40, and END is the BASIC word constituting the statement.

---

[1]An exception is the LET statement, where the BASIC word LET may be implied.

[2]Multiple statements are permitted within one line (see Section V).

The actual entry at the terminal of a sequence of statements of a BASIC program requires knowledge of control commands (described below), terminal operation (given in Section III), and elementary BASIC (described in Section IV).

CONTROL COMMANDS

Control commands are used to direct the BASIC system regarding the disposition or manipulation of a BASIC program. The system can be commanded, for example, to execute or list the program or save it for future use. Commands differ from statements in that they do not form a part of the program and are effective immediately upon being entered at the terminal. A control command is a "one-shot" instruction to the system as opposed to a statement which, although entered at the terminal only once, may be executed by the system many times. Control commands are not prefixed with line numbers and may be entered at any time the BASIC system is in control.

Control commands are part of the time-sharing system control language. Only those control commands and their formats most commonly used with BASIC are described below. All control commands and all possible control command formats for use with BASIC are described in Time-Sharing System General Information Manual.

Control commands most applicable to BASIC are as follows:

RUN

This command instructs the system to execute program statements in numerical sequence. (The execution of the program is commonly referred to as "running" the program, or as a "run" of a program.)

Another form of RUN is

RUNH

This form of the command has the same effect as RUN but with the addition of a header line which is printed before program execution. The header consists of the date and time of the run, with the time expressed in hours and hundredths of hours (hh.hh).

LIST

This command is given when the user wants his program to be printed. The command will result in a printout of the entire program, along with any additions or changes that may have been made prior to the use of LIST. If only a portion of the program is desired, the LIST command can be modified by line numbers indicating the portion desired, as follows:

LIST xxxx,yyyy

will result in a printout of the program between line numbers xxxx and yyyy.

LIST xxxx

will result in the printout of statements beginning with statement xxxx through the end of the program.

LIST ,yyyy

will result in a printout of statements from the beginning of the program through statement yyyy.

DONE

The user terminates his session with the BASIC system by the use of this command but may still retain use of the terminal for selection of another time-sharing system (or reselection of BASIC).

BYE

When the user wishes to terminate his session with the computer, this command is given. He will then receive a summary of the amount of resources used for this session and the total resources used by his account to date. His terminal will then be disconnected from the system.

SAVE filename

This command permits the user to save a program for future use. Filename can be any combination of alphanumeric, period, and minus sign characters, but cannot exceed eight characters. This command is given just prior to discontinuing the immediate use of the program.

RESAVE filename

If the user wishes to make changes in a program which has been previously saved, he must make use of the RESAVE command to save the changed program; i.e., the SAVE command cannot be used to resave an altered program. Note that the original program will be purged upon use of the RESAVE command. The RESAVE command may also be used to place a current file on an existing permanent file.

NEW

This command is given when the user wishes to continue the use of BASIC by building a new program.

OLD filename

This command is given if the user wishes to select another saved program as his current program. Other forms of the OLD command are as follows.

OLD filename (xxxx,yyyy)

The statements numbered xxxx to yyyy, inclusive, of the program saved under the name filename are brought into the user's working storage for processing.

OLD $\underline{\text{filename}}_1$; $\underline{\text{filename}}_2$;...; $\underline{\text{filename}}_n$

The n named programs are adjoined in the order given and are brought into the user's working storage. (The line numbers of the resultant program are not resequenced.) The contents of the current file can be included in the new file by the use of the name "*" in the file name list. If the list is too long for one line, it may be continued on the next line if a semicolon is the last nonblank character before the carriage return.

OLD $\underline{\text{filename}}_1$...; $\underline{\text{filename}}_2$...: $\underline{\text{filename}}_n$...

The files and/or file segments specified and separated by the semicolon are adjoined in the order listed and become the current file. Filenames which are separated by a colon are "weaved" and then adjoined to the current file. For example, a file containing statement numbers 10, 20, 30, 40.... and another file with numbers 5, 15, 25...., when "weaved" make a file with statements numbered 5, 10, 15, 20, 25.... If the same statement number occurs in more than one of the files being "weaved", both statements appear in the "weaved" file and in the order in which the files were named.

The asterisk (*), designating current file, may appear as a filename anywhere in the OLD list.

OLD $\underline{\text{filename}}_1$($xxxx_1$,$yyyy_2$);...;
$\underline{\text{filename}}_n$($xxxx_n$,$yyyy_n$)

The segments of the named files specified by line numbers xxxx through yyyy are adjoined in the order given and replace the user's current program. (The line numbers of the resultant program are not resequenced.) If the list is too long for one line, it can be continued on the next line if a semicolon is the last nonblank character before the carriage return. For example, the command

```
OLD PROGRAM1(10,85);PROGRAM4
```

will cause the statements numbered 10 through 85 of the file PROGRAM1, and the statements of the file PROGRAM4 to be concatenated in that order, to become the (new) current program.

GET filedescr (applicable permissions)

where filedescr represents the full file description (user identification, catalog and file names, and any required passwords) and permissions are READ and WRITE.

The permanent file designated by filedescr is accessed and made available to the user.

RESEQUENCE

This command causes the line numbers of the current program to be resequenced. Resequencing begins with line number 10 and is incremented by steps of 10. Statement-number references within the program (such as GOTO, GOSUB, and IF statements) are modified correspondingly. Another form of RESEQUENCE is

RESEQUENCE n,m,x-y

The line numbers of the current program are resequenced, beginning with line number n and with increments of m. Either n or m may be omitted; the value 10 will be assumed in either case.

x and y are used only if partial resequencing is desired. x indicates the starting line number and y the ending line number for resequencing. x- would result in resequencing from line x to the end of the file; -y would result in resequencing from the beginning of the file to line y.

AUTOMATIC

This command causes the automatic creation of line numbers, beginning at the point at which the automatic mode is entered (or re-entered), with line numbers initially starting at 10 and incremented in steps of 10. These line numbers are generated by the system, appear in the terminal copy, and are written in the file, just as though the user had typed them himself.

Another form of AUTOMATIC is

AUTOMATIC n,m

Automatic creation of line numbers begin with line number n and are incremented by m.

Normally, the generated line number will be followed by a blank. Any nonblank, nonnumeric characters affixed to the end of the command word will suppress the blank. For example:

AUTOMATICNB or AUTOMATICX

No further commands are recognized while the system is in automatic mode. The automatic mode is cancelled by a carriage return immediately following the issuance of an asterisk and line number.

DELETE

This command is given when the user wishes selective lines of his current file deleted. The DELETE command must be accompanied by operands to indicate lines to be deleted. For example:

```
DELETE a,f-k
```

will result in lines numbered a, and f through k being deleted.

This form of the command,

```
DELETE;*
```

will result in all lines of the current file being deleted.

TAPE

This command implies that statements are to be entered from the paper-tape reader instead of the keyboard. See "Entering the Program From Paper Tape", Section III, for detailed instructions.

STOPPING PROGRAM EXECUTION

Most BASIC programs are designed to process data, display the results, and then halt. In these cases, the system will automatically return to a "ready-for-further-input" mode, indicated by the word READY and an asterisk printed out by the terminal. All control commands described above are accepted in this mode.

Some programs, however, are designed to ask for keyboard input during execution. The input is processed, the results displayed, and the program then continues. When a program is awaiting numeric input, the response S (or any word beginning with the letter S, e.g., STOP) will cause termination of the program. *

SECTION III

TERMINAL OPERATION AND PROCEDURES

TERMINAL OPERATION

The assumption is made in this manual that the terminal available to the user is a Teletype[1] Model 33 or 35. With this terminal, the communication between user and computer is displayed by means of typed copy on paper. The Teletype keyboard is a standard typewriter keyboard for the most part, but there are special-purpose keys that the user must be familiar with. These are as follows:

Key	Function
RE-TURN	Depressing this key returns the carriage and transmits the typed line to the system. The computer ignores the typed line until this key is depressed.
CTRL plus X	When these keys are depressed simultaneously, the terminal deletes the entire line being typed. The word DEL is printed and the carriage is returned. The line is ignored by the computer.
SHIFT PLUS @ P	The @ symbol is located on the P key and is generated when depressed with either shift key. It is used to delete the character or space immediately preceding the @. If this key is depressed n times, the n preceding characters or spaces will be deleted. For example:
	ABCWT@@DE will be treated as ABCDE when RETURN is depressed.
	AB C@@@CDE will be treated as ACDE when RETURN is depressed.

[1]For a complete description of the Teletype unit, refer to instruction manual accompanying the unit.

BR-EAK

Depressing this key causes the system to discontinue printing or computation. One type of terminal requires that a BRK-RLS (break-release) button be depressed following the use of BREAK in order that operations continue. BREAK should be used sparingly and with discretion.

Other operational controls, not on the keyboard, are necessary to the operation of the terminal -- power on-off, connection to a phone line, and selection of operating mode. The location and operation of these controls differ with the type of terminal in use. The user must receive on-site instruction or must study the instruction manual for his terminal to gain familiarity with these operational controls.

CONNECTING TERMINAL TO THE COMPUTER

In order to connect with the computer from a terminal, proceed as follows:

1. Turn unit on and obtain a dial tone.
2. Dial one of the numbers at the Time-Sharing Center.

When the connection is made, a high-pitched tone is received, then no tone at all, and the terminal prints out an indication that the computer is available and that communication with the computer, through the terminal, can be now made.

GETTING ON (LOG-ON) PROCEDURE

With the terminal connected to the computer, the system initiates a "log-on" procedure. During this procedure the terminal will ask for information and a proper response must be made, each response followed by a carriage return (achieved by depressing the RETURN key). First, the terminal will ask for a user's identification. This is a string of characters that is assigned to uniquely identify the user to the computer for the purpose of identifying his programs and accounting for the user's charges.

The terminal will next ask for a password. The area on which the password is printed will be scored over by the terminal to make the password illegible. The purpose of this password is to assure the computer that it is "talking" to the legitimate user and not someone else using his identification. The password is his protection against unauthorized use of his user identification.

The terminal will then ask the user to select the system he wishes to use (in this case, BASIC). If an invalid system name is given, the system will print the message SYSTEM UNKNOWN and repeat the request for a name until a valid name is given. After a valid response, the terminal will ask if the user is going to work with an OLD or NEW program, to which the user must reply, either with OLD or NEW.

A NEW program is one in which the user will enter all of the program statements at this session at the terminal. An OLD program is a program that has been previously generated at other sessions at the terminal and has been saved for future use. If the user's response is OLD to the question OLD or NEW, the system will ask him for the OLD file name. This will be the same name that he had previously used when saving his program with the control command SAVE.

After the terminal prints READY and an asterisk on a following line, the user may begin entering his new program, add or modify statements in his old program, or use one of the control commands (e.g., LIST or RUN). A typical log-on sequence follows:

```
HIS SERIES 6000, SERIES 600 ON 05/28/71 AT 9.183
USER ID--DOE
PASSWORD
ABCDEFGHDKEF
SYSTEM  ? BASIC
OLD OR NEW-NEW
READY
*
```

This example illustrates the most elementary use of the OLD/NEW selection of programs.

ENTERING THE PROGRAM

After the terminal prints READY, it indicates its availability for input from the user by printing an asterisk on the next line at the left margin. Thereafter, each carriage return generates an asterisk at the left margin of each succeeding line, indicating readiness for input. Each statement should begin with a line number (after the asterisk) that contains no more than eight digits and contains no spaces or nondigit characters. The RETURN key must be depressed at the completion of each line of input to achieve a carriage return and transmission of the information to the computer.

The program input for a simple program, following READY and the subsequent asterisk, would appear as follows (refer to Section IV for information pertaining to the formatting of statements).

```
READY
*10 FOR I = 1 to 45
*20 PRINT 2**I;
*30 NEXT I
*40 END
*RUN
```

This program would print the powers of 2 for exponents 1 through 45 upon the receipt of the control command RUN.

ENTERING THE PROGRAM FROM PAPER TAPE

If the user wishes to enter his program from paper tape, he must respond with the control command TAPE after READY. The procedure for using paper tape is as follows:

1. Place paper tape in terminal tape reader.
2. Select tape-input operating mode, if required.
3. Start tape reader.
4. Input from paper tape will be accepted until one of the following occurs:

   a. tape reader is turned off,
   b. tape runs out,
   c. tape jams in tape reader, or
   d. an X OFF character is encountered on the tape.

Refer to <u>Time-Sharing System General Information Manual</u> for a description of the preparation of paper tape.

CORRECTING THE PROGRAM

If the user, while entering his program, has made errors which are self-evident, he can correct his program in the midst of his typing or before giving the RUN command as follows:

- A new statement may be substituted for a statement containing errors by retyping the statement number and a corrected version of the statement. The first version of the statement will be ignored in the running or listing of the program.

- A statement may be eliminated from the program by typing its number and depressing the RETURN key. This statement will be ignored in the running or listing of the program.

- The current line being typed can be deleted by depressing CTRL and X keys simultaneously. That line will be ignored.

- Typing errors, if perceived during the typing process, may be corrected by use of the @ symbol. The character or space immediately preceding the @ will be deleted. If this key is depressed n times, the n preceding characters or spaces will be deleted.

- Additional statements may be inserted into the program by typing them with line numbers which indicate their places within the program sequence. For example, if one or more new statements are desired between statements 30 and 40, they could be assigned line numbers from 31 to 39. In the running or listing of the program, the new statements will be properly sequenced.

If language errors (statements violating the BASIC language format) are made by the user in entering his program and are not perceived, error messages of a diagnostic nature will be printed upon use of the control command RUN, so as to aid the user in making corrections. Error messages may be grouped as follows:

- Compilation error messages - those that are printed during program entry. They prevent further entries or program execution until the errors are corrected.

- Execution error messages - those that are printed during program execution, and may or may not stop program execution.

Section VI lists all error messages along with interpretations of the messages.

Section VII contains a sample program and information on how errors can be located and corrected (debugged) so as to attain a successful program execution.

RUNNING THE PROGRAM

After typing in the complete program, the user types the control command RUN and depresses the RETURN key. If there are no format errors, the computer will execute the statements and the terminal will print out the results. If it is obvious to the user that wrong answers are being given, he can depress the BREAK key and output will cease.

The BRK-RLS button must then be depressed in order to permit further use of the system. If logical errors were made by the user in constructing his program, the results will be erroneous or may not appear at all. Logical errors do not generate error messages. They must be found by analyzing the program. Upon completion of program execution and its resulting output (if any), the terminal prints READY to indicate the system's availability for further input. If the user wishes to modify his program, he may now do so by retyping only those statements he wishes changed to achieve the desired modification. When the control command RUN is again given, a new output will be produced. The modification process can be repeated as often as the user wishes. The control command LIST may be used at any time the user wishes to inspect the current content of his program, it will show the result of any modifications.

If the user wishes to save his program for use at another time, he must use the control command SAVE filename, the system will respond with

DATA SAVED-filename

where filename is the name under which the program is saved. If the user wishes to discontinue working with his present problem but wishes to continue the use of BASIC, he may use either the command NEW or OLD. If NEW is typed, the system will respond with READY and the user can then enter a new program. If OLD is typed, the system will ask for OLD NAME-. When the old program name filename is supplied, the system will respond with READY. Modifications can be made, as with a NEW program, and the program can be listed. Upon the control command RUN, the old program will be run. (The entry OLD filename will bypass the request OLD NAME-.)

Note

> The old program must be a BASIC program and one which has been saved at a previous session at the terminal. Access to programs other than the user's own requires use of the ACCESS subsystem as described in Section V, "File Access".

If, while BASIC is requesting input from the terminal, the user types the control command DONE, the time-sharing system will sign the user off the BASIC system but will permit him to select another system within the confines of the time-sharing system and continue with the use of the computer. If, during execution of a BASIC program, the program halts and asks for new data input, a response of STOP (or S) will break off program execution.

GETTING OFF (LOG-OFF) PROCEDURE

If, while BASIC is requesting input from the terminal, the user types the control command BYE, the time-sharing system will "log-off" the user and disconnect the terminal. The time-sharing system will then give the user a summary of the amount of time and resources used for this run and the total amount of the user's resources used to date.

AUTOMATIC TERMINATION FROM TERMINAL

The user will be automatically terminated from the system for any of the following reasons:

- If he responds twice with an invalid user identification. The terminal will reply after the first invalid use with the message ILLEGAL ID--RETYPE--. If the user responds with an invalid user identification a second time, he will be terminated.
- If he responds twice with an invalid password. The terminal will reply after the first invalid use with the message ILLEGAL PASSWORD--RETYPE--. If the user responds with an invalid password a second time, he will be terminated.
- If more than one minute passes without a response to user identification or password request.
- If he leaves the terminal in an idle state for over ten minutes.
- If his resources are overdrawn by more than 10 per cent. The message, RESOURCES EXHAUSTED. CANNOT ACCEPT YOU, will be printed by the terminal before termination takes place.

EXAMPLE OF TERMINAL OPERATION AND PROCEDURES

The following elementary example illustrates steps in terminal operation and procedures required for entering and running a BASIC program.

HIS SERIES 6000, SERIES 600 ON 05/28/71 AT 9.891 CHANNEL 0020	Acknowledgement of terminal connection to time-sharing system
USER ID --DOE PASSWORD-- ABCDEFGHIJK SYSTEM ? BASIC OLD OR NEW-NEW READY	Log-on procedure
* Sequence of * Statements	Sequence of statements
* RUN	Control command to execute statements
(Output)	Output data resulting from execution of statements
READY	Indication of system's continued availability
*BYE **RESOURCES USED $ 0.32, USED TO DATE $ 35.00=10% **TIME SHARING OFF AT 10.006 ON 05/28/71	Log-off procedure and accounting

SECTION IV

ELEMENTARY BASIC

STATEMENT DEFINITION

Each BASIC statement consists of the following elements arranged in the order given:

> Statement (or line) number - by its ascending order, indicates the processing sequence of the statement.
>
> BASIC word - specifies the computer operation to be performed.
>
> Parameters - in most statements are variables, expressions, and numbers used in or to direct the operation performed by the statement.

MATHEMATICAL NOTATION AND OPERATIONS WITHIN A STATEMENT

Variable Representation

In the BASIC language, a variable can be represented by

1. a letter
2. a letter and a digit
3. either of the above, followed by the character $

For example A,Z,K6, and X may represent variables, but AR, Z12, 6K, and 22 can not. The inadvertent use of the digit 0 for the letter O (and vice versa) in a variable will cause errors in a program; use of the letter O or the digit 0 in variable representation is not recommended. The user may find choice of a letter as a mnemonic for a variable helpful; for example, P for price, S for sales, and N for numbers.

Variables with $'s are restricted to the assignment of strings (alphanumeric data) and are referred to as "string variables", in contrast to variables without the $ that are referred to as "numeric variables". Numeric variables, when used as a starting point in calculations (e.g., for a counter), have an initial value of zero. String variables have an initial value of zero when used for character count.

A BASIC variable is assigned a value, during the execution of a program, from the numbers given in a related LET, FOR, READ, or INPUT statement. It retains this value during the processing, unless it is reassigned a new value by another of these statements.

List and Table Variables

Subscripted variables are represented in BASIC as

variable name (subscript)
or
variable name (subscript, subscript)

where the subscript can be an integer, variable, or an arithmetic expression such as (1+K) or (A(3,7),B-C). The subscript must always be enclosed by parentheses. Subscript values should begin at 1 (i.e., not 0).

A list variable designates an element of a one-dimensional array that can be represented by such as P(15), P(H) or L(20). Before a list variable can be used in any statement, the maximum value of its subscript (i.e., size of list) must be specified in a DIM statement; otherwise a list of 10 or less is implied.

A table variable designates an element of a two-dimensional array that can be represented by such as S(15,17) or T(20,30). Before a table variable can be referenced in any statement, the maximum value of its subscripts must be specified in a DIM statement; otherwise, subscripts of 10 or less are implied.

Specification of the values of subscripts for list variables or table variables in DIM statements is not required if subscripts of 10 or less occur. BASIC provides for automatic dimensioning in such cases. Automatic dimensioring assigns a value of 10 for the subscript of the list variable and a value of 10 by 10 for the array of a table value. If a subscript with a value greater than 10 is used with a list or table variable and the list or table variable is not dimensioned in a DIM statement, an error message will be generated. Conversely, if values of subscripts less than 10 are specified in DIM statements, no adverse programming effects result.

Use of Numbers

A number may be positive or negative, may contain up to nine digits, and must be in decimal form. BASIC would accept 0.01, 2, -3.675, 123456789, -.987654321, and 483.4156 as numbers, but would reject 14/3 (this is an expression) or 32,437 (as representing 32437). Numbers are stored as single-precision floating-point values. Thus, the maximum value that can be represented accurately is 134217727; larger values are only approximated since digits beyond the eighth position are not reliable.

A number can also be expressed in "E notation", which is equivalent to expressing it as a power of 10. For example, in E notation,

0.00123456789	may be	0.123456789E-2 or 12.3456789E-4
1967	expressed	1.967E3 or 19.67E2
10,000,000	as	1E7 or 100E5

The decimal point can be positioned anywhere within the number as long as the integer following the E indicates its correct position. Note that E and an exponent alone cannot represent a number. For example, E7 cannot be written as a number to represent 10,000,000; it must be written as 1E7 to indicate 1 multiplied by 10 to the 7th power.

Arithmetic Operations

Five arithmetic operations can be performed by BASIC. Each of the following symbols represents an arithmetic operation that can be included in an expression.

Operator symbol	denotes	as illustrated by
+	addition	A + B
-	subtraction	A - B
*	multiplication	A * B
/	division	A / B
↑ or **	raise to a power	A ↑ B or A ** B

Relational Symbols

Six relational tests can be made with BASIC. Symbols representing these relationships can be used in statements when comparisons are required. The symbols and illustration of their use follow.

Relational symbol	denotes	as illustrated by
=	is equal to	A = B
<	is less than	A < B
<= or =<	is less than or equal to	A <= B or A =< B
>	is greater than	A > B
>= or =>	is greater than or equal to	A >= B or A => B
<> or ><	is not equal to	A <> B or A >< B

Those terminals that lack the < (less than) or > (greater than) characters can make use of an alphabetic code to obtain required relational symbols. See Appendix E.

Use of Expressions

The computer performs its primary function (that of computation) by evaluating expressions which are contained within program statements. These expressions are similar to those used in standard mathematical notation with the exception that all BASIC expressions must be complete within a statement and a statement is restricted to a single line. Expressions are made up of numbers, variables, operations, and functions by themselves or in conjunction with one another.

The user must be careful to make sure that he groups together those things which he wants together. He must also understand the order in which the computer does its work. For example, if he types A + B * C ↑ D, the computer will first raise C to the power D, multiply this result by B and then add A to the resulting product. This is the same convention as is usual for A + B times C raised to the power D. If this is not the order intended, then he must use parentheses to indicate a different order. For example, if it is the product of B and C that he wants raised to the power D, he must write A + (B * C) ↑ D; or, if he wants to multiply A + B by C to the power D, he writes (A + B) * C ↑ D. He could even add A to B, multiply their sum by C, and raise the product to the power D by writing ((A+B) * C) ↑ D. The order of arithmetic priorities is summarized in the following rules.

1. The expression inside parentheses is computed before the parenthesized quantity is used in further computations.

2. In the absence of parentheses in an expression involving addition, multiplication, and the raising of a number to a power, the computer first raises the number to the power, then performs the multiplication, and the addition comes last. Division has the same priority as multiplication, and subtraction the same as addition.

3. In the absence of parentheses in an expression involving only multiplication and division, the operations are performed from left to right, as they are read. The computer performs addition and subtraction from left to right.

Practically, extensive use of parentheses will tend to eliminate most ambiguities that may arise.

Mathematical Functions

BASIC provides for standard mathematical functions. Each is represented by a 3-letter mnemonic of its name and is followed by an expression enclosed in parentheses. The user need only enter the function in a statement to obtain its computed value in a run of a program.

Function	means find the
SIN(X)	sine of X
COS(X)	cosine of X
TAN(X)	tangent of X
COT(X)	cotangent of X
ATN(X)	arctangent of X
EXP(X)	e to the power X
LOG(X)	natural logarithm of X
CLG(X)	common logarithm of X
ABS(X)	absolute value of X
SQR(X)	square root of X

In these definitions, the letter X represents an expression, which, for the trigonometric functions, implies an angle measured in radians. If the value of X in LOG(X), CLG(X), or SQR(X) is negative, then the negative sign is ignored, the positive value is used, and an error message is printed.

Four additional mathematical functions are included in BASIC.

Function	means
INT(X)	truncate X
RND(X)	produce a random number
SGN(X)	sign determination
DET(X)	provide determinant of last matrix inverted

Refer to Section V, "Additional Functions", for details concerning the use of these functions.

In addition, the user may employ the DEF statement to define one or more of his own functions.

Miscellaneous Functions

A set of miscellaneous functions is available for use to provide a variety of non-mathematical operations. These are as follows:

Function	means obtain
TIM(X)	elapsed processor time
CLK$	time of day
DAT$	calendar date
NUM(X)	count of matrix data elements
SST(X$,Y,Z)	selected characters of a string (substring)
TAB(X)	character print position
SPC(X)	space print position
LEN(X$)	number of characters in string
LIN(X)	last line number encountered in reading/writing file
ASC(X)	numeric value of character or abbreviation
STR$(N)	expression to string conversion
VAL(S$)	string to expression conversion
TST(S$)	nonzero output if string can be interpreted as a number
HPS(X)	horizontal point position of next field, in current line, of file being written

Refer to Section V, "Additional Functions", for details concerning the use of these functions.

STATEMENT DESCRIPTIONS

Purpose: A concise statement of the operation it performs.

Format: The general form for its use in the program, with the literal entries in CAPITAL letters and descriptive names for variable entries in lower-case letters enclosed within the symbols <> . Parentheses are to be inserted as indicated. Note that an expression can be either a simple variable or a formula.

Examples: Typical uses are given to explain and clarify the format. Statement numbers are arbitrary and are used for illustrative purposes.

Rules: Requirements and cautions concerning the use of the statement.

Remarks: Pertinent comments related to the uses of the statement.

Arithmetic Statement

DEF

Purpose: To define a function that is to be used repeatedly within a given program.

Format: DEF FN_ (variable) = < expression >

Example: *10 DEF FNG(Z) = 1 + SQR(1+Z*V)

Rules:

1. The variable must be unsubscripted.
2. Up to 26 functions can be defined within a single program; i.e., FNA, FNB, ..., FNZ.
3. The space following FN is to be filled with any alpha character.

Remarks: If a function requires more than one line for its definition, a multiple-line defined function can be written. Refer to "Multiple-Line DEF Statement" in Section V.

Refer to "Defining Functions" in Section V for details of the use of the DEF statement.

Arithmetic Statement

LET

Purpose: To evaluate an expression and assign the resultant value to a specified variable.

Format: LET <variable> = <expression>

Examples:

```
1.  *10  LET X=X+1
2.  *20  LET W7=(W-X4+3)*(Z-A)/(A-B)-17
3.  *30  LET X(6)=0
```

Remarks: The LET statement is not a statement of algebraic equality; it is an assignment or replacement statement.

A variable defined in a LET statement may be subscripted or unsubscripted.

Multiple variable replacement is permitted within a LET statement. For example:

```
*10 LET A=B=C
*20 LET A=B=C=100
*30 LET A(I)=B(X+Y/Z)=C(J)
*40 LET A(B(J))=B(J)=C(5)
*50 LET E$=F$=G$
*60 LET E$=F$=G$="MULTIPLE REPLACEMENT"
*70 LET H$(B(J))=H1$="EXAMPLES"
```

Replacement is executed on a right-to-left basis. A numeric BASIC variable may not be replaced by a string variable and vice versa. Multiple replacement is limited to 20 elements within one LET statement.

The BASIC word LET may be implied; i.e., the statement

```
*10 X=X+1
```

implies LET precedes the variable X and is a valid assignment statement.

Arithmetic Statement

MAT

Purpose: To request the system to compute or manipulate a matrix.

Format: MAT READ < variable or comma-separated variables>

MAT PRINT < variable or comma-separated variables>

MAT INPUT < variable >

MAT < variable > = operation

Remark: A detailed description of the use of MAT statements in operations upon matrices is given in Section V, under the heading "Matrices".

Specification Statement

CHANGE

Purpose: To permit translation of data from numeric code representation to its equivalent string character and, conversely, string character to numeric code.

Format: CHANGE <variable> TO <variable>

Examples:

1. *10 CHANGE A TO A1$

   Elements of numeric variable A are converted to characters and stored in string A1$.

2. *20 CHANGE Z5$ TO X

   Characters in string Z5$ are converted to their numeric equivalents and stored in the elements of X.

Rules:

1. One variable must be a numeric variable, the other a string variable.

2. The number of characters to be converted is limited to 132.

3. If a numeric variable has not been previously dimensioned, it will automatically be dimensioned by 10.

4. When the conversion is to be from a numeric code list to a character string, it is necessary for the user to provide a count of the number of elements to be converted. This is done prior to the CHANGE command by an assignment statement which stores the desired count in element (0) of the numeric array.

   For example:

   ```
   *10 LET A(0) = 15
   *20 CHANGE A TO A1$
   ```

   directs the program to convert fifteen of the numeric elements in list A to their related characters and concatenate them in string A1$.

   If the count specified for conversion is smaller than the number of items in the numeric list, the remaining characters will be truncated; if the count given is larger, the string will contain irrelevant information.

5. When a string is converted to numerics, a count is not specified. The complete string will be converted if the numeric array is of sufficient length. If the array dimension is smaller than the string length, an error message will occur at execution time. If the string characters do not fill the entire array, the remaining array elements remain unchanged.

6. A table of characters and equivalent codes can be found in Section V under "Alphanumeric Data and String Manipulation."

Remarks: An explanation of the use of alphanumeric data and string manipulation within a BASIC program is given in Section V under the heading "Alphanumeric Data and String Manipulation". Use of the CHANGE statement is given there in detail.

Specification Statement

DATA

Purpose: To specify numeric values for variables in a READ statement.

Format: DATA <number or comma-separated numbers>

Example:

```
*10 READ A,B,X,L1,Z
     . .
     . .
*100 DATA 1,3.4,7,-167.921,1.9E5
```

Rules:

1. Only numbers (positive or negative) are allowed; numbers may be written conventionally or with E-notation.
2. The numbers in the DATA statement must be in the same sequence as the respective variables in the associated READ statement (in the example, X = 7).
3. The numbers may be in one or more DATA statements, but the sequence must correspond to that for the variables in the READ statement. That is, the DATA statement in the example could be replaced by as many as five DATA statements.

Remarks: DATA and READ statements are always used jointly.

The collection of all numbers in all of the DATA statements of a program is referred to as a "data block."

The placement of DATA statements in a program is arbitrary; common practice is to collect all of the DATA statements in one place in the program.

Specification Statement

DIM

Purpose: To define the dimension(s) of a list or table and thereby reserve sufficient space in the computer.

Format:
1. For a list

   DIM < variable > (subscript)

2. For a table

   DIM < variable > (subscript, subscript)

Examples:
1. *10 DIM H(35)

This statement reserves 35 computer locations.

2. *20 DIM Q(5,25)

This statement reserves 125 computer locations, since it involves 5 items times 25 items, as in 5 x 25 table.

Space for more than one list and/or table may be defined in a single DIM statement.

    *30 DIM M(50), R(25,35), T(10,10)

Rules:

1. A subscripted variable must appear in a DIM statement to achieve explicit dimensioning; otherwise, automatic dimensioning (subscript value of 10 or less) is implied.
2. DIM statements defining variables must precede the use of these variables.
3. The dimension(s) of a list or table in a DIM statement must be expressed explicitly; expressions are not to be used as subscripts.
4. For a list, the variable can be numeric or string; for a table, the variable must be numeric.

Input/Output Statement

INPUT

Purpose: To permit the input of desired values of variables during program execution time.

Format: INPUT < variable or comma-separated variables >

When, in the execution of the program, this statement is reached, a question mark is printed. The user must then enter a number or sequence of numbers before the program can continue.

Example:

*10 INPUT X,Y,Z	is entered into the program as a statement
?	but only a question mark appears during execution; the user is then to type the comma-separated values of X, Y, and Z after the question mark.

Rules:

1. Each INPUT statement must be positioned logically ahead (in the order of processing) of the statement that is to use the data values requested.
2. The numbers listed after the question mark must also be separated by commas.
3. The numbers must be typed in the same sequence as the variables to which they are assigned.

Input/Output Statement

PRINT

Purpose: To instruct the system to perform one of the following print operations:

1. Print out the result of computations.
2. Print out text, verbatim, to supply such items as messages, information, or labels.
3. Print out a combination of uses 1 and 2.
4. Skip a line in the printout of program execution.

Format: Every PRINT statement begins with the BASIC word PRINT but may vary in form, dependent upon the print operation required.

Example of Use 1:

```
*10 PRINT X,SQR(X)
```

will result in the printing of the value of X, and a few spaces to the right of that number, its square root.

```
*20 PRINT B*C,EXP(A),Y/Z,E+F,X↑2
```

will result in the printout of 5 computed values.

Example of Use 2:

Whenever text is to be printed verbatim during the execution of a program, it is enclosed within quotation marks in the statement; whatever is enclosed will be reproduced, including spaces and punctuation. This verbatim text is referred to as a label.

```
*40 PRINT "NO UNIQUE SOLUTION"
```

results in the printout

```
NO UNIQUE SOLUTION
```

Input/Output Statement

PRINT

Example of Use 3:

```
*50 PRINT "THE VALUE OF X IS", X
```

results in the printout, if X = 3,

```
THE VALUE OF X IS 3
```

```
*60 PRINT "THE SQUARE ROOT OF" X, "IS" SQR(X)
```

results in the printout, if X = 625,

```
THE SQUARE ROOT OF 625 IS 25
```

Example of Use 4:

When a statement such as

```
*70 PRINT
```

is encountered by the program during its execution, the terminal carriage is advanced one line at that stage of program execution.

Remarks: The form in which BASIC prints numbers is not under the control of the user. The following items apply to the printing of numbers when PRINT statements are utilized.

1. When a number is an integer, the decimal point is not printed.
2. When a computed value consists of an integer with more than seven digits, BASIC prints
   - the first significant digit
   - followed by a decimal point
   - the next five digits (the integer is rounded)
   - the letter E
   - followed by a space
   - and finally, a number indicating the power of 10 (how many places the decimal point is to be moved to the right).

   For example, the integer

   32437528259 becomes 3.24375E 10 when printed.
3. No more than seven significant digits are printed.

4. Numbers less than 1.0 are printed with a decimal point followed by up to seven significant digits.

   For example,

   .1234567

   would be printed exactly as shown, whereas the number

   .01234576978

   would be rounded and printed as

   .0123458

5. Numbers less than 0.0001 are printed in E-format.

   For example,

   .00001234567

   would be rounded and printed as

   1.23457E-05

The PRINT statement may be modified by the use of:

commas

semicolons

function TAB(X)

function SPC(X)

in order to vary the format of the output. Refer to Section V for details concerning the PRINT statement modification.

Input/Output Statement

PRINT USING

Purpose: To instruct the system to print out a formatted line.

Format: PRINT USING < statement number, output list >

where:

"statement number" is number of a statement in the program which contains format control characters and printable constants; "output list" consists of comma-separated arguments to be printed in sequential order.

Example:

```
*10 A = 100
*20 B = 200
*30 C = -300
*40 D$ = "END OF LIST"
*50 PRINT USING 60,A,B,C,D$
*60: #### #### #### 'LLLLLLLLLLLL
*70 END
*RUN

ƀƀ100ƀƀ200ƀ-300ƀEND OF LIST
```

Rules:

1. The statement number named in a PRINT USING statement points to an "image" statement which formats the line to be printed. The image statement is of the form

   statement number: image

2. The image of an image statement (colon-separated from the statement number) consists of format control characters and printable constants.

3. Format control characters are as follows:

   ' (apostrophe) - a 1-character field that is filled with the first character in an alphanumeric string, regardless of string length.

   # (pound sign) - the replacement field for a numeric character; each # specifies a space for one digit; a # specifies space for the minus sign if sign is present.

   ↑↑↑↑ (four up-arrows) - specifies scientific notation for a numeric field (E-format).

4. Printable constants are all characters other than format control characters.

Input/Output Statement

PRINT USING

Remarks: The image of an image statement may consist of one or more of the following fields:

integer
decimal
exponential
alphanumeric
literal

Refer to Section V, "Formatting Line Output", for details concerning use of the PRINT USING statement.

Input/Output Statement

READ

Purpose: To read values listed in DATA statements and assign them to specified variables.

Format: READ <variable or comma-separated variables>

Example:

```
*10 READ A,B,X,L1,Z
    .  .
    .  .
    .  .
*100 DATA 1,2,7,2,-167.921
```

Rules:

1. Each READ statement must be positioned logically ahead (in the order of processing) of the arithmetic or PRINT statement that is to use the data requested.

2. The variables in a READ statement must be in the same sequence as the respective values in the associated DATA statement (in the example, 7 will be assigned to X).

Remarks: READ and DATA statements are always used jointly. If there are not enough numbers in the data block (collection of DATA statements) for the variables in a READ statement, then the program is assumed to be finished, no further processing of data occurs, message OUT OF DATA is printed, and the program terminates processing.

If a READ statement is executed more than once, as if in a loop, the data block supplies the next available number for each execution, unless a RESTORE statement is executed.

Input/Output Statement

RESTORE

Purpose: To restore the data block to its original state, so that it may be read by a logically subsequent READ statement and thus used for further processing.

Format: RESTORE

Example: In the following portion of a program

```
*100 READ N
*110 FOR I = 1 TO N
*120 READ X
  .  .
  .  .
*200 NEXT I
  .  .
  .  .
*560 RESTORE
*570 READ X
*580 FOR I = 1 TO N
*590 READ X
  .  .
  .  .
*650 DATA 4, 15, 35, 23, 9
*660 END
```

the data is read, the data block is then restored to its original state, and the data is then read again for processing. Statement 570 is used to pass over the value of N, since it is already known.

Remarks: When the program is executed, the data from the DATA statements are saved in memory as a data block. The data is then assigned to variables via a READ statement in the sequence given. The RESTORE statement directs the computer to reassign data starting from the beginning of the data block; if this statement were not present in the above example, then the system would stop processing at statement 570 and print out the message OUT OF DATA.

Special uses of RESTORE (RESTORE* and RESTORE$) are described in Section V under "Alphanumeric Data and String Manipulation".

Loop and Subroutine Statement

CALL

Purpose: To call a program, previously saved on a permanent file, for use as a subroutine within the primary program.

Format: CALL <filename, password>

Example:

```
*10 DEF FNP(X,Y)=SQR(X*X+Y*Y)
*20 CALL SUB1
*30 DATA 3
*40 END
```

Program SUB1, previously saved, is as follows:

```
*10 READ B,C
*20 IF B=0 THEN 70
*25 CALL SUB2
*30 LET A=FNP(B,C)
*40 PRINT "HYPOTENUSE=";A
*50 GOTO 10
*60 DATA 4,0,0
*70 RETURN
```

Program SUB2, previously saved, is as follows:

```
*10 IF B > 0 THEN 40
*20 PRINT "NEGATIVE ARGUMENT"
*30 STOP
*40 IF C < 0 THEN 20
*50 RETURN
*60 END
```

Rules:

1. All variables and functions must be common to the primary (calling) program and the called programs.
2. The return from a called program to the calling program must be by the way of a RETURN statement.

Remarks: A password is required only if one is attached to the <u>filename</u>.

Multiple returns are permitted within a called program. The return is always to the statement immediately following the CALL statement. A called program may call other programs.

Loop and Subroutine Statement

CALL

An END or STOP statement to terminate execution may be in either the calling or called program.

Line numbers in calling or called programs are completely independent.

DATA statements are compiled from the primary program first, and then from each of the called programs in the order in which the CALL statements are encountered.

A total of 15 different programs may be called from the primary and called programs.

Loop and Subroutine Statements

FOR and NEXT

Purpose: The FOR statement is the initial statement of a program loop and it specifies the variable used to count the iterations through the loop, its range of values, and the step-size for each pass through the loop. The NEXT statement is the last statement in the loop and it directs the processing to either repeat the loop or continue sequential execution if the specified number of iterations have been completed.

Format:

```
FOR <variable> = <expression> TO <expression>
STEP <size expression>
  .  .
  .  .
  .  .
NEXT <variable>
```

<variable> specifies an unsubscripted loop-control variable. <expression> TO <expression> specifies the range of values to be assigned to the variable. The first expression sets the initial value of the variable; the second expression sets the final value of the variable. For a positive step-size, the loop will be repeated until the variable reaches a value greater than or equal to the final value. For a negative step-size, the loop will be repeated until the variable reaches a value less than or equal to the final value. STEP <size expression> specifies the increment or decrement to be added to the loop-control variable on each pass through the loop; if STEP and its size expression are omitted, the increment is assumed to be 1.

Examples:

1.
```
*30 FOR X = 1 TO 25
  .  .
  .  .
*80 NEXT X
```

2.
```
*120 FOR X4 = (17+COS(Z)/3) TO 3*SQR(10) STEP N*Z
  .  .
  .  .
*235 NEXT X4
```

3.
```
*240 FOR Z = 8 TO 3 STEP -1
  .  .
  .  .
*300 NEXT Z
```

Loop and Subroutine Statements

FOR and NEXT

4.
```
*450 FOR J = -3 TO 12 STEP 2
  .  .
  .  .
*500 NEXT J
```

5.
```
*30 FOR X = 0 TO 25 STEP A
  .  .
  .  .
*80 NEXT X
```

Rules:

1. If the range requires a negative step and it is omitted, the body of the loop will be executed once for the initial value of the variable. The variable is tested after the first time the implied step (+1) is added, and will be found to exceed the termination condition.

2. Paired FOR and NEXT statements must specify the same loop-control variable.

Loop and Subroutine Statements

GOSUB and RETURN

Purpose: GOSUB - To direct the system to the first statement of a subroutine sequence that is located elsewhere in the program (i.e., to "call" a subroutine).

RETURN - To return the processing to the next statement following the GOSUB statement used to call the subroutine.

Format: GOSUB < number of first statement of subroutine >

Example:

```
*80 GOSUB 200
*90 LET X = 5
   . .
   . .
*200 LET X = INT(A/B)
   . .
   . .
*350 RETURN
```

Statement 350 will return the processing to statement 90.

Remarks: A subroutine may be placed anywhere within a program but should only be entered by the way of a GOSUB statement. Return from a subroutine must be by the way of a RETURN statement; no other type of statement can be used.

Logic Statement

GOTO

Purpose: To transfer unconditionally to a statement other than the next one in the processing sequence.

Format: GOTO <statement number>

Example: *50 GOTO 20

Remark: The GOTO statement may be used as a means of delegating a program to return repeatedly to blocks of instructions.

Logic Statement

IF-----THEN
or
IF-----GOTO

Purpose: To direct the system to either go to a designated out-of-sequence statement if a certain condition is met or proceed to process in sequence, thus providing a 2-way conditional switch.

Format: IF < expression > relation < expression > $\left\{\begin{array}{l}\text{THEN}\\ \text{GOTO}\end{array}\right\}$ < statement number >

Examples:

1. *10 IF SIN(X) = M THEN 80 or
   *10 IF SIN(X) = M GOTO 80

2. *20 IF G=0 THEN 65 or
   *20 IF G=0 GOTO 65

In each example, if the condition is met, then the computer transfers to the designated statement number; otherwise, it proceeds to process the next statement in sequence.

Rule: BASIC provides six relational tests. The following symbols representing relationship can be used in IF----THEN or IF----GOTO statements when comparisons are required.

Relational Symbol	denotes	as illustrated by
=	is equal to	A = B
<	is less than	A < B
< = or = <	is less than or equal to	A < = B or A = < B
>	is greater than	A > B
> = or = >	is greater than or equal to	A > = B or A = > B
< > or > <	is not equal to	A < > B or A > < B

Those terminals that lack the < (less than) or > (greater than) characters can make use of an alphabetic code to obtain required relational symbols. See Appendix E.

Logic Statement

ON-----THEN
or
ON-----GOTO

Purpose: To direct the system to go to designated statements, thus providing a multiple switch.

Format: ON <expression> {THEN / GOTO} <statement numbers>

Examples:

1.
```
*10 ON X GOTO 100,200,150
```

   if X=1, the system branches to statement 100
   if X=2, to statement 200
   if X=3, to statement 150

   The value of X is dependent upon conditions set in another part of the program.

2.
```
*110 FOR X = 1 TO 3
*120 ON X GOTO 200,300,400
*200 PRINT"A"
*210 GOTO 500
*300 PRINT"B"
*310 GOTO 500
*400 PRINT"C"
*500 NEXT X
*600 STOP
*900 END
*RUN
    A
    B
    C
```

Rules:

1. Any number of statement numbers may follow THEN or GOTO, providing they fit on one line.
2. Statement numbers following THEN or GOTO may be repeated.

Remarks: The expression can be a variable or a formula. The variable must be an integer ranging from one to the number of statement numbers specified. For a formula, computation is made and its integer part is taken as the value. If the integer part is less than one or is larger than the number of statement numbers specified, an error message is printed.

Logic Statement

STOP

Purpose: To stop the execution of the program.

Format: STOP

Example:

```
*250 STOP
  .   .
  .   .
*340 STOP
  .   .
  .   .
*990 END
```

This example illustrates that there may be more than one STOP statement within a program, and if any one is processed, the program is terminated.

Remark: STOP is the equivalent of GOTO XXXX, where XXXX is the line number of the END statement in the program.

Logic Statement

END

Purpose: To indicate the end of a program.

Format: END

Example: *990 END

Rules:

1. The END statement is optional in a program.
2. The END statement, if used, must have the highest line number of the program.
3. The END statement, if omitted, is simulated when the RUN command is given and if an end-of-file situation is detected.

Remarks: In the execution of the program, the system recognizes the END statement as a command to terminate output. The END statement may be reached during program execution by normal sequential processing, or by program control being transferred to it by means of a GOTO or STOP statement.

Utility Statement

CHAIN

Purpose: To permit sequential compilation and execution of a series of BASIC programs.

Format: CHAIN <filename, password, line number>

Examples:
1. *10 CHAIN FILE1,PASS1,100
2. *20 CHAIN A$,PASS2
3. *30 CHAIN B$,1234,

Rules:

1. The filename can be expressed in the following manner:

   a. in ASCII characters, a limit of eight characters

   b. enclosed in quotes; i.e., "filename"

   c. as an alphanumeric variable, subscripted or unsubscripted, with the values of the variable and subscript (if any) assigned at compilation or execution times.

2. If a file with a password is named in a CHAIN statement, the password must accompany the filename.

3. The CHAIN statement permits chaining to a line number within a file.

4. Each CHAIN statement is restricted to one filename.

5. If a password is all numeric and no line number is specified, the password must be delimited by a trailing comma; otherwise, the password will be interpreted as a line number.

Remarks: The current file and a file named in a CHAIN statement must be files saved prior to any attempt to perform the chaining function.

If a line number is given in a CHAIN statement, it must be given as a numeric value.

There is no limit to the number of programs the user desires to compile and execute by means of CHAIN statements.

The use of double quotes to enclose a filename permits compatibility with programs written for other systems.

Utility Statement

TRACE ON

TRACE OFF

Purpose: To instruct the system to print out the line numbers, at execution time, of those statements enclosed between a TRACE ON and TRACE OFF statement.

Format:

```
TRACE ON
    .
    .          sequence of statements
    .
TRACE OFF
```

Example:

```
*10 LET X=0
*20 IF X > 0 GOTO 80
*30 TRACE ON
*40 LET X=15
*50 PRINT "PHASE 1"
*60 GOTO 20
*70 TRACE OFF
*80 PRINT "PHASE 2"
*90 END
```

When RUN is given as a command, program execution will be as follows:

```
* AT 40
* AT 50
PHASE 1
* AT 60
PHASE 2
```

Remarks: A TRACE ON statement may be used without a TRACE OFF statement; i.e., the END statement simulates a TRACE OFF statement. If a TRACE OFF statement is encountered before a corresponding TRACE ON statement, that TRACE OFF statement is ignored.

Multiple TRACE ON-TRACE OFF statements may be made within one program.

Documentation Statement

REM

Purpose: To permit the insertion of an explanatory remark in a program.

Format: REM <followed by the remark>

Example:

```
*50 REM  INSERT DATA IN LINES 900-1000.
*60 REM  THE FIRST NUMBER IS N, THE
*70 REM  NUMBER OF POINTS REQUIRED.
```

Remarks: The computer stores the text of the REM statement and does not process it. A GOSUB, IF-----THEN, or GOTO statement can refer to a REM statement by referencing its statement number. When a remark exceeds a line, a statement number and REM must be typed on each succeeding line before continuing the remark.

Programs containing distinctive parts such as subroutines or loops should have these parts labeled by means of REM statements. Such labeling readily identifies sections of a lengthy program and permits the user to rapidly scan the program if corrections or additions are required.

A BASIC PROGRAM EXAMPLE

General

The first step in writing a BASIC program is to analyze the problem and determine the exact operations that must be performed to produce the desired results. Having determined the required operations, it is then necessary to convert them into BASIC statements.

This example describes the preparation of a BASIC program that will calculate and print out the average number of miles traveled by a vehicle per gallon of gasoline, given:

Old Miles	New Miles	Gallons of Gasoline Used	Average Number of Miles per Gallon
3332	3553	14.8	?
	3801	7.4	?
	3926	15.2	?
	4091	11.3	?
	4275	10.9	?
	4460	9.8	?
	4628	9.8	?
	4864	12.3	?
	5250	13.6	?
	5617	6.7	?
	6112	10.0	?
	6379	14.0	?
Overall average miles traveled per gallon of gasoline			?

Analyzing the Problem

An analysis of the problem indicates that the following operations should be performed to arrive at the solution:

1. Show five column headings across the typeout as follows:

   Old Miles
   New Miles
   Miles Traveled
   Gallons of Gasoline Used
   Average Miles Traveled per Gallon of Gasoline

2. Write given "old miles" value in column one.

3. Write first given "new miles" value in column two.

4. Write first given "gallons of gasoline" value in column four.

5. Subtract value in column one from the value in column two and write the result in column three.

6. Divide value in column three by value in column four and write the result in column five. This is average number of miles traveled per gallon of gasoline.

7. Move down to second line in each column.

8. Write first given "new miles" value in column one.

9. Write second given "new miles" value in column two.

10. Write second given "gallons of gasoline" value in column four.

11. Subtract last value in column one from last value in column two and write result in column three.

12. Divide last value in column three by last value in column four and write result in column five.

13. Move down to third line in each column.

.
.
.

Continue writing of appropriate values in proper columns and making computations until all data is utilized. Move down to next line after completing each "average miles traveled per gallon of gasoline" computation and writing of result in column five.

.
.
.

14. Divide total number of miles traveled by total gallons of gasoline used and title the result "Overall average miles traveled per gallon of gasoline".

Converting to BASIC Language

Having determined the required operations, it is now necessary to convert the operations into BASIC statements.

The following relationships and abbreviations will facilitate the writing of the program:

$M = N-L$ and $A = \frac{M}{G}$ where:

M = miles traveled

L = old miles

N = new miles

A = average miles per gallon

G = gallons of gasoline

The following sequence of statements can now be written.

```
5   REM TOTAL MILES/GALS
10  PRINT"OLD MILES ";"NEW MILES ";"MITR ";"GAL GAS ";"AMPG"
20  PRINT"-----------------------------------------------"
30  READ L
40  LET L1 = L
50  READ N
60  IF N=0 THEN 190
70  READ G
80  LET M=N-L
90  IF M=0 THEN 120
100 LET A=M/G
110 IF A<>0 THEN 130
120 PRINT "YOUR TANK HAS A HOLE IN IT"
130 IF A < 35 THEN 150
140 PRINT "I DONT BELIEVE IT"
150 PRINT L;N;M;G;A
160 LET L=N
170 LET G1=G1+G
180 GOTO 50
190 PRINT "TOTAL MILES/GALS",(L-L1)/G1
200 DATA 3332,3553,14.8,3801,7.4,3926,15.2,4091,11.3,4275
210 DATA 10.9,4460,9.8,4628,9.8,4864,12.3,5250,13.6,5617
220 DATA 6.7,6112,10.0,6379,14.0,0
230 END
```

Explanation of the Statements

5 REM TOTAL MILES/GALS

Identifies the program; does not enter into the execution process.

```
10    PRINT "OLD MILES ";"NEW MILES ";"MITR ";"GAL GAS ";"AMPG"
```

```
20    PRINT "---------------------------------------------"
```

Statements 10 and 20 direct the system to print verbatim that information enclosed by quotation marks.

```
30    READ L
```

Assigns the first value in the data block to variable L; i.e., 3332 to L (old mileage).

```
40    LET L1=L
```

Assigns the existing value of L which is 3332, to L1. The value assigned to L will change as the program execution progresses but the value assigned to L1 will remain 3332. It is necessary to preserve the 3332 value for calculating total miles traveled; statement 190 directs the computer to make this computation.

```
50    READ N
```

Assigns the next value in the data block to variable N; i.e., 3553 to N (new mileage).

```
60    IF N=0 THEN 190
```

Directs the system to execute statement 190 instead of statement 70 if the value assigned to N in statement 50 was 0; i.e., last entry in data block.

```
70    READ G
```

Assigns the next value in the data block to variable G; i.e., 14.8 to G (gallons of gasoline)

```
80    LET M=N-L
```

Directs the system to subtract the value of L from the value of N and assign the difference to variable M (miles traveled).

```
90    IF M=0 THEN 120
```

Directs the system to execute statement 120 instead of statement 100 if the value assigned to M in statement 80 was 0.

100 LET A=M/G

Directs the system to divide the value of M by the value of G and assign the resulting value to A (average miles per gallon).

110 IF A<>0 THEN 130

Directs the system to execute statement 130 next instead of statement 120 if the value assigned to A in statement 100 was any value other than 0.

120 PRINT "YOUR TANK HAS A HOLE IN IT"

Directs the system to print out, verbatim, that information enclosed by quotation marks. This statement is executed only if the value assigned to A in statement 100 was 0, or if the value assigned to M in statement 90 was 0.

130 IF A < 35 THEN 150

Directs the system to execute statement 150 instead of statement 140 if the value assigned to A in statement 100 was less than 35.

140 PRINT "I DONT BELIEVE IT"

Directs the system to print out, verbatim, information enclosed by quotation marks. This statement is executed only if the value assigned to A in statement 100 was equal to or greater than 35.

150 PRINT L, N, M, G, A

Directs the system to print, in column form, the values of L, N, M, G, and A assigned in statements 30, 50, 80, 70, and 100, respectively.

160 LET L=N

Assigns the existing value of N (new mileage) to L (old mileage) in preparation for the next calculation.

170 LET G1=G1+G

The objective of this statement is to establish a means for recording the accumulative gallons of gasoline used for the entire trip. As there was no READ statement to assign a value, the computer initially set G1 to zero.

On the first pass through the data block, G was assigned the value 14.8. This statement directs the computer to add the value of G (14.8 in this instance, assigned in statement 70) to the initial value of G1 (zero), establishing a new value for G1 (14.8). On the second pass through the data block the next value of G (7.4) will be added to the existing value of G1 (14.8) establishing another new value for G1 of 22.2. This summation of G and G1 will be repeated on subsequent passes as long as there are new values of G in the data block.

180 GOTO 50

Directs the system to go to line 50, thus repeating the same sequence of statements over again to find the average miles traveled per gallon of gasoline for the next refueling. Eventually, a value of N equal to zero will be achieved and statement 60 will be executed. At that point, control of the program will be given to statement 190.

190 PRINT "TOTAL MILES/GAL", (L-L1)/G1

The system is instructed to calculate and print the overall miles traveled per gallon of gasoline for the entire trip.

The statement accomplishes this by directing the system to subtract L1 (3332 from statement 40) from L (6379 - the last old mileage assignment in the data block) and then divide the difference by G1 (accumulative gallons of gasoline calculated in statement 170).

200, 210, 220 DATA

Data statements are not executed. They are used to enter the data required for the subsequent execution of the program. The arrangement in which the data is entered in the statement is critical because the computer must be directed to store the data in a sequence compatible with the requirements of the program statements.

230 END

Directs the system to end the execution of the program.

Entering and Running the Program

The sequence of statements representing the problem and its solution can now be entered at the terminal. The complete program would appear as below, assuming no errors have been made. To run the program, the control command RUN is given.

```
*5   REM TOTAL MILES/GALS
*10  PRINT "OLD MILES ";"NEW MILES ";"MITR ";"GAL GAS ";"AMPG"
*20  PRINT"--------------------------------------------------"
*30  READ L
*40  LET L1 = L
*50  READ N
*60  IF N=0 THEN 190
*70  READ G
*80  LET M=N-L
*90  IF M=0 THEN 120
*100 LET A=M/G
*110 IF A<>0 THEN 130
*120 PRINT "YOUR TANK HAS A HOLE IN IT"
*130 IF A < 35 THEN 150
*140 PRINT "I DONT BELIEVE IT"
*150 PRINT L;N;M;G;A
*160 LET L=N
*170 LET G1=G1+G
*180 GO TO 50
*190 PRINT "TOTAL MILES/GALS",(L-L1)/G1
*200 DATA 3332,3553,14.8,3801,7.4,3926,15.2,4091,11.3,4275
*210 DATA 10.9,4460,9.8,4628,9.8,4864,12.3,5250,13.6,5617
*220 DATA 6.7,6112,10.0,6379,14.0,0
*230 END
*RUN

OLD MILES   NEW MILES   MITR   GAL GAS   AMPG
---------------------------------------------------------


   3332        3553      221      14.8    14.93243
   3553        3801      248       7.4    33.51351
   3801        3926      125      15.2    8.223684
   3926        4091      165      11.3    14.60177
   4091        4275      184      10.9    16.88073
   4275        4460      185       9.8    18.87755
   4460        4628      168       9.8    17.14286
   4628        4864      236      12.3    19.187
   4864        5250      386      13.6    28.38235
I DONT BELIEVE IT
   5250        5617      367       6.7    54.77612
I DONT BELIEVE IT
   5617        6112      495      10      49.50000
   6112        6379      267      14      19.07143
TOTAL MILES/GALS                 22.43741
```

SECTION V

ADVANCED BASIC

GENERAL

Advanced BASIC presupposes knowledge, on the part of the user, of the details contained in Section IV concerning the general use of BASIC statements and assumes that the user has acquired some skill in the application of these statements.

This chapter provides additional information pertaining to the use of statements and is intended for the more experienced user who wishes to obtain more flexibility in his programs or needs to solve more complex problems.

FLEXIBILITY IN PROGRAM OUTPUT FORMAT

General uses of the PRINT statement and PRINT USING statement were described in Section IV. For the advanced programmer, forms of the PRINT statement and PRINT USING statement are available that permit more flexibility in the formatting of the program output.

Formatting Output With a Comma or Semicolon

The end of a PRINT statement signals the end of the line, unless a comma or a semicolon is the last character of the statement.

For example, statement 20 in the program entry

```
*  10 FOR I = 1 TO 15
*  20 PRINT I
*  30 NEXT I
*  40 END
*  RUN
```

will result in output of 15 numbers printed on 15 lines, thus:

```
    1
    2
    3
    4
    5
    6
    7
    8
    9
   10
   11
   12
   13
   14
   15
```

The use of a comma after a variable in a PRINT statement implies data placed in a zone format upon printout. BASIC provides for a line comprising five zones, each zone being referred to as a standard zone. By the use of a comma after a variable, data is allotted to zones and the data is right-justified within the zone. Thus, by rewriting statement 20 as

```
* 20 PRINT I,
```

The resulting format will be

```
    1        2        3        4        5
    6        7        8        9       10
   11       12       13       14       15
```

The statement

```
* 10 PRINT X, Y
```

will result in the printing of the value of X in the first standard zone, the value of Y in the second standard zone and the return to the next line, while

```
* 20 PRINT X, Y,
```

will result in the printing of these two values in the first and second standard zones and no return; the next value called for in a subsequent PRINT statement will be printed in the third standard zone.

The statement

```
* 30 PRINT X, Y, Z, A, B, C
```

will result in the printing of the first five values in the five standard zones across the page; the sixth value will be printed in the first zone beneath the first value. Five values are the limit to a printout line, each value being restricted automatically within the confines of its zone upon printout. (Refer to the remarks of the PRINT statement description in Section IV.)

The use of a semicolon after a variable in a PRINT statement implies a variation of the standard zone format. Spacing is compacted to obtain more zones on the line. Minimum size zone is 7 columns and can contain a number up to 4 characters. The next larger size zone is 9 columns and contains up to 7 characters. All other fixed point numbers are printed as 12-column zones. Negative numbers are preceded by a minus sign in the first column of a zone.

For the following program (note use of semicolon in statement 20) the printout of values would be in compacted zones as illustrated.

```
* 10 FOR I = 1 TO 15
* 20 PRINT I;
* 30 NEXT I
* 40 END
* RUN
  1   2   3   4   5   6   7   8   9   10   11   12
 13  14  15
```

Commas and semicolons may be used within the same PRINT statement. The statement

```
* 50 PRINT X,Y; Z,
```

will result in the values of X and Z being printed in standard zones, while the zone of value Y would be compacted.

Text to be printed verbatim is referred to as a label. A label is printed just as it appears in the PRINT statement left-justified in a zone. If two or more labels appearing in the same PRINT statement are comma-separated, the first label will be printed left-justified in the first zone and each succeeding label will be printed left-justified in the next succeeding available zone.

The statement

```
* 10 PRINT "X VALUE", "SIZE", "RESOLUTION"
```

results in the printout

```
X VALUE                 SIZE                  RESOLUTION
```

Semicolon (or null-separated) labels in the same PRINT statement are printed with no character separation.

The statement

```
* 20 PRINT "OLD MILES"; "NEW MILES"
```

results in the printout

```
OLD MILESNEW MILES
```

If a label exceeds the length of a line, the line must be ended by quotation marks and its carryover on the next line or lines treated as additional PRINT statements.

Spacing Within an Output Line with Functions TAB(X) and SPC(X)

When used in a PRINT statement, functions TAB(X) and SPC(X) give the user additional control of spacing within an output line. These functions may be used as any field within the PRINT statement.

The TAB function is expressed in the form:

PRINT TAB (expression); < data to be printed >

It will cause the printing of the next data field at the character position indicated by the value of the expression plus one.

The SPC function is expressed in the form:

PRINT SPC (expression); < data to be printed >

The number of spaces, equal to the value of the expression, will be inserted in the print line. If this number causes the print position to exceed 72, the carriage will return and the print position indicator will be set at 1.

Exception conditions:

TAB:

1. When the expression results in a number which is less than the current character position where the carriage is sitting, the TAB function is ignored.

2. When the expression results in the number which is greater than the line limit, the TAB function is ignored.

SPC:

When the expression results in a number which, when added to the current character position on the line, exceeds the line limit, the current line will be printed and the current character position will be reset to the first position on the next line.

Examples:

```
* 10 PRINT X, TAB(20); Y; TAB(40); Z
```

will result in the values of X starting right-justified in the first zone, the values of Y starting at position 21, and the values of Z starting at position 41.

In the example

```
*  10 PRINT TAB(20); "DATA"
*  20 END
*  RUN
```

the resulting printout is DATA positioned as follows:

Position	20	21	22	23	24
		D	A	T	A

```
*  20 PRINT TAB(10*SIN(X)+10); X
```

will result in the value of X being printed in the position specified by the value of the expression (10*SIN(X)+10).

In the example

```
*  10 FOR X = 1 TO 5
*  20 PRINT X; SPC(X); "+"
*  30 NEXT X
*  40 END
*  RUN
```

the resulting printout is

```
1 +                     (separated by 1 space)
2  +                                (2 spaces)
3   +                               (3 spaces)
4    +                              (4 spaces)
5     +                             (5 spaces)
```

Formatting Line Output

A line of output (a printed line) may be formatted by the user by means of the PRINT USING and PRINT # USING statements.

The fields which compose the image of the image statement pointed to by the PRINT USING and PRINT # USING statements may be made up of the following types:

integer	exponential	literal
decimal	alphanumeric	

Format control characters depict the fields within the image statement; the fields are separated by one or more literal characters (which may be blanks).

Each character following the colon of a PRINT USING or PRINT # USING statement is treated as a print position, specifying either a literal or control character.

To facilitate explanation of format control characters and fields, the examples following make use of the PRINT USING statement only. The PRINT USING statement directs the system to immediately produce a visible result at the terminal upon program execution. In contrast, the PRINT # USING statement, making use of the same format control characters and fields, formats data on the designated file, which may later be made visible by the way of the LIST command.

INTEGER TYPE FIELD

Each numeric of an integer type field is indicated by a pound sign (#); the field width must also include a # for the algebraic sign, plus or minus. Upon program execution, the numbers of an integer type field are right-justified within the field and rounded if they are not integral.

Example:

```
*  10 LET A = 123
*  20 LET B = 12.34
*  30 PRINT USING 40,A,B
*  40: #### ####
*  50 END
*  RUN

     123  12
```

If a number does not fit into the specified format, a field of asterisks of the length specified will be printed upon program execution.

Example:

```
* 10 LET A = 1234
* 20 PRINT USING 30,A,A
* 30: ##### ###
* 40 END
* RUN

    1234 ***
```

If an integer type field is preceded by a dollar sign ($), the $ will float up against the first non-zero digit in the field upon program execution.

Example:

```
* 10 LET A = 123
* 20 PRINT USING 30,A
* 30: $######
* 40 END
* RUN

    $123
```

DECIMAL TYPE FIELD

Each numeric of a decimal type field is indicated by a #; the field width must also include a # for the algebraic sign if minus. Upon program execution, the numbers of a decimal type field are right-justified within the field and the value is rounded to the number of places specified by the #'s following the decimal point.

Example:

```
* 10 LET A = 123.45
* 20 LET B = -3.456
* 30 LET C = -.017
* 40 PRINT USING 50,A,B,C
* 50: ###.##   ##.####   #.##
* 60 END
* RUN

    123.45   -3.4560   -.02
```

Note

The remarks concerning the use of the dollar sign and display of asterisks in regard to the integer type field also apply to the decimal type field.

EXPONENTIAL TYPE FIELD

An exponential type field is a decimal type field followed by four up-arrows (↑↑↑↑); the up-arrows serve to reserve space for placint an exponent. The field width must include a # for the algebraic sign if minus. For negative values, a minimum of two #'s should be specified to the left of the decimal point to provide for the minus sign and at least one digit. The value will be rounded as with decimal type fields.

Example:

```
*  10 LET A = 123000000
*  20 LET B = 123.456
*  30 LET C = -.0177
*  40 PRINT USING 50,A,B,C
*  50:###.##↑↑↑↑  #.####↑↑↑↑   ##.##↑↑↑↑
*  60 END
*  RUN

     123.00E 06   1.2346E 02   -1.77E-02
```

ALPHANUMERIC TYPE FIELDS

An alphanumeric type field may be specified in one of four possible ways, each of these indicated by the use of a single quote (') followed by one or more letters to indicate place of the alphanumeric string within the field. Note that the quote of the designated field is also a place holder. The fields are as follows:

'L...L indicates the string is to be left-justified within the field and blank-filled or truncated.

'R...R indicates the string is to be right-justified within the field and blank-filled or truncated.

'C...C indicates the string is to be centered within the field and blank-filled or truncated to the right. If an odd number of characters is to be centered in a specified format calling for an even number of characters, the string is centered one character to the left of a centered position.

'E...E indicates the string is to be left-justified within the field and the field is to be right-extended to accommodate the string if the string is longer than the field itself.

Example:

```
010  A$="ABCDEFG"
020  B$="ABCDEFGHIJKL"
030  PRINT"123456789012345678901234567890123456789012345678901234567890"
040  PRINT
050  PRINT USING 100,A$
060  PRINT USING 110,A$
070  PRINT USING 120,A$
080  PRINT USING 130,A$
090  PRINT USING 140,B$
100: 'LLLLLLLLL          LEFT JUSTIFIED IN A 10-CHAR FIELD
110: 'RRRRRRRRR          RIGHT JUSTIFIED IN A 10-CHAR FIELD
120: 'CCCCCCCCC          CENTER JUSTIFIED IN A 10-CHAR FIELD
130: 'EEEEEEEEE          EXTENDED FIELD LONGER THAN STRING
140: 'EEEEEEEEE          EXTENDED FIELD SHORTER THAN STRING
150  END
```

When executed, this program will print:

```
123456789012345678901234567890123456789012345678901234567890

ABCDEFG           LEFT JUSTIFIED IN A 10-CHAR FIELD
   ABCDEFG        RIGHT JUSTIFIED IN A 10-CHAR FIELD
 ABCDEFG          CENTER JUSTIFIED IN A 10-CHAR FIELD
ABCDEFG           EXTENDED FIELD LONGER THAN STRING
ABCDEFGHIJKL        EXTENDED FIELD SHORTER THAN STRING
```

LITERAL TYPE FIELD

A literal type field is composed of characters (other than control characters). Upon program execution, the field will appear exactly as indicated by the image statement.

Example:

```
*  10 LET A = 123.450
*  20 PRINT USING 30,A
*  30: THE VALUE OF A IS  $####.##
*  40 END
*  RUN

     THE VALUE OF A IS     $123.45
```

CONCATENATION OF MULTIPLE FORMATTED IMAGES

The output of multiple PRINT USING or PRINT # USING statements can be placed on one line by use of a comma or semicolon following an output list. Images will be concatenated end-to-end. When used in conjunction with MARGIN to extend the right-most character position, lines can be formatted beyond the normal length of 75 characters.

DEFINING FUNCTIONS

The user can define any function which he expects to use a number of times in his program by use of a DEF statement. The name of the defined function must be three alpha characters. The user may define up to 26 functions. One suggested method of accounting for the number of functions within a program is to label function names alphabetically; e.g., FNA, FNB...., FNZ.

The handiness of such a function can be seen in a program where the user frequently needs the function (e raised to -x squared). He would introduce the function by the statement:

```
*  10 DEF FNE (X) = EXP (-X**2)
```

and later on call for various values of the function by such statements as

```
*  100 LET A = FNE(.1)
*  100 LET B = FNE(3.45)
```

Such a definition can be a great time-saver when the user wants values of some function for a number of different values of the variable.

The function to find the length of the hypotenuse of a right triangle will serve as another example. Given sides of X and Y, the function can be formatted in the statement

```
* 20 DEF FNA(X,Y) = SQR (X**2 + Y**2)
```

The function can then be used in the program as often as desired. For example:

```
* 50 LET H = FNA(3,4)
* 60 LET G = FNA(A+6,B-3)
```

The PRINT statement

```
* 70 PRINT H,G
```

will then result in the printout of the two required answers.

The DEF statement must occur previous to the use of the function in the program, and the expression to the right of the equal sign may be any formula which can fit onto one line. It may include any combination of other functions, including those defined by different DEF statements, and it can involve other variables besides the one denoted as the argument of the function. Thus, assuming FNR is defined by:

```
* 10 DEF FNR(X) = SQR (2+LOG(X)-EXP(Y*Z)*(X+SIN(2*Z)))
```

the current value of Y and Z will be used in the computation of X.

A DEF statement can contain up to nine arguments; the total number of arguments for all DEF statements within a program is limited to 99.

MULTIPLE-LINE DEF STATEMENT

The user may find occassions for the use of the DEF statement wherein he wishes to assign arguments or values which cause the statement to exceed the length of a line. If a DEF statement requires more than one line for the definition of a function, the function may be introduced with a DEF statement in which no equal sign appears, continue in a series of lines in which arguments or values are assigned, and end in a line containing the word FNEND. The function is thus defined in a multiple-line DEF statement, the end of the statement indicated by the line FNEND. Local variables defined within a function definition bear no relation to similarly-named variables used outside the definition. Multiple-line DEF statements may not be nested. Transfers from inside a multiple-line DEF statement to outside, and vice versa, are not allowed.

The following examples illustrate the use of the multiple-line DEF statement.

Example 1

```
*10  DEF FNX(A,B)
*20  FNX=A
*30  IF A < B THEN 50
*40  FNX=B
*50  PRINT "FNX=";FNX
*60  FNEND
*70  X1=FNX(5,7)
*80  END
*RUN
```

Lines 10 through 60 constitute the DEF statement. The program results in the printout.

```
FNX=  5
```

Example 2

```
*10  C=3
*20  D=4
*30  DEF FNA(X,Y)C,D
*40  C=5
*50  D=10
*60  FNA=X
*70  IF X=Y THEN 90
*80  FNA=Y
*90  PRINT "C="C;"D="D;"FNA="FNA
*100 FNEND
*110 C1=FNA(9,7)
*120 PRINT"C="C;"D="D
*130 END
*RUN
```

Lines 30 through 100 constitute the DEF statement; therefore, the values of C and D outside the statement bear no relation to values of C and D assigned within the statement. The program results in the printout

```
C=  5  D=  10  FNA=  7
C=  3  D=   4
```

DATA INPUT DURING PROGRAM EXECUTION

There are times when it is desirable to enter data during the running of a program. This is particularly true when one person writes the program and saves it in the system and other persons are to supply the data when they wish to make use of the program. Data may be requested by means of an INPUT statement, which acts as a READ statement but does not draw numbers from a DATA statement. If, for example, the user is to supply values for X, Y, and Z into a program, the statement

```
INPUT X,Y,Z
```

will appear ahead of the first statement that is to use these variables. When the system encounters this statement during program execution, the terminal will print out a question mark. The user then types values for X, Y, and Z immediately after the question mark, each separated by a comma, depresses the return key, and the computer resumes program execution.

An INPUT statement can be used in conjunction with a PRINT statement to permit identification of variable values being requested. The user can employ the sequence

```
* 20 PRINT "WHAT ARE X, Y, Z";
* 30 INPUT X, Y, Z
```

and the terminal will print out the following during program execution:

```
WHAT ARE X, Y, Z?
```

to which the user must respond with values, on the same line. (Without the semicolon at the end of statement 20, the question mark would have been printed on the next line.)

If an INPUT statement is employed in a loop to repeatedly request input of a numeric value, program execution must be terminated by typing the letter S (or any word beginning with the letter S, e.g., STOP) after the question mark.

It may take a long time to enter large amounts of numeric values using INPUT statements. Therefore, INPUT statements should be used only when small quantities of values are to be entered, or when there is a requirement to enter values during the running of the program.

NOTE: The special case for matrix data input during program execution when use is made of the MAT INPUT statement is described in "Matrices" below.

A program to convert degrees Fahrenheit to Centigrade serves to illustrate the usefulness of the INPUT statement. Because this program is designed to loop back to the program beginning each time to demand another input, the user must type in the word STOP after the question mark at a time he wishes to terminate the program.

```
*   10 PRINT "FAHRENHEIT";
*   20 INPUT F
*   30 LET C = (F-32) * 5/9
*   40 PRINT "CENTIGRADE =" C
*   50 PRINT
*   60 GOTO 10
*   70 END
*   RUN

FAHRENHEIT ?32
CENTIGRADE = 0

FAHRENHEIT ?212
CENTIGRADE = 100

FAHRENHEIT ?STOP
```

MATRICES

A set of special statements is provided for operating upon matrices. These statements are identified by the word MAT, with which each such statement begins. Although the user can construct programs using only elementary BASIC to perform calculations on--or otherwise manipulate--matrices, the set of MAT statements simplifies the programming effort by shortening programs considerably.

The format of the MAT statements are:

MAT READ A,B,C,...	Read into matrices A, B, C,..., their dimensions having been previously specified. Data is read in row-wise sequence from standard-format DATA statements, and entered into the matrices. Each matrix may be totally or partially filled. Zeroes are automatically assigned to any unfilled positions.
MAT PRINT A,B,C,...	Print matrices A, B, C,... The semicolon, TAB, and SPC can be used, as in the normal PRINT statement. Double space is provided for between rows; between folded parts of the same row, single space is provided.
MAT INPUT A	Input desired values for elements of matrix A during program execution time.
MAT C = A + B	Add two matrices A and B and store result in matrix C.

MAT C = A - B	Subtract matrix B from matrix A and store result in matrix C.
MAT C = A * B	Multiply matrix A by matrix B and store result in matrix C.
MAT C = INV(A)	Invert matrix A and store resulting matrix in C.
MAT C = TRN(A)	Transpose (interchange rows and columns) matrix A and store resulting matrix in C.
MAT C = (K) * A or MAT C = A * (K)	Multiply matrix A by value represented by K. K may be either a number or an expression, but in either case it must be enclosed in parentheses.
MAT C = CON	Each element of matrix C is set to one.
MAT C = ZER	Each element of matrix C is set to zero.
MAT C = IDN	Diagonal elements of matrix C are set to one's, yielding an identity matrix.

The last three MAT statements may also be written with subscripts suffixed to the right-hand side; e.g., MAT C = ZER(I,J). The use of this form is described below.

Special rules apply to the dimensioning of matrices which occur in MAT instructions. DIM statements indicate what the maximum dimension of a matrix is to be. Thus

```
DIM M(20,35)
```

means that M may have up to 20 rows and up to 35 columns. The dimensions of all matrices occurring in MAT statements must be specified in DIM statements; otherwise, automatic dimensioning (subscript values of 10 or less) is implied.

Note

> Rows and columns of matrices are numbered 1 through n. That is, there is no row or column numbered 0 in matrices used in MAT statements.

The current dimension of a matrix may be determined either when it is initially defined by the dimension statement or by special usage of certain MAT statement forms. The four general forms which may be used to accomplish dynamic redimensioning are:

1. MAT READ A(M,N)
2. MAT A = ZER(M,N)
3. MAT A = CON(M,N)
4. MAT A = IDN(N,N)

The first, MAT READ, will redefine the current dimensions of matrix A as M rows and N columns and then read M*N data values to fill in the elements. More than one matrix may be redimensioned and read with a single statement.

The other three forms are used to redefine the current dimensions of a matrix (A) and then fill its elements with values as specified by the statement type.

The rules for dynamic redimensioning are as follows:

1. No dimension may be changed to a value that exceeds its original declaration in the DIM statement.

2. Using the statement types described above, dimensions may be redefined in either the upward or downward direction as long as the definition is within the bounds of item 1 above and the original declaration in the DIM statement.

For example, a matrix specified in the DIM statement as (6,4) might be redimensioned as (4,4), but not as (10,2) -- by rule 1 -- or (5,5) -- by rule 2.

In addition to use of a DIM statement, and possibly a declaration of current dimensions, the user must use MAT statements with care. For example, a matrix product MAT C = A * B may be illegal for one of two reasons: A and B may have dimensions such that the product is not mathematically defined, or even if it is defined, C may not have reserved enough space for the answer. In either case, the message IN XXXX DIM ERROR results, where XXXX is the line number of the statement in question.

The same matrix may occur on both sides of a MAT statement in cases of addition, subtraction, multiplication by a constant, or inversion, but not in any other statement forms. Legal forms are:

```
MAT A = A + B
MAT A = A - B
MAT A = (2.5)*A
MAT A = INV (A)
```

Also, note that the special form of matrix multiplication

```
MAT B = A * A
```

is legal.

Illegal forms are:

```
MAT A = B
MAT A = B*A
MAT A = TRN(A)
MAT A = A + B - C
```

The last example is an attempt to use more than one arithmetic operator in a MAT statement. Each matrix operation requires its own matrix statement.

A 2-dimensional string matrix (e.g., A1$(10,20)) is not permitted. No MAT operations are permitted for string variables.

The following program illustrates some simple operations upon matrices by the use of MAT statements.

```
*  10  DIM A(2,3), B(2,3), S(2,3)
*  20  DIM D(2,3), M(2,3), T(3,2)
*  30  MAT READ A,B
*  40  REM SUM OF MATRICES
*  50  MAT S = A + B
*  60  MAT PRINT S
*  70  REM DIFFERENCE OF MATRICES
*  80  MAT D = A - B
*  90  MAT PRINT D
*  100 REM MULTIPLY MATRIX
*  110 MAT M = (2) * A
*  120 MAT PRINT M
*  130 REM TRANSPOSE MATRIX
*  140 MAT T = TRN(B)
*  150 MAT PRINT T
*  160 DATA 1,2,3,4,5,6
*  170 DATA 6,5,4,3,2,1
*  180 END
*  RUN

 7          7          7

 7          7          7


-5         -3         -1

 1          3          5


 2          4          6

 8         10         12


 6          3

 5          2

 4          1
```

The MAT INPUT statement permits input of data, pertaining to the elements of a matrix, at program execution time. The function NUM(X) can be utilized to supply a count of the number of data elements entered; thus, the matrix array can be filled to any level desired (.i.e., user need not input data elements to fill the entire array). The count of NUM(X) always reflects the number of input data elements for the most recently executed MAT INPUT statement. If more than one line of values is required, the line (and subsequent lines, if needed) is terminated with an ampersand (&) to indicate continuation. The ampersand may or may not be comma-separated from the last value. The MAT INPUT statement may be used with either 1- or 2-dimensional arrays. The 1-dimensional array is filled beginning with element 1. Two-dimensional arrays are filled in a row sequence.

Two examples of the use of the MAT INPUT statement are as follows:

Example 1:

```
*  10 DIM S(100)
*  20 MAT INPUT S
*  30 PRINT S(1);" + ";S(2);" = ";S(1)+S(2)
*  40 LET T = S(1)+S(2)
*  50 FOR I = 3 TO NUM(X)
*  60 LET T = T + S(I)
*  70 PRINT"          + ";S(I);"  =  ";T
*  80 NEXT I
*  90 END

READY

*RUN

?1,2,3,4,5 &
?6,7,8,&
?9,10,11
    1   +    2   =   3
        +    3   =   6
        +    4   =  10
        +    5   =  15
        +    6   =  21
        +    7   =  28
        +    8   =  36
        +    9   =  45
        +   10   =  55
        +   11   =  66
```

Example 2:

```
*  10 DIM M1(3,4)
*  20 MAT INPUT M1
*  30 MAT PRINT M1;
*  40 END

READY

*RUN

?1,2,3,4,5,6,7

    1    2    3    4
    5    6    7    0
    0    0    0    0
```

ADDITIONAL FUNCTIONS

BASIC provides for the use of other functions in addition to the standard mathematical functions listed in Section IV.

These additional functions are as follows:

INT(X)	TIM(X)	NUM(X)	TAB(X)	LEN(X$)	STR$(N)
RND(X)	CLK$	SST(X$,Y,Z)	SPC(X)	LIN(X)	VAL(S$)
SGN(X)	DAT$			ASC(X)	TST(S$)
DET(X)					HPS(X)

Function INT(X)

Purpose: To truncate a number to integer form.

Format: INT (expression)

Examples:
```
* 10 PRINT INT (2.35)
* 20 PRINT INT (-2.35)
* 30 PRINT INT (2.9)
```

are three examples of this function placed in a PRINT statement and used to truncate a number. The resultant printouts would produce 2, -3, and 2, respectively.

Function RND(X)

Purpose: To generate random numbers for computational procedures requiring random variables.

Format: The general format is

RND (any variable or constant)

which will produce a random number between (but not including) 0 and 1.

If a great number of these random numbers are produced, it becomes apparent that they tend to fall uniformly over the range, for the numbers come from a uniformly distributed population.

Examples:
```
*  10 FOR L = 1 TO 20
*  20 PRINT RND(X),
*  30 NEXT L
*  40 END
*  RUN
```

might generate the following:

0.3199251	0.0590169	0.4018556	0.6280534	0.2292995
0.8075665	0.964758	0.2424602	0.066037	0.368314
0.3074467	0.4493044	0.7489442	0.4024822	0.301177
0.7088735	0.2340001	0.9746831	0.5227955	0.6405085

If random integers between 0 and 10 are desired, statement 20 can be changed to read

```
* 20 PRINT INT (10*RND(X)),
```

which results in

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

If statement 20 were changed to read

```
* 20 PRINT INT (20*RND(X)+5),
```

then the printout would contain random numbers between integers 5 and 25.

The range of random numbers generated, therefore, is dependent upon how function RND(X) is modified.

The function RND(X) lends itself readily to programs involving probability. For example, to simulate a 5-trial coin tossing contest, the following program can be written:

```
* 10 FOR T = 1 TO 5
* 20 IF RND(T)< =0.5 THEN 50
* 30 PRINT "HEADS"
* 40 GOTO 60
* 50 PRINT "TAILS"
* 60 NEXT T
* 70 END
```

The program execution will be a reasonable facsimile of the results of a coin tossed five times.

The use of the RND function as described above is appropriate when the same sequence of random numbers is to be generated each time a program is run. If the variable or constant used as an argument is a positive quantity and is not changed, the same sequence of random numbers is generated for each execution of the program.

The use of a negative argument for the RND function will cause an unpredictable series of random numbers to be generated each time the program is run. For example, if the user wishes different sequences of random numbers for each execution of his program, he may use one of the following techniques:

```
* 10 LET X = -1
* 20 FOR I = 1 TO 20
* 30 PRINT RND(X)
* 40 NEXT I
* 50 END

* 10 LET X = 1
* 20 FOR I = 1 TO 20
* 30 PRINT RND(-X)
* 40 NEXT I
* 50 END

* 10 FOR I = 1 TO 20
* 20 PRINT RND(-1)
* 30 NEXT I
* 40 END
```

Function SGN(X)

Purpose: To determine the sign of an expression.

Format: SGN (expression)

The function yields +1, -1, or 0, depending upon the value of the expression. The following list gives the options:

	(Value of expression)	Yields
SGN	(zero)	0
SGN	(positive,non-zero)	+1
SGN	(negative,non-zero)	-1

Examples: * 10 IF SGN(X) = 1 THEN 100

In this statement, the value of X must be positive to accomplish the transfer of processing to statement 100.

The statement

```
* 20 LET X = SGN(Y)*ABS(X)
```

assigns to X the sign resulting from the value of Y.

Function DET(X)

Purpose: To obtain the determinant of the last matrix inverted.

Format: DET (any variable or constant)

Examples:

```
*10  MAT B=INV(A)
*20  LET C=DET(X)
*30  PRINT C
```

The program, when executed, will invert matrix A, store the result in matrix B, and print out the value of C, the determinant of matrix B.

The determinant can be made an element of a more complex numeric expression.

```
*10  PRINT 2*DET(X)
*20  IF DET(X)=0 THEN 60
```

Any attempt to invert a singular matrix does not stop the program, but DET(X) is set to zero. For any program, the user must decide if a determinant is large enough to be meaningful.

Function TIM(X)

Purpose: To obtain elapsed processor time in seconds.

Format: TIM (any keyboard character)

Examples:

```
*50 PRINT "PROCESSOR TIME=";TIM(X);"SECONDS"
```

A program including such a statement, when executed, would contain a printout line

```
PROCESSOR TIME= <value> SECONDS
```

The processor time may be assigned a variable name.

```
*50  LET T=TIM(X)
*60  PRINT "PROCESSOR TIME =";T
```

Function CLK$

Purpose: To provide the time of day as a string.

Format: CLK$

Examples: *50 PRINT CLK$

A program including such a statement, when executed, would contain a printout line indicating time of day in hours ranging from 1 to 24 and in portions of hours, such as NN.NNN.

The time of day may also be assigned to a string variable.

```
*10 LET T$=CLK$
*20 PRINT T$
```

Function DAT$

Purpose: To provide the calendar date as a string.

Format: DAT$

Examples: *50 PRINT DAT$

A program including such a statement, when executed, would contain a printout line indicating the calendar date (month, date, year), such as

MM/DD/YY

The calendar date may also be assigned to a string variable.

```
*10 LET A$=DAT$
*20 PRINT A$
```

Function NUM(X)

Purpose: To supply count of number of data elements in response to request from MAT INPUT statement.

Format: NUM (any alphanumeric character)

Refer to MAT INPUT statement under "Matrices" in this Section for an example concerning use of NUM(X).

Function SST(X$,Y,Z)

Purpose: To extract selected characters of a string.

Format: SST(string variable, beginning character, number of characters)

Refer to the use of the LET statement under "Alphanumeric Data and String Manipulation", in this Section, for an explanation of the use of this function.

Function TAB(X)

Purpose: To position data field at indicated character position within an output line.

Format: TAB(expression), <data to be printed>

Refer to "Spacing Within An Output Line with Functions TAB(X) and SPC(X)", in this Section, for an explanation of the use of this function.

Function SPC(X)

Purpose: To insert spaces at indicated positions within an output line.

Format: SPC(expression); <data to be printed>

Refer to "Spacing Within an Output Line with Functions TAB(X) and SPC(X)", in this Section, for an explanation of the use of this function.

Function LEN(X$)

Purpose: To determine the number of characters in a specified string variable.

Format: LEN(string variable)

Examples:

```
*    10 READ A$,B$,C$
*    20 PRINT LEN(A$);LEN(B$);LEN(C$)
*    30 DATA LENGTH, OF, STRING
*    40 END
*    RUN
```

results in a printout of

```
 6   2   6
```

The value of LEN may be assigned to a variable.

```
*    10 LET X=LEN(A$)
*    20 PRINT"LENGTH OF STRING=";X
```

Function LIN(X)

Purpose: To provide the last line number encountered in reading from or writing to a file.

Format: LIN(file designator)

Examples:

```
* 10 FILES A
* 20 SCRATCH #1
* 30 FOR I=1 to 45
* 40 WRITE #1,I;
* 50 NEXT I
* 60 PRINT "LAST LINE WRITTEN IS ";LIN(1)
* 70 RESTORE #1
* 80 PRINT
* 90 FOR I=1 to 24
* 100 READ #1,X1
* 110 PRINT X1;
* 120 NEXT I
* 130 PRINT
* 140 PRINT "LAST LINE READ IS ";LIN(1)
* 150 END
* RUN
```

upon execution, the program will produce

```
LAST LINE WRITTEN IS 50

 1   2   3   4   5   6   7   8   9  10  11  12
13  14  15  16  17  18  19  20  21  22  23  24

LAST LINE READ IS 30
```

The listing of file A will show that it contains the following data:

```
000010  1,   2,   3,   4,   5,   6,   7,   8,   9,
000020  10,  11,  12,  13,  14,  15,  16,  17,  18,
000030  19,  20,  21,  22,  23,  24,  25,  26,  27,
000040  28,  29,  30,  31,  32,  33,  34,  35,  36,
000050  37,  38,  39,  40,  41,  42,  43,  44,  45,
```

The value of LIN may be assigned to a variable.

```
* 10 LET N=LIN(1)
* 20 PRINT "LAST LINE READ IS ";N
```

Function ASC(X)

Purpose: To provide the numeric value of a specified character or, for the case of non-printing characters, an abbreviation.

Format: ASC $\left\{\begin{array}{l}\text{(character)}\\ \text{(abbreviation)}\end{array}\right\}$

Examples:

```
*    10 PRINT "VALUE FOR A IS ";ASC(A)
*    20 PRINT "VALUE FOR CR IS ";ASC(CR)
*    30 END
*    RUN
```

which results in

```
VALUE FOR A IS 65
VALUE FOR CR is 13
```

The value of ASC may be assigned to a variable.

```
*    10 LET X=ASC(A)
*    20 PRINT "VALUE FOR A IS ";X
```

The conversion equivalents for characters and non-printing characters are listed in the table "Numeric Code Table" in this section.

Function STR$(N)

Purpose: To produce a string corresponding to a value of a number represented by an expression.

Format: STR$ (expression)

Examples: The value of STR$ may be assigned to a string variable

```
*10 LET X$=STR$(N)
```

or may be used directly

```
*20 PRINT STR$(N)
```

where N is a number, STR$ converts N to a string containing the same digits.

```
*10  LET N=77.233
*20  LET X$=STR$(N)
*30  LET Y$=STR$(63)
*40  PRINT X$;Y$
*50  END
```

when executed, the program results in

```
77.233     63
```

Use of STR$ implies placement of the string right-justified in the smallest zone into which it will fit. Blanks will occupy the remaining character positions of the zone.

Function VAL(S$)

Purpose: To produce a numeric value corresponding to the value of a string represented by a string variable.

Format: VAL (string variable)

Examples: The value of VAL may be assigned to a variable

```
*10 LET A=VAL(S$)
```

or may be used as an element of a numeric expression

```
*20 LET A1=2*VAL(S$)
*30 PRINT 3*VAL(S$)+A+A1
```

The string variable of VAL must be a valid constant. The program

```
*10  LET A$="12345"
*20  LET B$="12.95"
*30  LET C=VAL(A$)
*40  PRINT C;VAL(B$)
*50  END
```

when executed, results in

```
12345       12.95
```

Function TST(S$)

Purpose: To produce a 1 as output if a string represented by a string variable can be interpreted as a number, or produce a 0 if the string cannot be interpreted as a number.

Format: TST (string variable)

Examples: The value of TST may be assigned to a variable

```
*10 LET T=TST(S$)
```

or may be used as an element of a numeric expression

```
*20 PRINT VAL(S$)*TST(S$)
*40 IF TST(S$)=0 THEN 50
```

The program

```
*10  LET A$="49"
*20  LET T=TST(A$)
*30  IF T=0 THEN 50
*40  PRINT VAL(A$)
*50  END
```

when executed, results in

```
49
```

Function HPS(X)

Purpose: To provide a horizontal print position of the next field to be transmitted to a specified file.

Format: HPS (file designator)

Examples: The function can be assigned to a variable

```
*10  LET P=HPS(0)
```

or can be used as an element of a numeric expression

```
*20  PRINT 12+HPS(0)
```

The program

```
*10  FOR X=1 to 8
*20  PRINT X;
*30  NEXT X
*40  LET A=HPS(0)
*50  PRINT A
*60  END
```

when executed, results in

```
1  2  3  4  5  6  7  8  49
```

The horizontal print position of the file is 49.

```
*10  FILES OUT1
*20  SCRATCH #1
*30  FOR I=1 TO 5
*40  WRITE #1,I;
*50  NEXT I
*60  PRINT "HOR. PRINT POS. OF FILE 1=";HPS(1)
*70  END
```

This program when executed, results in

```
HOR. PRINT POS. OF FILE 1=44
```

A listing of file OUT1 would show

```
10   1, 2, 3, 4, 5,
```

The file designator for function HPS must be a numeric value between zero and 8 inclusive. Zero is interpreted as being the user's terminal.

The use of function HPS is limited to providing the horizontal print position for output. If the specified file is open for input, a zero horizontal print position is returned.

SUBROUTINES

When a particular part of a program is to be performed more than one time, or possibly at several different places in the overall program, the part or parts are most efficiently programmed as subroutines. Subroutines can be likened to programs within the main program which permit the user to partition his main program.

The subroutine is entered by the way of a GOSUB statement. For example,

```
* 90 GOSUB 210
```

directs the processing to jump to statement 210, the first statement of the subroutine. The last statement of the subroutine to be executed must be a RETURN statement directing the processing to return to the earlier part of the program. For example,

```
* 350 RETURN
```

will tell the processing to go back to the first statement numbered greater than 90 and to continue the program from there.

GOSUB statements may be used within subroutines to branch to still other subroutines. The following nonsense program illustrates the technique:

```
*    10 READ L
*    20 GOSUB 50
*    30 PRINT A,B,C,
*    40 STOP
*    50 REM THIS IS SUBROUTINE 1
*    60 LET A = 5
*    70 GOSUB 100
*    80 LET B = 10
*    90 RETURN
*   100 REM THIS IS SUBROUTINE 2
*   110 LET C = 15
*   120 FOR I = 1 TO L
*   130 LET C = I*C
*   140 NEXT I
*   150 RETURN
*   160 DATA 5
*   170 END
```

Statement 20 jumps the processing to Subroutine 1. Statement 70, in turn, transfers processing from Subroutine 1 to Subroutine 2. Statement 150 then returns the processing to the most recent point of departure -- statement 80. When statement 90 is encountered, processing is returned to statement 30. Statement 40 prevents the program from falling back into Subroutine 1 again and the program is terminated.

LOOPS

Frequently, there are operations in programming that must be repeated many times; therefore, some statements within a program must be executed many times. This repetition of a set of statements is referred to as a loop. For example, if a table were required of the first 100 positive integers and their square roots, it could be obtained by this program.

```
*     10 PRINT 1, SQR(1)
*     20 PRINT 2, SQR(2)
       .      .      .
       .      .      .
       .      .      .
*    990 PRINT 99, SQR(99)
*   1000 PRINT 100, SQR(100)
*   1010 END
```

By means of two BASIC statements, a programming loop can be written that will accomplish the same as the program with 101 statements but in only four statements; namely,

```
*   10 FOR X = 1 TO 100
*   20 PRINT X, SQR(X)
*   30 NEXT X
*   40 END
```

The FOR statement denotes the beginning of the loop, and it specifies the range (1 to 100) for the given variable (X) and in unit steps (implied step-size of 1 when STEP is not given) as the program keeps passing through the loop. If the steps were to be increments of other than 1, then statement 10 would include the word STEP followed by the required size. If the increments were, say 2, then the statement would be written as

```
*  10 FOR X = 1 TO 100 STEP 2
```

The NEXT statement (statement 30) terminates the loop and returns the system to statement 10, with the statement between being executed for each pass through the loop. When the loop has been executed the specified number of times (100, in the example), then it directs the system to the statement after the NEXT statement (statement 40).

The program loop described above is a simple one. The FOR and NEXT statements can be used effectively in more complex problems wherever iterations are required. For example, if integration of a function is required, the FOR statement can be used to define limits and set the count of iterations through the loop. Computation statements can then be made and the NEXT statement used to repeat the iteration until the count has been achieved.

It is possible, as well as useful, to have loops within loops. However, a loop cannot cross another loop. To illustrate:

This method of creating loops is allowed:

```
┌────────FOR X
│         .
│         .
│         .
│   ┌────FOR Y          For each pass through the X loop,
│   │     .
│   │     .             the Y loop is executed the
│   │     .
│   └────NEXT Y         specified number of times.
│         .
│         .
│         .
└────────NEXT X
```

For example, if the X loop had a range of 5 and the Y loop a range of 10, then for each pass through the X loop the Y loop is executed 10 times. When the X loop has been executed 5 times, the Y loop will have been executed 50 times (i.e., 10 Y passes per 1 X pass).

This method is also allowed:

```
FOR X
FOR Y
FOR Z
NEXT Z
FOR W
NEXT W
NEXT Y
FOR Z
NEXT Z
NEXT X
  .
  .
  .
END
```

This method is not allowed; note the cross-over of the loops:

```
FOR X
FOR Y
NEXT X
NEXT Y
```

Loops may also be created within a program by the use of GOTO and READ statements. If a READ statement contains a variable to which the user wishes to assign more than one value, a GOTO statement will direct the program to loop back to the READ statement and assign another value.

The loop will be performed as many times as there are values available in a DATA statement. When the values have all been assigned, execution of the program is terminated and the message OUT OF DATA is printed.

The following sample program illustrates the use of a GOTO-READ loop:

```
10 READ A,B,D,E
15 LET G = A*E-B*D
20 IF G = 0 THEN 65
30 READ C,F
37 LET X = (C*E-B*F)/G
42 LET Y = (A*F-C*D)/G
55 PRINT X,Y
60 GOTO 30
65 PRINT "NO UNIQUE SOLUTION"
70 DATA 1,2,4
80 DATA 2,-7,5
85 DATA 1,3,4,-7
90 END
```

This program has assigned one set of values to the variables A, B, D, E, but three values each to the variables C and F. Therefore, the solution should provide six answers. To achieve multiple answers, a loop is created by the way of statement 60. Here the program is directed back to statement 30 to assign new values to C and F from the data block.

The program and the resulting run would appear as follows:

```
*  10 READ A,B,D,E
*  15 LET G=A*E-B*D
*  20 IF G = 0 THEN 65
*  30 READ C,F
*  37 LET X = (C*E-B*F)/G
*  42 LET Y = (A*F-C*D)/G
*  55 PRINT X,Y
*  60 GOTO 30
*  65 PRINT "NO UNIQUE SOLUTION"
*  70 DATA 1,2,4
*  80 DATA 2,-7,5
*  85 DATA 1,3,4,-7
*  90 END
*  RUN

           4        -5.5
   0.6666667   0.1666667
   -3.666667   3,8333333

OUT OF DATA IN 30
```

LISTS AND TABLES

Often in the writing of a program the need arises to make use of a list of numbers. The user will find it most advantageous to give the list a single variable name rather than provide separate variables for each number in the list. For example, if 25 salesmen were to be listed in a program, the list could be called S and the salesmen would be represented by S and a subscript, ranging from S(1) to S(25). Thus S(5) would represent the fifth salesman in list S and S(25) would represent the 25th or last salesman in the list. The user may also find the need to make use of tables in his programs. Here again, a single variable name rather than separate variables for each entry of a table is most convenient. For example, P(3,J) would represent row 3, column J in table P; table P could be a 5 by 10 array. P(5,10) represents the entire table and could be dimensioned as such in a DIM statement.

Lists and tables thus permit the user to enter groups of numbers into his program that are to be worked upon concurrently. Such programs can be used over and over again, with the user updating the data each time he uses the program.

The usefulness of employing a list in a program can be illustrated by an example. A brush salesman has 10 kinds of brushes he carries in his sample case. At the end of the day, he wishes to compute the dollar value of the orders he has taken. The prices of the 10 brushes are as follows:

0.50, 1.75, 2.25, 2.75, 3.45, 4.00, 4.25, 4.75, 5.00, 5.25

In writing his program, the salesman enters his quantity of sales for individual brushes and then asks for a printout of total sales.

```
*    10 DIM P(10)
*    20 FOR I = 1 TO 10
*    30 READ P (I)
*    40 NEXT I
*    50 LET S = 0
*    55 FOR I = 1 TO 10
*    60 READ B
*    70 LET S = S + B * P(I)
*    75 NEXT I
*    80 PRINT "TOTAL SALES = $" S
*    90 DATA 0.50, 1.75, 2.25, 2.75, 3.45
*   100 DATA 4.00, 4.25, 4.75, 5.00, 5.25
*   110 DATA 0,5,7,3,12
*   120 DATA 25,15,30,10,35
*   130 END
```

At the end of each work day, the salesman updates DATA statements 110 and 120 to reflect his orders and obtain new sales totals.

The use of tables is simply the extension of the use of lists. Refer to Appendix D for a sample program using both a list and a table.

The user should be aware of the need to dimension a list or table to at least the minimum of the subscript value. But it may be expedient to dimension somewhat generously over the minimum to permit changes to an existing program. For example, the brush salesman would do well to change statement 10 in his program to:

```
*   10 DIM P(25)
```

This will enable him to use his program if he adds up to 15 additional kinds of brushes to his line.

No harm will be done if extra large dimensions are defined in DIM statements, but space in computers is limited and a realistic dimension is in the best interest of all users of the time-sharing system.

ALPHANUMERIC DATA AND STRING MANIPULATION

BASIC has the ability to manipulate alphanumeric information in addition to numeric data. Data consisting of alphanumerics and certain, special characters can be treated as if it were numeric data.

A sequence of alphanumeric data is referred to as a "string"; the string size, in turn, is limited to 132 valid characters. Initially, space for 20 characters is allocated; the space is then expanded if space for more characters is required. Manipulation of a string is by means of a "string variable", created by following any permissible BASIC variable with the character $. For example,

```
A$,K1$,X5$
```

are valid string variables. Manipulation, incidentally, should not be interpreted as meaning arithmetic operations; such operations cannot be performed on string variables.

The use of alphanumeric data and string manipulation are restricted to certain BASIC statements. The following is a list of these statements, each accompanied by explanation of alphanumeric data use and string manipulation as applicable. The use of quotes to enclose strings is recommended where doubt exists as to their use; superfluous quotes will be ignored by the system.

- DIM

  A user may set up a list of allied strings as a one-dimensional array. The DIM statement must then be used to reserve space. For example,

  ```
  * 10 DIM A$(15),B$(25)
  ```

  Space for fifteen 20-character strings are then reserved by A$ and twenty-five 20-character strings by B$. The user may then select particular strings within a string list; for example, A$(4) would be the fourth string in the A$ list and B$(6) the sixth string in the B$ list.

* LET

The LET statement may be used to assign the contents of one string variable to another string variable, assign a string constant to a string variable, concatenate strings, and extract selected characters of a string. Quotes must enclose any assigned string constant. An ampersand (&) is used to indicate string concatenation.

The statement

```
*10 LET R$=T$
```

assigns the contents of the string T$ to R$.

The statement

```
*10 LET G$ = "THIS IS A STRING"
```

assigns the string, THIS IS A STRING, to G$.

String concatenation is limited within one LET statement to two string variables or one string constant and one string variable.

The statements

```
*10 LET A$ = "JOHN DOE "
*20 LET B$ = "EMPLOYEE NUMBER 12345"
*30 LET C$ = A$ & B$
*40 PRINT C$
```

or

```
*10 LET A$ = "JOHN DOE "
*20 LET C$ = A$ & "EMPLOYEE NUMBER 12345"
*30 PRINT C$
```

when executed, will produce the printout

```
JOHN DOE EMPLOYEE NUMBER 12345
```

Extraction of selected characters of a string is achieved by use of the substring extraction function, which has the general format

SST (string variable, beginning character, number of characters)

where

1. String variable has been assigned contents of a string
2. Beginning character is numeric value to indicate position of character with which to begin extraction
3. Number value of characters to extract

Character positions of a string are numbered from left to right, 1 through 132. Based on three arguments supplied to the SST function, a substring will be extracted and stored left justified in the string variable specified to the left of the equal sign of the LET statement. Blanks within a string, of course, are considered as characters when the character count is made.

The statements

```
*10 LET A$ = "THIS IS A DEMONSTRATION OF THE SUBSTRING FUNCTION"
*20 LET B$ = SST(A$,1,10)
*30 LET C$ = SST(A$,11,14)
*40 LET D$ = SST(A$,25,25)
*50 PRINT B$
*60 PRINT C$
*70 PRINT D$
```

upon program execution, will produce printouts of

```
THIS IS A
DEMONSTRATION
OF THE SUBSTRING FUNCTION
```

- IF-----THEN or IF-----GOTO

Strings and string variables may be manipulated with these statements also. Only one string variable is permitted on each side of the relational symbol and the string must be enclosed by quotes. Relational symbols indicate relation in regard to alphabetic order.

Examples are as follows:

```
* 10 IF G$ = "THIS IS A STRING" THEN 30

* 10 IF G$ > H$ GOTO 30

* 10 IF "MAY" < >M$ THEN 30
```

- CHANGE

The change statement may be used to convert string characters to equivalent numeric code or vice versa.

The process involves two lists, one numeric, the other a string variable. When converting numeric codes to a character string, the numeric list is to contain the valid numeric equivalent of a single character in each element. Given the desired number of items to convert, the CHANGE command will perform the conversion and concatenate the resulting characters into the string variable.

In changing from a character string, the command stores the related numeric code for each character into the elements of the numeric array.

The following table lists the string characters and their equivalent numeric code.

Numeric Code Table

String Characters	Code No. (decimal)	String Characters	Code No. (decimal)
(blank)	32	@	64
!	33	A	65 (97)
"	34	B	66 (98)
#	35	C	67 (99)
$	36	D	68 (100)
%	37	E	69 (101)
&	38	F	70 (102)
'	39	G	71 (103)
(	40	H	72 (104)
)	41	I	73 (105)
*	42	J	74 (106)
+	43	K	75 (107)
,	44	L	76 (108)
-	45	M	77 (109)
.	46	N	78 (110)
/	47	O	79 (111)
0	48	P	80 (112)
1	49	Q	81 (113)
2	50	R	82 (114)
3	51	S	83 (115)
4	52	T	84 (116)
5	53	U	85 (117)
6	54	V	86 (118)
7	55	W	87 (119)
8	56	X	88 (120)
9	57	Y	89 (121)
:	58	Z	90 (122)
;	59	[	91
<	60	\	92
=	61	]	93
>	62	↑	94
?	63		

Numerics in parentheses indicate lower case

Additional symbols useful on output are:

(backward arrow)95	LF (line feed)10
EOT (end of transmission)4	CR (carriage return)13
BELL (rings bell in teletype)7	RUB-OUT (tape use only)127

Notes:

1. This is not a complete list - there are 128 characters numbered 0 through 127. Some of these numbers duplicate the above (on some teletypes) and some are just spaces.

2. The EOT character will hang up the phone if it is sent to a Model 33 Teletype.

The following sample program illustrates the use of the CHANGE statement.

```
*  10 DIM A(100)
*  20 FOR I = 1 TO 26
*  30 LET A(I) = 64 + I
*  40 NEXT I
*  45 REM AT THIS POINT THE A LIST IS 65,66,67...90
*  50 LET A(0)=20
*  60 REM CONVERT ONLY THE 1ST 20 CODES IN A
*  70 REM      TO EQUIVALENT CHARACTERS
*  80 CHANGE A TO B$
*  90 PRINT B$
* 100 END
* RUN

   ABCDEFGHIJKLMNOPQRST
```

Statement 80 has caused the conversion of numerics to their equivalent string characters. Statement 50 provides a count of the number of chracters the user wishes to convert.

- READ and DATA

READ and DATA statements are utilized in the conventional manner to manipulate alphanumeric data. A READ statement may be a mix of both numeric variables or string variables or may simply contain string variables. In turn, the DATA statement will list the sequence of data to correspond to the variables listed in the READ statement. Strings in a DATA statement must be enclosed in quotation marks if they begin with a digit or have an embedded comma. For example,

```
*  10 READ A,B$,C,D$,E$,F
        .
        .
*  90 DATA 85,XYZ,5,"4FG","MAY 26,1969",20
```

A leading blank in a string listed in the DATA statement is ignored unless the blank and its string are enclosed in quotes.

- PRINT

Strings are printed in the conventional manner; i.e., all forms of the PRINT statement are applicable when alphanumeric data is to be printed. For example,

```
*  10 READ A$,B$,C$
*  20 PRINT C$;B$;A$
*  30 DATA ING,SHAR,TIME-
*  40 END
*  RUN
```

will result in the printout of

```
TIME-SHARING
```

- INPUT

  The requirements for handling alphanumeric data in an INPUT statement correspond to those of the READ statement in that the INPUT statement may be a mix of both numeric and string variables or may contain only string variables. For example,

  * 10 INPUT X,Y$,Z

  If the string variable represents a string with an embedded comma, the string, when entered during program execution, must be enclosed in quotes. A leading blank in a string is ignored unless the blank and its string are enclosed in quotes.

- RESTORE

  Numeric data and string data are stored independently within two separate blocks of the BASIC system. The conventional RESTORE statement will restore both numeric and string data. If the user wishes to restore only numeric data he must use RESTORE followed by an asterisk:

  * 10 RESTORE*

  If the user wishes to restore only string data he must use RESTORE followed by the $ character:

  * 10 RESTORE$

Additional functions pertaining to string manipulation are available. These functions are CLK$ (to provide time of day) DAT$ (to provide calendar date), SST(X$,Y,Z) (to extract selected characters of a string), and LEN(X$) (to determine the number of characters in a specified string variable). Refer to "Additional Functions", in this chapter, for details concerning use of these functions.

ASCII DATA FILES

BASIC provides the means for creating files of data which may be read, written on, or otherwise manipulated, all within the confines of the BASIC subsystem. A data file to be used as input must be prepared in advance and must be saved before it can be used in a program. A data file on which output is to be written during execution of a program does not necessarily need to have been created before that program is executed. If not in the user's catalog of permanent files when needed for output, a file will be created as temporary, and may be changed to permanent status at log-off time. Refer to "Saving of Temporary Files" in this chapter. Data files can be created with or without line numbers. Data in a data file may range from zero to an unlimited number of characters.

All files will initially be in read mode. A file can be placed in write mode by the use of a SCRATCH # statement. Read mode may be re-established by use of the RESTORE # statement.

Data files are implemented by data file input/output statements which supplement BASIC language statements. These data file input/output statements may be categorized as follows:

- File preparation statements

```
FILES filename 1, password;....;filename n, password
FILES user-id/catalogname$password/.../
     filename$password,permissions
FILE # file designator, "filename, password"
```

- File read statements

```
READ # file designator, input list
INPUT # file designator, input list
```

- File write statements

```
WRITE # file designator, output list
PRINT # file designator, output list
PRINT # file designator, USING statement number, output list
```

- Matrix input statements

```
MAT READ # file designator, matrix input list
MAT INPUT # file designator, matrix input list
```

- Matrix output statements

```
MAT WRITE # file designator, matrix output list
MAT PRINT # file designator, matrix output list
```

- File manipulation statements

```
SCRATCH # file designator
RESTORE # file designator
BACKSPACE # file designator
```

- Utility statements

```
APPEND # file designator
MARGIN # file designator, expression
DELIMIT # file designator, { (character)    }
                           { (abbreviation) }

IF END # file designator,  { THEN } line number
                           { GOTO }

IF MORE # file designator, { THEN } line number
                           { GOTO }
```

ASCII DATA FILE INPUT/OUTPUT STATEMENT FORMATS

The formats of data file input/output statements are described below. All statements, excepting FILES (used for initial data file preparation), make use of a data "file designator", a numeric argument whose value is used to select the data file desired for current operation. The numeric argument may be an integer, variable (subscripted or unsubscripted) or an arithmetic expression. The file designator is always preceded by a pound sign (#).

File Preparation Statement

FILES

Purpose: To establish a relationship between numeric file designators and alphanumeric file names.

Format 1: FILES <filename 1,password;...;filename n,password>

Format 2: FILES <user-id/catalogname$password/.../filename$password, permissions>

Examples:

```
*10 FILES MONDAY;TUESDAY,PASS1
*10 FILES USERA/CAT1$PC/FIL1$PF1,R,W
```

Rules:

1. Semicolons are used as filename separators.
2. Filename passwords (if any) are separated from filenames by commas in Format 1 and by commas or dollar signs in Format 2. Where the slant (/) does not precede a password, a comma may be used.
3. An asterisk may be used in place of a filename, in which case the filename may be filled in via a FILE # statement (described below).
4. The filename of a data file must be referenced in a FILES statement before its first use within a program.
5. Multiple FILES statements are permissible within one program; one program is limited to eight named files.
6. Filenames may not be duplicated within a set of FILES statements for a given program.
7. For Format 2, there is a 3-level limitation of catalog structure on files to be accessed. To exceed this 3-level limitation, the ACCESS subsystem must be used. See "File Access" in this section.

Remarks: The FILES statement sets all named data files to read mode.

Format 1 limits the user to the ability of accessing files contained in his own master catalog. Format 2 permits the user to access files emanating from his own subcatalogs or from catalogs and subcatalogs belonging to another user. The user, of course, must know the other user's identification, catalog and file names, and any required passwords. General or specific permissions for files are established by the files originator. Legal permission combinations are:

READ
WRITE
APPEND
READ,WRITE
READ,APPEND

Additional examples of the use of Format 2 may prove helpful.

```
*10 FILES USER1/CAT1$PC1/CAT2/CAT3/FIL1$PF1,R,W
```

Three levels of catalog structure (the limit) are accessed to get to FIL1, another user's file. Read and write permissions for the file are requested.

```
*10.FILES FIL2;USERB/FIL3,R,W;FIL4,PW4
```

Three files are being accessed here. FIL2 and FIL4 are the user's own files. FIL3 is a file originated by a user identified as USERB. Read and write permissions are being requested for FIL3.

```
*10  FILES/CATU/FIL7;USERD/CATD$PW/FIL8,R,W
```

Two files are being accessed here. FIL7 is the user's own files located in his catalog CATU. FIL8 is a file originated by user USERD. Read and write permissions are being requested for FIL8.

*

File Preparation Statement

FILE #

Purpose: To permit replacement of a data file, or to permit specification of a data file indicated by an asterisk in a FILES statement.

Format: FILE # < file designator, "filename,password" >

Examples: 1.

```
*10 FILES A;B;C
      . .
      . .
      . .
*50 FILE #3 ,"D"
```

Data file C, the third file, is replaced by data file D.

2.

```
*10 FILES A;*;C
      . .
      . .
      . .
*50 FILE #2 ,"B"
```

The asterisk-indicated data file, the second file, is specified as data file B.

Rules:

1. The filename may be indicated as follows:
    a. filename and password (if any) enclosed in quotes
    b. string variables (subscripted or unsubscripted) for filename and password (if any)
    c. asterisk enclosed in quotes (see Remarks below)
2. A file named in a FILE # statement cannot appear in a FILES statement, unless the file has been released before its use in the FILE # statement.
3. One program is limited to eight named files.

Remarks: When a quote-enclosed asterisk is used as a "filename", the associated file designator is invalidated until such time that it is validated again by a subsequent FILE statement. For example:

```
*10 FILES A;B;C
      . .
      . .
      . .
*50 FILE #3 ,"*"
```

In statement 50, file designator 3 now refers to a null filename and cannot be used again until it is reset by another FILE # statement.

A colon (instead of a comma) may be used as the separator between file designator and "filename".

A string variable may be substituted for "filename" if the string variable contains the filename to be referenced. For example:

```
*10 FILES MONDAY;TUESDAY;WEDNESDAY
*20 LET A1$ = "SATURDAY"
*30 FILE #1,A1$
```

File Read Statement

READ #

Purpose: To read data from a data file into an input list.

Format: READ # <file designator, input list>

Example:
```
*10 FILES MONDAY;TUESDAY
*20 READ #1,X1,A1$,X2,A2$
```

If data file MONDAY is represented by

```
10 5.6, SEPTEMBER, 100.5, OCTOBER
```

at execution time, the real value of 5.6 would be read into X1, string SEPTEMBER into A1$, real value 100.5 into X2, and string OCTOBER into A2$.

Rules:

1. The input list must consist of delimiter-separated variables, numeric or string, any of which may be subscripted.
2. When an input list contains both numeric and string variables, data elements in the data file must correspond one-to-one to the input list.
3. If the file designator is zero, data will be read from internal data created by the program's DATA statement(s). For reading of internal data, there need not be a one-to-one correspondence between numeric and string variables in the input list and data file.
4. A colon may be used in the READ # statement instead of a comma to separate file designator from the input list.

Remarks: The line number of a data file is not part of the data read by a file read statement into an input list. At least one blank should separate the line number from data in the data file.

If an entire data file is not read because of insufficient variables in the input list of a file read statement, the word pointer will remain positioned after the last data item read until additional file read statement(s) are executed.

If the first character of an input string is a quote ("), the string must be terminated by a delimiter following the trailing quote. The resulting string consists of the characters enclosed by the quotes.

Data files to be read by the READ # statement require that elements of each data line be delimiter-separated. A delimiter may or may not end the line, the decision being left to the user.

File Read Statement

INPUT #

Purpose: To read data from a data file into an input list, treating line numbers as data items.

Format: INPUT # <file designator, input list>

Example:

```
*10 FILES MONDAY,TUESDAY
*20 INPUT #1,A,B,C,D,E
```

If data file MONDAY is represented by

```
  10 1,2,3,4,5
```

the statement

```
*30 PRINT A;B;C;D;E
```

would produce

```
  101 2 3 4 5
```

at program execution time.

Rules:

1. The input list must consist of comma-separated variables, numeric or string, any of which may be subscripted.
2. When an input list contains both numeric and string variables, data elements in the data file must correspond one-to-one to the input list.
3. A colon may be used in the INPUT statement instead of a comma to separate the file designator from the input list.
4. If the file designator is zero, at execution time the program will ask for data from the user's terminal. In response to a question mark, the user supplies data elements to correspond to the input list.

Remarks: Embedded blanks within a line number will cause misinterpretation in reading of a line number.

If the first character of an input string is a quote ("), the string must be terminated by a specified delimiter following the trailing quote. The resulting string consists of the characters enclosed by the quotes.

File Write Statement

WRITE #

Purpose: To generate a data file in which each line contains a line number and data elements delimiter-separated.

Format: WRITE # < file designator, output list >

Example:

```
*10 FILES SUNDAY; MONDAY; ABC
*20 READ #2, X1, A1$
*30 SCRATCH #3
*40 WRITE #3, X1, A1$
```

If data file MONDAY is represented by

```
10  5, OCTOBER, 1969
```

the WRITE # statement generates a new data file ABC with contents of

```
10  5, OCTOBER
```

Data file ABC may be a temporary or permanent file.

Rules:

1. The output list may consist of numeric or string variables (any of which may be subscripted), or arithmetic expressions.
2. The format conventions of the normal PRINT statement apply to the WRITE # statement.
3. If the file designator is zero, the generated data file will be written out to the user's terminal upon program execution, with no SCRATCH # statement required.
4. A colon may be used in the WRITE statement instead of a comma to separate the file designator from the output list.
5. The standard line length is equal to 75 characters, including line numbers. The MARGIN statement can be used to adjust a line from 2 to 132 characters.

Remarks: The WRITE # statement generates a data file that begins with line number 10 and increments by 10 for each additional line. Each line number is separated from the first data element of the line by at least one blank. Data elements, in turn, are separated by delimiters (commas or user-specified delimiters).

When the TAB(X) function is used, the line number is included in the count for the tab position.

A data file generated by a WRITE # statement is equivalent to a data file saved in the conventional manner; i.e., the file can serve as input to other subsystems (e.g., LIST).

File Write Statement

PRINT #

Purpose: To generate a data file which contains no line numbers or delimiters on printout.

Format: PRINT # < file designator, output list >

Example:

```
*10 FILES SUNDAY;MONDAY;ABC
*20 INPUT #2,X1,A1$
*30 SCRATCH #3
*40 PRINT #3,X1,A1$
```

If data file MONDAY is represented by

```
5,OCTOBER,1969
```

the PRINT # statement generates a new data file ABC with contents of

```
5    OCTOBER
```

Rules:

1. The output list may consist of numeric or string variables (any of which may be subscripted), arithmetic expressions, or string constants (literals) in quotes.
2. The format conventions of the normal PRINT statement apply to the PRINT # statement.
3. If the file designator is zero, the generated data file will be printed out on the user's terminal upon program execution, with no SCRATCH # statement required.
4. A colon may be used in the PRINT # statement instead of a comma to separate the file designator from the output list.
5. The standard line length is equal to 75 characters including line numbers. The MARGIN statement can be used to adjust a line from 2 to 132 characters.
6. No delimiters are created by the PRINT # statement.

Remarks: The PRINT # and WRITE # statements are utilized in similar fashions. The difference lies in the manner in which the generated data file is printed out. With the use of the PRINT statement, no line numbers or data element delimiters (commas or semicolons) appear.

A data file generated by a PRINT # statement can serve as input to other subsystems (e.g., LIST).

File Write Statement

PRINT # USING

Purpose: To provide the ability to format data written to a data file.

Format: PRINT # < file designator > , USING < statement number, output list >

where:

"statement number" is number of a statement in the program which contains format control characters and printable constants; "output list" consists of comma-separated arguments to be printed in sequential order.

Example:

```
*10 FILES FORMAT
*20  SCRATCH #1
*30  A = 123.45
*40  B = -3.456
*50  C = -.017
*60  PRINT #1,USING 80,A,B,C
*70  PRINT #1,USING 90,A,B,C
*80:DECIMAL  FIELDS  ###.##      ##.###      #.###
*90:EXPONENT FIELDS  ##.###↑↑↑↑  ##.###↑↑↑↑  ##.###↑↑↑↑
*100 END

* RUN

DECIMAL  FIELDS      123.45      -3.456      -.017
EXPONENT FIELDS      12.345E 01  -3.456E 00  -1.700E-02
```

Rules:

1. The statement number named in a PRINT # USING statement points to an "image" statement which formats the line to be printed. The image statement is of the form

   statement number: image

2. The image of an image statement (colon-separated from the statement number) consists of format control characters and printable constants.

3. Format control characters are as follows:

' (apostrophe) - a 1-character field that is filled with the first character in an alphanumeric string, regardless of string length.

(pound sign) - the replacement field for a numeric character; each # specifies a space for one digit.

↑↑↑↑ (four up-arrows) - specifies scientific notation for a numeric field (E-format).

4. Printable constants are all characters other than format control characters.

Remarks: The image of an image statement may consist of one or more of the following fields:

integer
decimal
exponential
alphanumeric
literal

Refer to "Formatting Line Output" in this Section for details concerning use of format control statement.

Data to be retrieved from a data file via READ # or INPUT # statements should not be placed on the file by a PRINT # USING statement. Data files containing data formatted by PRINT # USING statements are intended for terminal printout only by the way of the LIST command.

Matrix Input Statement

MAT READ #

Purpose: To read data from data file into a matrix input list.

Format: MAT READ # <file designator, matrix input list>

Example:

```
*10 FILES A;B
*20 DIM M1(3,3),M2(5,7)
*30 MAT READ #1,M1,M2
```

If data file A is represented by

```
10 1,2,3,.....,10,
 .    .
 .    .
 .    .
50 ........ 48,49,50,
```

M1 will contain the matrix

```
1  2  3
4  5  6
7  8  9
```

M2 will contain the matrix

```
10  11  12  13  14  15  16
17  18  19  20  21  22  23
24  25  26  27  28  29  30
31  32  33  34  35  36  37
38  39  40  41  42  43  44
```

Rules:

1. String variables may not be used in the matrix input list.

2. Matrices in the matrix input list may have their dimensions specified in a DIM statement or in the MAT READ # statement itself.

3. When a matrix in the matrix input list is not dimensioned, a 10 by 10 matrix is assumed.

4. Files to be read by a MAT READ # statement must contain line numbers.

5. A colon may be used in the MAT READ # statement instead of a comma to separate the file designator from the matrix input list.

Remarks: If the file designator is zero, internal data is to be read from user-supplied DATA statement(s) within the program.

If there are not enough data elements in a data file to fill a designated matrix, the matrix will be filled out with zeros.

Matrix Input Statement

MAT INPUT #

Purpose: To read data from a data file into a matrix input list, treating line numbers as data items.

Format: MAP INPUT # < file designator, matrix input list >

Example:

```
*10 FILES M1
*20 DIM M2(3,3)
*30 MAT INPUT #1,M2
```

If data file M1 contains

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

M2 will contain the matrix

```
101   2   3
  4   5   6
  7   8   9
```

Rules:

1. String variables may not be used in the matrix input list.
2. Matrices in the matrix input list may have their dimensions specified in a DIM statement or in the MAT INPUT # statement itself.
3. When a matrix in the matrix input list is not dimensioned, a 10 x 10 matrix is assumed.
4. A colon may be used in the MAT INPUT # statement instead of a comma to separate the file designator from the matrix input list.

Remarks: If the file designator is zero, at execution time the program will ask for data from the user's terminal. In response to a question mark, the user supplies data elements to correspond to the input list.

If there are not enough data elements in a data file to fill a designated matrix, the matrix will be filled out with zeros.

Matrix Output Statement

MAT WRITE #

Purpose: To write matrices specified in a matrix output list to designated data file(s).

Format: MAT WRITE #< file designator, matrix output list>

Example:

```
*10 FILES A;B;C
*20 DIM M1(3,3)M2(5,7)
*30 MAT READ #1,M1,M2
*40 SCRATCH #2
*50 MAT WRITE #2,M1,M2
```

Matrices M1 and M2, read from data file A, are written to data file B.

Rules:

1. String variables may not be used in the matrix output list.

2. Matrices in the matrix output list must have their dimensions specified in a DIM statement; they cannot be dimensioned in a MAT WRITE # statement.

3. A colon may be used in the MAT WRITE statement instead of a comma to separate the file designator from the matrix output list.

Remarks: The MAT WRITE # statement generates a data file that begins with line number 10 and increment by 10 for each additional line. Each line number is separated from the first data element of the line by a blank.

Matrix Output Statement

MAT PRINT #

Purpose: To write matrices specified in a matrix output list to a designated data file which contains no line numbers or delimiters on printout.

Format: MAT PRINT #< file designator, matrix output list>

Example:

```
*10 FILES M1,M2
*20 MAT INPUT #1,A(2,3)
*30 SCRATCH #2
*40 MAT PRINT #2,A
```

If data file M1 is represented by

1,2,3,4,5,6

The MAT PRINT # statement generates a new data file M2 which consists of

```
1    2    3
4    5    6
```

Rules:

1. String variables may not be used in the matrix output list.

2. Matrices in the matrix output list must have their dimensions specified in a DIM statement; they cannot be dimensioned in a MAT PRINT # statement.

3. A colon may be used in the MAT PRINT # statement instead of a comma to separate the file designator from the matrix output list.

Remarks: The MAT PRINT # and MAT WRITE # statements are utilized in similar fashions. With the use of the MAT PRINT # statement, no line numbers or data element delimiters appear.

A data file generated by a MAT PRINT # statement can serve as input to other subsystems (e.g., LIST).

If the file designator is zero, the generated data file will be printed out at the user's terminal upon program execution.

File Manipulation Statement

SCRATCH #

Purpose: To place a data file in write mode.

Format: SCRATCH # <file designator>

Example:

```
*10 FILES DEBITS;CREDITS
*20 READ #1,X1,X2,X3
*30 SCRATCH #2
*40 WRITE #2,X1,X2,X3
```

Data file CREDITS is placed in write mode by SCRATCH # statement 30, prior to being written on by WRITE # statement 40.

Remarks: A SCRATCH # statement deletes all data previously contained in the designated file; i.e., for files created by WRITE #, MAT WRITE #, or PRINT # statements.

If the data file CREDITS is a file not previously created and saved, the file system will query the user as to the disposition of the file.

File Manipulation Statement

RESTORE #

Purpose: To position the data pointer for the designated data file to the beginning of the file and permit the file to be read.

Format: RESTORE # <file designator>

Examples: 1.

```
*10 FILES A;B;C
*20 READ #1,X1,X2,X3
*30 RESTORE #1
*40 READ #1,Y1,Y2,Y3
```

RESTORE # statement 30 permits data from data file A to be read again.

2.

```
*10 FILES A;B;C
*20 READ #1,X1,X2,X3
  .     .
  .     .
*50 SCRATCH #1
*60 WRITE #1,Y1,Y2,Y3
*70 RESTORE #1
*80 READ #1,X1,X2,X3
```

RESTORE # statement 70 places data file A in read mode and permits data just written to be read.

Remarks: If a designated data file is in write mode as the result of a SCRATCH # statement, a RESTORE # statement repositions the data pointer to the beginning of the file and places the file in read mode.

File Manipulation Statement

BACKSPACE #

Purpose: To position the data pointer for the designated data file backward one delimiter.

Format: BACKSPACE # <file designator>

Example: If data file A contains

```
10 1,2,3,4,5,

20 6,7,8,9,10,
```

The program

```
*10 FILES A;B;C
*20 READ #1,X1,X2,X3,X4,X5,X6,X7
*30 FOR I = 1 to 4
*40 BACKSPACE #1
*50 NEXT I
*60 READ #1,Y1,Y2,Y3,Y4
*70 PRINT X1,X2,X3,X4,X5,X6,X7
*80 PRINT Y1,Y2,Y3,Y4
*90 END
*RUN
```

will produce

```
1 2 3 4 5 6 7

4 5 6 7
```

Remarks: The BACKSPACE # statement places the designated file in read mode.

If the designated file is backspaced past the beginning of the file, the data pointer will be positioned to the beginning of the file.

Utility Statement

APPEND

Purpose: To permit data to be added to a designated file.

Format: APPEND # <file designator>

Example:

```
*10 FILES A;B;C
*20 READ #1,X1,X2,A1$
*30 APPEND #2
*40 WRITE #2,X1,X2,A1$
```

APPEND # statement 30 places data file B in write mode and permits WRITE # statement 40 to append data to data already on B.

Remarks: When the APPEND # statement is executed, the data pointer for the designated file will be moved immediately past the last data item on the file. The file is also placed in write mode, ready to accept the next WRITE # statement.

Utility Statement

MARGIN #

Purpose: To permit the specification of the rightmost character position for a designated file.

Format: MARGIN # <file designator, expression>

Example:

```
*10 FILES A;B;C
*20 SCRATCH #1
*30 SCRATCH #2
*40 MARGIN #1,20
*50 MARGIN #2,M*N-5
*60 WRITE #1,X1,X2,X3,X4
*70 WRITE #2,Y1,Y2,Y3,Y4
```

Rules:

1. The standard line (record) length for files created by WRITE # or PRINT # statements is 75 characters, including the line number. By use of the MARGIN # statement, the user may explicitly specify a maximum line length for a designated file to be any value between 2 and 160 characters. If the specified line length exceeds the physical capability of the terminal in use, the result may be a character overprint at the end of the line.

2. A colon may be used in the MARGIN # statement instead of a comma to separate the file designator from the expression.

3. A file designator of zero is interpreted as being the user's terminal.

Utility Statement

DELIMIT #

Purpose: To permit the use of a delimiter other than a comma in a designated file.

Format: DELIMIT # < file designator, $\left\{ \begin{array}{l} \text{(character)} \\ \text{(abbreviation)} \end{array} \right\}$ >

Example:

```
*10 FILES INPUT;OUTPUT
*20 READ #1,A,B,C,D,E,F
*30 DELIMIT #2,(LF)
*40 SCRATCH #2
*50 WRITE #2,A;B;C;D;E;F
```

If data file INPUT contains

```
10 1,2,3,4,5,6
```

a printout of data file OUTPUT would produce

```
10 1
      2
         3
            4
               5
                  6
```

Rules:

1. The standard delimiter separating data elements in a data file is the comma. The DELIMIT # statement may specify any character, or abbreviation for non-printing character(s).

2. Non-printing character abbreviations (e.g.,CR for carriage return; LF for line feed) are those specified by USASCII. Refer to Appendix C for a list of octal/USASCII conversion equivalents.

3. A DELIMIT # statement must be used prior to its associated READ # or WRITE # statement.

Remarks: A PRINT # statement will result in the printout of designated data without delimiters (or line numbers) regardless of whether standard or non- standard delimiters are used.

Utility Statement

IF END #----THEN
or
IF END #----GOTO

Purpose: To provide for a means of testing for the end of data when reading a data file.

Format: IF END # <file designator> {THEN | GOTO} <statement number>

Example:

```
*10 FILES A;B
*20 READ #1,X1,X2,X3
*30 PRINT X1,X2,X3,
*40 IF END #1 THEN 60
*50 GOTO 20
*60 PRINT "OUT OF DATA IN FILE A"
*70 END
*RUN
```

If data file A contains

```
10 1,2,3,4,5,6,7,

20 8,9,10,
```

the executed program will produce

```
1 2 3 4 5 6 7 8 9 10 0 0

OUT OF DATA IN FILE A
```

Rules: A comma or a colon may be used in an IF END #---THEN statement to separate the file designator from the THEN portion of the statement.

Remarks: If data elements (or string data) of a data file are exhausted before the input list in a READ # or MAT READ # statement is satisfied, the list will be filled out by zeros (or null) upon program execution.

The IF END #---THEN statement directs the system to go to a designated out-of-sequence statement when no more data remains on the file.

Utility Statement

IF MORE #---THEN
or
IF MORE #---GOTO

Purpose: To provide for a means of testing to determine whether at least one valid data element remains on a data file when reading the file.

Format: IF MORE # <file designator> {THEN / GOTO} <statement number>

Example:

```
*10 FILES A;B
*20 READ #1,X1,X2,X3
*30 PRINT X1,X2,X3,
*40 IF MORE #1 THEN 20
*50 PRINT "OUT OF DATA IN FILE A"
*60 END
*RUN
```

If data file A contains

```
10 1,2,3,4,5,6,7,

20 8,9,10,
```

the executed program will produce

```
1 2 3 4 5 6 7 8 9 10 0 0

OUT OF DATA IN FILE A
```

Rule: A comma or a colon may be used in an IF MORE # ---THEN statement to separate the file designator from the THEN portion of the statement.

Remarks: If data elements (or string data) of a data file are exhausted before the input list of a READ # or MAT READ # statement is satisfied, the list will be filled out by zeros (or null) upon program execution.

The IF MORE #---THEN statement directs the system to go to a designated out-of-sequence statement when more data remains on the file.

BINARY FILES

BASIC permits the user to perform file input/output with files made up in binary format. This mode of operation presupposes a sophisticated user whose knowledge encompasses the makeup of binary-type files and who has the need to create programs that have special applications.

The use of binary input/output, as contrasted with the use of alphanumeric (ASCII) input/output, provides the user with advantages in program execution speed and file space compactness. However, data cannot be placed on a binary file directly from the user's terminal, nor can a binary file be listed (by means of the LIST command) so as to verify its content.

Binary files can be either sequential or random and can be written, read, backspaced, scratched, and restored. Data can be appended to the end of a sequential binary file. Any word on a random binary file is accessible for reading or writing without the need for traversing the file space which precedes the word. When a random binary file is to be created, file space must be obtained by means of the ACCESS subsystem (see "File Access" below).

A word pointer is maintained in the file control block of each binary file so as to indicate the next word of the file to be read or written. Each binary file consists of a number of words, zero through n-1. For sequential files, the word pointer is initially set to word zero and moved forward with each READ: and WRITE: statement. The word pointer can be moved backward by means of the RESTORE:, SCRATCH:, and BACKSPACE: statements. This same forward and backward movement of the word pointer through statement manipulation exists for random files, with the exception that the user can alter the position of the word pointer by means of an additional statement--SET:. If the user wishes to begin reading and writing of a random file at a position other than word zero, he can position the word pointer to any position within the file with the SET: statement and begin his reading or writing at that point. The current position of the word pointer for a random file and the current length of a random file can be determined by use of functions LOC and LOF.

Each numeric data element on a binary file occupies one word and is in single-precision, floating-point format. Alphanumeric strings, which may vary in length from 1 to 132 characters, are placed on binary files with a string control word on either end of the string. Each string will thus occupy two words for control, plus enough words to contain the actual string of characters at four characters per word. The user must exercise caution in manipulating the word pointer on random binary files containing strings. A SET: statement could inadvertently position the word pointer to the middle of a string, causing an error in the next read or write. The user must take care to position the word pointer to a leading string control word and see to it that extended strings do not destroy data already on a file.

All sequential files will initially be in read mode. A file can be placed in write mode by the use of SCRATCH: statement. Read mode may be re-established by the use of the RESTORE: statement. Read/write mode does not apply to random files, which may be read or written at any point at any time.

Binary files are implemented by binary file input/output statements which supplement BASIC language statements. These binary file input/output statements are categorized as follows and, unless indicated, apply to both sequential and random binary files:

- File preparation statements

  FILES filename 1,password;....;filename n,password

  FILES user-id/catalogname$password/.../
  filename$password,permissions

  FILE: file designator, "filename,password"

- File read statement

  READ: file designator,input list

- File write statement

  WRITE: file designator,output list

- Matrix input statement

  MAT READ: file designator,matrix input list

- Matrix output statement

  MAT WRITE: file designator,matrix output list

- File manipulation statements

  SCRATCH: file designator

  RESTORE: file designator

  BACKSPACE: file designator

- Utility statements

  APPEND: file designator
  (for sequential files only)

  IF END: file designator $\left\{\begin{array}{l}\text{THEN}\\\text{GOTO}\end{array}\right\}$ line number

  IF MORE: file designator $\left\{\begin{array}{l}\text{THEN}\\\text{GOTO}\end{array}\right\}$ line number

  SET: file designator TO expression
  (for random files only)

The current position of the word pointer for a random binary file or its current length can be determined by the use of special functions. These functions are as follows:

- Word pointer location

  LOC(file designator)

- File length

  LOF(file designator)

Upon program execution, these functions contained within a program cause the printout of integers, indicating the desired word numbers.

Note

For all practical purposes, the IF END: and IF MORE: statements are applicable to sequential files only. Random files have no logical end-of-data; the entire random file supposedly contains good data and is accessible at any point for reading and writing. Thus, if a random file has a current size of three blocks (960 words) and has data written in only the first 10 words, the IF END: and IF MORE: statements cannot be used to determine when the end of the first 10 words has been reached. The remaining 950 words are accessible data despite the fact that they are empty.

BINARY FILE INPUT/OUTPUT STATEMENT FORMATS

The formats of binary file input/output statements are described below. All statements, excepting FILES (used for initial binary file preparation) make use of a "file designator", a numeric argument whose value is used to select the binary file desired for current operation. The numeric argument may be an integer, a variable (subscripted or unsubscripted), or an arithmetic expression. The file designator is always preceded by a colon.

File Preparation Statement

FILES

Purpose: To establish a relationship between numeric file designators and alphanumeric file names.

Format 1: FILES < filename 1,password;...;filename n,password >

Format 2: FILES < user-id/catalogname$password/.../ filename$password,permissions >

Examples: *10 FILES MONDAY;TUESDAY,PASS1

*10 FILES USERA/CAT1$PC/FIL1$PF1,R,W

Rules:

1. Semicolons are used as filename separators.

2. Filename passwords (if any) are separated from filenames by commas in Format 1 and by commas or dollar signs in Format 2. Where the slant (/) does not precede a password, a comma may be used.

3. An asterisk may be used in place of a filename, in which case the filename may be filled in via a FILE: statement (described below).

4. The filename must be referenced in a FILES statement before its first use within a program.

5. Multiple FILES statements are permissible within one program; one program is limited to eight named files.

6. Filenames may not be duplicated within a set of FILES statements for a given program.

7. For Format 2, there is a 3-level limitation of catalog structure on files to be accessed. To exceed this 3-level limitation, the ACCESS subsystem must be used. See "File Access" in this section.

Remarks: The FILES statement sets all named sequential binary and ASCII files to read mode.

Format 1 limits the user to the ability of accessing files contained in his own master catalog. Format 2 permits the user to access files emanating from his own subcatalogs or from catalogs and subcatalogs belonging to another user. The user, of course, must know the other user's identification, catalog and file names, and any required passwords. General or specific permissions for files are established by the files' originator. Legal permission combinations are:

Read
Write
Append
Read,Write
Read, Append

Additional examples of the use of Format 2 may prove helpful.

```
*10 FILES USER1/CAT1$PC1/CAT2/CAT3/FIL1$PF1,R,W
```

Three levels of catalog structure (the limit) are accessed to get to FIL1, another user's file. Read and write permissions for the file are requested.

```
*10 FILES FIL2;USERB/FIL3,R,W;FIL4,PW4
```

Three files are being accessed here. FIL2 and FIL4 are the user's own files. FIL3 is a file originated by a user identified as USERB. Read and write permissions are being requested for FIL3.

```
*10 FILES/CATU/FIL7;USERD/CATD$PW/FIL8,R,W
```

Two files are being accessed here. FIL7 is the user's own files located in his catalog CATU. FIL8 is a file originated by user USERD. Read and write permissions are being requested for FIL8.

File Preparation Statement

FILE:

Purpose: To permit replacement of a binary file by another binary filename, or to permit specification of a binary file indicated by an asterisk in a FILES statement.

Format: FILE: < file designator, "filename,password" >

Examples: 1.

```
*10 FILES A;B;C
      . .
      . .
      . .
*50 FILE: 3,"D"
```

Binary file C, the third file, is replaced by binary file D.

2.

```
*10 FILES A;*;C
      . .
      . .
      . .
*50 FILE: 2,"B"
```

The asterisk-indicated binary file, the second file, is specified as binary file B.

Rules:

1. The filename may be indicated as follows:

   a. filename and password (if any) enclosed in quotes

   b. string variables (subscripted or unsubscripted) for filename and password (if any)

   c. asterisk enclosed in quotes (see Remarks below)

2. A file named in a FILE: statement cannot appear in a FILES statement, unless the file has been released before its use in the FILE: statement.

3. One program is limited to eight named files.

Remarks: When a quote-enclosed asterisk is used as a "filename", the associated file designator is invalidated until such time that it is validated again by a subsequent FILE statement. For example:

```
*10 FILES A;B;C
      . .
      . .
      . .
*50 FILE: 3,"*"
```

In statement 50, file designator 3 now refers to a null filename and cannot be used again until it is reset by another FILE: statement.

A colon (instead of a comma) may be used as the separator between file designator and "filename".

A string variable may be substituted for "filename" if the string variable contains the filename to be referenced. For example:

```
*10 FILES MONDAY;TUESDAY;WEDNESDAY
*20 LET A1$ = "SATURDAY"
*30 FILE: 1,A1$
```

File Read Statement

READ:

Purpose: To read binary data from a permanent binary file into an input list.

Format: READ: <file designator, input list>

Example: The binary file SUNS contains a list of the names of basketball players, with each player's score average following his name. The beginning of the file (the first three names) could appear as follows:

Data	Word	Octal Representation
Control word	0	001600000700
HAWK	1	110101127113
INS	2	111116123040
Control word	3	001400000700
30	4	012740000000
Control word	5	001600000400
WALK	6	127101114113
Control word	7	001400000440
20	8	012500000000
Control word	9	001600001000
GOOD	10	107117117104
RICH	11	122111103110
Control word	12	001400001000
25	13	012620000000

The following program will produce the names of the first three players and their score averages.

```
*10 FILES SUNS
*20 FOR I = 1 to 3
*30 READ:1,N$,S
*40 PRINT USING 60,N$,S
*50 NEXT I
*60:'LLLLLLLLLL    ###
*70 PRINT
*80 PRINT "MORE TO COME"
*90 END
*RUN

HAWKINS         30
WALK            20
GOODRICH        25

MORE TO COME
```

Rules:

1. The input list must consist of delimiter-separated variables, numeric or string, any of which may be subscripted.

2. When an input list contains both numeric and string variables, data elements in the binary file must correspond one-to-one to the input list.

3. A colon may be used, instead of a comma, to separate the file designator from the input list.

Remarks: If an entire binary file is not read because of insufficient variables in the input list file read statement, the word pointer will remain positioned after the last data item read until additional file read statement(s) are executed.

File Write Statement

WRITE:

Purpose: To write binary data on a permanent binary file.

Format: WRITE: <file designator,output list>

Example:

```
*10 FILES PHX1
*20 H1 = H2 = 5
*30 H3 = 6\H4 = 6.2
*40 S1$="BINARY"
*50 S2$="DATA"
*60 SCRATCH:1
*70 WRITE:1,H1,H2,H3,H4,S1$,S2$
*80 END
```

Upon program execution, the following data would be placed in binary file PHX1.

Data	Word	Octal Representation
5	0	006500000000
5	1	006500000000
6	2	006600000000
6.2	3	006614631463
Control word	4	001600000600
BINA	5	102111116101
RY	6	122131040040
Control word	7	001400000600
Control word	8	001600000400
DATA	9	104101124101
Control word	10	001400000400

The file's word pointer would be at word 11 of the file.

Rules:

1. The output list may consist of numeric or string variables (any of which may be subscripted), or arithmetic expressions.
2. The format conventions of the normal PRINT statement apply to the WRITE: statement.
3. A colon may be used in the WRITE: statement instead of a comma to separate the file designator from the output list.

Remarks: The word pointer for the referenced binary file is incremented by one after each word is written on the file.

Matrix Input Statement

MAT READ:

Purpose: To read data from permanent binary file into a matrix input list.

Format: MAT READ: <file designator,matrix input list >

Example: Assume that binary file INTEGERS contains the numbers 0 through 10 in its first 11 words. The following program can be used to read data from file INTEGERS into a matrix input list.

```
*10 FILES*;INTEGERS
*20 DIM M8(6)
*30 READ:2,N1,N2
*40 MAT READ:2,M8
*50 MAT PRINT M8
*60 END
```

Upon execution, the program would produce the following printout:

```
2

3

4

5

6

7
```

Rules:

1. String variables may not be used in the matrix input list.
2. Matrices in the matrix input list must have their dimensions specified in a DIM statement or in the MAT READ: statement itself.
3. When a matrix in a matrix input list is not dimensioned, a 10 by 10 matrix is assumed.
4. A colon may be used, instead of a comma, to separate the file designator from the matrix input list.

Remarks: If there are not enough data elements in a binary file to fill a designated matrix, the matrix will be filled out with zeros.

Matrix Output Statement

MAT WRITE:

Purpose: To write matrices specified in a matrix output list to designated permanent binary file.

Format: MAT WRITE: <file designator,matrix output list>

Example: Assume that binary file ABCD has been created via ACCESS as a random file. The following program can be used to write a matrix output list to file ABCD.

```
*10 FILES ABCD
*20 DIM T(2,3)
*30 T(1,1)=1\T(1,2)=2\T(1,3)=3
*40 T(2,1)=4\T(2,2)=5\T(2,3)=6
*50 SCRATCH:1
*60 SET:1 TO 4
*70 MAT WRITE:1,T
*80 END
```

Statement 60 could not be used if ABCD was not random.

Upon execution, file ABCD will contain matrix T as follows:

Data	Word	Octal Representation
	0	400000000000
	1	400000000000
	2	400000000000
	3	400000000000
1	4	002400000000
2	5	004400000000
3	6	004600000000
4	7	006400000000
5	8	006500000000
6	9	006600000000

Rules:

1. String variables may not be used in the matrix output list.

2. Matrices in the matrix output list must have their dimensions specified in a DIM statement; they cannot be dimensioned in a MAT WRITE: statement.

3. When a matrix in the matrix output list is not dimensioned, a 10 by 10 matrix is assumed.

4. A colon may be used, instead of a comma, to separate the file designator from the matrix output list.

File Manipulation Statement

SCRATCH:

Purpose: To place a binary file in write mode.

Format: SCRATCH: <file designator >

Example:

```
*10 FILES ABC;XYZ
*20 READ:1,X1,X2,X3
*30 SCRATCH:1
*40 WRITE:1,X1,X2,X3
```

Binary file ABC is placed in write mode by SCRATCH: statement 30, prior to being written on by WRITE: statement 40.

Remarks: A SCRATCH: statement deletes all data previously contained in the designated file; i.e., data written by WRITE: or MAT WRITE: statements.

The SCRATCH: statement can be used with both sequential and random binary files. For sequential files, the word printer is set to zero and the file is placed in write mode. For random files, the entire file is filled with floating point zeros and the word pointer is set to zero. The read/write mode does not apply to random file; therefore, the SCRATCH: statement need not be utilized with a random file unless the user wishes to clear the entire random file to zeros.

File Manipulation Statement

RESTORE:

Purpose: To position the word pointer for the designated binary file to the beginning of the file and permit the file to be read.

Format: RESTORE: <file designator>

Example:

```
*10 FILES HUGO
*20 R1=8.8
*30 R2=9.9
*40 R3=10.10
*50 R1$="THIS LINE SHOULD APPEAR TWICE"
*60 SCRATCH:1
*70 WRITE:1,R1,R2,R3,R1$
*80 RESTORE:1
*90 READ:1,S1,S2,S3,S1$
*100 PRINT R1$;R1;R2;R3
*110 PRINT S1$;S1;S2;S3
*120 END
*RUN
```

will produce the printout

```
THIS LINE SHOULD APPEAR TWICE  8.8  9.9  10.1
THIS LINE SHOULD APPEAR TWICE  8.8  9.9  10.1
```

RESTORE: (statement 80) places binary file HUGO in read mode and permits data just written to be read.

Remarks: If a designated binary file is in write mode as a result of a SCRATCH: statement, a RESTORE: statement repositions the word pointer to the beginning of the file. The file is placed in read mode if it is sequential.

File Manipulation Statement

BACKSPACE:

Purpose: To position the word pointer for the designated binary file backward one data element.

Format: BACKSPACE: <file designator>

Example:

```
*10 FILES HIPPO
*20 A1=1\A2=2\A3=3
*30 E1$="IS A"
*40 E2$=" CROWD"
*50 SCRATCH:1
*60 WRITE:1,A1,A2,A3,E1$,E2$
*70 FOR J=1 TO 3
*80 BACKSPACE:1
*90 NEXT J
*100 READ:1,B3,G1$,G2$
*110 PRINT B3;G1$;G2$
*120 END
*RUN
```

will produce the printout

```
3 IS A CROWD
```

Remarks: The BACKSPACE: statement places the designated binary file in read mode if the file is sequential. If the designated binary file is backspaced past the beginning of the file, the word pointer will be positioned to the beginning of the file.

Utility Statement

APPEND:

Purpose: To permit data to be added to a designated, sequential binary file.

Format: APPEND: <file designator>

Example: Assume that the binary file SEE is a sequential file containing the integers 1 through 15.

```
*10 FILES A;B;SEE
*20 APPEND:3
*30 FOR I=16 TO 20
*40 WRITE:3,I
*50 NEXT I
*60 END
*RUN
```

The executed program will append the integers 16 through 20 to the file SEE.

Rules: The APPEND: statement applies to sequential files only.

Remarks: The APPEND: statement will set the word pointer for the designated file to the position immediately following the last data word. The file is then placed in write mode, ready to accept the next WRITE: statement.

Utility Statement

IF END:----THEN
or
IF END:----GOTO

Purpose: To provide a means of testing for end of data when reading a binary file.

Format: IF END: <file designator> {THEN | GOTO} <statement number>

Example:

```
*10 FILES ZORRO
*20 K1=1
*30 A$="EACH STRING "
*40 B$="HAS A "
*50 C$="LEADING AND TRAILING "
*60 D$="CONTROL "
*70 E$="WORD"
*80 SCRATCH:1
*90 WRITE:1,A$,B$,C$,D$,E$
*100 RESTORE:1
*110 IF END:1 THEN 150
*120 READ:1,V$
*130 PRINT V$;
*140 GOTO 110
*150 END
*RUN
```

The executed program will produce the printout

```
EACH STRING HAS A LEADING AND TRAILING CONTROL WORD
```

Rules: A comma or a colon may be used in an IF END:---THEN statement to separate the file designator from the THEN portion of the statement.

Remarks: The IF END:---THEN statement directs the system to go to a designated out-of-sequence statement when no more data remains on the file.

Utility Statement

IF MORE:----THEN
or
IF MORE:----GOTO

Purpose: To provide for a means of testing to determine whether at least one valid data element remains on a binary file when reading the file.

Format: IF MORE: <file designator> {THEN | GOTO} < statement number>

Example:

```
*10 FILES ZORRO
*20 K1=1
*30 A$="EACH STRING "
*40 B$="HAS A "
*50 C$="LEADING AND TRAILING "
*60 D$="CONTROL "
*70 E$="WORD"
*80 SCRATCH:1
*90 WRITE:1,A$,B$,C$,D$,E$
*100 RESTORE:1
*110 READ:1,V$
*120 PRINT V$;
*130 IF MORE:1 THEN 110
*140 END
*RUN
```

The executed program will produce the printout

```
EACH STRING HAS A LEADING AND TRAILING CONTROL WORD
```

Rules: A comma or a colon may be used in an IF MORE:---THEN statement to separate the file designator from the THEN portion of the statement.

Remarks: If data elements (or string data) of a binary file are exhausted before input list of a READ: or MAT READ: statement is satisfied, the list will be filled out by zeros upon program execution.

The IF MORE:---THEN statement directs the system to go to a designated out-of-sequence statement when more data remains to be read on the file.

Utility Statement

SET:

Purpose: To permit the word pointer for a random binary file to be positioned so that data can be read or written at any point on the file.

Format: SET: <file designator> TO <expression>

Example: Assume random binary file ORKIN is created via the ACCESS system and its size is three blocks (3 x 320 = 960 words).

```
*10 FILES ORKIN
*20 SET:1 TO 620
*30 FOR P=1 TO 36
*40 WRITE: 1,P
*50 NEXT P
*70 FOR K=655 TO 620 STEP -1
*80 SET:1 TO K
*90 READ:1,N
*100 PRINT N;
*120 NEXT K
*130 END
*RUN
```

Upon execution, the program will write the integers 1 through 36 on file ORKIN, beginning at word 620 and ending at word 655. In addition, the contents of words 620 through 655 are verified and the integers (in reverse order) are printed out as follows:

```
36  35  34  33  32  31  30  29  28  27  26  25  24  23  22

21  20  19  18  17  16  15  14  13  12  11  10   9   8   7

 6   5   4   3   2   1
```

Rules: The SET: statement applies to random binary files only.

Remarks: The expression in the SET: statement is evaluated and its integer portion, if greater than or equal to zero, stored in the word pointer of the designated file. If the integer portion is negative, an explanatory error message and program termination result.

The text previously contained on this page has been moved to page 5-10. *

* The text previously contained on this page has been moved to page 5-10.

MULTIPLE STATEMENTS WITHIN ONE LINE

While each statement of a program must be confined to a single line, the user may make multiple statements within a single line, utilizing one line number. Statements within a line are separated by means of a reverse slant (\). For example, the line

```
*10 A=12\B=37\C=SQR(A+B)\PRINT A,B,C
```

is equivalent to four statements and is identified by line number 10.

If a multiple-statement line is used in a program employing loops or transfers, a transfer can only be made to the first of the multiple statements. For example,

```
* 10 LET N=0
* 20 READ X,Y,Z\PRINT X,Y,Z\N=N+1\RESTORE
* 30 IF N < 5 THEN 20\DATA 1,2,3
* 40 END
```

SAVING TEMPORARY FILES

When the user terminates his session at the terminal with a log-off sequence, the system is scanned for the user's temporary files. The message

n TEMPORARY FILES CREATED

is issued, n being the number of files. Each temporary file name is listed, followed by a question mark. The user may respond as follows:

1. carriage return -- implies that this file is to be released; pass to the next file if more temporary files exist.
2. NONE -- implies this and all succeeding files are to be released.
3. SAVE filename -- specifies that this file is to be saved as one of the user's permanent files; pass to the next file if more temporary files exist.

SAVING AND EXECUTING OBJECT FILES

The RUN command may be used to save a file in its object (binary) code form and/or execute a program with such a file. Basic forms of the RUN command to achieve these purposes are as follows:

1. RUN = objfile

   The user's current file will be compiled and saved as an object file on random file objfile.

2. RUN = catalog/objfile

   Same as item 1 except that catalog/filename structure is used.

3. RUN objfile

   The contents of random file objfile are loaded into memory and executed. Compilation has already been performed.

4. RUN catalog/objfile

   Same as item 3 except that catalog/filename structure is used.

5. RUN filename = objfile

   The file filename will be compiled, saved as an object file on random file objfile, and executed.

6. RUN filename = objfile (NO GO)

   The file filename will be compiled and saved as an object file on random file objfile. No execution will take place if (NO GO) option is utilized.

For example,

    RUN JDOE/RACE,R = MYFILE

will compile file RACE; RACE will then be saved as object file MYFILE and MYFILE will be executed.

    RUN MYFILE

will execute object file MYFILE.

If a catalog/filename structure is used, a maximum of three levels is permitted. Legal permission combinations for the catalog/filename structure are:

Read
Write
Append
Execute
Read,Write
Read, Append

The user should note that, as a general rule, object programs are not transferable from software release to software release.

FILE ACCESS

For the normal time-sharing user, all files (programs) will be defined by his user identification and a unique file name for each of his set of files. Since the user identification which was given to the time-sharing system on the log-on procedure, and the file name (OLD program name), completely define the file for a normal situation, the time-sharing system will automatically give the user access to his own files stored by use of the SAVE control command. However, if the user wishes to make use of other files (for instance, those saved by another user), it is necessary for him to previously have accessed these files. One method of accessing other users files is by a time-sharing subsystem called ACCESS. This subsystem will allow the time-sharing user to access files that have been saved by others, or that have been stored in the file system by means other than the control command SAVE (e.g., batch-world files), and to place these files at his disposal for a session at the terminal. If this feature is required, the user must select ACCESS before he goes to the BASIC system. The ACCESS subsystem is described in Time-Sharing System General Information Manual.

An alternate method of accessing other users files is by means of the control command GET; refer to "Control Commands" in Section II for a description of the command.

SECTION VI

ERROR MESSAGES

ERROR MESSAGES AS A RESULT OF ERRORS OF FORM

One or more of the following diagnostic error messages are printed at the terminal whenever an error of form occurs; that is, whenever a BASIC language rule is violated. The messages will be printed after the control command RUN is given.

There are two groups of error messages:

Compilation - these may be printed during program compilation and prevent further entry or execution.

Execution - these may be printed during program execution and may or may not stop execution.

Compilation Errors

Note

When a subroutine referenced by a CALL statement, or a CHAIN link, is being compiled and an error is detected, the message

IN FILE filename

will follow the error message.

Error Message	Interpretation
BAD SOURCE FILE	Attempt made to read source program having invalid format.
FOR WITHOUT NEXT	Missing NEXT statement.
ILLEGAL FOR LOOP	Statement sequence cannot perform requested loop.
ILLEGAL TERMINATION OF XXXX	Statement XXXX has not been terminated correctly.

IN XXXX DEF STATEMENT TABLE EXCEEDED	More than 26 DEF statements being utilized in program.
IN XXXX DUP DIMENSION	Variable appears in DIM statement more than once.
IN XXXX DUP FUNCTION	Same function name defined more than once.
IN XXXX NAME NOT DECLARED IN DIM	Dimensioned variable not previously defined in DIM statement.
IN XXXX NEXT WITHOUT FOR	Missing FOR statement.
IN XXXX SYNTAX ERROR IN FILE DESCR	An error has been detected in formatting of FILES statement; causes most likely: 1. Filename greater than eight characters. 2. User-id, catalog name or password greater than 12 characters. 3. Termination of a file descriptor string with a delimiter. 4. Illegal permission. 5. Greater than 3-level catalog file description.
NULL SOURCE FILE	Attempt made to compile nonexistent current program.
↑ STATEMENT ERROR	This message encompasses the majority of errors that may occur in the formatting of a statement. The arrow points to the portion of the statement containing the error. Study this portion of the message to determine cause of the error.
SYSTEM FAULT BASIC IN XXXX	Self-explanatory.
NO LINE NUMBER XXXX	Statement XXXX appearing in a GOTO, GOSUB, or IF----THEN statement does not appear as a line number in the program.

Execution Errors

Error Message	Interpretation
BLOCK SERIAL NUMBER ERROR AT XXXX	The file being read was out of position - system malfunction.
DUPLICATE FILE NAME XXXX	The program already has a file whose name is a duplicate of the one specified by a FILE statement. A program cannot have two file designators referring to the same file name.
EXPECTED LINE NUMBER AT XXXX	A sequence number could not be found on a line where one was expected.

FATAL ERROR IN FILE I/O AT XXXX	A fatal error occurred in a read, write, backspace, or forward space sequence. This is a system malfunction.
FILE CLASS ERROR AT XXXX	Using mixture of ASCII and binary I/O operations on same file.
FILE DESIGNATOR NOT BETWEEN ONE AND EIGHT AT XXXX	There is a limit of eight active files per program. Therefore a file designator must fall between 0 and 8 inclusively, where 0 represents the file being processed at the user's terminal.
FILE NOT ASCII AT XXXX	An attempt was made to read a non-ASCII file. Only formatted ASCII files may be read.
FILE NOT BINARY AT XXXX	Attempting to perform binary I/O operation on ASCII file.
FILE NOT DEFINED BY FILES OR FILE STATEMENT AT XXXX	A file designator points to a vacant slot in the program's possible list of files. EXAMPLE: READ #6,X where file #6 has not been named by a FILES or FILE statement.
FILE NOT IN WRITE MODE AT XXXX	An attempt was made to transmit output to a file while the file was in input mode. Use a SCRATCH statement before output.
FILE NOT IN READ MODE AT XXXX	An attempt was made to transmit data from a file while the file was in output mode. Use a RESTORE statement before reading from a file which was previously in output mode.
FILE NOT RANDOM AT XXXX	Attempting to use SET, LOF, or LOC with ASCII or sequential binary file.

*

FILE NOT SEQUENTIAL BINARY AT XXXX	Attempting to use APPEND: statement with ASCII or random file.
ILLEGAL ARG IN XXXX	Argument in VAL function not a valid constant.
ILLEGAL ENTRY TO SUBROUTINE	A RETURN statement was executed without a corresponding GOSUB or CALL, or more than 15 GOSUB or CALL statements were executed without corresponding RETURN statements.
ILLEGAL FIELD IN XXXX	In an image statement for PRINT USING, there were too few places specified for the left of a decimal point. A special character such as a dollar sign ($) or minus sign (-) was needed and there was no space, or a scientific notation (E-type) did not specify at least one place to the left of the decimal as required.
ILLEGAL INPUT AT XXXX	Data read from a file did not correspond in type to that expected by the source statement I/O list. The I/O list and data to be read must be compatible.
* ILLEGAL INPUT FORMAT. RETYPE?	Invalid response has been given to INPUT statement request. Execution stops.
ILLEGAL MARGIN AT XXXX	The argument of a MARGIN statement was not between the allowable limits of 2 to 160 characters, inclusively.
INCONSISTENT FIELD IN XXXX	In PRINT USING statement XXXX, the I/O list and fields specified by the image statement do not agree in type and/or number. Check the I/O list and image statement for compatibility.
INCORRECT FILE FORMAT AT XXXX	A line of the file being read is in incorrect format. The most probable error is an incorrectly formatted sequence number for the line.

INVALID CHANGE IN XXXX	An invalid CHANGE statement has been executed at statement number XXXX. The string is longer than the array, or the number of array entries to be converted to characters is less than 1 or greater than 132. If conversion is a numeric array to a string variable, check to see that element zero of one array specifies the number of conversion characters.
INVALID COMPUTED GOTO IN XXXX	A computed GOTO has been executed with a negative or zero index, or the index is too large to correspond to one of the switch points (sequence numbers).
INVALID SUBSTRING ARGUMENT IN XXXX	An argument in statement XXXX is invalid; either null string, beginning character position greater than string length, or number of characters exceed length of string to right of beginning character.
IN XXXX DIM ERROR	Dimension of variable used in matrix calculation inconsistent with dimension declared in DIM statement. Execution stops.
IN XXXX DIV CHECK	A division by zero has been attempted. System supplies + infinity and execution continues.
IN XXXX DUPLICATE FILE NAME	A FILES statement has attempted to specify a file name already in use by this program.
IN XXXX EXP(B) GRT 88.028 SET RESULT = ARGUMENT	Argument of exponential function greater than 88.028. System supplies argument value and continues execution.
IN XXXX EXP ERROR 00 ** (-C) - SET RESULT = 0	Computation of form 0 ** (-1) has been attempted. The system sets result to zero and continues execution.
IN XXXX EXP OVERFLOW	Floating point overflow. System supplies + or - infinity and execution continues.
IN XXXX EXP UNDERFLOW	Floating point underflow. System supplies zero and execution continues.

IN XXXX FNEND WITHOUT DEF	Multiple-line DEF statement not initiated by DEF.
IN XXXX LOG(-B) NOT ALLOWED -EVAL FOR +B	Program has attempted to calculate logarithm of a negative number. System supplies logarithm of absolute value and execution continues.

IN XXXX LOG(0) NOT ALLOWED SET RESULT = 0	Program has attempted to calculate logarithm of 0. The system sets result to zero and execution continues.
IN XXXX MORE THAN 8 FILES	A FILES or FILE # statement has exceeded the limit of 8 files per program.
IN XXXX MORE THAN 20 REPLACEMENTS	The limit of 20 equal signs for a multiple replacement statement has been exceeded.
IN XXXX NESTED DEF	Multiple-line DEF statements may not be nested.
IN XXXX SIN/COS ARG GRT 2 ** 27 - SET RESULT = 0	Argument of function greater than 2 ** 27. System sets result to zero and execution continues.
IN XXXX SQR(-B) ILLEGAL -EVAL FOR +B	Program has attempted to extract square root of negative number. System supplies square root of absolute value and execution continues.
LINE NUMBER GREATER THAN 8 CHARACTERS AT XXXX	Self-explanatory.
MORE THAN 15 FILES REFERENCED BY CALL STATEMENT IN XXXX	The limit of 15 file statements referenced by CALL statements has been exceeded.

NO CHARACTERS IN STRING VARIABLE AT XXXX	A null string variable was used in statement XXXX. A string variable must be set before use in statements such as PRINT, FILE, or CHAIN.
NULL FILE AT XXXX	Attempt made to read file at statement XXXX which does not contain data.
OUT OF DATA IN XXXX	READ statement for which there is no data has been encountered. May mean normal end of program, and should be ignored in those cases. Otherwise, it means not enough data has been supplied. In either case, execution stops.
SUBSCRIPT ERROR IN XXXX	A subscript in statement XXXX is out-of-bounds; either negative, greater than specified in DIM statement, or greater than 10 if implied dimension was used.
UNFINISHED DEF	Multiple-line DEF statement not ended by FNEND.
WORD POINTER OUTSIDE FILE AT XXXX	Value of word pointer is negative, or greater than size of file.
WORD POINTER POSITION ERROR AT XXXX	String control word encountered when not expected, or string control word not present where needed. May be caused by input list not matching file contents as to data type, or by an error in positioning the word pointer via SET: statement.
Filename or Password HAS TOO MANY CHARACTERS IN XXXX	In a CHAIN statement, a filename or password represented by a string variable is too long. The limit for filename and password is 8 and 12 characters, respectively.

*

<50> CANNOT ADD LINKS TO TEMP FILE AT XXXX (reason)	Additional links were needed for a temporary file for output purposes, but could not be obtained. Note reason given and refer to HELP subsystem, code 50, if necessary.
<50> CANNOT GROW PERM FILE AT XXXX	A perm file needed to grow for output purposes, but could not. Note the reason given and refer to HELP subsystem, code 50, if necessary.
<50> ERROR IN CREATING TEMP FILE AT XXXX	A temp file could not be created for output purposes. Note the reason given and refer to HELP subsystem, code 50, if necessary.
<50> UNABLE TO ACCESS FILE AT XXXX (reason)	A file could not be accessed correctly for I/O. If the reason for failure is not self-explanatory as given, refer to the HELP subsystem, code 50.
064 EXECUTE TIME LIMIT EXCEEDED	Execution time limit specified by user or installation exceeded.

ERROR MESSAGES AS A RESULT OF SYSTEM ERRORS

Error messages may also be generated as a result of errors made in the use of the time-sharing system.

System error messages will be indicated by a number code accompanying the message. For example:

009 -- SYSTEM UNKNOWN

The system HELP (a subsystem of the time-sharing system) will provide an explanation of any number - coded error message along with suggestive corrective actions for some errors.

SECTION VII

ERROR LOCATION AND CORRECTION

GENERAL

Locating and correcting errors (or "bugs") in a program is referred to as "debugging." Occasionally (especially in smaller programs) the first run of a new program will be free of errors; but is is more common to have some errors present and therefore the need to correct them exists.

In BASIC, possible errors are of two types:

Form - caused by violating BASIC language rules; can stop the processing of the program.

Logical - caused by erroneous statements of facts, data, equations, etc., made by the user; do not stop the processing but will cause erroneous output data or perhaps no data at all.

Errors of form result in diagnostic error messages being printed. Section VI contains a list of these error messages and their interpretations. Logical errors, however, are not easily detected and generally require a great deal of diligent effort and patience on the part of the user for their detection.

In any case, when errors are detected, the program containing them can be debugged by:

a. retyping statements,
b. inserting new statements, and/or
c. deleting incorrect or superfluous statements from the program.

No overall set of rules can be given that will provide the user with complete means for debugging programs. For the most part, the experienced programmer accumulates techniques from one program to the next. For the majority of programs, the user may find the following procedure helpful.

Before attempting the execution of a program, obtain a printout of the statement sequence by means of the control command LIST. The printout will be an edited version of the sequence, incorporating all corrections, additions, and changes made to the program during entry. The printout can then be readily scanned for possible program errors that may have been otherwise obscured.

Error detection in lengthy programs can best be accomplished by means of PRINT statements requesting a printout of intermediate results. These test results can then be examined and necessary corrections made to the program before voluminous amounts of data are generated. In addition, the user might ask by way of PRINT statements for printout of sample data, results of which the user is already aware or which can be readily checked by hand computation. These testing PRINT statements may then be deleted when they are no longer required.

DEBUGGING A SAMPLE PROGRAM

A sample program and debugging methods are given below. The errors, their detection, and their correction may appear forced but the sample is merely intended to provide guidelines for debugging.

Consider the following problem: Find the maximum point on the sine curve between 0 and 3 radians by searching along the X axis. The computer will be directed to test successive values in intervals of 0.1, 0.01, and 0.001. Thus, the computer is to find the sine of 0, 0.1, 0.2, 0.3,....., 2.8, 2.9 and 3, and to determine which of these 31 values is the largest. Then it is to repeat the search with a 0.01 interval, which involves 301 numbers this time. Then, the search is to be repeated for a 0.001 interval, which involves 3001 numbers. At the end of each search, the computer is directed to print: (a) the value of X1 that has the largest sine, (b) the sine of that number, and (c) the interval of search.

Prior to going to the terminal, the program to accomplish the above problem is organized and written down.

```
10 READ D
20 LET X1 = 0
30 FOR X = 0 TO 3 STEP D
40 IF SIN(X)< = M THEN 100
50 LET X1 = X
60 LET M = SIN(X)
70 PRINT X1, X, D
80 NEXT X
90 GOTO 20 100 DATA 0.1, 0.01, 0.001 110 END
```

The program entries and executions are listed below, with explanations of the debugging process to the right of the program.

```
*   10 READ D
*   LET X1 = 0
*   30 FOR X = 0 TO 3 STEP D
*   40 IF SINR(X) <=M THEN 100
*   50 LET X1 = X
*   60 LET M = SIN(X)
*   70 PRINT X1, X, D
*   80 NEXT X1
*   90 GOTO 20
*  100 DATA 0.1,0.01,0.001
*  110 END
*  RUN
```

The sequence of statements are entered and the command to execute the program is given.

```
40 IF SINR (X) < = M THEN 100
         ↑
         STATEMENT ERROR
         ILLEGAL FOR LOOP
FOR WITHOUT NEXT
UNDEFINED NUMBER 20
```

The statements are edited and error messages are printed. Entering statement 20 defines the number in statement 90 and rectifies the omission of the statement number in the second statement.

```
READY
*  20 LET X1 = 0
*  40 IF SIN(X) < = M THEN 80
*  RUN
```

Statement 40 is corrected. The RUN command is again given.

```
         ILLEGAL FOR LOOP
FOR WITHOUT NEXT

READY
*  80 NEXT X
*  RUN
```

The requested loop is formatted incorrectly. The variable in statement 80 is made to duplicate that in statement 40. The RUN command is again given.

```
     0.1  0.1  0.1
     0.2  0.2  0.1
     0.3  0.3  0.1
     0.4  0.4  0.1
     0.5  0.5  0.1
     0.6  0.6  0.1
     0.7  0.7  0.1
     0
*  20 LET M = -1
*  RUN
```

Program execution is taking place but every value of X is being printed. Execution is halted by depressing BREAK.

A review of the statements indicates that in the first time through the loop (statements 30-80) SIN(0) is compared with M but no value has been assigned to M.

Furthermore, statement 50 "remembers" the value of X each time the loop is executed, so that statement 20 is useless.

Statement 20 is used to assign a value to M less than the maximum value of the sine and the RUN command is again given.

```
    0     0  0.1
  0.1   0.1  0.1
  0.2   0.2  0.1
  0.3   0.3  0.1
  0.4
*  70
*  85 PRINT X1,M,D
*  RUN
```

Again X1, the current value of X, and the interval are being printed. Execution is halted by depressing BREAK.

Statement 70 is within the loop and requests the printout of these items.

Statement 70 is deleted by typing its number and statement 85 is formatted with M substituted for X. The RUN command is again given.

```
   1.6  0.9995736  0.1
   1.6  0.9995736  0.1
   1.6  0.9995736  0.1
   1.6  0.9995736  0.1
   1.6  0.9995736  0
*  90 GOTO 10
*  RUN
```

The same operation is being repeated. Execution is halted by depressing BREAK.

The loop is being repeated for a D value of 0.1. Examination reveals that statement 90 regenerates the same loop by directing the sequence back to statement 20.

Statement 90 is corrected by directing the sequence back to statement 10 for a new value of D. The RUN command is again given.

1.6	0.9995736	0.1	The execution results in a
1.57	0.9999997	0.01	printout of X1, M and D and
1.57099	1	0.001	the program is terminated.

```
OUT OF DATA IN 10
```

APPENDIX A

GLOSSARY OF TIME-SHARING TERMS

Available File Table (AFT)

Table provided by the time-sharing system which lists all files made available to the user for his current session at the terminal.

Binary Files

Files implemented by binary file input/output statements which supplement BASIC language statements.

Catalog

Time-sharing subsystem containing a list of user's file names.

Central Processor

This is the central computer of the time-sharing system whose functions are to receive user input from the data communications processor; process user commands, execute user programs; and transmit user output to the data communications processor...all in a multiprogrammed time-sharing mode of operation.

Character

A digit, letter of the alphabet, or symbol of the terminal's keyboard.

Command Language

A set of orders or instructions which request functions to be performed for a program being executed at a terminal.

Communications Processor

One of the group of peripheral computers attached to the time-sharing system used to monitor the telephone lines for which it is responsible; collect user input lines; form messages; transmit the messages to the central processor; and transmit lines of message output to the user.

Current Program

Program which user is working upon at his current session at the terminal.

Data Files

Files implemented by data file input/output statements which supplement BASIC language statements.

Delimiter

Keyboard characters used to separate parts of a statement.

Executive Program

The system program which controls time-sharing system operation.

File

A generic term for all data stored within and processed by the time-sharing system. Files are of two major classifications: system files and user files.

File System

The repository for all permanent files kept within the time-sharing system.

Input Line

A group of characters ending with a carriage return (RETURN), that are entered by the user from his terminal.

Permanent Files

Permanent files are files which are stored in the file system, by the normally explicit control command SAVE.

Source Program

A user file composed of program statements written in one of the time-sharing compiler languages (e.g., BASIC). Each program statement is entered as a line of input via the terminal. Input lines begin with a 1 to 8-digit line number and end with a carriage return character.

System Files

System files include the program files (such as the executive modules, compilers, command processing routines, and supporting subroutines) which control time-sharing system operation, and the data files (such as catalogs, tables, and libraries) used by the time-sharing system to support its operation.

Temporary Files

Temporary files are files which are processed by the time-sharing system under the direction of a user connected to the system. Temporary files may be new files entered into the time-sharing system by the user; or they may be copies of permanent files which have been called out of the file system by the user for processing. Temporary files reside in the central processor's core memory. They are stored in the file system only by explicit control command SAVE. Temporary files are created and released dynamically in the time-sharing system in the course of the user command processing. Temporary files in existence when the user signs off are released.

Terminal

A keyboard device designed to send and receive programs or data.

User Files

User files include both temporary and permanent source program files, and data files entered into the time-sharing system by the user.

APPENDIX B

SAMPLE BASIC PROGRAMS

While each program in this appendix is titled to indicate a specific use, the set of programs could serve as a general guide to the use of BASIC and is intended to illustrate some problem-solving possibilities and programming techniques.

The sample programs are as follows:

1. Creating a Table of Roots (Program with Loops)
2. Determining Greatest Common Divisor (Program with Subroutine)
3. Computing Total Sales (Program with a List and Table)
4. Plotting a Sine Function (Program with Plot of Function)
5. Calculating True Annual Interest (Program Requiring Input During Execution)

SAMPLE PROGRAM NO. 1

CREATING A TABLE OF ROOTS

(Program With Loops)

A program which creates a table of roots provides the opportunity to study the use of loops. The range of numbers for which roots are desired are 1 to 15. The roots desired are square root, cube root, and fourth root. The statement sequence and run of the program are as follows:

```
*  10 FOR X = 1 to 15
*  20 PRINT X,
*  30 FOR R = 2 to 4
*  40 PRINT X**     (1/R),
*  50 NEXT R
*  60 PRINT
*  70 NEXT X
*  80 END
*  RUN
```

1	1	1	1
2	1.414214	1.259921	1.189207
3	1.732051	1.44225	1.316074
4	2	1.587401	1.414214
5	2.236068	1.709976	1.495349
6	2.44949	1.817121	1.565085
7	2.645751	1.912931	1.626577
8	2.828427	2	1.681793
9	3	2.080084	1.732051
10	3.162278	2.154435	1.77828
11	3.316625	2.223980	1.821160
12	3.464102	2.289428	1.86121
13	3.605551	2.351335	1.898829
14	3.741657	2.410142	1.934336
15	3.872983	2.466212	1.96799

Statements 10 and 70 create the outer loop and determine the range of numbers. Statements 30 and 50 create the inner loop and determine the roots. Note the use of PRINT statement 60 to advance the output a line each time the inner loop is executed and thus line the numbers up with their roots.

This brief program is an indication of the power of loops by which hundreds of computations can be made by executing a few statements repeatedly.

SAMPLE PROGRAM NO. 2

DETERMINING GREATEST COMMON DIVISOR

(Program With Subroutine)

The following example is a program for determining the greatest common divisor (GCD) of three integers (using the Euclidean algorithm) and illustrates the use of subroutines. The first two numbers are selected in statement 30 and 40 and their GCD is determined in the subroutine, statements 200-310. The GCD just found is called X in statement 60, the third number is called Y in statement 70, and the subroutine is entered from statement 80 to find the GCD of these two numbers. This number is, of course, the GCD of the three given numbers and is printed out with them, as directed by statement 90.

```
*  10 PRINT TAB(13);"A";TAB(28);"B";TAB(43);"C";TAB(58);"GCD"
*  20 READ A,B,C
*  30 LET X = A
*  40 LET Y = B
*  50 GOSUB 200
*  60 LET X = G
*  70 LET Y = C
*  80 GOSUB 200
*  90 PRINT A,B,C,G
* 100 GOTO 20
* 110 DATA 60,90,120
* 120 DATA 38456,64872,98765
* 130 DATA 32,384,72
* 200 LET Q = INT(X/Y)
* 210 LET R = X-Q*Y
* 220 IF R = 0 THEN 300
* 230 LET X = Y
* 240 LET Y = R
* 250 GOTO 200
* 300 LET G = Y
* 310 RETURN
* 320 END
* RUN

            A         B         C       GCD
           60        90       120        30
        38456     64872     98765         1
           32       384        72         8

OUT OF DATA
```

SAMPLE PROGRAM NO. 3

COMPUTING TOTAL SALES

(Program With a List and Table)

Below is a listing and run of a program which uses both a list and a table. The program computes the total sales of each of five salesmen, all of whom sell the same three products. The list P gives the price/item of the three products and the table S tells how many items of each product each man has sold. Product number 1 sells for $1.25 per item, number 2 for $4.30 per item, and number 3 for $2.50 per item; salesman number 1 sold 40 items of the first product, 10 of the second, and 35 of the third, and so on. The program reads in the price list in statements 10, 20, 30, using data in line 160, and the sales table in lines 40-80, using data in statements 170-190. The same program could be used again, modifying only statement 160 if the prices change, and only statements 170-190 to enter the sales in another month.

```
*   5 DIM S(3,5),P(3)
*  10 FOR I = 1 TO 3
*  20 READ P(I)
*  30 NEXT I
*  40 FOR I = 1 TO 3
*  50 FOR J = 1 TO 5
*  60 READ S(I,J)
*  70 NEXT J
*  80 NEXT I
*  90 FOR J = 1 TO 5
* 100 LET S = 0
* 110 FOR I = 1 TO 3
* 120 LET S = S + P(I)*S(I,J)
* 130 NEXT I
* 140 PRINT "TOTAL SALES FOR SALESMAN",J,"$",S
* 150 NEXT J
* 160 DATA 1.25,4.30,2.50
* 170 DATA 40,20,37,29,42
* 180 DATA 10,16,3,21,8
* 190 DATA 35,47,29,16,33
* 200 END

*RUN

TOTAL SALES FOR SALESMAN      1     $   180.5
TOTAL SALES FOR SALESMAN      2     $   211.3
TOTAL SALES FOR SALESMAN      3     $  131.65
TOTAL SALES FOR SALESMAN      4     $  166.55
TOTAL SALES FOR SALESMAN      5     $   169.4
```

SAMPLE PROGRAM NO. 4

PLOTTING A SINE FUNCTION

(Program With Plot of Function)

Functions may be readily plotted by the use of a keyboard character to depict the plot. In this example, the plot of a sine function at approximately 10-degree intervals is illustrated. The program consists essentially of a loop in which the character is positioned by means of the computation of the TAB expression. The range of the variable X is in radians and each step of the range is printed in line with its positioned character.

```
* 10 FOR X = 0 TO 6.2832 STEP 0.17
* 20 PRINT TAB(10*SIN(X)+15);"+";TAB(40);INT((X+.005)*100)/100
* 30 NEXT X
* 40 END
* RUN
            +                                0
             +                            0.17
              +                           0.34
               +                          0.51
                +                         0.68
                 +                        0.85
                  +                       1.02
                  +                       1.19
                   +                      1.36
                   +                      1.53
                   +                       1.7
                   +                      1.87
                  +                       2.04
                  +                       2.21
                 +                        2.38
                +                         2.55
               +                          2.72
              +                           2.89
             +                            3.06
            +                             3.23
           +                               3.4
          +                               3.57
         +                                3.74
        +                                 3.91
       +                                  4.08
       +                                  4.25
      +                                   4.42
      +                                   4.59
      +                                   4.76
      +                                   4.93
      +                                    5.1
       +                                  5.27
        +                                 5.44
         +                                5.61
          +                               5.78
            +                             5.95
              +                           6.12
```

SAMPLE PROGRAM NO. 5

CALCULATING TRUE ANNUAL INTEREST RATE

(Program Requiring Input During Execution)

This program is an example of a program that has been formatted by the user so as to require an input upon program execution. The program has then been saved by use of the control command SAVE under the file name TRUINT. To inspect the program, the file name is supplied and the control command LIST is given. The program depicted below will be printed out at the terminal.

```
OLD NAME-TRUINT
READY
*LIST

10 PRINT "THIS PROGRAM WILL CALCULATE THE TRUE ANNUAL INTEREST"
11 PRINT "RATE CHARGED ON AN INSTALLMENT LOAD. YOU SUPPLY THE"
12 PRINT "VALUES OF FOUR VARIABLES: A=AMOUNT OF LOAN (IN $),"
13 PRINT "P=AMOUNT OF EACH PAYMENT ($), N=THE TOTAL NUMBER"
14 PRINT "OF PAYMENTS DUE, AND K=THE NUMBER OF PAYMENTS DUE"
15 PRINT "IN ONE YEAR.";
17 PRINT "WHAT ARE A,P,N,K";
18 INPUT A, P, N, K
19 PRINT
20 IF N=1 THEN 60
21 IF P*N >= A THEN 27
22 PRINT
23 PRINT "THAT'S NOT REASONABLE. THE PAYMENTS ADD UP"
24 PRINT "TO LESS THAN THE AMOUNT OWED. TRY AGAIN"
25 PRINT
26 GOTO 17
27 LET R=0
28 LET D=100
29 GOSUB 38
30 IF P=P1 THEN 48
31 IF P>P1 THEN 34
32 LET R=R-D
33 GOTO 35
34 LET R=R+D
35 LET D=D/2
36 IF D<0.0001 THEN 48
37 GOTO 29
38 LET R1=R/(100*K)
39 LET Q=1+R1
40 IF N*LOG(Q)/LOG(10)<= 75 THEN 43
41 LET P1=A*R1
42 RETURN
43 IF Q>1 THEN 46
44 LET P1=A/N
45 RETURN
```

```
46 LET P1=A*Q↑N*R1/(Q↑N-1)
47 RETURN
48 LET R=0.01*INT(0.5+100*R)
49 IF R<199.5 THEN 55
50 PRINT
51 PRINT "ARE YOU SURE THE DATA IS RIGHT? THE INTEREST"
52 PRINT "RATE WOULD BE OVER 200 PERCENT. TRY AGAIN"
53 PRINT
54 GOTO 17
55 PRINT "THE TRUE ANNUAL INTEREST RATE = ";R
56 PRINT
57 PRINT
58 PRINT "ANOTHER CASE?  (TYPE 'S' TO STOP NOW)."
59 GOTO 17
60 LET R=(P/A-1)*K
61 LET R=100*R
62 GOTO 48
63 END
```

To use the program, the control command RUN is given in place of LIST. Upon execution, statement 18 requests assignment of values to the variables A, P, N, and K. Complete execution of the program, one which will provide for a value representing true annual interest rate, will be held up until the input request is complied with by the way of the user typing in his inputs on the line containing the question mark.

A run of this program with an assignment of values to variables A, P, N, and K is depicted below.

```
*RUN
THIS PROGRAM WILL CALCULATE THE TRUE ANNUAL INTEREST
RATE CHARGED ON AN INSTALLMENT LOAD.  YOU SUPPLY THE
VALUES OF FOUR VARIABLES:  A = AMOUNT OF LOAD (IN $),
P = AMOUNT OF EACH PAYMENT ($), N = THE TOTAL NUMBER
OF PAYMENTS DUE, AND K = THE NUMBER OF PAYMENTS DUE
IN ONE YEAR.  WHAT ARE A,P,N,K  ?600.00, 31.99, 21, 12

THE TRUE ANNUAL INTEREST RATE = 12.61

ANOTHER CASE?  (TYPE 'S' TO STOP NOW).

WHAT ARE A,P,N,K  ?S
```

APPENDIX C.

OCTAL/ASCII CONVERSION EQUIVALENTS

Octal No.	ASCII Char.	Octal No.	ASCII Char.	Octal No.	ASCII Char.	Octal No.	ASCII Char.
000	NULL	040	SP	100	@	140	GRA
001	SOH	041	EXP	101	A	141	a
002	STX	042	"	102	B	142	b
003	ETX	043	#	103	C	143	c
004	EOT	044	$	104	D	144	d
005	ENQ	045	%	105	E	145	e
006	ACK	046	&	106	F	146	f
007	BELL	047	'	107	G	147	g
010	BSP	050	(	110	H	150	h
011	HT	051	)	111	I	151	i
012	LF	052	*	112	J	152	j
013	VT	053	+	113	K	153	k
014	FFD	054	,	114	L	154	l
015	CR	055	-	115	M	155	m
016	SO	056	.	116	N	156	n
017	SI	057	/	117	O	157	o
020	DLE	060	0	120	P	160	p
021	DC1	061	1	121	Q	161	q
022	DC2	062	2	122	R	162	r
023	DC3	063	3	123	S	163	s
024	DC4	064	4	124	T	164	t
025	NAK	065	5	125	U	165	u
026	SYN	066	6	126	V	166	v
027	ETB	067	7	127	W	167	w
030	CAN	070	8	130	X	170	x
031	EM	071	9	131	Y	171	y
032	SUB	072	:	132	Z	172	z
033	ESC	073	;	133	LBK	173	LBR
034	FS	074	<	134	RSL	174	VTL
035	GS	075	=	135	RBK	175	RBR
036	RS	076	>	136	CFX	176	TILDE
037	US	077	?	137	_	177	DEL

DEFINITIONS

COMMUNICATIONS CONTROL

ACK	Acknowledgment
CAN	Cancel
DC1	Device Control 1
DC2	Device Control 2
DC3	Device Control 3
DC4	Device Control 4
DLE	Data Link Escape
EM	End of Medium
ENQ	Enquiry
EOT	End of Transmission
ESC	Escape (Alternate Mode)
ETB	End of Transmission Block
ETX	End of Text
NAK	Negative Acknowledgment
SOH	Start of Heading
STX	Start of Text
SUB	Substitute Character
SYN	Synchronous Idle

FORM EFFECTORS

BSP	Backspace
CR	Carriage Return
FFD	Form Feed
HT	Horizontal Tabulation
LF	Line Feed
VT	Vertical Tabulation

ITEM SEPARATORS

FS	File Separator
GS	Group Separator
RS	Record Separator
US	Unit Separator

TEXT MATERIAL

BELL	Bell, or other attention signal
CFX	^
DEL	Delete (Rubout)
EXP	!
GRA	`
LBK	[
LBR	{
TILDE	~
NULL	Null
RBK	]
RBR	}
RSL	\
SI	Shift In
SO	Shift Out
SP	Space
VTL	Vertical Line

APPENDIX D

SUMMARY OF BASIC LANGUAGE CHARACTERISTICS

BASIC STATEMENTS

Arithmetic Statements

DEF	- defines a repeatedly used function
LET	- requests a computation or manipulation upon an arithmetic variable
MAT	- requests a computation or manipulation upon a matrix

Specification Statements

CHANGE	- converts string characters to numerical code or vice versa
DATA	- specifies numeric values for variables listed in a READ statement
DIM	- reserves space for list or table

Input/Output Statements

CHAIN	- compiles and executes series of programs
INPUT	- delays input of values to variables until program is in execution; program will request input of data by terminal user or a user's file when statement is executed
PRINT	- prints computed results; prints text - prints computed results and text - skips lines - formats output data
PRINT USING	- formats output line
READ	- reads values from a DATA statement or user's file and assigns them to designated variables
RESTORE	- restores previously processed blocks of input data from DATA statements

Loop and Subroutine Statements

CALL	- directs processing sequence to a subroutine previously saved
FOR	- is first statement of a loop and sets conditions of loop
NEXT	- is last statement of loop
GOSUB	- directs processing sequence to a subroutine
RETURN	- returns processing sequence from a subroutine

Logic Statements

GOTO	- unconditionally transfers the processing sequence to a designated statement
IF-----THEN or IF---GOTO	- conditionally transfers the processing sequence to a designated statement
ON-----THEN or ON---GOTO	- conditionally transfers the processing sequence to designated statements
STOP	- stops the execution of the program
END	- indicates end of program

Utility Statements

CHAIN	- compiles and executes series of programs
TRACE ON TRACE OFF	- prints line numbers of statements between TRACE ON/TRACE OFF statements

Documentation Statement

REM - inserts a remark into the statement sequence

ARITHMETIC OPERATIONS

Operator symbol	denotes
+	addition
-	subtraction
*	multiplication
/	division
↑ or **	raise to a power

RELATIONAL SYMBOLS

Relational symbol	denotes
=	is equal to
<	is less than
<= or =<	is less than or equal to
>	is greater than
>= or =>	is greater than or equal to
< > or > <	is not equal to

MATHEMATICAL FUNCTIONS

Function	Operation
SIN(X)	sine of X
COS(X)	cosine of X
TAN(X)	tangent of X
COT(X)	cotangent of X
ATN(X)	arctangent of X
EXP(X)	e to the power X
LOG(X)	natural logarithm of X
CLG(X)	common logarithm of X
ABS(X)	absolute value of X
SQR(X)	square root of X
INT(X)	truncate X
RND(X)	produce a random number
SGN(X)	sign determination
DET(X)	provide determinant of last matrix inverted

MISCELLANEOUS FUNCTIONS

Function	Operation
TIM(X)	elapsed processor time
CLK$	time of day
DAT$	calendar date
NUM(X)	count of matrix data elements
SST(X$,Y,Z)	selected characters of a string (substring)
TAB(X)	character print position
SPC(X)	space print position
LEN(X$)	number of characters in string
LIN(X)	last line number encountered in reading/writing file
ASC(X)	numeric values of character or abbreviation
STR$(N)	expression to string conversion
VAL(S$)	string to expression conversion
TST(S$)	nonzero output if string can be interpreted as a number
HPS(X)	horizontal print position of next field, in current line, of file being written

ASCII DATA FILE STATEMENTS

- File preparation statements

```
FILES filename 1, password;....;filename n, password
FILES user-id/catalogname$password/.../
      filename$password,permissions
FILE # file designator, "filename, password"
```

- File read statements

```
READ # file designator, input list
INPUT # file designator, input list
```

- File write statements

```
WRITE # file designator, output list
PRINT # file designator, output list
PRINT # file designator, USING statement number,
        output list
```

- Matrix input statements

```
MAT READ # file designator, matrix input list
MAT INPUT # file designator, matrix input list
```

- Matrix output statements

  MAT WRITE # file designator, matrix output list
  MAT PRINT # file designator, matrix output list

- File manipulation statements

  SCRATCH # file designator
  RESTORE # file designator
  BACKSPACE # file designator

- Utility statements

  APPEND # file designator
  MARGIN # file designator, expression
  DELIMIT # file designator, $\left\{\begin{matrix}\text{(character)} \\ \text{(abbreviation)}\end{matrix}\right\}$
  IF END # file designator $\left\{\begin{matrix}\text{THEN} \\ \text{GOTO}\end{matrix}\right\}$ line number
  IF MORE # file designator $\left\{\begin{matrix}\text{THEN} \\ \text{GOTO}\end{matrix}\right\}$ line number

BINARY FILE STATEMENTS

- File preparation statements

```
FILES filename 1,password;....;filename n,password
FILES user-id/catalogname$password/.../
      filename$password,permissions
FILE: file designator, "filename,password"
```

- File read statement

```
READ: file designator,input list
```

- File write statement

```
WRITE: file designator,output list
```

- Matrix input statement

```
MAT READ: file designator,matrix input list
```

- Matrix output statement

```
MAT WRITE: file designator,matrix output list
```

- File manipulation statements

```
SCRATCH: file designator

RESTORE: file designator

BACKSPACE: file designator
```

- Utility statements

```
APPEND: file designator
        (for sequential files only)

IF END: file designator {THEN} line number
                        {GOTO}

IF MORE: file designator {THEN} line number
                         {GOTO}

SET: file designator TO expression
     (for random files only)
```

BINARY FILE FUNCTIONS

Function	Operation
LOC (file designator)	word pointer location
LOF (file designator)	file length

APPENDIX E

ALPHABETIC CODES FOR RELATIONAL SYMBOLS

Relational Code	Denotes	As Illustrated By
EQ	is equal to	A EQ B
LT	is less than	A LT B
LE	is less than or equal to	A LE B
GT	is greater than	A GT B
GE	is greater than or equal to	A GE B
NE	is not equal to	A NE B

I N D E X

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form*

TITLE: SERIES 600/6000 TIME-SHARING BASIC

ORDER No.: BR36, REV. 1

DATED: SEPTEMBER 1971

ERRORS IN PUBLICATION:

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION:

(Please Print)

FROM: NAME ______________________ DATE: ______________

COMPANY ______________________

TITLE ______________________

____

____

*Your comments will be promptly investigated by appropriate technical personnel, action will be taken as required, and you will receive a written reply. If you do not require a written reply, please check here. ☐

FIRST CLASS
PERMIT NO. 39531
WELLESLEY HILLS,
MASS. 02181

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

POSTAGE WILL BE PAID BY:

HONEYWELL INFORMATION SYSTEMS
60 WALNUT STREET
WELLESLEY HILLS, MASS. 02181

ATTN: PUBLICATIONS, MS 050

Honeywell