20854A

Timeshared BASIC/2000, Level F

Reference Manual

series
2000

HP 20854A
Timeshared/BASIC 2000,
Level F

Reference Manual

HEWLETT *hp* PACKARD

List of Effective Pages

The list of Effective Pages gives the most recent date on which the technical material on any given page was altered. If a page is simply re-arranged due to a technical change on a previous page, it is not listed as a changed page. Within the manual, changes are marked with a vertical bar in the margin.

Preface

This manual is a reference text for people who want to use the Hewlett-Packard 20854A Timeshared BASIC/2000, Level F programming language in a timeshared environment at user terminals.

This manual is designed to serve both beginners and experienced BASIC programmers: the sample programs were chosen for their teaching value (beginners are encouraged to try them "on-line"), and the manual's sections progress from simple elements of BASIC to advanced usage.

Sections I and II introduce the timeshared BASIC system and the essentials of BASIC programming. Section III discusses advanced features of the programming language. Sections IV through VIII describe files, matrices, strings, logical operations, and formatted output, respectively. Section IX contains the Backus-Naur Form syntax for the programming language, and other information for the professional user. Several appendices and an index conclude the manual.

While this manual is designed to support the use of the 20854A Timeshared BASIC/2000, Level F programming language, it can be used as a programmer's guide to several, earlier, versions of Hewlett-Packard Timeshared BASIC programming languages. The following table lists the 20854A options which are equivalent to the earlier programming languages.

If the 20854A ordered is:	Then the programming features and capabilities in this manual are equivalent to these languages:	
20854A	2000F (options 200 and 500)	
20854A-001	2000F (options 205 and 505)	
20854A-002	2000F (options 210 and 510)	2000C (High-speed option)
20854A-003	2000F (options 215 and 515)	2000C (High-speed option)
20854A-004	2000F (1600 BPI tape)	

Printing History

New editions incorporate all update material since the previous edition. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover changes only when a new edition is published. If minor corrections and updates are incorporated, the manual is reprinted but neither the date on the title page and back cover nor the edition change.

First Edition . . . . Oct 1974
Second Edition . . . . Dec 1975

DEC 1975

Contents

Text Conventions

SAMPLE	EXPLANATION
PLEASE LOG IN	All capitals in examples indicates computer-output information. . .
20 PRINT X, Y LIST	or a statement or command typed by the programmer.
This section. . .	Mixed upper and lower case is used for regular text.
line number PRINT X, Y	Lower case italics indicates a general form, derived from BASIC syntax requirements (Sect. IX).
return linefeed *control* *break*	Represents the terminal keys: Return, Linefeed, Control, and Break.
[ ]	An element enclosed in brackets is optional.

SECTION I
Introduction to Timeshared BASIC

HP 20854A Timeshared BASIC/2000, Level F programming language is used in a timeshared environment at user terminals. The Timeshared BASIC system (TSB) uses two computers - - a main computer for actual computation and an Input/Output processor computer to control access to the main computer. Additional peripheral equipment is associated with the system at the central site and is under control of the system operator. Up to 32 user terminals can be connected directly (hardwired) to the TSB system or connected remotely through dial-up telephone modems.

This section describes how to log on and log off, how to enter statements and commands and how to make corrections. Simple programs are used for illustration, but the actual programming language is described in Section II.

This manual assumes that the user is familiar with the terminal's keyboard. Special keys with particular functions in the TSB system are described in this section. The characteristics of particular types of user terminals are given in Appendix E. A user's terminal may be one of several types. Some terminals are equipped with a paper tape punch and reader. The user can enter programs into the system either through the keyboard or through the paper tape reader. System output can be typed out on the terminal as well as punched on paper tape. In addition, a line printer may be connected to the system. If a line printer is available, system output can be printed on it. The system is designed so that any user should experience no more than a few seconds delay between entering a command and receiving a response from the system, even when all terminals are active.

The user can work in a simple interactive mode, entering and running programs and reading the results from the user terminal, or he can take advantage of the large storage capacity of the TSB system by using library programs and by storing his own programs for later use.

In this section only, characters typed by the computer are underlined to distinguish them from user input. Subsequent sections assume that this distinction is clear to the user.

SPECIAL KEYS

Key	Function
break	Terminates a running program, listing, or punching operation. This key may appear on the keyboard as INTRPT, BRK, ATTN, etc., depending on the user terminal type.
control	Converts normal keys to non-printing control character keys. This key may appear as CTRL, CTL, CONTRL, etc., depending on the user terminal type.
linefeed	Causes the user terminal to advance one line. This key may appear as LINEFEED, LF, etc., depending on the user terminal type.
return	Must be pressed after every statement and command and after some control characters. It terminates the line and causes the terminal's printing element to return to the first print position. TSB responds with a linefeed if the entered line is acceptable. This key may appear as RETURN, CR, etc., depending on the user terminal type.
←	Backspace. Deletes one preceding character for each ← typed in. This key may be represented by the underscore (_) character on some types of user terminal.
control C (C^c)	Terminates an input loop during program execution. It must be followed by *return*. Effectively, C^c causes a jump to the END statement. TSB responds by printing DONE followed by a *return* and *linefeed*.
control N (N^c)	Generates a *linefeed* when used in a PRINT statement.
control O (O^c)	Generates a *return* when used in a PRINT statement.
control Q (Q^c)	Diverts output to user's terminal when the line printer is designated as the output device.
control W (W^c)	Returns output to the line printer if output was previously diverted via Q^c.
control X (X^c)	Deletes a line being typed from the user terminal. TSB responds by printing a backslash (\ ) followed by a *return* and *linefeed*.

PROMPT CHARACTERS

The TSB system uses a set of prompting characters to signal to the user that certain input is expected or that a specific action is completed.

Character	Meaning
?	User input is expected during execution of an INPUT statement.
??	Further input is expected during execution of an INPUT statement.
???	A BASIC command was mistyped; re-enter it correctly.
\	Issued by TSB in response to the control character X^c. Indicates that the line being typed just prior to entry of X^c is deleted from the user's work area.

LOGGING ON AND OFF

Connection to the Computer

To log on to the TSB system, connection must be established between the user terminal and the computer. There are several ways of doing this, depending on the type of user terminal equipment used.

ACOUSTIC COUPLER AND TELEPHONE:

1. Set terminal mode to ON-LINE and power switch to ON.

2. Set coupler power switch to ON.

3. If coupler has a duplex switch, set to FULL or FULL/UP.

4. If coupler has a line switch, set to ON-LINE.

5. Remove telephone handset and dial the computer telephone number.

6. When the computer responds with a high pitched tone, place the handset into the coupler receptacle (the correct handset position should be marked on the coupler).

HALF-DUPLEX COUPLER AND TELEPHONE:

1. Set terminal mode to ON-LINE and power switch to ON.

2. Set coupler power switch to ON.

3. If coupler has a line switch, set to ON-LINE.

4. Remove telephone handset and dial the computer telephone number.

5. When the computer responds with a high pitched tone, place the handset into the coupler receptacle (the correct handset position should be marked on the coupler).

DATA SET:

1. Set terminal mode to ON-LINE and power switch to ON.

2. Press the TALK button on the Data Set.

3. Remove the handset and dial the computer telephone number.

4. When the computer responds with a high pitched tone, press the DATA button on the Data Set to light it, and replace the handset in its cradle.

> *Note:* *When connection is via telephone lines, the user must log on within a time period (normally two minutes) determined by the system operator.*

DIRECT CONNECTION (HARDWIRED):

Set terminal mode to ON-LINE and power switch to ON.

Checking the Connection

This step is optional. The TSB system does not respond once connection is established. If you wish to determine that connection has been made, type X^C. If the terminal and the computer are connected, the system responds with " \ ". For further verification, type any numeral followed by *return*. The TSB system will respond:

PLEASE LOG IN *return linefeed*

Identification Code

An identification code is assigned to you by the system operator. The code consists of a single letter followed by a three-digit number. When logging on, the identification code along with a password and sometimes a terminal type parameter must be specified.

Password

The password is also assigned to you by the system operator. It consists of from one to six printing or non-printing characters. The password can be kept confidential by using non-printing characters. For example, on the terminal the password $SEC^C R^C ET^C$ prints as:

    SEE

> *Note:* *The character E^C should not be used in a password. It is a reserved*
> *character for some types of terminals.*

Terminal Type Parameter

The terminal type parameter informs TSB of the type of terminal being logged on. Failure to specify the correct parameter may result in a loss of characters. Terminal type is specified as one digit as follows:

    Not specified, or 0 = HP 2600A or HP 2749A/B, HP 2640A or IBM 2741 (default)
    1 = Execuport 300
    2 = ASR-37
    3 = TermiNet 300, TermiNet 1200, or HP 2762A/B
    4 = Memorex 1240

Logging On

Once the terminal is connected and ready, the user may log on. To log on, type the HELLO command. For example:

    HELLO-H200, JOHN, 1

H200, JOHN and 1 are sample parameters representing the identification code, the password, and the terminal type. A comma must be typed between them. TSB responds with a system message or the word READY. In either case, the user is logged on and can enter BASIC commands or statements.

ERRORS DURING LOGGING ON: If a mistake is made during logging on, the system responds with an appropriate error message. For example, if you forget to type the hyphen while entering the HELLO command:

```
HELLOH200,JOHN,1
```

TSB responds with the message:

```
ILLEGAL FORMAT
```

Re-enter the command in the correct form.

If the password is entered incorrectly:

```
HELLO-H200,JHN,1
```

TSB responds:

```
ILLEGAL ACCESS
```

Re-enter the command with the correct password.

The messages ILLEGAL ACCESS and ILLEGAL FORMAT indicate that some or all of the current input is not acceptable to the system.

Spelling mistakes, format errors and incorrect parameters can be corrected while the line is being entered if the error is noticed before return is pressed. The backspace character ($\leftarrow$) (underscore (_) on some terminals) can be used to correct a few characters just typed, or the control character X^c can be used to cancel the entire line and start over.

Suppose the command HELLO is misspelled during entry. The backspace ($\leftarrow$) will delete the last character. The user retypes the character correctly and finishes the line. When you press *return*, the line is entered correctly.

```
HELO←LO-H200,JOHN,1
```

If several characters have been typed after the error, the backspace character must be typed for each character to be deleted. In the following example, four characters are deleted:

```
HELO-H2←←←←LO-H200,JOHN,1
```

Another method is to use X^c to cancel the entire line. X^c must be typed before *return* is pressed. To cancel a line, type X^c. The system responds with a backslash at the end of the line and then produces a *return* and *linefeed*. The correct command can be entered on the new line:

```
HELO-\
HELLO-H200,JOHN,1
```

Logging Off

When a session at the terminal is completed, the user logs off with the BASIC command BYE.
To log off, type:

        BYE

TSB responds by printing the total number of minutes the user was logged on. For example:

        014 MINUTES OF TERMINAL TIME

THE BASIC LANGUAGE

There are many types of languages. English is a natural language used to communicate with people.
To communicate with a computer system we use a formal language, that is, a combination of
simple English and algebra. BASIC is a formal language used to communicate with the TSB
system. The TSB system employs BASIC statements with which to write programs and
BASIC commands for controlling program operation.

Commands

BASIC commands instruct the TSB system to perform certain control or utility functions such as
storing and listing programs or logging on and off the system. Commands differ from statements
used to write a program in the BASIC language. A command instructs the system to perform some
action immediately, while a statement is an instruction to perform an action only when the program
is executed (run). A statement is always preceded by a statement number; a command never is.

Any BASIC command can be entered once the logging on procedure is successfully completed.
Each command is a single word that can be abbreviated to the first three characters on entry.
Embedded blanks are ignored. If a command is misspelled, TSB returns three question marks.

Following entry of each command, *return* must be pressed to signal that command entry is complete.

Some commands have parameters to further define command operation. For instance, BYE is a
command that signals completion of a user session at the terminal and results in logging the user off
the system and disconnection of the terminal from the system. BYE has no parameters. Another
command, LIST, results in a display of the current program in the user's work area. It may have
parameters to specify that only part of the program is to be printed.

Statements

BASIC statements are used to write a BASIC program that will be subsequently executed. Each
statement within the program performs a particular function. Every statement entered becomes
part of the current program and is kept until explicitly deleted or the user logs off the TSB system.
In addition, programs may be saved in one of the system libraries for further use.

A statement is always preceded by a statement number. This number is an integer between 1 and 9999. The statement number indicates the order in which the statements will be executed. Statements are ordered by BASIC from the lowest to the highest statement number. This order is maintained by the TSB system. Thus, it is not necessary for the user to enter statements in execution order so long as the statement numbers are in that order.

Following entry of each statement, *return* must be pressed to inform the system that the statement is complete. The system generates a *return* and a *linefeed* to the next line to signal that the statement is acceptable. If an error is made while entering the statement, the computer prints an error message.

BASIC statements are free form; blanks are ignored. For example, the following statements are equivalent:

```
30  PRINT  S
30        PRINT        S
30PRINTS
   3  0  P  R  I  N  T  S
```

Error Messages

If an error is made in a statement line and the line is entered with *return*, TSB responds with a message. The message consists of the word ERROR.

For example, if the line:

```
30  PRING  S
```

is entered, the system will respond:

```
ERROR
```

The user may press *return* and re-enter the statement in the correct form. If the error is not obvious, type any character after the message followed by *return*. The system will respond with a diagnostic message:

```
30  PRING  S
ERROR:  MISSING  ASSIGNMENT  OPERATOR
```

Typing a colon causes the diagnostic message to be printed. Any other character could have been typed with the same result.

Changing or Deleting a Statement

If an error is made before return is pressed, the error can be corrected with the backspace character (←) or the line may be cancelled with X^C. (See "Errors During Logging On", above.) After *return* is pressed, the statement can be changed or deleted.

To change a statement, simply type the same statement number followed by the desired statement. To change this statement:

        30 PRINT X

Retype it as:

        30 PRINT S

A change of this type can be made any time before the program is run.

To delete a statement, type the statement number followed by a *return*:

        30

Statement 30 is deleted.

The DELETE command described in Section III is useful to delete a group of statements.

BASIC Programs

Any statement or group of statements that can be executed constitutes a program. The last statement (the statement with the highest statement number) of every program must be an END statement. The following is an example of a simple BASIC program:

```
        15    PRINT 35+5
        25    END
```

15 and 25 are statement numbers. PRINT is a key word or instruction that tells TSB the kind of action to perform. In this case, it prints the result of the expression that follows. 35+5 is an arithmetic expression. It is evaluated by the system, and when the program is run, the result is printed. END is also a key word. It informs TSB that this is the end of this program. An END statement is required as the last statement within every program.

Usually, a program contains several statements. The following four statements are a program:

```
10   INPUT A,B,C,D,E
20   LET S=(A+B+C+D+E)/5
30   PRINT S
40   END
```

This program, which calculates the average of five numbers, is shown in the order of its execution. It could be entered in any order if the statement numbers assigned to each statement were not changed. The following program executes exactly like the program above:

```
40   END
20   LET S=(A+B+C+D+E)/5
10   INPUT A,B,C,D,E
30   PRINT S
```

Generally, it is a good idea to number statements in increments of 10. This allows room to insert additional statements as needed.

User's Work Area

When statements are entered at the terminal, these statements become part of the user's work area. All statements in the user's work area constitute the current program.

Any statement in the user's work area can be edited or corrected; the resulting statement will then replace the previous version in the user's work area. When the user logs off the TSB system, the work area is cleared. Commands are available with which to retain the contents of the user's work area in the user's library.

Listing a Program

At any time while a program is being entered, the LIST command can be used to produce a listing of the statements that have been accepted by the TSB system. LIST causes the system to print a listing of the current program at the terminal.

After deleting or changing a line, LIST can be used to check that the deletion or correction has been made. For example, a correction is made while entering a program:

```
10   U←INPUT A,B,C,D,E
20   PR←←LET S=(A+B+C+D+E)/5
30   PRINT S
40   END
```

To check the corrections, list the program:

```
LIST


10   INPUT A,B,C,D,E
20   LET S=(A+B+C+D+E)/5
30   PRINT S
40   END
```

Should the statements be entered out of order, the LIST command will cause them to be printed in ascending order by statement number. For example, the program is entered in this order:

```
20 LET S=(A+B+C+D+E)/5
30 PRINT S
40 END
10 INPUT A,B,C,D,E
```

The list will be in correct order of execution:

```
LIST


10   INPUT A,B,C,D,E
20   LET S=(A+B+C+D+E)/5
30   PRINT S
40   END
```

Running a Program

After the program is entered and, if desired, checked with LIST, it can be executed with the RUN command. RUN will be illustrated with two sample programs.

The first program has two statements:

```
15 PRINT 35+5
25 END
```

When run, the result of the expression 35+5 is printed:

```
RUN


40

DONE
```

Because the program contains a PRINT statement, the result is printed when the program is run. When execution of a program is complete the system prints the message DONE at the user's terminal.

The second sample program averages a group of five numbers. The numbers must be input by the user:

```
10  INPUT  A,B,C,D,E
20  LET  S=(A+B+C+D+E)/5
30  PRINT  S
40  END
```

Each of the letters following the word INPUT and separated by commas names a variable that will contain a value input by the user from his terminal. When the program is run, TSB signals that input is expected by printing a question mark. The user enters the values following the question mark. They are entered with a comma between each successive value.

The statement LET S = (A+B+C+D+E)/5 assigns the value of the expression to the right of the equal sign to the variable S on the left of the equal sign. The expression first adds the variable values within parentheses and then divides them by 5. The result is the value of S.

When the program is run, the user enters input values and the computer prints the result:

```
RUN


?7,5,6,8,9
 7

DONE
```

Deleting a Program

If a program that has been entered and run is no longer needed, it can be deleted from the user's work area with the SCRATCH command. SCRATCH deletes whatever program has been entered by the user during the current session.

The first program entered was:

```
15   PRINT  35+5
25   END
```

This program should be scratched before entering the next program. Otherwise, statement numbers may overlap causing undesirable results. In the latter case, when RUN is typed, the program will execute in order of the statement numbers. The program will execute until the first END statement is encountered. For example, if the program above remains in the user's work area and the user enters a new program, as follows:

```
10    INPUT A,B,C,D,E
20    LET S=(A+B+C+D+E)/5
30    PRINT S
40    END
```

Typing RUN produces the following results:

```
RUN


?7,5,6,8,9
 40

DONE
```

The program executes statements 10 through 25, accepting input from the user but printing the result of the expression 35+5.

A listing of the current program would appear:

```
LIST


10    INPUT A,B,C,D,E
15    PRINT 35+5
20    LET S=(A+B+C+D+E)/5
25    END
30    PRINT S
40    END
```

Documenting a Program

Remarks that explain or comment can be inserted in a program with the REM statement. Any remarks typed after the word REM will be printed in the program listing but will not affect program execution. As many REM statements can be entered as are needed.

The sample program to average five numbers can be documented with several remarks:

```
5 REM   THIS PROGRAM AVERAGES
7 REM   FIVE NUMBERS.
15 REM FIVE VALUES MUST BE INPUT.
25 REM S CONTAINS THE AVERAGE.
```

The statement numbers determine the position of the remarks within the existing program. A listing will show them in order:

```
LIST


5    REM   THIS PROGRAM AVERAGES
7    REM   FIVE NUMBERS.
10   INPUT A,B,C,D,E
15   REM FIVE VALUES MUST BE INPUT.
20   LET S=(A+B+C+D+E)/5
25   REM S CONTAINS THE AVERAGE.
30   PRINT S
40   END
```

When run, the program will execute exactly as it did before the remarks were entered.

SECTION II
The Essentials of BASIC

This section contains enough information to allow you to use BASIC in simple applications, without using the capability of storing programs.

Proceed at your own pace. The information in the vocabulary and operators subsections is included for completeness; experienced programmers may skip these. Programmers with some knowledge of BASIC may also concentrate on capabilities of the TSB system presented in the commands subsection.

The "Operators" subsections contain brief descriptions, rather than explanations, of the logical operators. The novice should not expect to gain a clear understanding of logical operators from this presentation. Section VII presents more details and examples of TSB logical operations. Readers wishing to make best use of TSB logical capabilities should consult this section. Those unfamiliar with logical operations should also refer to an elementary logic text.

A simple program is included at the end of this section for reference; it contains a running commentary on the uses of many of the BASIC statements presented in the section.

TERM: NUMBER

Defined: A positive or negative decimal number whose magnitude is between an approximate minimum of 10^{-38} (2^{-129}) and an approximate maximum of 10^{38} (2^{127}). Zero is included.

The precision of all numbers in TSB is 6 to 7 decimal digits (23 binary digits). If the user types a BASIC statement which contains a number that is not representable in TSB, the system will print a warning and change the number in the statement to the closest representable one.

If an executing program makes a calculation which results in a non-representable number, that number will be set to the closest representable positive number and a warning message will be printed.

TERM: E NOTATION

Defined: A means of expressing numbers having more than six decimal digits in the form of a decimal number raised to some power of 10.

E notation is used to print numbers greater than six digits. (See PRINT.) It may also be used to input any number. When entering numbers in E notation, leading and trailing zeros may be omitted from the number; the + sign and leading zeros may be omitted from the exponent.

EXAMPLES:

1.00000E+06 is equivalent to 1000000 and is read:
"1 times 10 to the sixth power" ($1x10^6$).

1.02E+4 is equal to 10200
1.02000E-04 is equal to .000102

TERM: SIMPLE VARIABLE

Defined: A letter (from A to Z); or a letter immediately followed by a number (from 0 to 9).

Variables are used to represent numeric values. For instance, in the statement:

        10 LET M5 = 96.7

M5 is a variable; 96.7 becomes the value of the variable M5.

There are two other types of variables in TSB, array and string variables; their use is explained in Sections V and VI respectively.

EXAMPLES:

A0	*B*
M5	*C2*
Z9	*D*

TERM: EXPRESSION

Defined: A combination of variables, constants and operators which has a numeric value.

EXAMPLES:

(P + 5)/27

(where P has previously been assigned a numeric value.)

Q - (N + 4)

(where Q and N have previously been assigned numeric values.)

TERM: ARITHMETIC EVALUATION

Defined: The process of calculating the value of an expression.

THE ASSIGNMENT OPERATOR

SYMBOL: =

GENERAL FORM: LET *variable = expression*

                 variable = expression

The assignment operator assigns an arithmetic or logical value to a variable.

When used as an assignment operator, = is read "takes the value of," rather than "equals". It is, therefore, possible to use assignment statements such as:

        100 LET X = X+2

This is interpreted by TSB as: "LET X take the value of (the present value of) X, plus two."

Several assignments may be made in the same statement, as in statements 10 and 50 below.

See Section VII, "Logical Operations" for a description of logical assignments.

EXAMPLES:

```
10  LET A = B2 = C = 0
20  LET A9 = C5
30  Y = (N-(R+5))/T
40  N5 = A + B2
50  P5 = P6 = P7 = A = B = 98.6
```

ARITHMETIC OPERATORS

SYMBOLS: ↑ * / + –

Each symbol represents an arithmetic operation, as:

exponentiate: ↑

multiply: *

divide: /

add: +

subtract: –

The "–" symbol is also used as a sign for negative numbers. It is good practice to separate arithmetic operations with parentheses when unsure of the exact order of precedence. The order of precedence (hierarchy) is:

↑

* /

+ –

with ↑ having the highest priority. Operators on the same level of priority are acted upon from left to right in a statement. See "Order of Precedence" in this Section for examples.

EXAMPLES:

```
40. LET N1 = X-5
50  LET C2 = N↑3
60  LET A = (B-C)/4
70  LET X = ((P↑2)-(Y*X))/N+Q
```

RELATIONAL OPERATORS

SYMBOLS: = # <> > < >= <=

Relational operators determine the logical relationship between two expressions, as

equality: =

inequality: # *or:* <>

greater than: >

less than: <

greater than or equal to: >=

less than or equal to: <=

Expressions using relational operators are logically evaluated, and assigned a value of "true" or "false" (the numeric value is 1 for true, and 0 for false).

When the = symbol is used in such a way that it might have either an assignment or a relational function, TSB assumes it is an assignment operator. For a description of the assignment statement using logical operators, see Section VII, "Logical Operations."

EXAMPLES:

```
100 IF A=B THEN 900
110 IF A+B >C THEN 910
120 IF A+B < C+E THEN 920
130 IF C>= D*E THEN 930
140 IF C9<= G*H THEN 940
150 IF P2#C9 THEN 950
160 IF J <> K THEN 950
```

MIN AND MAX OPERATORS

SYMBOLS: MIN
             MAX

The MIN or MAX operator selects the larger or smaller value of two expressions.

In the examples below, statement 110 selects and prints the larger value: since X = 7.5 and Y = 12.0, the value of Y is printed. The evaluation is made first, then the statement type (PRINT) is executed.

EXAMPLES:

```
 10 LET A=A9=P2=P5=C2=X=7.5
 20 LET B5=D8=Q1=Q4=Y=B=12.0
    .
    .
    .
 80 PRINT (A MIN 10)
 90 LET B=(A MIN 10)+100
100 IF (A MIN B5) > (C2 MIN D8) THEN 10
110 PRINT (X MAX Y)
120 IF (A9 MAX B) <= 5 THEN 150
```

Note: *It is not necessary for the novice to understand the nature of*
logical evaluation of relational operators at this point. The
comments below are for the experienced programmer.

Expressions using relational operators are logically evaluated, and assigned a value of "true" or "false" (the numeric value is 1 for true, and 0 for false).

When the = symbol is used in such a way that it might have either an assignment or a relational function, TSB assumes it is an assignment operator. For a description of the assignment statement using logical operators, see Section VII, "Logical Operations."

EXAMPLES:

```
100  IF  A=B  THEN  900
110  IF  A+B >C  THEN  910
120  IF  A+B < C+E  THEN  920
130  IF  C>= D*E  THEN  930
140  IF  C9<= G*H  THEN  940
150  IF  P2#C9  THEN  950
160  IF  J <> K  THEN  950
```

MIN AND MAX OPERATORS

SYMBOLS: MIN
             MAX

The MIN or MAX operator selects the larger or smaller value of two expressions.

In the examples below, statement 110 selects and prints the larger value: since X = 7.5 and Y = 12.0, the value of Y is printed. The evaluation is made first, then the statement type (PRINT) is executed.

EXAMPLES:

```
 10  LET  A=A9=P2=P5=C2=X=7.5
 20  LET  B5=D8=Q1=Q4=Y=B=12.0
     :
     :
 80  PRINT (A MIN 10)
 90  LET  B=(A MIN 10)+100
100  IF  (A MIN B5) > (C2 MIN D8) THEN 10
110  PRINT (X MAX Y)
120  IF  (A9 MAX B) <= 5 THEN 150
```

THE AND OPERATOR

SYMBOL: AND

The AND operator forms a logical conjunction between two expressions. If both are "true", the conjunction is "true"; if one or both are "false", the conjunction is "false".

> *Note:* *It is not necessary for the novice to understand how this operator works. The comments below are for experienced programmers.*

The numeric values are: "true" = 1, "false" = 0.

All non-zero values are "true". For example, statement 90 below would print either a 0 or a 1 (the logical value of the expression X and Y) rather than the actual numeric values of X and Y.

Control is transferred in an IF statement using AND, only when all parts of the AND conjunction are "true". For instance, example statement 80 requires four "true" conditions before control is transferred to statement 10.

See Section VII, "Logical Operations" for a more complete description of logical evaluation.

EXAMPLES:

```
60 IF A9<B1 AND C#5 THEN 100
70 IF T7#T AND J=27 THEN 150
80 IF P1 AND R>1 AND N AND V2 THEN 10
90 PRINT X AND Y
```

THE OR OPERATOR

SYMBOL: OR

The OR operator forms the logical disjunction of two expressions. If either or both of the expressions is "true", the OR disjunction is "true"; if both expressions are "false" the OR disjunction is "false".

The numeric values are: "true" = 1, "false" = 0.

All non-zero values are "true"; all zero values are "false".

Control is transferred in an IF statement using OR, when either or both of the two expressions evaluate to "true".

See Section VII, "Logical Operations" for a more complete description of logical evaluation.

EXAMPLES:

```
100  IF  A>1  OR  B<5  THEN  500
110  PRINT  C  OR  D
120  LET  D  =  X  OR  Y
130  IF  (X  AND  Y)  OR  (P  AND  Q)  THEN  600
```

THE NOT OPERATOR

SYMBOL: NOT

The NOT operator logically evaluates the complement of a given expression.

Note: It is not necessary for the novice to understand how this operator works. The comments below are intended for experienced experienced programmers.

If A = 0, then NOT A = 1; if A has a non-zero value, NOT A = 0.

The numeric values are: "true" = 1, "false" = 0; for example, statement 65 below would print "1", since the expression NOT (X AND Y) is "true".

Note that the logical specifications of an expression may be changed by evaluating the complement. In statement 35 below, if A equals zero, the evaluation would be "true" (1); since A has a numeric value of 0, it has a logical value of "false", making NOT A "true".

See Section VII, "Logical Operations" for a more complete description of logical evaluation.

EXAMPLES:

```
30 LET X = Y = 0
35 IF NOT A THEN 300
45 IF (NOT C) AND A THEN 400
55 LET B5 = NOT P
65 PRINT NOT (X AND Y)
70 IF NOT (A=B) THEN 500
```

EXECUTION ORDER OF PRECEDENCE

The order of performing operations follows:

↑	*highest precedence*
NOT unary + unary −	
* /	
+ −	
MIN MAX	
Relational Operators	
AND	
OR	*lowest precedence*

If two operators in an expression are on the same level, the order of execution is left to right within the statement.

5 + 6*7	is evaluated as:	5 + (6x7)
7/14*2/5	is evaluated as:	$\dfrac{(7/14)x2}{5}$

Unary + and − may be used; parentheses are assumed by TSB. For example:

A++B	is evaluated as:	A+(+B)
C+−D	is evaluated as:	C+(−D)

Leading unary plus signs are omitted from output resulting from program execution, but remain in program listings.

A MIN B MAX C MIN D is evaluated as:

((A MIN B) MAX C) MIN D

Operations enclosed in parentheses are performed before any operations outside the parentheses. When parentheses are nested, operations within the innermost pair of parentheses are performed first.

STATEMENTS

Be sure you know the difference between statements and commands.

Statements are instructions to the system. They are contained in numbered lines within a program, and execute in the order of their line numbers. Statements cannot be executed without running a program. They tell the system what to do while a program is running.

Commands are also instructions. They are executed immediately, do not have line numbers, and may not be used in a program. They are used to manipulate programs, and for utility purposes, such as logging on and off.

Here are some examples mentioned in Section I:

Statements	Commands
LET	HELLO
PRINT	BYE
INPUT	LIST

Do not attempt to memorize every detail in the "Statements" subsection; there is too much material to master in a single session. By experimenting with the sample programs, and attempting to write your own programs, you will learn more quickly than by memorizing.

The ASSIGNMENT Statement

GENERAL FORM:

statement number LET *variable = number or expression or string or variable. . .*

or

statement number variable = number or expression or string or variable. . .

The ASSIGNMENT statement used to assign or specify the value of a variable. The value may be an expression, a number, string or a variable of the same type.

Note that LET is an optional part of the assignment statement.

The assignment statement must contain:

1. The variable to be assigned a value.

2. The assignment operator, an = sign.

3. The number, expression or variable to be assigned to the variable.

Statement 20 in the example below shows the use of an assignment to give the same value (0) to several variables. This is a valuable feature for initializing variables in the beginning of a program.

EXAMPLES:

```
10  LET  A  =  5.02
20  X  =  Y7  =  Z  =  0
30  B9  =  5*  (X↑2)
40  LET  D  =  (3*C2↑N)/(A*(N/2))
```

REM Statement

GENERAL FORM: *statement number* REM *any remark or series of characters*

The REM statement allows insertion of a line of remarks or comment in the listing of a program.

The REM statement must be preceded by a line number. Any series of characters may follow REM.

REM lines are saved as part of a BASIC program, and printed when the program is listed or punched; however, they are ignored when the program is executing.

Remarks are easier to read if REM is followed by a punctuation mark, as in the example statements.

EXAMPLES:

```
10 REM--THIS IS AN EXAMPLE
20 REM: OF REM STATEMENTS
30 REM-----/////*****!!!!!
40 REM. STATEMENTS ARE NOT EXECUTED BY TSB
```

GO TO and Multibranch GO TO Statements

GENERAL FORM:

statement number GO TO *statement number*

statement number GO TO *expression* OF *sequence of statement numbers*

GO TO is used to transfer control to the statement specified.

GO TO *expression*. . .rounds the *expression* to an integer *n* and transfers control to the *n*th statement number following OF.

GO TO may be written: GOTO or GO TO.

GO TO must be followed by the statement number to which control is transferred, or *expression* OF, and a sequence of statement numbers.

GO TO overrides the normal execution sequence of statements in a program.

If there is no statement number corresponding to the value of the *expression*, the GO TO statement is ignored.

Useful for repeating a task infinitely, or "jumping" to (GOing TO) another part of a program if certain conditions are present.

GO TO should not be used to enter FOR-NEXT loops; doing so may produce unpredictable results or fatal errors.

EXAMPLES:

```
10  LET X = 20
    .
    .
    .
40  GO TO X+Y OF 410,420,430
50  GOTO 100
80  GOTO 10
90  GO TO N OF 100,150,180,190
```

IF . . . THEN Statement

GENERAL FORM: *statement number* IF *expression* THEN *statement number*

Control is transferred to a specified statement if a specified condition is true.

Sometimes described as a conditional transfer; GO TO is implied by IF . . . THEN, if the condition is true. In the example above, if X < 10, the message in statement 60 is printed.

Because numbers are not always represented exactly in the computer, the = operator should be used carefully in IF . . . THEN statements. < = , > =, etc. should be used in the IF expression, rather than = , whenever possible.

If the specified condition for transfer is not true, then the program will continue executing in sequence. In the example below, if X>=10, the message in statement 40 will be printed.

See "Logical Operations," Section VII for a more complete description of logical evaluation.

SAMPLE PROGRAM:

```
10  LET N = 10
20  READ X
30  IF X < N THEN 60
40  PRINT "X IS 10 OR OVER"
50  GO TO 80
60  PRINT "X IS LESS THAN 10"
70  GO TO 20
80  END
```

FOR...NEXT Statement

GENERAL FORM:

statement number FOR *simple variable* = *initial value* TO *final value*

      or

statement number FOR *simple variable* = *initial value* TO *final value* STEP *step value*

     •
     •
     •

statement number NEXT *simple variable*

> *Note:* *The same simple variable must be used in both the FOR and NEXT statements of a loop.*

The FOR...NEXT statements allow repetition of a group of statements within a program.

Initial value, final value and step value may be any expression.

The simple variable is assigned the value of the initial value; the value of the simple variable is increased by 1 (or by the optional step value) each time the loop executes. When the value of the simple variable passes the final value, control is transferred to the statement following the NEXT statement. If the loop is not executed (as in "FOR X = 1 TO 2 STEP-1"), none of the statements in the loop is executed, and control is transferred to the statement following the NEXT statement.

The initial value, final value and step value are all evaluated only when the loop is entered. Thus, the final value and step value do not depend on results of executing the loop. For instance, "FOR I = 1 TO 100 STEP I" generates a constant step size of 1 (first value for I), not a step size depending on the current value of I (1, 2, 4, 8, ...).

STEP and step value are optional. For further details on the STEP feature, see "FOR...NEXT with STEP" in Section III.

NESTING FOR. . .NEXT LOOPS: Multiple FOR. . .NEXT statement loops may be used in the same program; they may also be nested (placed inside one another). There are two important features of FOR. . .NEXT loops:

1.   FOR. . .NEXT statement loops may be nested.

```
                              ┌────── 10 FOR A1 = 1 TO 5
                              │  ┌─── 20 FOR B2 = N TO P
      Range of loop A1  ↗     │  │ ┌─ 30 FOR C3 = X TO Y STEP R
                              │  │ │      .
      Range of loop B2 ──→    │  │ │      .
                              │  │ │      .
      Range of loop C3 ──→    │  │ └─ 80 NEXT C3
                              │  └─── 90 NEXT B2
                              └────── 100 NEXT A1
```
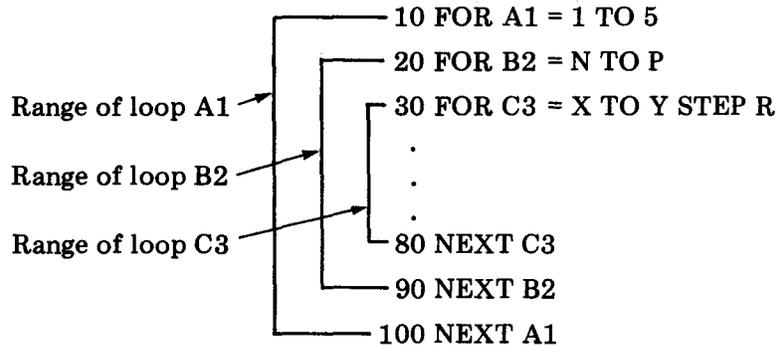
2.   The range of FOR. . .NEXT statement loops may not overlap. The loops in the example above are nested correctly. The following example shows improper and illegal nesting.

```
                              ┌────── 10 FOR I = 1 TO 5
                              │          .
                              │          .
                              │          .
                              │  ┌─── 30 FOR J = 1 TO N
      The range of loops │    │  │      .
      I and J overlap.   │    │  │      .
                              │  │      .
                              │  └─── 50 NEXT I
                              │          .
                              │          .
                              │          .
                              └────── 90 NEXT J
```
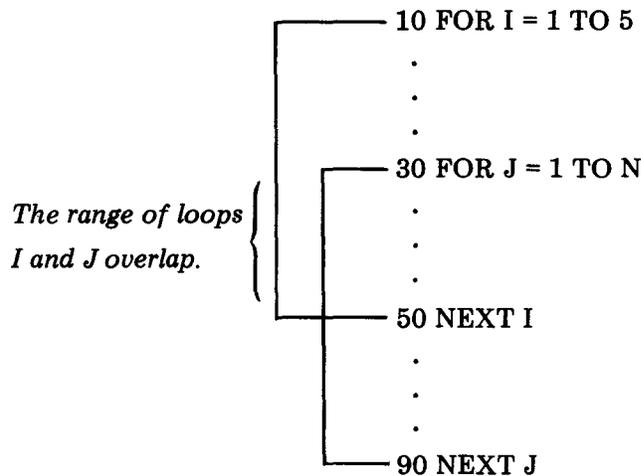
Sample Program with a variable number of loops:

```
40    PRINT "HOW MANY TIMES DO YOU WANT TO LOOP";
50    INPUT A
60    FOR J=1 TO A
70    PRINT "THIS IS LOOP";J
80    READ N1,N2,N3
90    PRINT "THESE DATA ITEMS WERE READ:"N1;N2;N3
100   PRINT "SUM ="N1+N2+N3
110   NEXT J
120   DATA 5,6,7,8,9,10,11,12
130   DATA 13,14,15,16,17,18,19,20,21
140   DATA 22,23,24,25,26,27,28,29,30
150   DATA 31,32,33,34
160   END
```

```
100  FOR  P1  =  1  TO  5
  .
  .
  .
170  NEXT  P1

120  FOR  R2  =  N  TO  X  STEP  1
  .
  .
  .
150  NEXT  R2

110  FOR  Q7  =  N  TO  X
130  FOR  S  =  1  TO  X  STEP  Y
140  NEXT  S
160  NEXT  Q7
```

READ, DATA, and RESTORE Statements

GENERAL FORM:

statement number READ *variable , variable, ...*

statement number DATA *number or string, number or string, ...*

statement number RESTORE

statement number RESTORE *statement number*

The READ statement instructs TSB to read an item from a DATA statement. READ statements require at least one DATA statement in the same program.

The DATA statement is used for specifying data in a program. The data is read in sequence from first to last DATA statements, and from left to right within the DATA statement. TSB maintains a data pointer as each item is read. Items in a DATA statement must be separated by commas. String and numeric data may be mixed. Programmers mixing string and numeric data may find the TYP function useful. See "The TYP Function", Section IV.

DATA statements may be placed anywhere in a program. The data items will be read in sequence as required. DATA statements do not execute; they merely specify data.

The RUN command automatically sets the data pointer to the first data item.

The RESTORE statement resets the data pointer to the first data item, allowing data to be re-read. RESTORE followed by a statement number resets the pointer to the first data item, beginning at the specified statement.

If you are not sure of the effects of READ, DATA, and RESTORE, try running the sample programs.

EXAMPLES:

Sample Program # 1

In this program, each data item is read only once. TSB keeps track of data with a pointer. When the READ statement is encountered for the first time, the pointer indicates that the first item in the DATA statement is to be read; then, the pointer is moved to the second item of data, and so on. After the loop has executed five times, the pointer remains at the end of the data list.

```
15   FOR I=1 TO 5
20   READ A
40   LET X=A↑2
45   PRINT A;" SQUARED =";X
50   NEXT I
95   DATA 5.24,6.75,30.8,72.65,89.72
99   END
```

Sample Program # 2

In this program, statements 55 through 80 are inserted into the program. The RESTORE statement resets the pointer to the first data item, allowing data to be re-read for the second portion of the program.

```
15   FOR I=1 TO 5
20   READ A
40   LET X=A↑2
45   PRINT A;" SQUARED =";X
50   NEXT I
55   RESTORE
60   FOR J=1 TO 5
65   READ B
70   LET Y=B↑4
75   PRINT B;" TO THE FOURTH POWER =";Y
80   NEXT J
95   DATA 5.24,6.75,30.8,72.65,89.72
99   END
```

INPUT Statement

GENERAL FORM: *statement number* INPUT *variable, variable, ...*

The INPUT statement requests data to be input from the user terminal for subsequent assignment to a variable. When the INPUT statement is encountered, the program comes to a halt and a question mark is printed on the user's terminal. The program does not continue execution until the input requirements are satisfied.

Only one question mark is printed for each INPUT statement. The statements:

```
10  INPUT  A,  B2,  C5,  D,  E,  F,  G
```

and

```
20  INPUT  X
```

each cause a single question mark to be printed. Note that the question mark generated by statement 10 requires seven input items, separated by commas, while that generated by statement 20 requires only a single input item.

The only way to stop a program when input is required is to enter C^C followed by a carriage return. Note that C^C aborts the program; it must be restarted with the RUN command.

RELEVANT DIAGNOSTICS: One of the following responses may be printed on the user terminal following user input:

??	indicates that more input is required to satisfy the INPUT statement.
???	indicates that TSB cannot decipher your input.
EXTRA INPUT-WARNING ONLY	indicates that extra input was entered; excess data items have been ignored; the program is continuing execution.

Sample Program:

```
5   FOR M=1 TO 2
10    INPUT A
20    INPUT A1,B2,C3,Z0,Z9,E5
30    PRINT "WHAT VALUE SHOULD BE ASSIGNED TO R";
40    INPUT R
50    PRINT A;A1;B2;C3;Z0;Z9;E5;"R=";R
60    NEXT M
70    END
```

------------------------------ RESULTS ------------------------------

```
RUN


?1
?2,3,4,5,6,7
WHAT VALUE SHOULD BE ASSIGNED TO R?27
 1     2     3     4     5     6     7     R= 27
?1.5
?2.5,3.5,4.5,6.,7.2
??8.1
WHAT VALUE SHOULD BE ASSIGNED TO R?-99
 1.5          2.5          3.5          4.5          6     7.2
 8.1          R=-99

DONE
```

PRINT Statement

GENERAL FORM:

statement number PRINT *expression , expression , . . .*

> *or*

statement number PRINT *"any text" ; expression ; . . .*

> *or*

statement number PRINT *"text" ; expression ; "text" , "text" ; . . .*

> *or*

statement number PRINT *any combination of text and/or expressions and/or* TAB, LIN, *and* SPA

> *or*

statement number PRINT

The PRINT statement causes the value(s) of the expression(s) to be output to the terminal device. In addition, it causes the terminal device to skip a line when used without an operand and causes text within quotes to be printed literally.

The terminal device may be a user terminal or the line printer.

Note the effects of , and ; on the output of the sample program. If a comma is used to separate PRINT operands, up to five fields will be printed per line. These five fields begin in columns 0, 15, 30, 45, and 60. If semicolon is used, up to twelve "packed" numeric fields will be output per line; the exact number depends on the size of each numeric field. If semicolons are used between text in quotes, it is possible to print a full 72 characters on a line.

A carriage return and linefeed are output after the execution of any PRINT statement unless the list of items to be printed is terminated by a comma or semicolon, in which case the next PRINT statement will begin on the same line.

Values output by PRINT statements are in one of four possible numeric formats, depending on the value. These values and their formats are:

Numeric Value Ranges	Output Format	Examples
$-999 \leqslant$ integer $\leqslant 999$	–dddbb	b733bb –214bb
$-32767 \leqslant$ integer $\leqslant -1000$ and $1000 \leqslant$ integer $\leqslant 37267$	–dddddbbb	–32767bbb –1000bbbb b1000bbbb b37267bbb
.000001 $\leqslant$ all other integers or real numbers $\leqslant 999999.5$	–ddddddbbbb (one d is a decimal point, trailing 0's are suppressed)	–131072.bbbb b131072.bbbb b14.6bbbbbbb –.003456bbbb
any number, n, where n $<$ .000001 or $999999.5 \leqslant$ n	–d.dddddE±ddbbb	–6.91120E+15bbb
d is one decimal digit – is the sign, if negative ± is the appropriate sign b is a space		

Insertion of the special functions TAB, SPA, and LIN into the output list provides carriage control:

TAB *(expression)*   Causes the carriage to move to the specified print column (0-71). No action is taken if the move would be to the left. The carriage moves to the beginning of the next line if expression >71. (To TAB beyond column 72 see PRINT USING statement.)

SPA *(expression)*   Causes carriage to skip specified number of spaces ("print that number of blanks"). A negative expression does nothing. If more spaces are requested than remain in the line, the carriage moves to the beginning of the next line.

LIN *(expression)*   Generates a *carriage return* and the specified number of *linefeeds*. If the expression is negative, then no *carriage return* is generated. LIN (0) produces a single *carriage return*.

0^c printed in a character string causes a *carriage return* to be output instead.

N^c printed in a character string causes a *linefeed* to be output instead.

The PRINT USING statement, which provides increased output formatting capabilities, is described in Section VIII.

Sample Program 1

```
10   FOR N=-5 TO 30
20   PRINT 2↑N;
30   NEXT N
40   END
```

- RESULTS -

```
RUN

 .03125       .0625       .125        .25         .5          1       2
4       8       16      32      64      128     256     512     1024        2048
4096        8192        16384       32768.          65536.          131072.         262144.
524288.         1.04858E+06     2.09715E+06     4.19430E+06     8.38861E+06
1.67772E+07     3.35544E+07     6.71089E+07     1.34218E+08     2.68435E+08
5.36871E+08     1.07374E+09
DONE
```

Sample Program 2

```
10    LET A=B=C=D=E=F=G=14
20    LET D1=E9=20
30    PRINT A,D1,B,C,E9
40    PRINT A/B,B/C/D1+E9
50    PRINT "NOTE THE POWER TO EVALUATE AN EXPRESSION AND PRINT "
60    PRINT "THE VALUE IN THE SAME STATEMENT."
70    PRINT
80    REM: "PRINT" WITH NO OPERAND CAUSES THE TELEPRINTER TO
81    REM: SKIP A LINE.
90    PRINT "'A' DIVIDED BY 'E9' =";A/E9
100   PRINT
110   PRINT "11111","22222","33333","AAAAA","BBBBB","CCCCC"
120   PRINT "11111";"22222";"33333";"AAAAA";"BBBBB";"CCCCC"
130   PRINT A,B,C,D,D1,E,F,E9,G
140   PRINT A;B;C;D;D1;E;F;E9;G
150   PRINT
160   PRINT TAB(8);"CARRIAGE";SPA(5);"CONTROL";LIN(2);"FUNCTIONS"
170   END
```

------------------------------- RESULTS ------------------------------------

RUN


```
 14              20              14              14              20
 1               20.05
NOTE THE POWER TO EVALUATE AN EXPRESSION AND PRINT
THE VALUE IN THE SAME STATEMENT.

'A' DIVIDED BY 'E9' = .7

11111           22222           33333           AAAAA           BBBBB
CCCCC
1111122222333 33AAAAABBBBBCCCCC
 14              14              14              14              20
 14              14              20              14
 14     14      14      14      20      14      14      20      14

         CARRIAGE        CONTROL

FUNCTIONS

DONE
```

END and STOP Statements

GENERAL FORM:

any statement number STOP
any statement number END
highest statement number in program END

Terminates execution of the program and returns control to TSB.

The highest numbered statement in the program must be an END statement.

END and STOP statements may be used in any portion of the program to terminate execution.

END and STOP have identical effects; the only difference is that the highest numbered statement in a program must be an END statement.

EXAMPLES:

```
200  IF  A  #  27.5  THEN  350

   .
   .
   .

300  STOP

   .
   .
   .

350  LET  A  =  27.5

   .
   .
   .

500  IF  B  #  A  THEN  9999

   .
   .
   .

550  PRINT  "B  =  A"
600  END
9999 END
```

SAMPLE PROGRAM

If you understand the effects of the statement types presented up to this point, skip to the "COMMANDS" section.

The sample program on the next two pages uses several BASIC statement types.

Running the program gives a good idea of the various effects of the PRINT statement on terminal device output. If you choose to run the program, you may save time by omitting the REM statements.

After running the program, compare your output with that shown under "RUNNING THE SAMPLE PROGRAM". If there is a difference, LIST your version and compare it with the one presented on the next two pages. Check your PRINT statements for commas and semicolons; they must be used carefully.

```
10    REMARK: "REMARK" OR "REM" IS USED TO INDICATE REMARKS OR
20    REMARK: COMMENTS THE USER WANTS TO INCLUDE IN THE TEXT
30    REM: OF HIS PROGRAM.  THE COMPUTER LISTS AND PUNCHES THE
40    REM: "REMARK" LINE, BUT DOES NOT EXECUTE IT.
50    REM: "PRINT" ALONE GENERATES A "RETURN" AND "LINEFEED".
60    PRINT
70    PRINT "THIS PROGRAM WILL AVERAGE ANY GROUP OF NUMBERS"
80    PRINT "YOU SPECIFY.  IT WILL ASK ALL NECESSARY QUESTIONS"
90    PRINT "AND GIVE INSTRUCTIONS.  PRESS THE RETURN KEY AFTER"
110   PRINT "YOU TYPE YOUR REPLY."
120   PRINT LIN(2)
140   REM: FIRST, ALL VARIABLES USED IN THE PROGRAM ARE IN-
150   REM: ITIALIZED TO ZERO (THEIR VALUE IS SET AT ZERO).
160   LET A=N=R1=S=0
180   REM: NOW THE USER WILL BE GIVEN A CHANCE TO SPECIFY HOW
190   REM: MANY NUMBERS HE WANTS TO AVERAGE.
200   PRINT "HOW MANY NUMBERS DO YOU WANT TO AVERAGE";
210   INPUT N
220   PRINT
230   PRINT "O.K., TYPE IN ONE OF THE ";N;"NUMBERS AFTER EACH"
240   PRINT "QUESTION MARK.  DON'T FORGET TO PRESS THE RETURN"
241   PRINT "KEY AFTER EACH NUMBER!!!"
250   PRINT LIN(2)
260   PRINT "NOW, LET'S BEGIN"
270   PRINT
280   PRINT
300   REM: "N" IS NOW USED TO SET UP A "FOR-NEXT" LOOP WHICH
310   REM: WILL READ 1 TO N NUMBERS AND KEEP A RUNNING TOTAL.
320   FOR I=1 TO N
330   INPUT A
340   LET S=S+A
350   NEXT I
360   REM: "I" IS A VARIABLE USED AS A COUNTER FOR THE NUMBER
370   REM: OF TIMES THE TASK SPECIFIED IN THE "FOR-NEXT" LOOP
380   REM: IS EXECUTED.  "A" IS A VARIABLE USED TO REPRESENT
400   REM: THE NUMBER TO BE AVERAGED.  THE VALUE OF "A" IS
410   REM: CHANGED EACH TIME THE USER INPUTS A NUMBER.
```

```
420    REM: "S" WAS CHOSEN AS THE VARIABLE TO REPRESENT THE
430    REM: SUM OF ALL NUMBER TO BE AVERAGED.  AFTER THE LOOP
440    REM: IS EXECUTED "N" TIMES, THE PROGRAM CONTINUES.
460    REM: A SUMMARY IS PRINTED FOR THE USER.
470    PRINT
480    PRINT
490    PRINT N;"NUMBERS WERE INPUT."
500    PRINT
510    PRINT "THEIR SUM IS:";S
520    PRINT
530    PRINT "THEIR AVERAGE IS:";S/N
540    PRINT
550    PRINT
570    REM: NOW THE USER WILL BE GIVEN THE OPTION OF QUITTING
580    REM: OR RESTARTING THE PROGRAM.
590    PRINT "DO YOU WANT TO AVERAGE ANOTHER GROUP OF NUMBERS?"
600    PRINT
610    PRINT "TYPE 1 IF YES, 0 IF NO"
620    PRINT "BE SURE TO PRESS THE RETURN KEY AFTER YOUR REPLY."
630    PRINT
640    PRINT "YOUR REPLY";
650    INPUT R1
660    IF R1=1 THEN 120
670    REM: THE FOLLOWING LINES ANTICIPATE A MISTAKE IN THE
671    REM: REPLY.
680    IF R1#0 THEN 700
690    GOTO 720
700    PRINT "TO REITERATE, YOU SHOULD TYPE 1 IF YES, 0 IF NO!!!"
710    GOTO 640
720    END
```

Running The Sample Program

```
RUN



THIS PROGRAM WILL AVERAGE ANY GROUP OF NUMBERS
YOU SPECIFY.  IT WILL ASK ALL NECESSARY QUESTIONS
AND GIVE INSTRUCTIONS.  PRESS THE RETURN KEY AFTER
YOU TYPE YOUR REPLY.



HOW MANY NUMBERS DO YOU WANT TO AVERAGE?5

O.K., TYPE IN ONE OF THE  5    NUMBERS AFTER EACH
QUESTION MARK.  DON'T FORGET TO PRESS THE RETURN
KEY AFTER EACH NUMBER!!!
```

```
NOW, LET'S BEGIN


?99
?87.6
?92.7
?79.5
?84


   5    NUMBERS WERE INPUT.

THEIR SUM IS: 442.8

THEIR AVERAGE IS: 88.56


DO YOU WANT TO AVERAGE ANOTHER GROUP OF NUMBERS?

TYPE 1 IF YES, 0 IF NO
BE SURE TO PRESS THE RETURN KEY AFTER YOUR REPLY.

YOUR REPLY?2
TO REITERATE, YOU SHOULD TYPE 1 IF YES, 0 IF NO!!!
YOUR REPLY?0

DONE
```

COMMANDS

Remember the difference between commands and statements (See "Statements" in this section).

Commands are direct instructions to the system, and are executed immediately. They are used to manipulate programs, and for utility purposes.

Note that all TSB commands may be abbreviated to their first three letters. If information is required or permitted after a command, a hyphen "–" must be included. For example, when **logging in:**

```
HEL-H200,SECCRECT
```

Do not try to memorize all of the details in the COMMANDS subsection. The various commands and their functions will become clear to you as you begin writing programs.

HELLO Command

GENERAL FORM:

HELLO-*idcode, password, terminal type*

HEL-*idcode, password*

The HELLO command is used to log on to the TSB system. The user's idcode and password are assigned by the system operator. The terminal type parameter informs TSB of the type of user terminal being logged on. Terminal type is specified as one numeric digit as follows:

> Not specified, or 0 = HP 2600A, HP 2749A/B, IBM 2741, or HP 2640A (default)
> 1 = Execuport 300 or Texas Instruments Silent 700
> 2 = ASR-37 or GE TermiNet 30
> 3 = GE TermiNet 300, GE TermiNet 1200, or HP 2762A/B
> 4 = Memorex 1240

Failure to specify the correct type number may result in a loss of characters.

Several users with the same idcode may be logged on to the computer simultaneously, using different terminals.

EXAMPLES:

```
HELLO-D007,POST C,2
HEL-Z123,TSB
HEL-A453,GEORGE,3
```

BYE Command

GENERAL FORM: BYE

The BYE command is used to log off the TSB system.

Entry of this command logically disconnects the user from the TSB system. Telephone connection is broken.

EXAMPLE:

```
BYE
009 MINUTES OF TERMINAL TIME
```

ECHO Command

GENERAL FORM:

ECHO-ON

    or

ECHO-OFF

The ECHO command allows use of half-duplex terminal.

Users with half-duplex terminal equipment must first log on, then type the ECHO-OFF command; then input and output becomes legible.

ECHO-ON returns a user to the full-duplex mode.

This command may be abbreviated to its first three letters.

EXAMPLES:

```
ECHO-OFF
ECH-ON
```

RUN Command

GENERAL FORM:

RUN
RUN- *statement number*

Entry of the RUN command starts execution of a program at the lowest numbered statement when used without specifying a statement number. It starts execution of a program at the specified statement when a statement number is used.

Note that when RUN- *statement number* is used, all statements before the specified statement will be skipped. Variables defined in statements which have been skipped are therefore considered to be undefined by TSB, and may not be used until they are defined in an assignment, INPUT, ENTER, READ, or LET statement.

A running program may be terminated by pressing the *break* key; or, to terminate a running program at some point when input is required, type C^C.

EXAMPLE:

RUN
OR
RUN- 300

LIST Command

GENERAL FORM:

LIST
LIST- *statement number*
LIST- *statement number* , *statement number*
LIST- , *statement number*
LIST- *statement number* , *statement number* ,P
LIST- P
LIST- *statement number* , P
LIST- , *statement number* , P

This command produces a listing of all statements in a program (in statement number sequence) when no statement number is specified.

When a statement number is specified, the listing begins at that statement.

When a second statement number is specified, listing ends with that statement.

When a "," and a statement number appear, listing starts at the beginning and ends with the specified statement.

When "P" is specified, the listing is spaced for cutting into 11-inch sheets sized for binding or filing. "P" must be the final parameter, and must be preceded by a comma if it follows other parameters.

A listing may be stopped by pressing the *break* key. Library programs designated "RUN ONLY" (protected) by the System Master or Group Master cannot be listed. LIST may be abbreviated to its first three letters.

EXAMPLE:

```
LIST
LIST-100
LIST-100, 200
```

SCRATCH Command

GENERAL FORM:

SCRATCH

   or

SCR

This command deletes (from memory) the program currently being accessed from the user terminal. The user's work area is cleared including the program name.

Scratched programs are not recoverable. For information about saving programs on paper tape or in your personal library, see the NAME and SAVE commands in Section III, and the PUNCH command in this section.

EXAMPLE:

```
SCRATCH
OR
SCR
```

RENUMBER Command

GENERAL FORM:

REN

or

REN-*number assigned to first statement*

or

REN-*number assigned to first statement , interval between new statement numbers*

or

REN-*number assigned to first statement, interval between new statement numbers, starting statement number, ending statement number*

or

REN-*number assigned to first statement, interval between new statement numbers, starting statement number*

The RENUMBER command is used to renumber statements in the current program.

Statement numbers referenced within GO TO, GOSUB . . . Return, IF . . . THEN, RESTORE, and PRINT USING statements are automatically replaced with the appropriate new number.

Starting statement number and ending statement number refer to line numbers in the original program at which the renumbering is to start and end.

If ending statement number is not specified, it is assumed to be the last statement in the program.

If starting statement number is not specified, it is assumed to be the first statement in the program.

If both starting and ending statement numbers are omitted, the entire program is renumbered.

If no interval is specified, the new numbers are spaced at intervals of 10, from the beginning statement.

If no parameters are stated, the entire program is renumbered starting with statement 10 at intervals of 10.

RENUMBER can not be used to change the *order* of statements in a program.

Any parameter may be omitted, but all parameters following it must also be omitted.

EXAMPLES:

```
RENUMBER
REN
REN-100
REN-10,  1
REN-20,  50
REN-10,  10,  50,  100
```

PUNCH and XPUNCH Commands

GENERAL FORM:

PUN

PUN- *statement number*

PUN- *statement number* , *statement number*

PUN- *statement number* , *statement number* , P

PUN- , *statement number*

PUN- P

XPU

XPU- *statement number*

XPU- *statement number* , *statement number*

XPU- *statement number* , *statement number* , P

XPU- , *statement number*

XPU- P

These commands punch a program onto paper tape if the user terminal has a paper tape punch;
also punches the program name, and leading and trailing feed holes on the tape; lists the program
as it is punched. Punching can begin and/or end at specified statements; "P" provides the pagination
option (see LIST).

If the user terminal is not equipped with a paper tape reader/punch, only a listing is produced.

Remember to press the paper tape punch "ON" button before pressing the *return* after PUNCH.

XPUNCH produces the same results as punch, but adds an X-OFF character at the end of each line (before *return linefeed*) to enable other BASIC programs to read the paper tape as data. (See Appendix B.)

EXAMPLES:

```
PUNCH
PUN- 100, 200
PUN- 100, 200, P
PUN-65
PUN-, 300
XPUNCH
XPU- 65, P
XPU- P
```

TAPE Command

GENERAL FORM:

TAPE

 or

TAP

The TAPE command informs the system that following input (a group of BASIC statements) is from paper tape.

TAPE suppresses any diagnostic messages which are generated by input errors, as well as the automatic *linefeed* after *return*. The KEY command, KEY *return*, (or any other command except HELLO, BYE, or SCRATCH), causes the diagnostic messages to be output to the user terminal, ending the TAPE mode.

TSB responds to the TAPE command with a *linefeed* after which the user may activate the tape reader START switch.

This command is illegal if entered from an IBM 2741 Communication Terminal.

EXAMPLES:

```
TAPE
TAP
```

KEY Command

GENERAL FORM: KEY

The KEY command informs the system that following input will be from the user terminal keyboard; used only after a TAPE (paper tape input) sequence is complete; causes error messages suppressed by TAPE to be output to the terminal.

Any command (except HELLO, BYE, or SCRATCH), followed by a *return* has the same effect as KEY. Commands substituted for KEY in this manner are not executed if diagnostic messages indicating syntax errors in BASIC statements were generated during tape input.

EXAMPLE:

KEY

LPRINTER Command

GENERAL FORM:

LPRINTER [-*character string*]

                or

LPR [-*character string*]

The LPRINTER command requests that system designate the line printer as the user's output device.

If successful, a *linefeed* occurs at the user's terminal. The line printer performs a page eject and the *character string*, if specified, is printed. The *character string* may be 1-132 characters in length.

Once assigned to a user, the line printer is designated as that user's output device and output generated by the next entered command is printed on the line printer. After the entered command is executed, or when the program ends (or is terminated by the user) line printer control is returned to the system and the message *LP FREE* is displayed on the user's terminal.

Whenever an L^c is encountered in output to the line printer, the line printer performs a page eject. (For example, whenever L^c is embedded in a LPRINTER command *character string*, or in a program listing or program output.)

LINE PRINTER CARRIAGE CONTROL: The line printer connected to the system may be one of the following:

Model No.	Carriage Width	Print Speed
HP 2610A	132 columns	200 lpm
HP 2613A	136 columns	300 lpm
HP 2614A	132 columns	600 lpm
HP 2617A	136 columns	600 lpm
HP 2618A	132 columns	1200/1800 lpm
HP 2767A	80 columns	300 lpm
HP 2778A	120/132 columns	300 lpm

Data designated for line printer output should not exceed the carriage width of the line printer because overprinting or truncation occurs (depending on the model used).

During execution of PRINT or PRINT USING statements (with the line printer designated as the output device), those special characters which normally cause a *return* cause a line print with no paper advance. Similarly, characters which normally cause a *linefeed* or *return/linefeed* cause a line print with paper advance.

All string characters except control characters and the DEL character are printed on the line printer. Lowercase characters are printed as uppercase characters on line printers supporting the 64-character ASCII subset. Support of the 96-character ASCII subset can be obtained as an option for the 2610A or 2614A; both lowercase and uppercase characters are printed if this option is selected.

During execution of an ENTER statement (with the line printer designated as the output device), an asterisk is displayed on the user's terminal to signal that data input is expected. Similarly, during execution of an INPUT statement, an asterisk followed by a question mark is displayed on the user's terminal. In addition, ANNOUNCE command messages from the system operator are not sent to a user who has the line printer assigned as his output device.

When the line printer is designated as the output device, execution errors which result only in a warning message cause the message to be printed on the line printer; execution with line printer output continues. Errors fatal to execution result in loss of line printer control; execution halts and control returns to the user's terminal. In addition, a system power failure results in loss of line printer control.

CONTROL CHARACTERS: After line printer control has been established with the LPRINTER command, mixing of output devices is permitted through use of two control characters:

Q^c Suspends line printer output and routes subsequent output to the user's terminal

W^c Resumes line printer output

These control characters must be entered before typing a command or when entering data during execution of an INPUT or ENTER statement.

Messages	Meaning
LP BUSY	Displayed on user's terminal in response to LPRINTER command if line printer is currently assigned to another user.
LP DOWN	Displayed on user's terminal if line printer becomes disabled during output (printer power failure, out-of-paper condition, etc.). Line printer output resumes after problem is corrected.
LP NOT AVAILABLE	Displayed on user's terminal in response to LPRINTER command if line printer is not connected to system (notify system operator) or if line printer power is not on.
LP FREE	Displayed on user's terminal when command or program ends or when program is terminated by the user.
ILLEGAL FORMAT	Displayed on user's terminal if the string specified in the LPRINTER command exceeds 132 characters.

EXAMPLE:

A user logs on the TSB System from an ASR-37 user terminal with idcode B003 and password PSWD. He creates a BASIC program, requests control of the line printer, and enters the RUN command.

The user terminal display appears:

```
HEL-B003,PSWD,2            User enters HELLO command

READY                      System response
10 FOR X = 1 TO 10         User creates BASIC program
15 PRINT "TESTLP ";            statement
20 NEXT X                        by
25 PRINT "END OF TEST"        statement
30 END                        until complete
LPR-PRINTER TEST           User enters LPRINTER command
                              (Response:  CR/LF)
RUN                        User enters RUN command
LP FREE                    System response on completion of RUN

DONE
```

The line printer output appears:

:
:

(PAGE EJECT)

:
:

PRINTER TEST
TESTLP TESTLP TESTLP TESTLP TESTLP TESTLP TESTLP TESTLP TESTLP TESTLP
END OF TEST

TIME Command

GENERAL FORM: TIME

This command causes TSB to inform the user of terminal time used since log on, and total time used for the account.

Time used by each idcode is recorded automatically by TSB. The system operator controls the accounting system. Consult your system operator for information about your system's accounting methods.

EXAMPLE:

TIME
CONSOLE TIME = 12 MINUTES.   TOTAL TIME = 1193 MINUTES.

MESSAGE Command

GENERAL FORM:

MESSAGE-*character string*

            or

MES-*character string*

The MESSAGE command sends a character string to the system operator, preceded by the user's port number.

Can be used to request information from the system operator, or to have programs sanctified, desecrated, copied, bestowed, or loaded from or dumped to magnetic tape (see Appendix D).

If the system operator's message storage area is full, the message:

CONSOLE BUSY

will be printed on the user's terminal, indicating that the message has not been sent and should be entered again.

The maximum message size is 68 characters; additional characters are ignored.

EXAMPLE:

MES-PLEASE SANCTIFY PROGRAM "DUMMY", USER J122.

Break Key

GENERAL FORM: *break* (Press the *break* key.)

Pressing the *break* key terminates a program being executed (RUN) unless the *break* capability has been disabled (see "BRK Function" in Section III), or terminates the execution of any command producing output.

Pressing the *break* key signals the computer to terminate a program, producing the message: STOP.

Depending on the type of terminal, this key may appear on the keyboard as INTRPT, BRK, ATTN, INTERRUPT, etc.

When *break* is pressed during a listing, the message STOP is output.

Pressing *break* will not terminate the program if it is awaiting input from the keyboard while executing an INPUT or ENTER statement. In this case the only means of ending the program is typing:

C^C

which produces the DONE message.

break will not delete a program except the HELLO program, or a program which was chained to from HELLO. Type RUN to restart the program. (See also COM, Section III.)

SECTION III
Advanced BASIC

This section describes more sophisticated capabilities of BASIC.

The experienced programmer has the option of skipping the "Vocabulary" subsection, and briefly reviewing the commands and functions presented here. The most important features of the TSB system - - files, matrices, and strings are explained in the next three sections.

The inexperienced programmer need not spend a great deal of time on programmer-defined and standard functions. They are shortcuts, and some programming experience is necessary before their specifications become apparent.

TERM: ROUTINE

Defined: A sequence of program statements which produces a certain result.

Routines are used for frequently performed operations. Using routines saves the programmer the work of defining an operation each time he uses it, and saves computer memory space.

A routine may also be called a program, subroutine, or sub-program.

The task performed by a routine is defined by the programmer.

Examples of routines and subroutines are given in this section.

TERM: ARRAY

Defined: An ordered collection of numeric data. A single program can have up to about 4900 total array elements (numeric values).

An array variable is any single alphabetic character, A through Z. Subscripted variables define elements in an array. A_1, written A(1), is the first element in the single-dimensioned array called A. In array A below, the value of A(1) is 5.0:

Array A

Element	Value
1	5.0
2	3.2
3	1.1
4	0.3

Double subscripts are used to define elements in two-dimensioned arrays, referring to a row and column position within an array. Element B(1,3) in array B has the value appearing in the first row, third column. In this case the value of B(1,3) is 4.

Array B

	Column 1	Column 2	Column 3
Row 1	6	5	4
Row 2	3	2	1
Row 3	0	9	8

Array B is a three-by-three array. Arrays need not be square.

If a one-dimensional array has more than ten elements or a two-dimensional array (matrix) has more than 10 rows and 10 columns, a DIM (dimension) statement is required. The DIM statement is described in Section V, which covers matrices; a matrix is a rectangular array of elements subject to mathematical operations according to specified rules.

TERM: STRING

Defined: 0 to 72 characters enclosed by quotation marks.

Quotation marks may not be used within a string, except when the string is input using an ENTER statement, described later in this section.

Sample strings: "ANY CHARACTERS!?*/---"
                   "TEXT 1234567 . . ."

TERM: FUNCTION

Defined: The mathematical relationship between two variables (X and Y for example) such that for each value of X there is one and only one value of Y.

The independent variable is called an argument; the dependent variable is the function value. For instance, in the statement:

        100 LET Y = SQR(X)

X is the argument; the function value is the square root of X; and Y takes the value of the positive root.

TERM: WORD

Defined: The equivalent of approximately two BASIC characters or one-half of a number.

The term "word" is used to define the basic unit of computer storage. The TSB system operates on computers having a word structure of 16 binary bits. Each *character* in BASIC occupies 8 bits of computer storage; each *number* (when used in computation) occupies 32 bits. A *numeral* that appears in a literal string (Section VI) is not used for computation, and is considered to be a *character*.

Therefore, *two characters* will fit into one computer word, while *one number* will require two computer words. Actually, the TSB system requires a few additional computer words of storage, so programs and files will require slightly more storage than one word for each two characters or two words for each number. Each user has a working area of 10,000 words. The user need not normally be concerned about computer words.

STORING AND DELETING PROGRAMS

Up to this point manipulation of programs has been limited to the "current" program, that is, the program being written or run at the moment. The only means of saving a program introduced thus far is the PUNCH command.

The commands on the following pages allow the user to create his own library of programs on the Time Shared BASIC system. Library programs are easily accessed, modified, and run.

The experienced programmer need only review the commands briefly — they do what their names imply: NAME, SAVE, etc.

A word of caution for the inexperienced programmer: it is wise to make a "hard" copy (on paper tape) of programs you wish to use frequently. Although it is easy and convenient to store programs "on-system", you will make mistakes as you learn, and may accidentally delete programs. It is much less time consuming to enter a program from paper tape than to rewrite it!

LENGTH Command

GENERAL FORM: LEN

The LENGTH command causes TSB to print the number of words in the program currently being accessed from the terminal. This is the amount of "storage space" needed to SAVE the program.

Each user has a working space of over 10,000 words (20,000 characters or 5,000 numbers). LENGTH is a useful check on total program length when writing Long programs. During execution, programs have temporary tables, buffers, etc. which require additional storage space. This larger total length is not permitted to exceed the user's working area. See "Memory Allocation by a User," Section IX.

EXAMPLES:


```
LENGTH
3172 WORDS
LEN
0151 WORDS
```

NAME Command

GENERAL FORM:

NAME-*Program name of 1 to 6 characters*

or

NAM-*Program name of 1 to 6 characters*

This command assigns a name to the program currently being accessed from the teleprinter.

The first character of the program named may not be $ or *. These symbols are used to access the System Library ($) and the Group Library (*). The comma (,) and quote mark (") may not be used in the name of a program.

The program name must be used in certain TSB operations (see the SAVE, CSAVE, KILL, GET, and APPEND commands in this section).

> *Note:* *If NAME- is entered with no program name or if the hyphen is omitted, the program cannot be stored with the SAVE or CSAVE commands.*

EXAMPLES:

```
NAME-PROG.1
NAM-ADDER
NAM-MYPROG
```

SAVE AND CSAVE Commands

GENERAL FORM: CSAVE or CSA

SAVE or SAV

These commands are used to save a copy of the current program in the user's private library. (CSAVE stores the program in semi-compiled form so that it will CHAIN more quickly. See CHAIN.)

A program must be named before it can be saved. (See NAME, this section.)

No two programs in a user's library may have the same name. The procedure for saving a changed version of a program is as follows (the program name is SAMPLE):

KILL-SAMPLE   (Deletes the stored version)
NAME-SAMPLE   (Names the current program)
SAVE   (Saves the current program, named SAMPLE)

For instructions on opening a file, see Section IV, "FILES."

EXAMPLES:

```
SAVE
SAV
CSA
```

GET, GET-$, and GET-* Commands

GENERAL FORM

GET- *name of a program in user's library*

GET-$ *name of system library program*

GET-* *name of group library program*

GET- retrieves the specified program, making it the program currently accessed from the user's terminal.

GET-$ retrieves the specified program from the system library, making it the program currently accessed from the user's terminal.

GET-* retrieves the specified program from the group library.

GET- performs an implicit SCRATCH. The program that was the current program prior to using GET- can not be recovered from the system unless it was previously saved with either SAVE or CSAVE.

For more information on public library programs, see "Library" and "Group" in this section.

EXAMPLES:

```
GET-MYPROG
GET-$PUBLIC
GET-$NAMES
GET-*DATES
```

KILL Command

GENERAL FORM:

KILL-*program or file to be deleted*

          or

KIL-*program or file to be deleted*

This KILL command deletes the specified program or file from the user's library. (Does not delete the program currently being accessed from the user's terminal, even if it has the same name.)

> **Caution:** Files have only one version, the stored one. A killed
>   file is not recoverable.

A file may not be killed while it is being accessed by another user.

KILL should be used carefully, as the killed program can not be recovered from the system unless the killed program was also the current program.

The SCRATCH command deletes the program currently being accessed from the user terminal while KILL deletes a program or file stored on-system. The stored and current versions of a program occupy separate places in the system. They may differ in content, even though they have the same name.

The sequence of commands for changing and storing a program named PROG** is:

        GET-PROG** (Retrieves the program.)
        (make changes)
        KILL-PROG** (Deletes the stored version.)
        SAVE   (Saves the current version.)

EXAMPLE:


```
KILL-PROG12
KIL-EXMPLE
KIL-FILE10
```

APPEND Command

GENERAL FORM:

APPEND-*program name*

or

APP-*program name*

or

APP-$*system library program name*

or

APP-**group library program*

This command retrieves the named program from the user's own library, or the group or public libraries and appends it (attaches it) to the program currently being accessed from the user's **terminal.**

The lowest statement number of the appended program must be greater than the highest statement number of the current program.

Programs saved by a CSAVE command may not be referenced in an APPEND command.

> Caution: If an appended public library program is "run-only",
> the entire program to which it is appended becomes
> "run-only". ("Run-only" programs may not be
> listed, punched, or saved.)

The $ preceding system library program names is needed to append them; the * is needed to append group library programs. For details, see "Library" in this section.

EXAMPLES:

```
APPEND-MYPROG
APP-MYPROG
APPEND-$PUBLIC
APP-$SYSLIB
APP-*GPROG
```

DELETE Command

GENERAL FORM:

DEL-*statement number at which deletion starts*

or

DEL-*statement no. at which deletion starts , statement no. at which deletion ends*

DEL-*statement number* erases the current program statements after and including the specified statement. DEL-1 has the same effect as SCRATCH, except that DEL-1 does not remove the protected-program state, or the program name.

DEL-*statement number, statement number* deletes all statements in the current program between and including the specified statements. If both statement numbers are the same, only that statement is deleted.

It is sometimes useful to save or punch the original version of a program which is being modified, before using the DELETE statement.

Deleted statements are not recoverable (unless a copy of the program in the user swap area also resides on the disc).

EXAMPLES:

```
DELETE-27
DEL-27, 50
DEL-27,27
```

LIBRARY, GROUP, and CATALOG Commands

GENERAL FORMS:

LIBRARY

or

LIB

GROUP

or

GRO

CATALOG

or

CAT

These commands are used to print an alphabetic listing of programs and files stored by the system. LIBRARY or LIB produces a list of *system* programs and files. GROUP or GRO produces a list of *group* programs and files. CATALOG or CAT produces a list of programs and files stored in the user's own program library.

In the examples below, the code letters preceding LENGTH indicate:

> F - the entry is a file.
>
> C - the entry is a program in semi-compiled form. If neither a C nor an F appears, the entry is a *program*.
>
> P - the entry is "protected," may be either a program or a file.
>
> S - the entry is "sanctified," may be either a program or a file. (See Appendix D.)

Code letters may be combined as in the first entry, AAA in the LIBRARY listing.

Length is given in words for programs, records for files.

Protected system or group *programs* may be run but not listed, saved or punched. Protected system or group *files* may not be accessed by other users. A user's own programs may not be protected, but may be sanctified by the operator.

Each user has access to the three libraries described. He has complete control over his own library, using any of the commands used to store, delete, or retrieve programs and files.

Each user is part of a group, all having IDcodes with the same letter and same first digit. The user whose IDcode ends in 00 is the group librarian, or Group Master. The Group Master is responsible for maintaining the group library, entering and deleting programs in the same manner as the System Master controls the system library.

The system library is under the control of the System Master, user A000. Only the System Master (actually any user with access to the password for IDcode A000) can enter programs or files into the system library, or delete programs and files from the system library.

The System Master and all Group Masters have the responsibility of controlling access to their libraries. Regular users can not make entries to, deletions from, or changes to either the system library or their group library. The System Master and all Group Masters have access to special commands called PROTECT, which makes specified programs available on a run-only basis and files unavailable to regular users, and UNPROTECT, which reverses the procedure. These special commands are described in the 20854A Timeshared BASIC/2000, Level F system operator's manual, part number 02000-90074.

A user can call a program from the system library by typing GET-$, followed by the program name exactly as it appears in the LIBRARY, or append the program by typing APP-$ followed by the program name. GET-* and APP-* are used to access group programs.

Files are accessed with the FILES statement, described in Section IV.

Any of these listings may be terminated by pressing the *break* key.

The system prints an error message if the user attempts to access a non-existent program, list or punch or save a protected program, or GET or APPEND a file.

EXAMPLES:

LIBRARY

NAME		LENGTH	NAME		LENGTH	NAME		LENGTH	NAME		LENGTH
AAA	FPS	2	AB	F	230	BAA	F	2	BAB	P	13
BAC		6	BAD	C	18	BB	F	46	BBA	F	2
BBB	F	46	BFILE	F	128	BUDGE		12	BUDGET		3431
BUDGEU		12	C	F	31	C.R	S	1220	CB	F	230
CC	F	31	CCC	F	31	D	F	100	F1	F	64
FFF	F	34	GARY1		95	GARY2		83	GARY3		188
GOGO	P	151	G5	F	128	STRING	F	1	XY	F	256

GROUP

NAME		LENGTH	NAME		LENGTH	NAME		LENGTH	NAME		LENGTH
B	F	30	B1	F	128	B2	F	128	BLOCK2	F	128
CAICAL		4004	CALC	C	4081	MBLOCK		1655	SP1	F	400

CATALOG

NAME		LENGTH	NAME		LENGTH	NAME		LENGTH	NAME		LENGTH
BLOCK2	F	128	CHECK	C	55	SP1	F	800	TEST		3

SUBROUTINES AND FUNCTIONS

The following pages show TSB features useful for repetitive operations — subroutines, programmer-defined and standard functions.

The programmer-controlled features, such as multibranch GOSUB's, FOR . . . NEXT with STEP, and DEF FN become more useful as the user gains experience, and learns to use them as shortcuts.

Standard mathematical and trigonometric functions are convenient timesavers for programmers at any level. They are treated as numeric expressions by TSB.

The utility functions TAB, SPA, LIN, SGN, TYP, and LEN also become more valuable with experience. They are used to control or monitor the handling of data by TSB, rather than for performing mathematical chores.

GOSUB. . .RETURN Statement

GENERAL FORM:

statement number GOSUB *statement number starting subroutine*

.
.
.

statement number RETURN

The GOSUB statement transfers control to the specified statement number.

The RETURN statement transfers control to the statement following the GOSUB statement which transferred control.

GOSUB . . . RETURN eliminates the need to repeat frequently used groups of statements in a **program**.

The portion of the program to which control is transferred must end with a RETURN statement.

RETURN statements may be used at any desired exit point in a subroutine. There may be more than one RETURN per GOSUB.

Variables have the same meaning as in the main program.

EXAMPLE:

```
50  READ  A2
60  IF  A2<100  THEN  80
70  GOSUB  400
 .
 .
 .
380  STOP   (STOP frequently precedes the first statement of a subroutine,
                  to prevent accidental entry.)
390  REM--THIS  SUBROUTINE  ASKS  FOR  A  1  OR  0  REPLY.
400  PRINT  "A2  IS>100"
410  PRINT  "DO  YOU  WANT  TO  CONTINUE";
420  INPUT  N
430  IF  N  #0  THEN  450
440  LET  A2  =  0
450  RETURN
 .
 .
 .
600  END
```

Multibranch GOSUB Statement

GENERAL FORM:

statement number GOSUB *expression* OF *sequence of statement numbers . . .*

GOSUB *expression* rounds the *expression* to an integer *n* and transfers control to the *n*th statement number following OF.

Subroutines should be exited only with a RETURN statement.

The *expression* indicates which of the specified subroutines will be executed. For example, statement 20, above transfers control to the subroutine beginning with statement 300. The *expression* specifies which statement in the sequence of five statements is used as the starting one in the subroutine.

The *expression* is evaluated as an integer. Non-integer values are rounded to the nearest integer.

If the *expression* evaluates to a number greater than the number of statements specified, or less than 1, the GOSUB is ignored.

Statement numbers in the sequence following OF must be separated by commas.

EXAMPLE:

```
 50  READ  A2
 60  IF  A2<100  THEN  80
 70  GOSUB  400
  .
  .
  .
380  STOP   (STOP frequently precedes the first statement of a subroutine,
                 to prevent accidental entry.)
390  REM--THIS  SUBROUTINE  ASKS  FOR  A  1  OR  0  REPLY.
400  PRINT  "A2  IS>100"
410  PRINT  "DO  YOU  WANT  TO  CONTINUE";
420  INPUT  N
430  IF  N  #0  THEN  450
440  LET  A2  =  0
450  RETURN
  .
  .
  .
600  END
```

Multibranch GOSUB Statement

GENERAL FORM:

statement number GOSUB *expression* OF *sequence of statement numbers . . .*

GOSUB *expression* rounds the *expression* to an integer *n* and transfers control to the *n*th statement number following OF.

Subroutines should be exited only with a RETURN statement.

The *expression* indicates which of the specified subroutines will be executed. For example, statement 20, above transfers control to the subroutine beginning with statement 300. The *expression* specifies which statement in the sequence of five statements is used as the starting one in the subroutine.

The *expression* is evaluated as an integer. Non-integer values are rounded to the nearest integer.

If the *expression* evaluates to a number greater than the number of statements specified, or less than 1, the GOSUB is ignored.

Statement numbers in the sequence following OF must be separated by commas.

EXAMPLES:

```
20 GOSUB 3 OF 100,200,300,400,500
60 GOSUB N+1 OF 200,210,220
70 GOSUB N OF 80,180,280,380,480,580
```

Nesting GOSUB Statements

Nested GOSUB . . . RETURN statements allow selective use of subroutines within subroutines.

GOSUB statements may be nested logically to a level of nine. More than nine exits without a return may cause an error message.

RETURN statements may be used at any desired exit point in a subroutine. Note, however, that nested subroutines are exited in the order in which they were entered. For example, if subroutine 250 (below) is entered from subroutine 200, 250 is exited before subroutine 200.

EXAMPLES:

```
100 GOSUB 200
  .
  .
200 LET A = R2/7
210 IF A THEN 230
220 GOSUB 250
  .
  .
250 IF A>B THEN 270
260 RETURN
270 GOSUB 600
  .
  .
```

FOR...NEXT with STEP Statement

GENERAL FORM:

statement number FOR *simple variable* = *expression* TO *expression* STEP *expression*

This statement allows the user to specify the size of the increment of the FOR variable.

The step size need not be an integer. For instance,

100 FOR N = 1 to 2 STEP .01

is a valid statement which produces approximately 100 loop executions, incrementing N by .01 each time. Since no binary computer represents all decimal numbers exactly, round-off errors may increase or decrease the number of steps when a non-integer step size is used.

A step size of 1 is assumed if STEP is omitted from a FOR statement.

A negative step size may be used, as shown in statement 40 below.

EXAMPLES:

```
20 FOR 15 = 1 TO 20 STEP 2
40 FOR N2 = 0 TO -10 STEP -2
80 FOR P = 1 TO N STEP R
90 FOR X = N TO W STEP (N↑2-V)
   .
   .
   .
```

DEF FN Statement

GENERAL FORM:

statement no. DEF FN *single letter A to Z (simple var.)* = *expression*

This command allows the programmer to define functions.

The simple variable is a "dummy" variable whose purpose is to indicate where the actual argument of the function is used in the defining expression. After a function has been defined, the value of that function is referenced whenever the function is used by the programmer. For example, in this sequence M is a dummy variable:

```
10  LET Y = 100
20  DEF FNA (M) = M/10
30  PRINT FNA (Y)
40  END
RUN
10
```

When FNA (Y) is called for in statement 30, the formula defined for FNA in statement 20 is used to determine the value printed.

A maximum of 26 programmer-defined functions are possible in a program (FNA to FNZ).

Any operand in the program may be used in the defining expression; however, such circular definitions as:

```
10  DEF FNA (Y) = FNB (X)
20  DEF FNB (X) = FNA (Y)
```

cause infinite looping.

See the vocabulary at the beginning of this section for a definition of "function."

EXAMPLES:

```
60  DEF FNA (B2) = A↑2 + (B2/C)
70  DEF FNB (B3) = 7*B3↑2
80  DEF FNZ (X) = X/5
```

General Mathematical Functions

The use of common mathematical functions is facilitated by pre-defining them as follows:

ABS *(expression)*   the absolute value of the expression

EXP *(expression)*   the constant *e* raised to the power of the expression value in statement 642 below, *e*↑N)

INT *(expression)*   the largest integer ≤ the expression (INT (–3.5) would result in –4)

LOG *(expression)*   the logarithm of the expression to the base *e*

RND *(expression)*   a random number between 0 and 1

SQR *(expression)*   the positive square root of the positively valued expression

SGN *(expression)*   returns: a 1 if the expression is greater than 0, a 0 if the expression equals 0, a –1 if the expression is less than 0.

All these functions may be used as expressions or as parts of expressions. LOG and SQR expressions must have a positive value or a fatal error will occur. A random sequence can be achieved if the sequential call to RND has a positive argument. Specification of a negative argument gives a predictable result. A sequence of random numbers is repeatable if the initial call to RND has a negative argument and is followed by a sequential call to RND with a positive argument.

EXAMPLES:

```
642 PRINT EXP(N)/ ABS(N)
652 IF RND (0)>=.5 THEN 900
662 IF INT (R) # 5 THEN 910
672 PRINT SQR (X); LOG (X)
```

Trigonometric Functions

The use of common trigonometric functions is facilitated by pre-defining them, as:

SIN   *(expression)*   the sine of the expression (in radians)

COS   *(expression)*   the cosine of the expression (in radians)

TAN   *(expression)*   the tangent of the expression (in radians)

ATN   *(expression)*   the arctangent (in radians) of the expression.

The trigonometric functions may be used as expressions, or parts of an expression.

The expressions (arguments) for SIN, COS, and TAN are interpreted as angles measured in radians. ATN returns the angle in radians.

EXAMPLES:

```
500  PRINT SIN(X); COS(Y)
510  PRINT 3*SIN(B); TAN (C2)
520  PRINT ATN (22.3)
530  IF SIN (A2) <1 THEN 800
540  IF SIN (B3) = 1 AND SIN(X) <1 THEN 90
```

The LEN Function

GENERAL FORM:

The LEN function may be used as an expression, or part of an expression. The function form is

   LEN (*string variable*)

The LEN function returns the length (number of characters) currently assigned to a string variable.

Note the difference between the LEN function and the LENGTH command. The command is used outside a program, and returns the working length of the current program in two-character words. The LEN function may be used only in a program statement.

EXAMPLES:

```
580  IF LEN (B$) >= 21 THEN 9999
800  IF LEN (C$) = R THEN 1000
850  PRINT LEN (N$)
880  LET P5 = LEN (N$)
```

The TIM Function

GENERAL FORM: TIM (X)

where if   X = 0, TIM (X) = current minutes (0 to 59)

X = 1, TIM (X) = current hour (0 to 23)

X = 2, TIM (X) = current day (1 to 366)

X = 3, TIM (X) = current year (0 to 99)

The TIM function returns the current minute, hour, day or year.

Note the difference between the TIM function and the TIME command. The TIME command is used outside a program and gives the console time and total time used. The TIM function can only be used within a program statement.

The argument must be an integer in the range 0-3. Otherwise, an error results.

EXAMPLES:

```
580 IF TIM (0) - A > 15 THEN 9000
700 LET A3 = TIM (B)
800 PRINT "MINUTE",TIM (0),"HOUR",TIM (1),"DAY",TIM (2),"YEAR",TIM (3)
```

CHAIN Statement

GENERAL FORM:

statement number CHAIN *"character string"*

or

statement number CHAIN *string variable*

or

statement number CHAIN *"character string"* , *expression*

or

statement number CHAIN *string variable* , *expression*

This statement is used to link programs together. "Character string" or string variable specifies a program in the user's own library, the group library or the system library, which is retrieved (replacing the current program) and run.

Strings and string variables are described in Section VI. As applied to the CHAIN statement, "character string" is the name of a program in one of the libraries; string variable is an alphabetic character followed by a $ that leads to a character string that is the name of a program. Expression is a line number in the named program. In the examples below, lines 20, 97, and 150 contain character strings. The other examples contain string variables.

If the first character of the program name, however defined, is $, the system will search the system library; if the first character is *, the system will search the user's group library. If the first character is neither $ or *, the system will search the user's own library. Note that the $ has different meanings as the first character in a program name and when used to define a string variable.

If expression is not specified, the program will be retrieved from the proper library and executed normally — examples 20 and 50. Expression may be an actual line number as in examples 150 and 230, may be a variable as in example 97, or may be computed as in example line 200.

In any of the above cases common storage is allocated if used. (See COM.) Before execution can begin, the program chained to must be compiled. Programs which are often chained to should be stored in semi-compiled form by use of the CSAVE command. This significantly reduces the time required to execute CHAIN statements.

Execution of the CHAIN statement can produce the same errors produced in executing the GET command. Such errors terminate execution of the program attempting the chaining, which will remain as the current program, with its common area (if any) intact.

EXAMPLES:

```
 20  CHAIN  "PROG2"
 50  CHAIN  V$
 97  CHAIN  "---", A
150  CHAIN  "MELVIN", 80
200  CHAIN  N$,Q+14
230  CHAIN  A$,110
```

COM Statement

GENERAL FORM:

statement number COM *list of variables, dimensioned arrays and strings*

The COM statement is used to designate data that can be passed between two or more programs without intermediate storage. A number of programs may be run sequentially, all accessing and possibly changing data in the common area.

The equivalence of common variables in different programs is determined by their relative order in the COM statements. Thus, if one program contains the statement

        10 COM A,B1,C$(10)

and a second program contains the statements

        1 COM X
        2 COM Y,Z$(10)

and the two programs are run in order, identifiers A and X refer to the same variable, as do identifiers B1 and Y, C$ and Z$.

There are certain restrictions on the use of COM:

1.   COM statements must be the lowest numbered statements in the program.

2.   A variable that is declared common in one program can be accessed by another program only if all preceding common variables in both programs are of the same type and size. If the common area in one program is smaller than that in another program to be run sequentially, only the common variables in the smaller area will be preserved.

3.   Arrays and strings which are to be in common must be dimensioned in the COM statement and they must not also appear in DIM statements.

Variables in COM should be initialized by the first program that uses them. After that, other programs containing equivalent COM definitions can be executed by GET and RUN or CHAIN. The COM variables will still have the same values. When a program with a common area terminates (whether normally, or because of an execution error or because the user presses *break*) the variables in common storage retain their values and will remain available until the user calls a program (GET command) with a different common area, enters a BASIC statement, or deletes a file (KILL command).

```
10 COM A,B,C,Q$(63),F(3,6),S1          (In program A)   All variables in common
10 COM J,K,L,C$(63),C(3,6),V           (In program B)


10 COM A,B,C,Q$(63),F(3,6),S1          (In program A)   Three variables in common
10 COM H,N,M,0                         (In program B)


10 COM A,B,C                           (In program A)   No variables in common
10 COM S$(45),A,B,C                    (In program B)


10 COM A,B,C                           (In program A)   All variables in common.
10 COM V                               (In program B)
30 COM B,C
```

ENTER Statement

GENERAL FORM:

statement number ENTER # *variable 1*

statement number ENTER *expression, variable 2, variable 3*

statement number ENTER # *variable 1, expression, variable 2, variable 3*

Allows the program to limit the time allowed for run-time data input, to check the actual time
taken to respond, to read in one string or numeric variable, to determine whether the input is of
the correct type, and/or to determine the current user's terminal number.

The form ENTER # sets variable 1 to the terminal number (between 0 and 31) of the user.

Expression sets the time limit; it should have a value between 1 and 255 seconds. Zero is treated as
1 and numbers greater than 255 are treated modulo 256. Timing starts when all previous statements
have been executed and any resultant output to the user terminal is completed.

Variable 2 returns the approximate time the user took to respond. If the user's response was of
the wrong type, such as alphabetic when numeric is expected, the value is the negative of the
response time. If the user failed to respond in time, the value is set to −256.

Variable 3, the data input variable, may be either a numeric or a string variable. A character string being entered should not be enclosed in quotes, but may contain quotes, leading or trailing blanks and embedded blanks. Only one data item can be entered per ENTER statement.

The ENTER statement differs from the INPUT statement in that a "?" is not printed on the user terminal, and the TSB System returns to the program if the user does not respond within a specified time limit. Also, the system does not generate a linefeed after the user types *return*.

A carriage return is a legitimate input to a string variable request.

A string that is too long to be assigned to a requested string variable is truncated from the right, and the response time is negated.

> *Note:* *If the user fails to respond in time, the variable which was to be input is not changed; if it was a number it may be undefined.*

EXAMPLES:

```
100  ENTER  #V
200  ENTER  A,B,C$
300  ENTER  #V,K1,K2,K3
400  ENTER  25,L,Q
```

The BRK Function

GENERAL FORM: BRK(x)

where $x < 0$ returns current status of the BREAK capability.

   $x = 0$ disables the BREAK capability.

   $x > 0$ enables the BREAK capability.

The BREAK capability (key) may be disabled or enabled by execution of the BRK function within the user's program. At the beginning of program execution, the BREAK capability is enabled (default). Once disabled, it remains disabled until program execution is completed, the program terminates because of an execution error, the BREAK command is entered by the system operator, C^C is typed into an input, or until the BRK function is executed with an argument greater than zero.

Because program execution may be completed before accumulated output is exhausted, care should be taken when re-enabling the BREAK capability. To ensure that program output will not be interrupted and lost, either an INPUT statement or an ENTER statement can be included in the program just prior to the statement containing the BREAK enable function. This will cause the

program to pause until output is complete before continuing execution. For example, the following program segment will disable the BREAK capability and print the value of "I" twenty times. On encountering the ENTER statement, the program will pause until printed output is complete before execution continues from statement 30.

```
 5 Y=BRK(0)
10 FOR I=1 TO 20
15 PRINT "I=";I
20 NEXT I
25 ENTER 1,A,B
30 Z=BRK(1)
      .
      .
      .
99 END
```

For arguments equal to or greater than zero, the value returned after evaluation of the expression depends on the previous condition of the BREAK capability. This value will be 1 if the capability was previously enabled, or 0 if the capability was previously disabled.

To find the current status of the BREAK capability, enter an argument less than zero. If currently enabled, a 1 is returned. If currently disabled, a 0 is returned.

If a program is in an infinite loop during execution and the BREAK capability is disabled, the system operator can enter a BREAK command to enable the BREAK capability.

For terminals connected to the system through telephone lines, a loss of carrier for longer than two seconds causes the user to be disconnected and automatically logged off the system. Similarly, hardwired terminals that drop carrier and/or data set ready signals when turned off cause the user to be automatically logged off the system. In either case, a disabled BREAK capability is returned to the enabled condition.

EXAMPLES:

```
935 LET B = BRK(0)
940 Z = BRK(A+M)
945 PRINT BRK(Y)
```

SECTION IV
Files

For those problems that require permanent data storage external to a particular program, the TSB system provides a data file capability. This allows flexible, direct manipulation of large volumes of data stored within the system itself. Special versions of the READ, PRINT, MAT READ, MAT PRINT, and IF statements allow you to read from and write onto mass storage files.

File programming offers two levels of complexity. Many problems can be solved using files treated simply as serial access storage devices. In this case, the program reads or writes a serial list of data items (either numbers or strings of characters) without regard to the underlying structure of the file. However, with additional programming effort, any file can be used as a random access storage device. In this case, the program breaks the file into a series of logical subfiles that can be modified independently.

This section deals with the serial use of files, then internal file structure and random access use. Explanatory programming samples follow each series of commands in this section.

TERM: FILE

Defined: An area of memory external to the program where numbers and strings of characters can be stored and retrieved. Files are created by, and belong to, a particular user.

The user determines the name and size of a file. Files vary in size from 1 record to a maximum determined by the device used to store them. The maximum size for files that are to be sanctified is 32 records. (See Appendix D.) A record contains between sixty-four and 256 16-bit words.

When a program stores some information in a file, the information remains there until it is changed or the file is eliminated. Any program of a particular user can be written to access this information.

Each program must declare its files with a FILES statement before it can access them. Each program can access up to 16 different files at one time. Files being accessed by a program can be changed by use of the ASSIGN statement.

For each file declared in the program, there is a file pointer that keeps track of the item in the file currently being accessed by that program. The RUN command causes all these pointers to be reset to the beginning of the file. The ASSIGN statement repositions the pointer to the beginning of a specified file. As the program reads or writes on a file, the pointer for the file is moved through the file.

SERIAL FILE ACCESS

This program writes all the data items out into the file in serial order. Each write operation begins where the previous one left off. Then, to retrieve one of these items, the program resets the pointer to the beginning of the file and reads through the items until it comes to the desired item. There is only one pointer for each file. When the pointer is repositioned by either a READ or a PRINT statement, it remains pointing to the next item in the file until it is repositioned by another file control statement.

Try this example. It should print out the same numbers you type in.

EXAMPLE OF SERIAL FILE ACCESS:


```
OPEN-GHIJK,50
```
The OPEN command creates a new file. GHIJK is the name of the file. The file is 50 records long.

```
NAM-PROG1
100 FILES GHIJK
```
The FILES statement links the file into the program. From now on, the file is referenced by number; GHIJK is file # 1.

```
200 INPUT A,B,C,D
300 PRINT #1;A,B,C,D
```
This is a serial file PRINT statement. It is identical to the normal PRINT statement except that a file number appears and the values of the variables are written onto the file, not the terminal.

```
400 INPUT A,B,C,D
500 PRINT #1;A,B,C,D
```
This PRINT stores the new values of the variables immediately following the previous values in the file.

```
600  READ  #1,1
```
This is a reset operation; it resets the pointer for file #1 to the beginning of the file.

```
700  READ  #1;  H1,H2,H3
```
This is a serial file READ statement. It assigns the first three values in the file to the three variables specified.

```
800  PRINT  H1,H2,H3
900  READ  #1;  H1,H2,H3,H4,H5
```
This READs the remaining five values in the file into the five variables given. The values in the file are not disturbed.

```
1000  PRINT  H1,H2,H3,H4,H5
2000  END
```

OPEN Command

GENERAL FORM:

OPEN- *1 to 6 character file name , number of records in file*

OPE- *1 to 6 character file name , number of records in file, record size*

OPE- *1 to 6 character file name , number of records in file*

The OPEN command creates a file with a specified number of records of a specified size, and assigns it a name.

The file that is opened is accessible only by the user idcode that opened it. The file remains open until the same user kills it.

> *Note: Unprotected system library files can be read by all users, and unprotected group files can be read by all members of the group.*

File names must conform to the same rules as program names except that a quotation mark (") may appear in a file name.

The size of the file may vary from a minimum of 1 record to a maximum determined by the peripheral devices on the system, the amount of unused storage, and the user's personal storage limit.

The size of a record must be between 64 and 256 words. If not specified, the system assumes 256 words. In any case, each record consumes 256 words of system storage.

If the system does not have enough storage space for the new file, the OPEN command is rejected and an error message is printed:

SYSTEM OVERLOAD

If the user does not have enough space left for the new file in the amount set for him by the system operator, the OPEN command is rejected and an error message is printed:

LIBRARY SPACE FULL

If the name given in the OPEN command equals the name of an existing file or program, the command is rejected and an error message is printed:

DUPLICATE ENTRY

The OPEN command marks each record of the new file as empty. If the system is heavily loaded, this process could take several minutes for very large files.

EXAMPLES:

```
OPEN-FILE27, 20, 64
OPEN-SAMPLE, 128
```

KILL Command

GENERAL FORM:

KILL-*file to be deleted*

KIL-*file to be deleted*

This command removes the named file from the user's library and releases the space it occupied for further storage. Users can kill only their own files.

Files have only one version, the stored one. When a file is killed, all the information in it is lost.

If the file named is currently being accessed by a user on another terminal, the KILL command is rejected and an error message is printed:

FILE IN USE

EXAMPLES:

```
KILL-NAMEXX
KILL-EXMPLE
KIL-FILE10
```

FILES Statement

GENERAL FORM:

statement number FILES *file name*$_1$*,file name*$_2$*,. . .,file name*$_{16}$

The FILES statement declares which files will be used in a program; assumes that the files will be opened (see OPEN command) before the program is RUN.

Up to four FILES statements can appear in a program, but only 16 files total can be declared (duplicate entries are legal). The files are assigned numbers (from 1 to 16) in the order they are declared in the program. In the EXAMPLES below, MATH is file #1 and #9, FILE27 is #7 and DATA is #10.

These numbers are used in the program to reference the files. For instance, in the same example,

100 PRINT #2; A

would print the value of A into the file named SCORE. This feature allows most programming to be done independently of the files to be used. The FILES statements may be added any time before running the program.

Public or group library files to be read (they cannot be written on) must also be declared in a FILES statement but with a $ or * preceding the file name. In the example, DATA is a public file; GRP is a group file. When * is used without a program name as one of the arguments in a FILES statement, the position occupied by the * symbol is reserved for a file to be specified later by an ASSIGN statement. ASSIGN statements are described on the following page.

Users with the same I.D. number can share files, but only one user can write on a file at a time. I.D. codes beginning with an "A" (e.g., A067) are an exception to the rule; they may read or write on files at the same time.

EXAMPLES:

```
10 FILES MATH, SCORE, AND, SQRT, NAMES
20 FILES *GRP, FILE27, SAMPLE
30 FILES MATH, $DATA, * , *
```

ASSIGN Statement

GENERAL FORM:

statement number ASSIGN *file name, file number, return variable, mask*

statement number ASSIGN *file name, file number, return variable*

The ASSIGN statement is used to change the file referred to by a specified file number during the execution of a program.

The parameters of an ASSIGN statement are:

file name	The name of a file - - a literal string of up to six characters (seven if the first character is $ or *) enclosed in quotes, or a string variable leading to a literal string. The symbol $ as a first character indicates a system file; * as a first character indicates a group file.
file number	A number, variable or expression whose value is between 1 and 16, indicating a file position. The file number should not exceed the **number of files declared in the FILES statements of the program.**
return variable	One of the following values will be returned to this variable when the statement is executed, depending upon the outcome of the execution:

0 - the file is available for reading and writing.

1 - the file is available on a read-only basis because it is being accessed by another terminal. For users A000 through A999, a return code of 1 indicates only that the named file is being accessed by another terminal. The file is still available for reading and writing.

2 - the file is available on a read-only basis because it is a system library or group library file.

3 - the requested file does not exist or it is protected (and the user attempting to ASSIGN it is not the owner).

4 - the file number in the ASSIGN statement is out of range; it does not correspond to one of the positions reserved by the FILES statements.

5 - the requested file has records which are larger than those of the file previously in this position.

If the value given to the return variable is 3, 4, or 5, any access to the requested file will cause a fatal error. If the return value is 2, any print attempt to the file will cause a fatal error. If the returned value is 1, a print attempt by any user other than Axxx users will cause a fatal error.

mask   An optional parameter that can be used to ensure security of data in the file. Mask can be either a literal string of up to six characters or a string variable of up to six characters used to form a mask through which data is written to or read from the file. If the same mask is used to read a data item that was used to write the item, the results are the same value that was written.

When the ASSIGN statement is executed, the named file replaces the file previously referenced by the file number in the statement. Subsequent file references using this number will apply to the new file. Data written to the old file will be intact.

EXAMPLES:

```
20 ASSIGN A$, 3, B1, C$
30 ASSIGN "NEWFL", S2, J
40 ASSIGN "$F2", 6, C, "AX1532"
```

Serial File PRINT Statement

GENERAL FORM:

statement number PRINT #*file number formula ; data item, data item,. . .*

This statement is used to print variables, numbers, or strings of characters consecutively on the specified file, starting after the last item previously read or printed.

The *file number formula* may be any expression; it is rounded to the nearest integer (from 1 through 16). If the value is n, then the nth file declared in the FILES statements (or the file most recently assigned to the nth position) is used.

The serial file PRINT always writes the indicated data items into the next available space in the file. However, since character strings may vary in length and each string must be wholly contained within a record, some space in each record may be left unused. You can calculate the number of words occupied by any string with a formula described under "Storage Requirements" in this section. Following the print list items an EOR (end-of-record) mark is written on the file.

After a serial file PRINT operation, the file pointer is updated so that it points to the next available space.

The information written in a file remains there even when the program terminates. Therefore, the user can return a day or week later and access the data at that time. If a program terminates because of an error or if the user types *break*, the files may not have been completely updated.

Matrices can also be written on files using a MAT PRINT # statement described in Section V.

EXAMPLES:

```
125 PRINT #5; A1,B2,C$
130 PRINT #5; D,E,F, "B,C,D,E"
140 PRINT #M+N; B
```

Serial File READ Statement

GENERAL FORM:

statement number READ #*file number formula ; data item, data item, . . .*

This statement is used to read numbers and strings into variables consecutively from the specified file, starting after the last item read.

The *file number formula* is evaluated as in the serial file PRINT.

Both strings and numbers can be read, but the order of variable types must match the order of data item types exactly. The TYP Function provides a means of determining the type of the next item.

The serial file READ moves from record to record within a file automatically, as necessary to find the next data item. After a READ, the file pointer is updated, and a subsequent READ will start with the next consecutive data item. Record boundaries and unused portions of records are ignored.

Matrices can also be read from files using a MAT READ # statement described in Section V.

> *Note:* *Following a serial file PRINT, the pointer must be reset*
> *to the beginning of the file before the data that was just*
> *written can be read. This is done using the reset operation*
> *described on the next page. A serial READ should not*
> *directly follow a serial PRINT.*

EXAMPLES:

```
65 READ #5; A,B,C
70 READ #3; B$
80 READ #N; A,B$, C(5,6)
90 READ #(N+1); A,B$,C
```

Resetting the File Pointer

GENERAL FORM:

statement number READ #*file number formula* , *1*

The READ statement in this form is used to reset the file pointer to the beginning of the file specified by the *file number formula*.

READ #N,1 is used after a serial PRINT to prepare for a serial READ.

> *Note:* *Do not use PRINT # 1,1 to reset, as this erases the first*
> *record of the file.*

EXAMPLES:

```
100 READ #1,1
200 READ #2,1
300 READ #M+N,1
```

The TYP Function

GENERAL FORM:

TYP may be used as an expression or as part of an expression; the function form is:

TYP *(file number formula)*

The TYP function determines the type of the next data item in the specified file so that the program can avoid a type mismatch on a file READ.

There are three possible responses:

> 1 = next item is number
>
> 2 = next item is character string
>
> 3 = next item is "end of file."

If the *file number formula* is negated (<0), the TYP function also detects "end of record" conditions (explained later under "Random Access") and returns a value of 4 for them.

If the *file number formula* equals zero, the TYP function references the DATA statements. In this case, TYP returns these values for the next data item:

> 1 = number
>
> 2 = string
>
> 3 = "out of data" condition.

EXAMPLES:

```
100 IF TYP (1)=2 THEN 1000
250 IF TYP (6)=3 THEN 500
300 GO TO TYP(B) OF 400,600,800
```

Listing Contents of a File

Here is a sample program that lists a file of unknown contents. It assumes that the file (DATUMS) has been previously filled serially by some other program.

```
NAM - LIST
100   FILES DATUMS
200   DIM A$[72]
300   IF END #1 THEN 1000
```
The IF END statement tells the program where to go if it comes to the end of file #1. Without this statement, the program would quit at the end of the file and give an error message.

```
500   IF TYP(1)=1 THEN 600
550   IF TYP(1)=2 THEN 700
```
TYP checks whether the next data item is a number (1) or a string (2).

```
600   READ #1;A
```
Reads a number from file #1 into variable A.

```
650   PRINT A
675   GOTO 500
700   READ #1;A$
```
Reads a string from file #1 into variable A$.

```
750 PRINT A$
775   GOTO 500
1000 PRINT "FILE LIST COMPLETED"
```
The program comes here when it reaches the end of file #1.

```
2000 END
```

TERM: END-OF-FILE

If a program attempts to PRINT beyond the physical end of a file or attempts to READ more values than are present in the file, the TSB system detects an end-of-file condition and terminates the program.

The OPEN command causes end-of-file marks to be written at the start of every record in the file. End-of-file marks can also be written by the user (as explained later under "END").

> *Note:* *If the user or an error (such as end-of-file) stops a program abnormally, it is not possible to know which file PRINT operations of the program were in fact performed.*

To avoid termination of a program because of end-of-file, use the IF END statement below. If this is done, all of the values preceding the end-of-file are transferred successfully.

IF END#...THEN Statement

GENERAL FORM:

statement number IF END #*file number formula* THEN *statement number*

This statement form defines a statement to be branched to if an "end-of-file" occurs on a specified file.

The IF END statement defines an exit procedure which remains in effect until another IF for the same file changes it, or until an ASSIGN statement containing the same file number is executed.

A different exit procedure can be defined for each file.

IF END is also used with random access to provide exit procedures when an "end-of-record" occurs. (See "Random Access.")

If a program does not contain an IF END statement for a file and an "end-of-file" occurs on that file, the program is terminated and an error message is printed:

END OF FILE/END OF RECORD IN STATEMENT *xxxx*

EXAMPLES:

```
300  IF  END  #N  THEN  800
310  IF  END  #2  THEN  830
320  IF  END  #3  THEN  9999
```

PRINT#...END Statement

GENERAL FORM:

statement number PRINT #*file number formula* ; *data item list* , END

This statement form places a logical "end-of-file" marker after the last value written on the file; END is ignored if it is not the last item in the statement.

The "end-of-file" marker written by this statement is a logical marker; each file also has a physical end-of-file which marks the physical boundary of the file. An EOR (end-of-record) mark is not written in this case.

The "end-of-file" mark is overlaid by the first item in the next serial PRINT statement. An "end-of-file" condition that aborts the program or triggers an IF END statement occurs only on an attempted READ operation beyond the available data or an attempted PRINT operation beyond the physical end-of-file.

END and IF END can be used to modify a serial file.

EXAMPLES:

```
 95 PRINT #N: A,B2,END
100 PRINT #(X+1); R3,S1,N$, "TEXT" , END
110 PRINT #2; G5,H$,P, END
```

STRUCTURE OF SERIAL FILES

When a file is opened, you can think of it as looking like this:

```
OPEN-INFO,5
```

INFO = | EOF | | PEOF |
          ↑

EOF is a mark that shows the end of the data.

PEOF is the physical end of the file, beyond which no data can be written.

↑ is the position of the file pointer.

When information is written into the file, the pointer moves and space in the file is used up.

```
100  FILES  INFO
200  PRINT  #1;  A,B,C,F$,Q1,  END
```

INFO = | A | B | C | F$ | Q1 | EOF | | PEOF |
                                  ↑
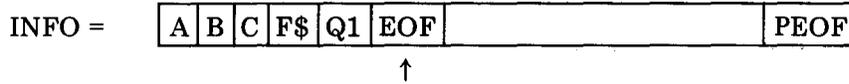
The file is filled solidly from the beginning.

When more information is printed, it follows the previous data and the pointer is changed.

```
300  PRINT  #1;  G1,G2,G$,H$,Z,END
```

INFO = | A | B | C | F$ | Q1 | G1 | G2 | G$ | H$ | Z | EOF | | PEOF |
                                                          ↑
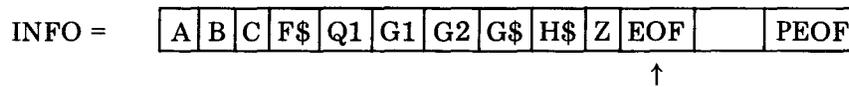
To read this data, the pointer must be reset.

```
400  READ  #1,1
```

INFO = | A | B | C | F$ | Q1 | G1 | G2 | G$ | H$ | Z | EOF | | PEOF |
         ↑
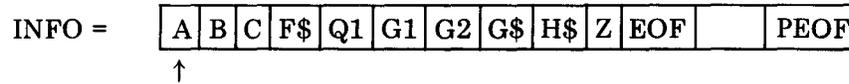
Now the data can be read.

```
500  READ  #1;  M1,M2
```

INFO = | A | B | C | F$ | Q1 | G1 | G2 | G$ | H$ | Z | EOF | | PEOF |
             ↑
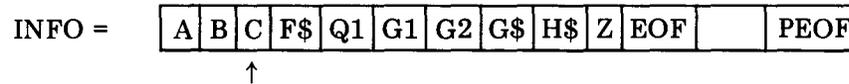
M1 now contains the value of A

M2 now contains the value of B

At this point, the program continues to read the data.

```
600 READ #1; D1
```

INFO = | A | B | C | F$ | Q1 | G1 | G2 | G$ | H$ | Z | EOF | | PEOF |
           ↑
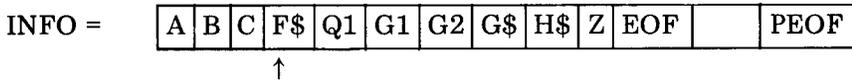
D1 now contains the value of C

However, if you PRINT anything in the file at this point, the rest of the file is effectively lost as far as serial access is concerned.

```
700 PRINT #1; D2,END
```

INFO = | A | B | C | D2 | EOF | | PEOF |
              ↑
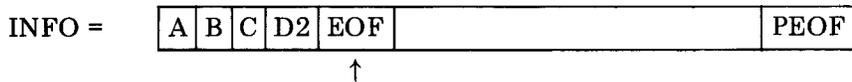
The correct way to modify an item in the middle of serial file is to READ all the succeeding items, then PRINT them and the new value out again.

```
700 READ #1; M$, P1, P2, P3, P$, R$, P4
```
        (read the values)
```
750 READ #1,1    (reset the pointer)
800 READ #1; A, B, C
```
        (move the pointer out to the correct item)

INFO = | A | B | C | F$ | Q1 | G1 | G2 | G$ | H$ | Z | EOF | | PEOF |
           ↑
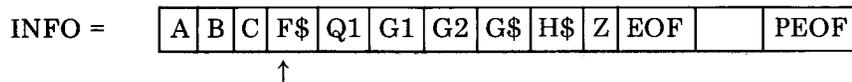
```
900 PRINT #1; D2    (print the new item)
1000 PRINT #1; P1, P2, P3, P$, R$, P4, END
```
        (print the old values out)

INFO = | A | B | C | D2 | Q1 | G1 | G2 | G$ | H$ | Z | EOF | | PEOF |
                    ↑
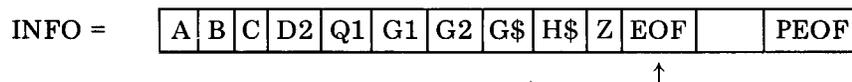
EXAMPLE OF SERIAL FILE MODIFICATION:

(When the file is opened, "end-of-file" markers are written into every record.)

```
OPEN-DATUMS,128

NAM-ADDIT


100   FILES DATUMS
200   DIM P$[72]
300   IF  END #1 THEN 1500
400   REM: THIS PROGRAM FIRST FINDS THE END OF FILE. IT ASKS
410   REM: THE USER FOR A STRING AND A NUMBER.  IF THIS IS
420   REM: NOT THE PHYSICAL END OF THE FILE, IT ADDS THEM TO
430   REM: THE END OF THE FILE.  THEN, THE PROGRAM ASKS THE
440   REM: THE USER IF HE WANTS TO ADD ANY MORE ITEMS.  IF
450   REM: THE USER ANSWERS YES, THE PROGRAM REPEATS THE
460   REM: INPUT AND WRITE LOOP.
800   READ #1;A$,A
850   GOTO 800
1500  IF  END #1 THEN 2000
1600  PRINT "STRING";
1650  INPUT A$
1700  PRINT "NUMBER";
1750  INPUT A
1800  PRINT #1;A$,A, END
1900  PRINT "MORE";
1950  INPUT A$
1960  IF A$="YES" THEN 1600
1970  STOP
2000  PRINT "PHYSICAL END OF THIS FILE"
5000  END
```

> *Note: If the file is empty, the first thing the program finds is an end-of-file. Therefore, it begins filling the file from the first location.*

The IF END statement (line 300) is changed once the end-of-file marker is found. The program is then looking for the physical end-of-file.

You can use the sample program for "Listing Contents of a File" to check the contents of the file.

TERM: RECORD

Defined: A physical division of a file; consisting of from 64 to 256 words.

The number of records in a file is subject to several constraints, but in no case may it exceed 32767.

File =

| Data | E O R | /// | DATA | E O R | /// | } { | DATA | E O F | /// |

↑ ↑ ↑ ↑ ↑

PEOR PEOR PEOR PEOR PEOF
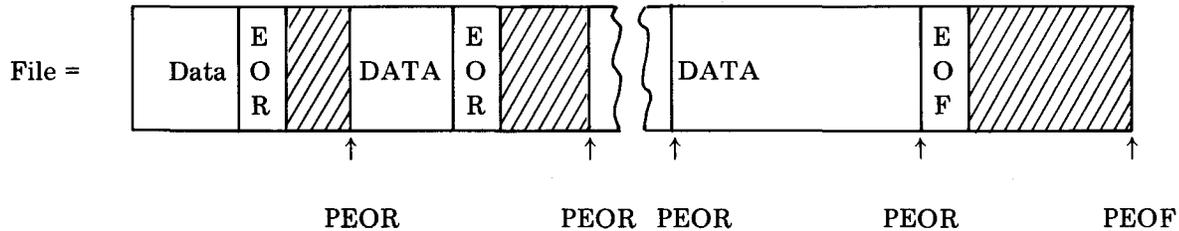
where PEOR = the physical end of the record.

  EOR = the end-of-record marker written by the system.

  EOF = the end-of-file marker written by the system.

  PEOF = the physical end of the file.

Following the data in a record, there is always an end-of-record marker. Every record also has a physical end. (When the record is completely full, this also acts as the logical end-of-record marker.)

During serial access the end-of-record markers act as skip markers that say to look in the next record for the data item, but during random access they cause an end-of-file condition. This will be explained later.

STORAGE REQUIREMENTS

Numerical data items require two words of storage space per item. If a full-size record is filled completely with numbers, it contains 128 items.

Strings can be of varying sizes: they require about 1/2 word of storage per character in the string. The exact formula for the number of words needed to store a string is:

If the number of characters is odd, then

$$1 + \frac{\text{number of characters in the string} + 1}{2}$$

If the number of characters is even, then

$$1 + \frac{\text{number of characters in the string}}{2}$$

Eight 62-character strings will completely fill a 256-word record. Strings and numbers can be mixed within a record, but each item must fit completely within the bounds of the record. For example, a 256-word record could contain five strings of 72 characters (each using 37 words) and a maximum of 35 numbers (leaving one word of the record unused).

MOVING THE POINTER

GENERAL FORM:

statement number READ #*file number formula* , *record number formula*

The statement in this form moves the pointer to the beginning of a specified record within a file; rounds the *file number formula* and the *record number formula* to integers.

The READ #M,N statement only generates an end-of-file condition at the physical end of the file, not for end-of-file markers.

After moving the pointer to the start of a record, you can use the serial READ and PRINT statements normally.

EXAMPLES:

```
200  READ  #1,N
300  READ  #M,N
400  READ  #3*J,9
```

To Determine the Length of a File

Here is a sample program that determines the number of records in a file. It uses the READ #M,N statement through the records until it comes to the physical end of the file.

```
NAM-LENGTH
10   REM: THIS PROGRAM PRINTS OUT THE LENGTH IN RECORDS OF ANY FILE.
20   FILES DATUMS
30   REM: 'DATUMS' IS THE FILE WHOSE LENGTH IS SOUGHT.
40   IF  END #1 THEN 80
50   FOR R=1 TO 32767
60   READ #1,R
70   NEXT R
80   PRINT "LENGTH IN RECORDS:";R-1
90   END
```

SUBDIVIDING SERIAL FILES

Serial files can be divided into smaller serial files by moving the pointer and using the PRINT END statement. For example, a file of six records could be treated as two files of three records.

a record

DATA	DATA	DATA	E O F	DATA	DATA	DATA	E O F

File:

first subdivision   second subdivision
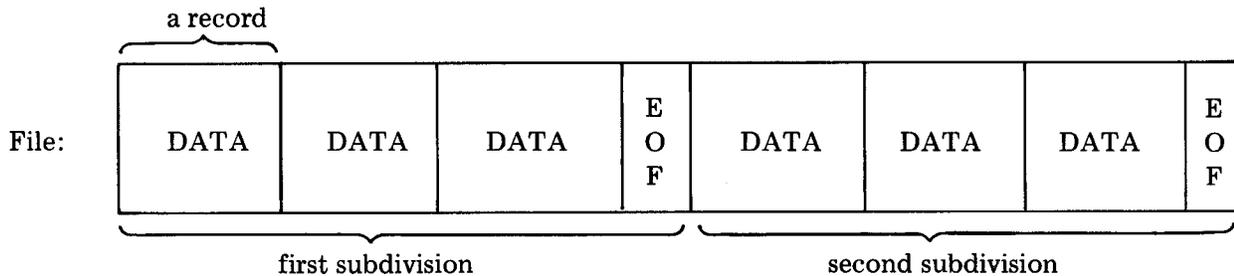
To switch from the first subdivision to the second, use this statement

      100 READ #1, 4

since the fourth record is the start of the second subdivision.

When using this technique, you must be careful that you do not print more data into the sub-division than it will hold. If you print too much, the data will overflow into the next subdivision and destroy its contents.

A logical extension of this concept is to make each subdivision equal to a single record. The TYP function detects end-of-record markers. The random access versions of PRINT# and READ# statements (described later) allow you to access random records within a file without overflowing the bounds of the record.

USING THE TYP FUNCTION WITH RECORDS

GENERAL FORM:

TYP is a function and can be used as an expression or a part of an expression.

TYP (*-file number formula*)

Returns a code telling the type of the next item in a specified file.

TYP(- X)   = 1 for a number

                 = 2 for a string

                 = 3 for an end-of-file

                 = 4 for an end-of-record

The *file number formula* must be negated to detect the end of record. If it is positive or zero, different results are returned. See "TYP Function" in this section.

EXAMPLES:

```
100   GO TO TYP(-1) OF 200, 250, 300, 400
2000 A=TYP(-5) + B*2
```

To List the Contents of a Record

Here is a sample program that lists the exact contents of any record in a file.

```
NAM-RLIST
1   REM: THIS PROGRAM LISTS THE CONTENTS OF ANY RECORD OF THE FILE.
5   DIM A$[72]
10   FILES DATUMS
20   IF  END #1 THEN 60
30   PRINT "RECORD NUMBER";
40   INPUT R
50   IF R>0 AND R=INT(R) THEN 80
60   PRINT "INVALID RECORD NUMBER."
70   GOTO 30
80   READ #1,R
100   GOTO TYP(-1) OF 110,150,220,200
110   PRINT "NUMBER:";
120   READ #1;X
130   PRINT X
140   GOTO 100
150   PRINT "STRING:";
160   READ #1;A$
170   PRINT A$
180   GOTO 100
200   PRINT "END OF RECORD MARK."
210   STOP
220   PRINT "END OF FILE MARK."
230   END
```

To Copy a File

Here is a sample program that copies one file into another using only the statements and functions covered so far: IF END, TYP, FILES, READ #M,N, serial READ, and serial PRINT.

```
NAM-COPY
1   REM: THIS PROGRAM COPIES FILE #1 INTO FILE #2.
10   FILES SAM1,SAM2
20   DIM A$[72]
30   IF  END #1 THEN 170
40   IF  END #2 THEN 180
50   FOR I=1 TO 32767
60   READ #1,I
70   PRINT #2,I
80   GOTO TYP(-1) OF 90,120,150,160
90   READ #1;X
100   PRINT #2;X
110   GOTO 80
120   READ #1;A$
130   PRINT #2;A$
140   GOTO 80
150   PRINT #2; END
160   NEXT I
170   STOP
180   PRINT "SECOND FILE TOO SMALL."
190   END
```

TERM: RANDOM FILE ACCESS

Defined: A READ or PRINT access of a file is "random" if it specifies a particular record within the file.

Serial Access: 100 READ #1;A,B,C
                  (Reads from the file pointer)

Random Access: 100 READ #1,5;A,B,C
                  (Moves to record 5 before reading)

When files are accessed serially, the record structure of files is ignored. Serial READ operations skip over end-of-record markers to the next record and act as if all data were in a continuous list.

The TSB System does, however, provide statements that take advantage of this record structure. The file pointer can be moved to the beginning of any record. Also, any record can be read or printed independently of the rest of the file using random access versions of READ# and PRINT# statements. The TYP function and IF END statement can detect end-of-record conditions. These extensions to BASIC constitute a random access file capability.

EXAMPLE OF RANDOM FILE ACCESS:

This sample program fills each record with two strings of up to 30 characters each and five numbers. Then it lists the contents of any record.

```
OPEN-RNDFL,20
```

```
NAM-PROG2
100    FILES RNDFL
150    DIM A$[30],B$[30]
200    IF   END #1 THEN 1000
300    FOR  J=1 TO 20
400    INPUT A$,B$,A,B,C,D,E
500    PRINT #1,J;A$,B$,A,B,C,D,E
600    NEXT J
700    PRINT "WHICH RECORD WOULD YOU LIKE TO SEE";
750    INPUT J
760    READ #1,J;A$,B$,A,B,C,D,E
770    PRINT A$
780    PRINT B$
790    PRINT A,B,C,D,E
800    GOTO 700
1000   END
```

This loop reads in two strings and five numbers from the user, then it writes the Jth record of the file.

This section will read and list the contents of record N.

PRINTING A RECORD

GENERAL FORM:

statement number PRINT # *file number formula, record number formula ; list of data items*

The PRINT statement in this form prints a specified list of data items into a specific record of a file, starting at the beginning of the record. (The *record number formula* is rounded to the nearest integer.)

The previous contents of the record are destroyed. An end-of-record marker is written after the data. If an END occurs in the data list, it acts as an end-of-record marker too. The random PRINT operation cannot change the contents of any record except the one specified. The entire list of data items must fit within the record. Otherwise, an end-of-file condition occurs which terminates the program and prints an error message:

> END OF FILE/END OF RECORD

An IF END statement establishes an exit procedure. See "IF END" in this section.

Matrices are printed using the random version of MAT PRINT# statement described in Section V. Note, however, that the matrix must fit within a *single* record, so a maximum of 128 numerical items can be printed. If this rule is violated, an end-of-file condition occurs.

EXAMPLES:

```
165  PRINT  #N,X;G2,H,I,"TEXT"
170  PRINT  #1,3;X,Y4,Z,6127,B
175  PRINT  #(N+2),(X+2);F,P5
180  PRINT  #2,5;A,B,C,D,END
```

READING A RECORD

GENERAL FORM:

statement number READ # *file number formula , record number formula ; list of data items*

The READ statement in this form reads data from a specified record of a file, starting at the beginning of the record. (The *file number formula* and *record number formula* are rounded to integers.)

The contents of the file are not changed.

If the READ operation encounters an end-of-record marker before filling all the data items, an end-of-file occurs. The program is terminated unless an IF END statement has been defined previously. (See IF END in this section.)

Matrices are read from records using a random version of MAT READ# statement described in Section V. If the READ operation requests more items than the record contains, an end-of-file condition occurs.

EXAMPLES:

```
100  READ  #2,3;A,B,C3,X$
110  READ  #N,2;N1,N2,N3
120  READ  #M,N;R2,P7,A$,T(35)
130  READ  #(M+1),(N+1);X,Y,Z
```

Modifying Contents of a Record

Principle: The contents of a record can be changed only by reading the entire record into the program, modifying the items desired, then printing it back on the file again.

> Caution: When the strings are replaced by longer strings, the
> result may no longer fit within a record. If this
> happens, an end-of-file condition occurs.

```
100  READ  #1,5;A,B,C,Z$
200  LET  A  =  Q*2+(M/5)
300  LET  Z$  =  M$
500  PRINT  #1,5;A,B,C,Z$
```

A,B,C, and Z$ are the entire contents of record 5.

Erasing a Record

GENERAL FORM:

statement number PRINT #*file number formula* , *record number formula*

The PRINT statement in this form erases the contents of a specified record in a file by printing an end-of-record marker at the beginning of the record.

The file pointer is moved to the start of the specified record.

Only the contents of the specified record are erased; the rest of the file is unchanged. The erased record still exists, however, and can be filled with new data.

Do not confuse this erase operation with the KILL command which permanently eliminates the entire file.

EXAMPLES:

```
320  PRINT  #M+N,  R+S
330  PRINT  #1,2
340  PRINT  #12,Q1
```

To Erase a File, Record by Record

Here is a sample program that uses the erase operation to erase an entire file, record by record.

```
NAM-ERASE
1   REM: THIS PROGRAM ERASES A FILE BY ERASING EVERY RECORD.
10  FILES X
20  IF  END #1 THEN 60
30  FOR I=1 TO 32767
40  PRINT #1,I
50  NEXT I
60  END
```

Updating a Record in a File

File programming is simplified if every record of a file has the same data structure. For example, each record might contain a string (e.g., a person's name) and a number (e.g., the amount of money he owes). Here is a sample program that manipulates such a file. The program searches through the file until it finds a specified string; then it updates the number in the record to a new value.

```
NAME-UPDATE
10  FILES DATA
20  DIM A$[72],B$[72]
30  IF  END #1 THEN 160
40  PRINT "NAME";
50  INPUT A$
60  FOR I=1 TO 32767
70  READ #1,I
80  IF TYP(-1)#2 THEN 150
90  READ #1;B$
100   IF B$#A$ THEN 150
110   PRINT "NEW NUMBER";
120   INPUT N
130   PRINT #1;N
140   STOP
150   NEXT I
160   PRINT "NAME NOT ON FILE."
170   END
```

An Alphabetically Organized File

Suppose the first item of every record in a file is a string. The records can be ordered alphabetically. Here is a program that inserts a new record where it alphabetically belongs. The rest of the file must be moved up one record. In this example, record 1 contains the record number of the last item.

```
10    FILES DATA
20    DIM G$[72],H$[72]
30    IF  END #1 THEN 290
40    READ #1,1;N
45    IF  END #1 THEN 270
50    READ #1,N+2
60    PRINT "STRING";
70    INPUT G$
72    IF N#0 THEN 80
74    R=2
76    GOTO 180
80    F=2
90    L=N+1
100   R=INT((F+L)/2)
110   READ #1,R;H$
120   IF G$<H$ THEN 210
130   IF G$>H$ THEN 230
140   FOR I=N+1 TO R STEP -1
150   READ #1,I;H$
160   PRINT #1,I+1;H$
170   NEXT I
180   PRINT #1,R;G$
190   PRINT #1,1;N+1
200   STOP
210   L=R
220   IF F#L THEN 100
225   GOTO 140
230   F=R
240   IF L-F>1 THEN 100
250   R=R+1
255   IF L-F#1 THEN 140
260   F=F+1
265   GOTO 100
270   PRINT "FILE FULL."
280   STOP
290   N=0
300   GOTO 45
310   END
```

FILE ACCESSING ERRORS

If a data error occurs while the computer is performing a requested file read or write, the program will be terminated and one of the following messages will be printed:

BAD FILE READ IN LINE nn

BAD FILE WRITE DETECTED IN LINE nn

As is the case with other errors which terminate a running program, the specific contents of any file written on during execution cannot be easily predicted.

Most of the information in the file on which the data error occurred may be recoverable. If file errors persist, the information should be copied item by item or record by record to another file.

SECTION V
Matrices

A matrix is a rectangular array of data elements arranged in rows and columns. Arrays are described in Section III. This section describes a series of special instructions used to manipulate matrices. Instructions starting with MAT refer to an entire matrix, or to two or more matrices. Instructions such as PRINT and INPUT refer to specific elements of the array by row and column. The DIM statement is used to define the dimensions of the matrix and to reserve storage space for it. Some typical matrix operations are:

MAT READ A,B,C	Read the three matrices, their dimensions having been previously defined. Data is stored in the matrix row by row.
MAT INPUT A,B	Input matrices A and B from the user terminal. Data is stored in the matrix row by row.
MAT C = ZER	Fill matrix C with zeros.
MAT C = CON	Fill matrix C with ones.
MAT C = IDN	Set up matrix C as an identity matrix.
MAT PRINT A,B;C	Print the three matrices, with A and C in the regular format, but B closely packed.
MAT B = A	Set matrix B equal to matrix A.
MAT C = A + B	Add two matrices, A and B; set matrix C to the result.
MAT C = A - B	Subtract matrix B from matrix A; set matrix C to the result.
MAT C = A*B	Multiply matrix A by matrix B; set matrix C to the result.
MAT C = TRN(A)	Transpose matrix A; set matrix C to the result.
MAT C = (K)*A	Multiply matrix A by K. K, which must be in parentheses, may be a formula; set matrix C to the result.
MAT C = INV(A)	Invert matrix A; set matrix C to the result.

| MAT PRINT #5;A | Print matrix A onto a file. |
| MAT READ #M,N+2;D | Read matrix D from a file, row by row — same restrictions as MAT READ. |

Use of these statements is described in this section. Formatted printing of matrices is described in Section VIII.

STATEMENTS

DIM Statement

GENERAL FORM:

statement number DIM *matrix variable (integer) . . .*

or

statement number DIM *matrix variable (integer , integer) . . .*

The DIM statement sets upper limits on the amount of working space used by an array or a matrix in the TSB system.

The *integers* refer to the number of array elements if only one dimension is supplied, or to the number of row and column elements respectively, if two dimensions are given.

A matrix (array) variable is any single letter from A to Z.

Some matrix operations permit the initialization of an array or matrix within the operation call if the dimensions are 10 elements or less for one dimensional arrays or 10 rows and 10 columns or less for two-dimensional arrays. Otherwise, a DIM statement is required.

The working size of a matrix may be smaller than its physical size. For example, an array declared 9 x 9 in a DIM statement may be used to store fewer than 81 elements; the DIM statement supplies only an upper bound on the number of elements. When the working size of a matrix is changed using one of the MAT statements described on the following pages, the values of excluded positions are lost.

The absolute maximum matrix size is about 4900 elements; a matrix of this size is practical only in conjunction with a very small program.

EXAMPLES:

```
110 DIM A (50), B(20,20)
120 DIM Z (5,20)
130 DIM S (5,25)
140 DIM R (4,4)
```

MAT . . .ZER Statement

GENERAL FORM:

statement number MAT *matrix variable* = ZER

or

statement number MAT *matrix variable* = ZER *(expression)*

or

statement number MAT *matrix variable* = ZER *(expression , expression)*

This statement sets all elements of the specified matrix equal to zero.

The matrix specified may be initially dimensioned within the MAT . . . .ZER statement if the dimensions are 10 elements or less for one-dimensional, or 10 rows and 10 columns or less for two-dimensional arrays. Otherwise, a DIM statement is required.

A new working size may be established; the new working size in a MAT . . .ZER is an implicit DIM statement within the limits set by the DIM statement on the total number of elements.

Since 0 has a logical value of "false", MAT . . .ZER is useful in logical initialization.

The *expressions* in new size specifications should evaluate to integers. Non-integers are rounded to the nearest integer value.

EXAMPLES:

```
305 MAT A = ZER
310 MAT Z = ZER (N)
315 MAT X = ZER (30, 10)
320 MAT R = ZER (N, P)
```

MAT . . .CON Statement

GENERAL FORM:

statement number MAT *matrix variable* = CON

<p style="text-align:center">or</p>

statement number MAT *matrix variable* = CON *(expression)*

<p style="text-align:center">or</p>

statement number MAT *matrix variable* = CON *(expression , expression)*

This statement sets up a matrix with all elements equal to one.

The matrix specified may be initially dimensioned with the MAT . . .CON statement if the dimensions are 10 elements or less for one-dimensional, or 10 rows and 10 columns or less for two-dimensional arrays. Otherwise, a DIM statement is required.

A new working size may be specified, within the limits of the original DIM statement on the total number of elements.

Note that since 1 has a logical value of "true", the MAT . . .CON statement is useful for logical initialization.

The expressions in new size specifications should evaluate to integers. Non-integers are rounded to the nearest integer value.

EXAMPLES:

```
205 MAT C = CON
210 MAT A = CON (N,N)
220 MAT Z = CON (5,20)
230 MAT Y = CON (50)
```

INPUT Statement

GENERAL FORM:

statement number INPUT *matrix variable (expression) . . .*

or

statement number INPUT *matrix variable (expression , expression) . . .*

The INPUT statement allows input of a specified matrix element(s) from the user terminal.

An *expression* should be an integer. Non-integers are rounded to the nearest integer value.

The subscripts *(expressions)* used after the matrix variable designate the row and column of the matrix element. Do not confuse these expressions with working size specifications, such as those following a MAT INPUT statement.

See MAT INPUT and DIM in this section for further details on matrix input.

See ENTER, Section III for an additional means of inputting specific matrix elements.

EXAMPLES:

```
600  INPUT  A(5)
610  INPUT  B(5,8)
620  INPUT  R(X), N$, A(3,3)
630  INPUT  Z(X,Y), P3, W$
640  INPUT  Z(X,Y), Z(X+1, Y+1), Z(X+R3, Y+S2)
```

MAT INPUT Statement

GENERAL FORM:

statement number MAT INPUT *matrix variable*

<div align="center">*or*</div>

statement number MAT INPUT *matrix variable (expression) . . .*

<div align="center">*or*</div>

statement number MAT INPUT *matrix variable (expression , expression) . . .*

The MAT INPUT statement allows input of an entire matrix from the user terminal.

The matrix specified may be initially dimensioned within the MAT INPUT statement if the dimensions are 10 elements or less for one-dimensional, or 10 rows and 10 columns or less for two-dimensional arrays. Otherwise, a DIM statement is required.

A new working size may be specified, within the limits of the DIM statement on total number of elements.

Do not confuse the size specifications following MAT INPUT with element specifications. For example, INPUT X(5,5) causes the fifth element of the fifth row of matrix X to be input, while MAT INPUT X(5,5) requires input of the entire matrix called X, and sets the working size at 5 rows of 5 columns.

Example statements 360 through 375 require input of the specified number of matrix elements, because they specify a new size.

Elements being input must be separated by commas.

When a MAT INPUT statement is executed, "?" is generated regardless of the number of elements. A "??" response to an input item means that more values are required.

MAT INPUT causes the entire matrix to be filled from teleprinter input in the (row, col.) order: 1,1;1,2;1,3; etc.

EXAMPLES:

```
355 MAT INPUT A
360 MAT INPUT B(5)
365 MAT INPUT Z(5,5)
370 MAT INPUT A(N)
375 MAT INPUT B(N,M)
```

Printing Matrices

GENERAL FORM:

statement number PRINT *matrix variable (expression) . . .*

<div align="center">or</div>

statement number PRINT *matrix variable (expression , expression) . . .*

A PRINT statement causes the specified matrix element(s) to be printed.

The expressions (subscripts) should be integers. Non-integers are rounded to the nearest integer value.

A trailing semicolon packs output into twelve elements per teleprinter line, if possible. A trailing comma prints five elements per line.

Subscripts following the matrix variable designate the row and column of the matrix element. Do not confuse these with new working size specifications, such as those following a MAT INPUT statement.

This statement prints individual matrix elements. MAT PRINT is used to print an entire matrix.

EXAMPLES:

```
800  PRINT  A(3)
810  PRINT  A(3,3);
820  PRINT  F(X);E$;  C5;R(N)
830  PRINT  G(X,Y)
840  PRINT  Z(X,Y),  Z(1,5),  Z(X+N,  Y+M)
```

MAT PRINT Statement

GENERAL FORM:

statement number MAT PRINT *matrix variable*

or

statement number MAT PRINT *matrix variable* , *matrix variable . . .*

A MAT PRINT statement causes an entire matrix to be printed.

Matrices referenced in a MAT PRINT statement must first be dimensioned in a DIM statement.

The MAT PRINT statement causes the matrix elements to be printed row by row across the page. Each matrix row starts a new line. The spacing between row elements is controlled by the use of , and ; in the same manner as for the PRINT statement. Rows containing more elements than can be printed on a line are continued on consecutive lines. Each row of a matrix is started on a new line and is separated from the previous row by a blank line. Thus the instruction:

      MAT PRINT A, B;C

will cause the three matrices to be printed A and C with five components to a line and B with up to twelve.

Singly subscripted arrays may be interpreted as column vectors. Vectors may be used in place of matrices, as long as the above rules are observed. Since a vector like V(I) is treated as a column vector by BASIC, a row vector has to be introduced as a matrix that has only one row, namely row 1. Thus

      DIM X(7), Y(1,5)

introduces a 7-component column vector and a 5-component row vector.

A column vector will be printed one element to the line with double spacing between lines. A row vector will be printed in the manner indicated by the form of the statement. For example: if V is a row vector then, "MAT PRINT V" or "MAT PRINT V," will print V as a row vector, five numbers to the line, while "MAT PRINT V;" will print V as a row vector with up to twelve numbers to the line.

EXAMPLES:

```
500 MAT PRINT A
505 MAT PRINT A;
515 MAT PRINT A,B;C
520 MAT PRINT A,B,C;
```

READ Statement

GENERAL FORM:

statement number READ *matrix variable (expression)*

or

statement number READ *matrix variable (expression , expression) . . .*

The READ statement causes the specified matrix element to be read from the current DATA statement.

Expressions (subscripts) should evaluate to integers. Non-integers are rounded to the nearest integer.

Subscripts following the matrix variable designate the row and column of the matrix element. Do not confuse these with working size specifications, such as those following MAT INPUT statement.

The MAT READ statement is used to read an entire matrix from DATA statements. See details in this section.

EXAMPLES:

```
900 READ A(6)
910 READ A(9,9)
920 READ C(X); P$; R7
930 READ C(X,Y)
940 READ Z(X,Y),P(R2, S5), X(4)
```

MAT READ Statement

GENERAL FORM:

statement number MAT READ *matrix variable*

<div align="center">or</div>

statement number MAT READ *matrix variable (expression) . . .*

<div align="center">or</div>

statement number MAT READ *matrix variable (expression , expression)*

The MAT READ statement reads an entire matrix from DATA statements.

The matrix specified may be initially dimensioned within the MAT READ statement if the dimensions are 10 elements or less for one-dimensional, or 10 rows and 10 columns or less for two-dimensional arrays. Otherwise, a DIM statement is required.

A new working size may be specified, within the limits of the original DIM statement.

MAT READ causes the entire matrix to be filled from the current DATA statement in the (row, col.) order: 1,1; 1,2; 1,3; etc. In this case the DIM statement controls the number of elements read.

EXAMPLES:

```
350 MAT READ A
370 MAT READ B(5),C,D
380 MAT READ Z (5,8)
390 MAT READ N (P3,Q7)
```

Matrix Addition

GENERAL FORM:

statement number MAT *matrix variable = matrix variable + matrix variable*

Matrix addition establishes a matrix equal to the sum of two matrices of identical dimensions; addition is element-by-element.

Each matrix referenced must be previously mentioned in a DIM statement. Dimensions of the resultant matrix must be the same as the component matrices.

The same matrix may appear on both sides of the = sign, as in example statement 320.

EXAMPLES:

```
310 MAT C = B + A
320 MAT X = X + Y
330 MAT P = N + M
```

Matrix Subtraction

GENERAL FORM:

statement number MAT *matrix variable = matrix variable – matrix variable*

Matrix subtraction establishes a matrix equal to the difference of two matrices of identical dimensions; subtraction is element-by-element.

Each matrix referenced must be previously mentioned in a DIM statement. Dimensions of the resultant matrix must be the same as the component matrices.

The same matrix may appear on both sides of the = sign, as in example statement 560.

EXAMPLES:

```
550 MAT C = A - B
560 MAT B = B - Z
570 MAT X = X - A
```

Matrix Multiplication

GENERAL FORM:

statement number MAT *matrix variable* = *matrix variable* * *matrix variable*

Matrix multiplication establishes a matrix equal to the product of the two specified matrices.

Each matrix referenced must be previously dimensioned.

Following the rules of matrix multiplication, if the dimensions of matrix B = (P,N) and matrix C = (N,Q), multiplying B*C results in a matrix of dimensions (P,Q).

Note that the resulting matrix must have an appropriate working size.

The same matrix variable may not appear on both sides of the = sign.

EXAMPLES:

```
930 MAT Z = B ⋇ C
940 MAT X = A ⋇ A
950 MAT C = Z ⋇ B
```

Scalar Multiplication

GENERAL FORM:

statement number MAT *matrix variable* = (*expression*) * *matrix variable*

Scalar multiplication establishes a matrix equal to the product of the matrix multiplied by a specified number, that is, each element of the original matrix is multiplied by the number.

Each matrix referenced must be previously dimensioned. The same matrix variable may appear on both sides of the = sign. Both matrices must have the same working size.

EXAMPLES:

```
110 MAT A = (5) ⋇ B
115 MAT C = (10) ⋇ C
120 MAT C = (N/3) ⋇ X
130 MAT P = (Q7⋇N5) ⋇ R
```

Copying a Matrix

GENERAL FORM:

statement number MAT *matrix variable* = *matrix variable*

A specified matrix may be copied into a matrix of the same dimensions; copying is element-by-element.

Each matrix referenced must be previously dimensioned. Both must have the same dimensions.

EXAMPLES:

```
405 MAT B = A
410 MAT X = Y
420 MAT Z = B
```

Identity Matrix

GENERAL FORM:

statement number MAT *array variable* = IDN

          or

statement number MAT *array variable* = IDN *(expression , expression)*

A MAT . . . IDN statement is used to establish an identity matrix (all 0's, with a diagonal of all 1's).

The matrix specified may be initially dimensioned within the MAT . . . IDN statement if the dimensions are 10 elements or less for one-dimensional, or 10 rows and 10 columns or less for two-dimensional arrays. Otherwise, a DIM statement is required.

A new working size may be specified within the limits of the original DIM statement. The IDN matrix must be two dimensional and square.

Specifying a new working size has the effect of a DIM statement.

```
Sample identity matrix:  1   0   0
                         0   1   0
                         0   0   1
```

EXAMPLES:

```
205 MAT A = IND
210 MAT B = IDN (3,3)
215 MAT Z = IDN (Q5, Q5)
220 MAT S = IDN (6, 6)
```

Matrix Transposition

GENERAL FORM:

statement number MAT *matrix variable* = TRN *(matrix variable)*

A MAT . . . TRN statement can be used to establish a matrix as the transposition of a specified matrix; transposes rows and columns.

Each matrix referenced must be previously dimensioned.

Sample transposition:

Original				Transposed		
1	2	3		1	4	7
4	5	6		2	5	8
7	8	9		3	6	9

Note that the dimensions of the resulting matrix must be the reverse of the original matrix. For instance, if A has dimensions of 6,5 and MAT C = TRN (A), C must have dimensions of 5,6. The same matrix can not be on both sides of the "=" sign.

EXAMPLES:

```
959 MAT Z = TRN (A)
969 MAT X = TRN (B)
979 MAT Z = TRN (C)
```

Matrix Inversion

GENERAL FORM:

statement number MAT *matrix variable* = INV *(matrix variable)*

A MAT . . . INV statement is used to establish a square matrix as the inverse of the specified square matrix of the same dimensions.

Each matrix referenced must be previously dimensioned.

A matrix may be inverted into itself, as in example statement 400 below.

In performing the inversion, the system must generate an additional internal matrix, so that an additional amount of storage equal to that needed for the original matrix is required. It may not be possible to invert an extremely large matrix.

Sample Inversion:

Original				Inverted		
1	3	4		-.392265	.18232	.110497
4	1	8		.243094	-.127072	4.41989E-02
6	9	1		.165746	4.97238E-02	-6.07735E-02

EXAMPLES:

```
380 MAT A = INV(B)
390 MAT C = INV(A)
400 MAT Z = INV(Z)
```

MAT PRINT # Statement

GENERAL FORM:

statement number MAT PRINT # *file number formula ; matrix variable . . .*

<center>*or*</center>

stat. no. MAT PRINT # *file no. form. , record no. form. ; matrix var. . . .*

The MAT PRINT # statement prints an entire matrix on a file, or on a specified record within a file.

Matrices referenced in a MAT PRINT # statement must be previously dimensioned.

A random matrix file print (i.e., with a record number specified) cannot transfer more than 128 numeric values because that is the maximum a record can hold. Attempting to exceed this generates an end-of-file condition.

A serial matrix file print, however, can transfer as many elements as will fit in the entire file.

> *Note: A matrix may also be printed with formatted output.*
> *See PRINT USING, Section VIII.*

EXAMPLES:

```
520 MAT PRINT #5; A
530 MAT PRINT #6, 3; B
540 MAT PRINT #4,M; A
550 MAT PRINT #N,M; A
```

MAT READ # Statement

GENERAL FORM:

statement number MAT READ # *file formula number ; matrix variable . . .*

or

statement no. MAT READ # *file formula no. , record no. formula ; matrix variable . . .*

or

statement no. MAT READ # *file form. no. , record no. form. ; matrix var. (expression) . . .*

or

stmt. no. MAT READ # *file form. no. , record no. form. ; matrix var. (expr. , expr.) . . .*

A MAT READ # statement reads a matrix from a file, or specified record within a file.

A new working size may be specified within the limits of the original DIM statement.

MAT READ # fills the entire matrix in a row-by-row sequence of elements as: 1,1; 1,2; 1,3; 1,4 . . .

Remember that a maximum of 128 numbers may be transferred on a random read.

EXAMPLES:

```
720  MAT  READ  #2;A
730  MAT  READ  #2,3;B
740  MAT  READ  #M,N;B(5)
750  MAT  READ  #M,N;B(P7,R5)
```

SECTION VI
Strings

A string is a set of characters such as "DDDDDE" or "45T,#". BASIC contains special variables and language elements for manipulating string quantities. This section explains how to use the string features of BASIC. There is little difference in the form of statements referencing numeric quantities and those referencing strings. One important difference, however, is the use of subscripts which is explained later.

Lower-case alphabetic characters can be input from or output to user terminals having this capability. When lower-case characters are output to a terminal not capable of printing them, most terminals will print such characters as the upper case equivalent. Lower-case characters are automatically converted to upper case by the system, except when they occur in strings or REM statements. Lower-case characters in strings used as file names in ASSIGN statements or program names in CHAIN statements are also converted to upper case when used.

The examples and comments in this section emphasize modifications in statement form or other special considerations in handling strings.

If you are familiar with the concepts "string", "string variable", and "substring", skip directly to "The String DIM Statement".

TERM: STRING

Defined: A set of 1 to 72 characters enclosed by quotation marks or the null string (no characters).

Typical Strings: "ABCDEFGHIJKLMNOP"

        "12345"

        "BOB AND TOM"

        "MARCH 13, 1970"

Null String: " "

Quotation marks cannot be used within a string because quotation marks are used as string delimiters.

> *Note: Quotation marks are accepted in strings by the ENTER statement.*

Apostrophes and control characters are legal as string characters.

A null string has no value, as distinguished from a blank space which has a value.

Strings are manipulated in string variables. For example:

```
100 A$   =   "THIS IS A STRING"
     ↑                  ↑
  string             string
  variable


200 B$   =   A$(1,10)
     ↑            ↑
  string       substring
  variable     (defined later)


300 C$   =   ""
     ↑         ↑
  string     null string
  variable
```

TERM: STRING VARIABLE

Defined: A variable used to store strings; consists of a single letter (A to Z) followed by a $. For example: A$, Z$, M$.

String variables must be declared before being used if they contain strings longer than one character. See "The String DIM Statement".

When a string variable is declared, its "physical" length is set. The "physical" length is the maximum size string that the variable can accommodate. For example:

    710 DIM A$(72),B$(20),C$(50)

During execution of a program, the "logical" length of a string variable varies. The "logical" length of the variable is the actual number of characters that the string variable contains at any point. For example:

100 DIM A$(72)	(Sets physical length of A$)
200 A$ = "SAMPLE STRING"	(Logical length of A$ is 13)
300 A$ = "LONGER SAMPLE STRING"	(Logical length of A$ is now 20)

TERM: SUBSTRING

Defined: A single character or a set of contiguous characters from within a string variable. The substring is defined by a subscript string variable.

A substring is defined by subscripts placed after the string variable. Characters within a string are numbered from the left starting with one. Subscripts must be positive, non-zero, and less than 32768. Non-integer subscripts are rounded to the nearest integer.

Two subscripts, separated by a comma, specify the first and last characters of the substring. For example:

        100 Z$ = "ABCDEFGH"
        200 PRINT Z$(2,6)

prints the substring

        BCDEF

A single subscript specifies the first character of the substring and implies that all characters following are part of the substring. For example:

        300 PRINT Z$(3)

prints the substring

        CDEFGH

Two equal subscripts specify a single character substring. For example:

        400 PRINT Z$(2,2)

prints the substring

        B

If subscripts specify a substring larger than the physical length of the original string, blanks are appended.

STRINGS AND SUBSTRINGS

A string can be made into a null string. This is done by assigning it the value of a substring whose second subscript is one less than its first. For example:

        100 A$ = B$(6,5)   *(A$ now contains a null string)*

This is the only case in which a smaller second subscript is acceptable in a substring.

Substrings can become strings. For example:

```
100 A$   =    "ABCDEFGH"
200 B$   =    A$(3,5)
300 PRINT B$
```

prints the string

```
CDE
```

because the substring of A$ is now a string in B$.

Substrings can be used as string variables to change characters within a larger string. For example:

```
100 A$   =    "ABCDEFGH"
200 A$(3,5)  =  "123"
300 PRINT A$
```

prints the string

```
AB123FGH
```

Strings, substrings, and string variables can be used with relational operators. They are compared and ordered as entries are in a dictionary. For example:

```
100 IF A$   =    B$ THEN 2000
200 IF A$   ≤    "TEST" THEN 3000
3000 IF A$(5,6)   ≥    B$(7,8) THEN 4000
```

See the STRING IF statement description in this section.

The String DIM Statement

GENERAL FORM:

statement number DIM *string variable (number of characters in string)*

The string DIM statement reserves storage space for strings longer than 1 character; also for matrices (arrays).

The number of characters specified for a string in its DIM statement must be expressed as an integer from 1 to 72.

Each string having more than 1 character must be mentioned in a DIM statement before it is used in the program.

Strings not mentioned in a DIM statement are assumed to have a length of 1 character.

The length mentioned in the DIM statement specifies the maximum number of characters which may be assigned; the actual number of characters assigned may be smaller than this number. See "The LEN Function" in this section for further details.

Matrix dimension specifications may be used in the same DIM statement as string dimensions (example statement 45 below).

EXAMPLES:

```
35 DIM A$ (72), B$(60)
40 DIM Z$ (10)
45 DIM N$ (2), R(5,5), P$(8)
```

The String Assignment Statement

GENERAL FORM:

statement number LET *string variable* = " *string value* "

or

statement number LET *string variable* = *string or substring variable*

or

statement number string variable = " *string value* "

or

statement number string variable = *string or substring variable*

The string assignment statement establishes a value for a string; the value may be a literal value in quotation marks, or a string or substring value.

Strings contain a maximum of 72 characters, enclosed by quotation marks. String variables having more than 1 character must be mentioned in a DIM statement.

Special purpose characters, such as ←, X^c, or quotation marks may not be string characters.

If the source string is longer than the destination string, the source string is truncated at the appropriate point.

EXAMPLES:

```
200 LET A$ = "TEXT OF STRING"
210 B$ = "XXX TEXT !!!"
220 LET C$ = A$(1,4)
230 D$ = B$(4)
240 F$(3,8)=N$
```

The String INPUT Statement

GENERAL FORM:

statement number INPUT *string or substring variable . . .*

The string INPUT statement allows string values to be entered from the user terminal.

Placing a single string variable in an INPUT statement allows the string value to be entered without enclosing it in quotation marks.

If multiple string variables are used in an INPUT statement, each string value must be enclosed in quotation marks, and the values separated by commas. The same convention is true for substring values. Mixed string and numeric values must also be separated by commas.

If a substring subscript extends beyond the boundaries of the input string, the appropriate number of blanks are appended.

Numeric variables may be used in the same INPUT statement as string variables (example statement 55 below).

> *Note:* The ENTER statement (Section III) can be used to input a
> character string. When using the ENTER statement for
> character strings, the string being entered should not be
> enclosed in quotation marks, but may contain quotation
> marks.

EXAMPLES:

```
50 INPUT R$
55 INPUT A$,B$, C9, D10
60 INPUT A$(1,5)
65 INPUT B$(3)
```

Printing Strings

GENERAL FORM:

statement number PRINT *string or substring variable , string or substring variable . . .*

A string PRINT statement causes the current value of the specified string or substring variable to be output to the user's terminal device. The terminal device may be a user terminal or the line printer.

String and numeric values may be mixed in a PRINT statement (example statements 115 and 130 below).

Specifying only one substring parameter causes the entire substring to be printed. For instance, if the value of B3 = 642 and C$ = "WHAT IS YOUR NAME?", example statement 120 prints:

WHAT IS

while statement 115 prints

YOUR NAME?END OF STRING 642

Numeric and string values may be "packed" in PRINT statements without using a "semicolon", as in example statement 115.

O^c and N^c generate a *return* and *linefeed* respectively when printed as string characters.

> *Note:* *The PRINT USING statement (Section VIII) can be used*
> *to provide greater control of format over strings and sub-*
> *strings.*

EXAMPLES:

```
105 PRINT A$
110 PRINT A$, B$, Z$
115 PRINT C$(8) "END OF STRING" B3
120 PRINT C$(1,7)
130 PRINT "THE TOTAL IS:";X5
```

Reading Strings

GENERAL FORM:

statement number READ *string or substring variable , string or substring variable , . . .*

A string READ statement causes the value of a specified string or substring variable to be read from a DATA statement.

A string variable (to be assigned more than 1 character) must be mentioned in a DIM statement before attempting to READ its value.

String or substring values read from a DATA statement must be enclosed in quotation marks, and separated by commas. See "Strings in DATA Statements" in this section.

Only the number of characters specified in the DIM statement may be assigned to a string. Blanks are appended to substrings extending beyond the string dimensions.

Mixed string and numeric values may be read (example statement 310 below); see "The TYP Function", Section IV for description of a data type check which may be used with DATA statements.

EXAMPLES:

```
300 READ C$
305 READ X$, Y$, Z$
310 READ Y$(5), A,B,C5,N$
315 READ Y$(1,4)
```

String IF Statement

GENERAL FORM:

statement no. IF *string var. relational oper. string var.* THEN *statement no.*

A string IF statement compares two strings. If the specified condition is true, control is transferred to the specified statement.

Strings are compared one character at a time, from left to right; the first difference determines the relation. If one string ends before a difference is found, the shorter string is considered the smaller one.

Characters are compared by their ASCII representation. (See STRING EVALUATION BY ASCII CODES, Section IX.)

If substring subscripts extend beyond the length of the string, null characters (rather than blanks) are appended.

String compares may appear only in IF . . . THEN statements and not in conjunction with logical operators (Section VII).

EXAMPLES:

```
340  IF  C$<D$  THEN  800
350  IF  C$>=D$  THEN  900
360  IF  C$#D$  THEN  1000
370  IF  N$(3,5)<R$(9)  THEN  500
380  IF  A$(10)="END"  THEN  400
```

The LEN Function

GENERAL FORM:

statement number statement type LEN *(string variable)* . . .

The LEN function supplies the current (logical) length of the specified string, in number of characters.

DIM merely specifies a maximum string length. The LEN function allows the user to check the actual number of characters currently assigned to a string variable.

Note that LEN is a directly executable command (see Section III), while LEN (. . . $) is a predefined function used only as an operand in a statement. The LEN command gives the working program length; the LEN function gives the current length of a string.

EXAMPLES:

```
469 PRINT LEN (A$ )
479 PRINT LEN (X$)
489 PRINT "TEXT"; LEN(A$); B$, C
499 IF LEN (P$) #5 THEN 600
509 IF LEN (P$) = 5 THEN 609
519 IF LEN (P$) = 5 OR LEN (P$) = 10 THEN 10
529 LET X$(LEN(X$)+1) = "ADDITIONAL SUBSTRING"
   .
   .
   .
600 STOP
609 PRINT "STRING LENGTH = "; LEN (P$)
```

Strings in DATA Statements

GENERAL FORM:

statement number DATA " *string text* ", " *string text* " . . .

The DATA statement specifies data in a program (numeric values may also be used as data).

String values must be enclosed by quotation marks and separated by commas.

String and numeric values may be mixed in a single DATA statement. They must be separated by commas (example 520 below).

Strings up to 72 characters long may be stored in a DATA statement.

See "The TYP Function", Section IV, for description of a data type (string, numeric) check which may be used with DATA statements.

EXAMPLES:

```
500 DATA "NOW IS THE TIME."
510 DATA "HOW", "ARE", "YOU,"
520 DATA 5.172, "NAME?", 6.47,5071
```

Printing Strings on Files

GENERAL FORM:

statement number PRINT # *file number , record number formula ; string variable . . .*

or

statement number PRINT # *file number formula , record number formula ; " string text "*

or

statement number PRINT # *file number formula ; string variable or substring variable . . .*

The PRINT # statement prints string or substring variables on a file.

String and numeric variables may be mixed in a single file or record within a file (example statement 360 below).

The formula for determining the number of 2-character words required for storage of a string on a file is:

$$1 + \frac{\text{number of characters in string}}{2} \quad \text{if the number of characters is even;}$$

$$1 + \frac{\text{number of characters in string} + 1}{2} \quad \text{if the number of characters is odd.}$$

See "The TYP Function", Section IV for description of a data type check.

EXAMPLES:

```
350  PRINT #5; "THIS IS A STRING."
355  PRINT #8; C$, B$, X$, Y$, D$
360  PRINT #7,3; X$, P$, "TEXT", 27.5,R7
365  PRINT #N,R; P$, N, A(5,5), "TEXT"
```

Reading Strings from Files

GENERAL FORM:

statement no. READ # *file no. formula , record no. formula ; string or substring variable . . .*

<div align="center">or</div>

statement no. READ # *file no. formula ; string or substring variable . . .*

The READ # statement reads string and substring values from a file.

String and numeric values may be mixed in a file and in a READ number statement; they must be separated by commas.

See "The TYP Function," Section IV for description of a data type check.

EXAMPLES:

```
710 READ #1, 5; A$, B$
715 READ #2; C$, A1, B2, X
720 READ #3,6; C$(5),X$(4,7),Y$
730 READ #N,P; C$, V$(2,7),R$(9)
```

SECTION VII
Logical Operations

Logical evaluation simply determines whether a given statement or expression is true or false. When applied to numeric values, any non-zero expression is considered "true"; a value of zero is considered "false."

When an expression or statement is logically evaluated by the TSB system, it is assigned one of two numeric values—a 1 if the expression or statement is logically "true," or a 0 if the expression or statement is logically "false."

Logical decisions are used to select one of two or more paths of execution through a program. Executing an IF statement, described in this section, causes the system to perform a specified statement next if the condition in the IF statement is true, and a different statement if the condition is false.

The truth or falsity of a statement or expression can also be determined and printed as a 1 for true or a 0 for false.

RELATIONAL OPERATORS

There are two ways to use the relational operators in logical evaluations:

1.   As a simple check on the numeric value of an expression.

EXAMPLES:

```
150 IF B=7 THEN 600
200 IF A9#27.65 THEN 700
300 IF (Z/10)>=0 THEN 800
```

When a statement is evaluated, when the "IF" condition is currently true (for example, in statement 150, if B=7), then control is transferred to the specified statement. When the condition is false, the next sequential statement after 150 is executed.

Note that the numeric value produced by the logical evaluation is unimportant when the relational operators are used in this way. The user is concerned only with the presence or absence of the condition indicated in the IF statement.

2. As a check on the numeric value produced by logically evaluating an expression, that is: "true" = 1, "false" = 0.

EXAMPLES:

```
610  LET  X=27
615  PRINT  X=27
620  PRINT  X#27
630  PRINT  X>=27
```

The example PRINT statements give the numeric values produced by logical evaluation. For instance, statement 615 is interpreted by TSB as "Print 1 if X equals 27, 0 if X does not equal 27." There are only two logical alternatives; 1 is used to represent "true," and 0 "false."

The numeric value of the logical evaluation is dependent on, but distinct from, the value of the expression. In the example above, X equals 27, but the numeric value of the logical expression X=27 is 1, since it describes a "true" condition.

BOOLEAN OPERATORS

There are two ways to use the Boolean operators.

1. As logical checks on the value of an expression or expressions.

EXAMPLES:

```
510  IF  A1  OR  B  THEN  670
520  IF  B3  AND  C9  THEN  680
530  IF  NOT  C9  THEN  690
540  IF  X  THEN  700
```

Statement 510 is interpreted: "if either A1 is true (has a non-zero value) or B is true (has a non-zero value) then transfer control to statement 670."

Similarly, statement 540 is interpreted: "if X is true (has a non-zero value) then transfer control to statement 700."

The Boolean operators evaluate expressions for their logical values only; these are "true" = any non-zero value, "false" = zero. For example, if B3 = 9 and C9 = –5, statement 520 would evaluate to "true," since both B3 and C9 have a non-zero value.

2. As a check on the numeric value produced by logically evaluating an expression, that is: "true" = 1, "false" = 0.

EXAMPLES:

```
490 LET B = C = 7
500 PRINT B AND C
510 PRINT C OR B
520 PRINT NOT B
```

Statements 500 - 520 returns a numeric value of either: 1, indicating that the statement has a logical value of "true," or 0, indicating a logical value of "false."

Note that the criteria for determining the logical values are:

true   =   any non-zero expression value

false   =   an expression value of 0.

The numeric value 1 or 0 is assigned accordingly.

EXAMPLES:

The following examples show some of the possibilities for combining logical operators in a statement.

It is advisable to use parentheses whenever possible when combining logical operators.

```
310 IF (A9 MIN B7)<0 OR (A9 MAX B7)>100 THEN 900
310 PRINT (A>B) AND (X<Y)
320 LET C = NOT D
330 IF (C7 OR D4) AND (X2 OR Y3) THEN 930
340 IF (A1 AND B2) AND (X2 AND Y3) THEN 940
```

The numerical value of "true" or "false" may be used in algebraic operations. This sequence counts the number of zero values in a file:

```
 90 LET X = 0
100 FOR I = 1 TO N
110 READ #1; A
120 LET X = X+(A=0)
130 NEXT I
140 PRINT N; "VALUES WERE READ."
150 PRINT X; "WERE ZEROES."
160 PRINT (N-X); "WERE NONZERO."
```

Note that X is increased by 1 or 0 each time A is read; when A = 0, the expression A = 0 is true, and X is increased by 1. N must have been given a value earlier in the program.

SECTION VIII
Formatted Output

The PRINT USING and IMAGE statements give the user more explicit and exact control over the format of his output. Numbers can be printed in three forms - - integer, fixed-point, or floating point - - with control of + and − signs. Strings may be printed in specified fields. Blanks can be inserted wherever needed. Carriage return and linefeed can be controlled. PRINT USING requires more programming effort than a simple PRINT, but it provides the ability to output data in whatever format the programmer needs.

DEFINITIONS

Term	Defined in TSB
FORMATTED OUTPUT	Similar to normal output (PRINT statement) except that, in addition to an expression list of output values, the PRINT USING statement also specifies a format string that determines the form in which the values are printed. Formatted output can be used to print lines of any character length, up to the maximum capability of the output device used.
EXPRESSION LIST	A list of expressions and string variables separated by commas and optionally interspersed with space functions. An expression list must not contain literal strings.
FORMAT STRING	A string of up to 72 characters, consisting of an optional carriage control character followed by a list of format specifications separated by commas or slashes (/).
FORMAT SPECIFICATION	A series of format characters and repetition factors that determines the format (field width, decimal point, sign, etc.) of one item in the expression list. Can also be a literal string in certain situations. Format specifications can be gathered into a repeatable group through the use of parentheses.

Term	Defined in TSB
FORMAT CHARACTERS	The characters A, X, D, S, ., and E are used to specify output fields for strings and numbers.
REPETITION FACTOR	An unsigned integer (e.g., 3, 6, 12, 32) that is placed before a format character or group of format specifications in order to repeat it (e.g., 3A = AAA; 2(3A,4A) = 3A,4A,3A,4A). The repetition factor must be between 1 and 72 inclusive.
SLASH	A delimiter (/) used to separate specifications when a carriage return-linefeed is desired before processing the next specification. Multiple slashes may be used (///).
LITERAL STRING	Any sequence of characters, other than quote marks (''), that is surrounded by quote marks and is to be printed as it appears.
SPACE FUNCTIONS	Three functions can appear in an expression list: TAB(x) - Tabs out to column x before printing next item. ($x > 72$ is legal only in a PRINT USING statement.) LIN(x) - Skips | x | lines before printing next item. (If $x < 0$, no carriage return is generated. If x=0, only a carriage return is generated.) SPA(x) - Skips x spaces before printing next item.
CARRIAGE CONTROL CHARACTERS	At the beginning of any format string there may appear one of three optional characters set off by a comma: + means to suppress linefeed. − means to suppress carriage return. # means to suppress carriage return and linefeed. These characters specify action to be taken after the PRINT USING statement is complete. If no character is specified, the default condition is a carriage return and linefeed.

Summary

```
┌─────────────────────────────┐          ┌───────────┐
│   FORMAT CHARACTERS         │          │    ( )    │
│   & REPETITION FACTORS      │          │           │
└─────────────────────────────┘          └───────────┘
              └──────────────┬──────────────┘
                   ┌─────────────────────┐
                   │    FORMAT SPECS     │
                   └─────────────────────┘
                              │
┌─────────────────────────┐   │   ┌──────────────────────────────┐
│   CARRIAGE CONTROL      │   │   │   DELIMITERS: / and ,        │
│   CHARACTERS            │   ▼   │                              │
└─────────────────────────┘       └──────────────────────────────┘
         └──────────────────┬──────────────────┘
                   ┌─────────────────────┐
                   │   FORMAT STRING     │
                   └─────────────────────┘
                              │
   PRINT USING                │        ┌─────┐  ┌────────────────────┐
                              ▼        │  ;  │  │  EXPRESSION LIST   │
                                       └─────┘  └────────────────────┘
         └──────────────────┬──────────────────────┘
                   ┌─────────────────────────────┐
                   │  PRINT USING STATEMENT      │
                   │  FORMATTED OUTPUT           │
                   └─────────────────────────────┘
```

EXAMPLES:

```
PRINT USING      "DDD.DDD"        ;            Z1
                 └────┬────┘                  └─┬─┘
                 FORMAT STRING           EXPRESSION LIST


PRINT USING    "2X,3(3D.3D,2X)"   ;        Z1,Z2,Z3
               └──────┬───────┘            └───┬────┘
                 FORMAT STRING           EXPRESSION LIST
```

STRING FORMAT SPECIFICATIONS

Format Characters Used

A - calls for one ASCII character to be output from a string in the expression list.

X - specifies that a blank be printed next.

nA - calls for n ASCII characters (n = repetition factor).

nX - specifies that n blanks be printed.

Combination Rules

Any combination of X's, A's, and repetition factors specifies a legal STRING FORMAT SPECIFICATION. When such a specification is encountered in a format, the next item in the expression list must be a string.

FORMAT EXAMPLES:

```
AAAA   ⎫
4A     ⎬  equivalent
2A2A   ⎭

4X          special case (all blanks, so no variable required)
AXAXAXA     alternate characters and blanks
2X20A
```

OUTPUT EXAMPLES:

Format Spec	Contents of String Variable	Format of Output
6A	ABCDEF	ABCDEF
5A	ABCDEF	ABCDE
8A	ABCDEF	ABCDEF ∧∧
2X6A	ABCDEF	∧∧ ABCDEF
AXAXAXAXAXA	ABCDEF	A∧B∧C∧D∧E∧F

The string is left-justified in the field and any leftover spaces are filled with blanks. If the string variable contains more characters than the specification allows, characters on the right are truncated.

INTEGER FORMAT SPECIFICATIONS

Format Characters Used

D - calls for one decimal digit to be printed from a number in the expression list.

nD - calls for n contiguous decimal digits to be printed from a number in the expression list.

X - specifies that a blank is to be printed within the field for the number (nX is also allowed).

S - specifies that the sign (+ or -) of the number is to be printed.

Combination Rules

Any combination of X and D is allowed, but at least one D must be present and only one S is allowed. When such a specification occurs in a format, the next item in the expression list must be a number. This number is rounded to an integer and printed right-justified. Although the requested number of digits will be printed, only six can be guaranteed to be significant.

FORMAT EXAMPLES:

```
DDDD  ⎫
4D    ⎬  equivalent
2DDD  ⎪
2D2D  ⎭

2DX3D
SDDD
S4D
DX3DS
```

OUTPUT EXAMPLES:

Format Spec	Value	Format of Output
4D	1234	1234
S4D	1234	+1234
4DS	1234	1234+
5D	1234	ʌ1234
4D	1234.8	1235
DXDDD	1234	1ʌ234
S10D	1234	ʌʌʌʌʌʌ +1234
DSDDD	1234	1+234
5D	-1234	-1234
4D	1234.2	1234

If there is not enough room in the field for the number (i.e., the number of digits is greater than the number of D's in the format spec), then the value is printed on a separate line in a floating-point format (SD.5DE) so that the programmer can analyze what went wrong.

If an S precedes all D's, the sign is printed immediately preceding the first digit of the number. If an S appears past the first D, the sign is printed at the location of the S.

If an S is not included in the format, then an extra D should be provided if the value could possibly be negative. When the value is negative, the – sign is always printed preceding the most significant digit and a space must be provided for it with a D or the field may overflow.

The ability to insert blanks can be combined with the ability to overprint (carriage control) in order to produce useful results. For example, large numbers can be printed with blanks left in the correct spots for commas to be inserted after each group of three digits (e.g., $10,937).

FIXED-POINT FORMAT SPECIFICATIONS

Format Characters Used

Same as INTEGER FORMAT, plus

. – specifies the location of the decimal point.

Combination Rules

Any combination of D and X to the left and right of the decimal point is allowed, but at least one D must be present and only one S and one "." are allowed. For this specification, the next item in the expression list must be a number. The digits to the right of the decimal point are rounded to fit in the field. Leading zeros to the left are suppressed, but trailing zeros are always printed.

FORMAT EXAMPLES:

```
DDD.DDD      ⎫
DDD.3D       ⎪
3D.3D        ⎬  equivalent
3D.DDD       ⎭

S3D.3D
DXDXDX.DDXD
XD6X4D.8D
DDSDD.3D
```

OUTPUT EXAMPLES:

Format Spec	Value	Format of Output
3D.4D	465.465	465.4650
4D.2D	465.465	∧465.47
4D.3D	-465.465	-465.465
SDD2D.D	465.465	∧+465.5
S2D.4D	.465	∧+0.4650
S.4D	.465	+.4650
D.4D	-.465	-.4650
2D.4D	-.465	-0.4650

If the number to be printed has no digits to the left of the decimal point but D's are provided to the left, then a zero ("0") will be printed in the rightmost D on the left side. If an S is provided to the left, it is moved to the right through D's and X's until it comes to the first non-blank character. If an S is not provided and the number is negative, then one of three things will happen: 1) no D's to the left causes overflow; 2) one D to the left will be used for the "-" sign and the "0" will not be printed; or 3) two or more D's to the left, then the "-" and "0" will be printed in the positions reserved by the rightmost two D's.

FLOATING-POINT FORMAT SPECIFICATIONS

Format Characters Used

Same as FIXED-POINT FORMAT, plus

E - signifies floating point format.

X - as defined earlier may follow E.

Combination Rules

Any legal INTEGER or FIXED-POINT format specification followed by an E is a legal FLOATING-POINT format. The E stands for "exponent" and signifies a four-character field consisting of an "E" followed by "+" or "-" and two decimal digits. When 10 is raised to the power printed after E and multiplied by the number in the integer or fixed-point field, the result is the value being output. This format is useful for numbers that are very large or very small. For example, $.00005 = .5 \times 10^{-4} = .5E-4$. X's may follow the E and they cause Blanks to be printed between the E and the exponent sign.

FORMAT EXAMPLES:

```
SD.5DE
DDD.DDDXEX
SD.8DXE
S6DE
S6D.E
S6D.XE
S6D.DDDE
```

OUTPUT EXAMPLES:

Format Spec	Value	Format of Output
SDXE	4.82716×10^{21}	+5∧E+21
DDDD.DDE	SAME	4827.16E+18
S5DX.X5DEX	SAME	∧∧∧+48 ∧.∧ 27159E∧+20
SD.5DE	SAME	+4.82716E+21
S.10DE3X	SAME	+.4827159382E∧∧∧+22

Note again that the format can specify an unlimited number of digits in a specification, but only six of these are guaranteed accurate. When more than six digits are requested, non-significant digits are printed as in the preceding examples.

To produce the output, the output value from the expression list is multiplied or divided by 10, the number of times necessary to fit the value into the field. It is then rounded from the right, and the exponent is adjusted to account for the multiplications or divisions.

If the format allows for more digits than there are significant digits in the output value, two rules are followed:

1. If there are more than 6D's on the right side of the decimal point, the leftmost digit is printed in the first D (if any) to the left of the decimal point or the first D to the right of the decimal point; extra D's beyond 7 on the right are filled with non-significant digits. In the following examples, the arrow indicates the position of the leftmost digit printed:

      DD.40D   XX.DD40D   40DD.40D
      ↑   ↑   ↑

2. If there are less than 7 D's on the right side of the decimal point, the leftmost digit is printed in the seventh D position from the right (or the leftmost if there are not 7). In the following examples, the arrow indicates the position of the leftmost digit printed:

      6DDDD.DDDD   DD.DD   D.6D
      ↑   ↑   ↑

POSITION OF THE SIGN

1. When S is used.

    If S precedes any D, the sign position is moved to the right through X's and D's and is printed immediately to the left of the first non-blank character. If the number to be printed is a fraction with no digits to the left of the decimal point and any D's appear on the left of the decimal point, then a "0" appears in the rightmost D and the sign floats up to that "0".

    If S is preceded by one or more D's, the sign is printed at the position of the S and does not **float.**

2. When S is not used.

    When the number is negative, an extra D must be present to reserve a place for the sign. The position of the sign is moved through unused D's and X's to the first non-blank character. If not enough D's are provided for all the significant digits and sign of a negative number, then the field overflows and the number is printed on a separate line in SD.5DE format.

GROUPED FORMAT SPECIFICATIONS

One or more format specifications can be gathered within parentheses to make a group. This group must be repeated by prefacing it with a repetition factor between 1 and 72 inclusive. Within the parentheses, the specifications must be separated by commas or slashes and the group must be set off from other specifications by a comma or slashes, just as if it were a single specification. Groups can be nested two levels deep.

EXAMPLES:

```
4(10A,2X,4D,2X//)
3(10D,2(3DX,4DX),4A)
3D.3D//3(20A,6D,4(2A2X)/)
```

FORMAT STRINGS

Defined: A collection of format specifications (or groups of format specifications) set off by commas or slashes and optionally preceded by a carriage control character set off by a comma. One format string is used by one PRINT USING statement. The first character of a format string must not be a slash (/) or a comma.

EXAMPLES:

```
+,20A,2X,S4D.2D
6D,2X,6DSX,13AXAX,2(4D,2X,3AX)
-,20A/20X20A/40A20X/
```

TERM: EXPRESSION LIST

Defined: The list of items to be printed using the format string. The items must be separated by commas (not semicolons), and the list must not contain any literal strings. The types of the items (numerical or string) must match the types of items called for in the format string. Space functions (SPA,LIN,TAB) may appear in the list.

PRINT USING Statement

GENERAL FORM:

statement number PRINT USING *format string ; expression list*

This statement is used to print out data according to a specified format.

The *format string* can be specified in one of three ways:

a.   an actual string ("6D,X20A")

b.   a string variable containing the format string (A$,B$(5,20))

c.   the statement number of an IMAGE statement containing the format string (200).

The *expression list* is a list of expressions separated by commas; the semicolon and expression list are optional.

When the PRINT USING statement is executed, the format string is examined and the carriage control character, if any, is saved. Each specification is extracted and examined. If it calls for a string or numerical item, the next expression in the expression list is taken and printed according to the specification.

If there are no more specifications or the specification is of the wrong type, execution of PRINT USING terminates.

If the specification does not require an item from the list (e.g., a blank or literal specification), the specification is printed without examining the expression list.

If the end of the format string is reached before the end of the expression list, processing continues from the beginning of the format string.

When all expressions have been printed, a carriage return and linefeed (modified by the carriage control character) are printed.

EXAMPLES:

```
300  PRINT  USING  200;A,B4,C$,TAB(50),67.78
400  PRINT  USING  A$;A,A3,C$,D$
500  PRINT  USING  "6DX,25A";A,A$
```

In the following examples, the variables have these values: A = +12345, B = -1234, C = 123, D = 12, E = -12345, F = 123456, G = -1, H = 1234.

```
100  PRINT  USING  "3(S6D2X)/ ";A,B,C,D,E,F
```

Output

```
^+12345 ∧∧∧∧ -1234 ∧∧∧∧∧ +123
∧∧∧∧ +12 ∧∧∧ -12345 ∧∧ +123456
```

```
100  PRINT  USING  "3(S6D2X)/";A,G
```

Output

```
∧+12345 ∧∧∧∧∧∧∧ -1
```

```
50   IMAGE  "MONEY ",6DX,"COST ",6DX,"INPUT ",6DX
100  PRINT  USING  50;H,D
```

Output

```
MONEY ∧∧∧ 1234 ∧COST ∧∧∧∧∧ 12
```

MAT PRINT USING Statement

GENERAL FORM:

statement number MAT PRINT USING *format string ; matrix list*

This statement is used to print out data from matrices in a specified format.

Matrices referenced in MAT PRINT USING statements must be previously dimensioned.

The *format string* is the same as in PRINT USING except that it must not contain any string specifications.

The *matrix list* is a list of matrices separated by commas. (The semicolon and matrix list are optional.) Space functions are allowed in the matrix list.

As in MAT PRINT, the matrices are printed in row by row order.

EXAMPLES:

```
200 MAT PRINT USING 300;A,B,SPA(M),C
350 MAT PRINT USING A$; B,N,M
400 MAT PRINT USING "SD.5DE2X";K
```

```
10   DIM A(5,5)
     .
     .
     .
100 PRINT USING "6(SD.5DE)/";A
```

FORMAT IN A STRING VARIABLE: One way to specify the format string in a PRINT USING or MAT PRINT USING statement is by using a string variable that contains the format string. This allows the programmer to change the format during the execution of the program. The following excerpt from a sample program shows what can be done:

```
100  LET A$ = "DD,^^^^^ DD"
110  IF X<Y THEN 130
120  A$(4,8) = "SD.E,"
130  PRINT USING A$; ....
```

If X is not less than Y, then the format string becomes

> DD,SD.E,DD

instead of

> DD,DD

IMAGE Statement

GENERAL FORM:

statement number IMAGE *format string*

The IMAGE statement is used to specify a format to be used in a PRINT USING statement.

An IMAGE statement is one means by which a literal string can be introduced into a format string. Literal strings are printed exactly as they appear in the format string, similar to the way blanks are printed in a blank specification.

The *format string* is any legal format string; it is not enclosed in quotes and can therefore contain literal strings as format specifications.

EXAMPLES:

```
100  IMAGE 6D,"LITERAL STRING",SD.5DE
200  IMAGE XDDXDD.DDE,20A,3D
```

USING CARRIAGE CONTROL

This example demonstrates the use of the LIN function (statement 5), the carriage control characters (statements 20, 40, and 60), and literal strings in IMAGE statements.

PROGRAM:

```
5   PRINT LIN(5)
10   PRINT   USING 20
20   IMAGE #,"# "
30   PRINT   USING 40
40   IMAGE -,"SUPPRESSES LINEFEED AND CARRIAGE RTN"
50   PRINT   USING 60
60   IMAGE +,"- SUPPRESSES CARRIAGE RTN"
70   PRINT   USING 80
80   IMAGE "AND + SUPPRESSES LINEFEED."
90   END
```

OUTPUT:

```
# SUPPRESSES LINEFEED AND CARRIAGE RTN
AND + SUPPRESSES LINEFEED.                - SUPPRESSES CARRIAGE RTN

DONE
```

NUMERICAL OUTPUT

This example program prints out the values of 2↑N and (-2)↑N, where N varies from -5 to 20.
Floating-point and integer formats are used (statement 350).

PROGRAM:

```
200   PRINT  USING 210
210   IMAGE " N ",3X,"2 TO THE N",3X,"-2 TO THE N"
300   FOR N=-5 TO 20
350   PRINT  USING "3D,2X,SD.5DE,2X,SD.5DE";N,2↑N,(-2)↑N
360   NEXT N
1000  END
```

OUTPUT:

```
N     2 TO THE N    -2 TO THE N
-5   +3.12500E-02   -3.12500E-02
-4   +6.25000E-02   +6.25000E-02
-3   +1.25000E-01   -1.25000E-01
-2   +2.50000E-01   +2.50000E-01
-1   +5.00000E-01   -5.00000E-01
 0   +1.00000E+00   +1.00000E+00
 1   +2.00000E+00   -2.00000E+00
 2   +4.00000E+00   +4.00000E+00
 3   +8.00000E+00   -8.00000E+00
 4   +1.60000E+01   +1.60000E+01
 5   +3.20000E+01   -3.20000E+01
 6   +6.40000E+01   +6.40000E+01
 7   +1.28000E+02   -1.28000E+02
 8   +2.56000E+02   +2.56000E+02
 9   +5.12000E+02   -5.12000E+02
10   +1.02400E+03   +1.02400E+03
11   +2.04800E+03   -2.04800E+03
12   +4.09600E+03   +4.09600E+03
13   +8.19200E+03   -8.19200E+03
14   +1.63840E+04   +1.63840E+04
15   +3.27680E+04   -3.27680E+04
16   +6.55360E+04   +6.55360E+04
17   +1.31072E+05   -1.31072E+05
18   +2.62144E+05   +2.62144E+05
19   +5.24288E+05   -5.24288E+05
20   +1.04858E+06   +1.04858E+06
```

REPORT GENERATION

This program is a sample report generator. It first requests a school number from the terminal, then reads and prints out information about the school's teachers from a file. Note that a carriage control character is used to advantage (statement 100), slashes (/) are used (statement 200), string and fixed-point fields are used (statement 210), and an error occurs in the output for the eighth teacher (number too large for field; therefore, it is printed in E format on a separate line).

PROGRAM:

```
10   REM: THIS PROGRAM GENERATES A REPORT ON TEACHERS.
50   DIM A$[25],B$[19],C$[19]
60   FILES SCH1,SCH2,SCH3,SCH4,SCH5
100    IMAGE #,"ENTER SCHOOL NUMBER:"
150    IMAGE "TEACHER",13X,"SUBJECT",13X,"SALARY",4X,"ATTND."
175    IMAGE "-------",13X,"-------",13X,"------",4X,"------"/
200    IMAGE "CENTRAL CITY SCHOOL DISTRICT"/"DAILY REPORT OF ",25A//
210    IMAGE 20A,20A,"$",DDD.DD,DD.DDDD
230    PRINT  USING 100
250    INPUT Z
260    READ #Z;A$,N
270    PRINT LIN(6)
500    PRINT  USING 200;A$
550    PRINT  USING 150
555    PRINT  USING 175
557    FOR A1=1 TO N
560    READ #1;B$,C$,A,B
600    PRINT  USING 210;B$,C$,A,TAB(50),B
620    NEXT A1
1000   END
```

OUTPUT:

```
ENTER SCHOOL NUMBER:?1



CENTRAL CITY SCHOOL DISTRICT
DAILY REPORT OF B. BAKER HIGH SCHOOL



TEACHER                 SUBJECT             SALARY          ATTND.

MISS BROOKS             ENGLISH             $450.34         12.5000
MISS CRABTREE           REM. READING        $400.00         64.3200
MISS GRUNDIE            HISTORY             $350.00          1.0010
MRS. HUMPREY            SPELLING            $700.00         99.9900
COLONEL MUSTARD         CRIMINOLOGY         $700.00         21.4500
MISS PEACH              LIFE PREPARATION    $232.00         23.2320
PROF. PLUM              AGRICULTURE         $777.77         65.0050
MISS H. PRYNNE          SOCIAL STUDIES      $100.25
+5.00500E+02

MISS SCARLETT           P.E.                $205.10         25.0000
MR. SIR                 HOME ROOM           $890.00         99.9000
MR. T. TIM              MUSIC               $ 10.99          0.0500
MR. WEATHERBY           ECONOMICS           $767.99         10.0400
```

FATAL ERRORS

These errors cause termination of execution of the PRINT USING statement. An appropriate message is printed, along with the format specification that caused the error.

1. The replicator is outside the range $1 \leqslant n \leqslant 72$.
2. Appearance of a D,S,E or . in a string specification.
3. Appearance of an A in an integer specification, a fixed specification, or a floating specification.
4. Appearance of any character other than A,X,D,S,E,/ or . in any specification but literal.
5. A comma followed by a slash.
6. More than two levels of parentheses.
7. No D in a fixed or floating specification.
8. An S in a blank specification.
9. String expression attempted to output in non-string format.
10. A slash followed by a comma.
11. Two or more E's or . in a specification.

12. Literal string not separated by delimiters.
13. Missing quotes in a literal.
14. Specifications enclosed in parentheses without a replicator.
15. Specified statement is not IMAGE.
16. Attempt to print number with string format.

NON-FATAL ERRORS

These errors do not cause termination of the PRINT USING statement. The action taken is indicated.

1. String specification field too narrow - - truncate string on right.

2. Field too narrow for integer or integer part of fixed specification - - number is printed in SD.5DE format on next line and printing resumes on following line.

3. Field too narrow for fraction part of fixed or floating specification - - round from right to fit into field.

4. Specification requires the printing of more than 46 digits - - 46 digits will be printed preceded by blanks filling the rest of the field.

5. More than one S - - subsequent S's are ignored.

SECTION IX

For the Professional

This section contains the most precise reference authority - - the syntax requirements of Time-shared BASIC. The syntax requirements are explicit and unambiguous. They may be used in all cases to clarify descriptions of BASIC language features presented in other sections.

The other subsections give technical information of interest to the sophisticated user.

SYNTAX REQUIREMENTS OF TSB

Legend

:: = "is defined as . . ."

| "or"

< > enclose an element of Time Shared BASIC

Language Rules

1. Exponents have 1 or 2 digit integers only.
2. A <parameter> primary appears only in the defining formula of a <DEF statement>.
3. A <sequence number> must lie between 1 and 9999 inclusive.
4. An array bound must lie between 1 and 9999 inclusive; a string variable bound must lie between 1 and 72 inclusive.

5.   The character string for a <REM statement> may include the character ".

6.   An array may not be transposed into itself, nor may it be both an operand and the result of a matrix multiplication.

7.   A character string that is not a literal can contain the character ", through the use of the ENTER statement.

8.   A replicator must lie between 1 and 72 inclusive.

> *Note:*   *Parentheses, ( ) , and square brackets [ ] , are accepted*
> *interchangeably by the syntax analyzer.*

\<constant>	::=	\<number>|+\<number>|-\<number>|\<literal string>
\<number>	::=	\<decimal number>|\<decimal number>\<exponent part>
\<decimal number>	::=	\<integer>|\<integer>.|\<integer>.\<integer>|.\<integer>
\<integer>	::=	\<digit>|\<integer>\<digit>
\<digit>	::=	Ø|1|2|3|4|5|6|7|8|9
\<exponent part>	::=	E\<integer>|E+\<integer>|E-integer (see rule 1)
\<literal string>	::=	"\<character string>"
\<character string>	::=	\<character>|\<character string>\<character> *(See Rule 7.)*
\<character>	::=	Any ASCII character except null, line feed, return, x-off, ←, ", rubout, and under-score
\<variable>	::=	\<simple variable>|\<subscripted variable>
\<simple variable>	::=	\<letter>|\<letter>\<digit>
\<letter>	::=	A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
\<subscripted variable>	::=	\<letter>(\<sublist>)
\<sublist>	::=	\<expression>|\<expression>,\<expression>
\<string variable>	::=	\<string simple variable>|\<string simple variable>(\<sublist>)
\<string simple variable>	::=	\<letter>$
\<expression>	::=	\<conjunction>|\<expression>OR\<conjunction>
\<conjunction>	::=	\<relation>|\<conjunction>AND\<relation>
\<relation>	::=	\<minmax>|\<minmax>\<relational operator>\<minmax>
\<minmax>	::=	\<sum>|\<minmax>MIN\<sum>|\<minmax>MAX\<sum>
\<sum>	::=	\<term>|\<sum>+\<term>|\<sum>-\<term>
\<term>	::=	\<subterm>|\<term>*\<subterm>|\<term>/\<subterm>
\<subterm>	::=	\<denial>|\<signed factor>

```
<leftpart>                      ::=  <variable>=|<leftpart><variable>=

<IF statement>                  ::=  IF<decision expression>THEN<sequence number>|
                                     IF END #<file formula>THEN<sequence number>

<decision expression>           ::=  <expression>|
                                     <comparison string 1><relational operator>
                                                       <comparison string 2>

<comparison string 1>           ::=  <string variable>

<comparison string 2>           ::=  <string variable>|<literal string>

<GOTO statement>                ::=  GOTO<sequence number>
                                     GOTO<expression>OF<sequence list>

<sequence list>                 ::=  <sequence number>|<sequence list>,<sequence number>

<GOSUB statement>               ::=  GOSUB<sequence number>|
                                     GOSUB<expression>OF<sequence list>

<RETURN statement>              ::=  RETURN

<FOR statement>                 ::=  FOR<for variable>=<initial value>TO<final value>|
                                     FOR<for variable>=<initial value>TO<final value>
                                                       STEP<step size>

<for variable>                  ::=  <simple variable>

<initial value>                 ::=  <expression>

<final value>                   ::=  <expression>

<step size>                     ::=  <expression>

<NEXT statement>                ::=  NEXT<for variable>

<STOP statement>                ::=  STOP

<END statement>                 ::=  END

<DATA statement>                ::=  DATA<constant>|<DATA statement>,<constant>

<READ statement>                ::=  READ<variable list>|READ<file reference>|
                                     READ<file reference>;<variable list>

<variable list>                 ::=  <read variable>|<variable list>,<read variable>

<read variable>                 ::=  <variable>|<destination string>

<INPUT statement>               ::=  INPUT<variable list>
```

```
<ENTER statement>                    ::=   ENTER #<variable>|
                                           ENTER<expression>,<variable>,<variable>|
                                           ENTER<expression>,<variable>,<string variable>|
                                           ENTER #<variable>,<expression>,<variable>,
                                                       <variable>|
                                           ENTER #<variable>,<expression>,<variable>,
                                                       <string variable>

<PRINT statement>                    ::    <type statement>|<file write statement>|
                                           PRINT<file reference>

<type statement>                     ::=   <print 1>|<print 2>

<print 1>                            ::=   PRINT|<print 2>,|<print 2>;|<print 3>

<print 2>                            ::=   <print 1><print expression>|<print 3>

<print 3>                            ::=   <type statement><literal string>

<print expression>                   ::=   <expression>|<special function>|
                                           <source string>

<A part>                             ::=   A|<A part>A|<replicator><A part>

<D part>                             ::=   D|<D part>D|<replicator><D part>

<X part>                             ::=   X|<X part>X|<replicator><X part>

<replicator>                         ::=   <integer>

<empty>                              ::=

<string spec. comp.>                 ::=   <A part>|<X part>

<string spec. 1>                     ::=   <string spec. comp.>|
                                           <string spec. comp.><string spec. 1>

<string spec. 2>                     ::=   <string spec. 1>|
                                           <empty>

<string spec.>                       ::=   <string spec. 2><A part><string spec 2>

<integer spec. comp.>                ::=   <D part>|<X part>|S|<empty>

<integer spec.>                      ::=   <D part>|<integer spec. comp>
                                               <integer spec.><integer spec. comp.>

<fraction spec.>                     ::=   <integer spec. comp.>|
                                           <fraction spec.><integer spec. comp.>

<fixed spec.>                        ::=   <integer spec.><fraction spec.>|
                                           <fraction spec.>.<integer spec.>
```

```
<floating spec>                  ::=  <fixed spec.>E|<integer spec.>E|
                                      <floating spec.><X part>

<format list element>            ::=  <string spec.>|<fixed spec.>|
                                      <floating spec.>|<integer spec.>|
                                      <X part>|<literal string>

<format list>                    ::=  <format list element>|
                                      <format list element>,<format list>|
                                      <replicator>(<format list>)|
                                      <format list>/<format list element>|
                                      <format list>/

<carriage control>               ::=  +|-|#

<format string>                  ::=  <format list>|<carriage control>,
                                          <format list>

<special function type>          ::=  TAB|LIN|SPA

<special function>               ::=  <special function type>(<expression>)

<expression list>                ::=  <expression list element>|
                                      <expression list>,<expression list element>

<expression list element>        ::=  =<expression>|<special function>

<PRINT USING 1>                  ::=  PRINT USING"<format string>"|
                                      PRINT USING<sequence number>|
                                      PRINT USING<string variable>

<PRINT USING statement>          ::=  <PRINT USING 1>;<expression list>|
                                      <PRINT USING 1>

<IMAGE statement>                ::=  IMAGE<format string>

<file write statement>           ::=  PRINT<file reference>;<write expression>|
                                      <file write statement>,<write expression>|
                                      <file write statement>;<write expression>|
                                      <file write statement><literal string>|
                                      <file write statement><literal string>
                                                      <write expression>

<write expression>               ::=  <expression>|END|<source string>

<RESTORE statement>              ::=  RESTORE|RESTORE<sequence number>

<DIM statement>                  ::=  DIM<dimspec>|<DIM statement>,<dimspec>
```

```
<COM statement>                    ::=  COM<com list element>|
                                         <COM statement>,<com list element>

<com list element>                 ::=  <simple variable>|<string simple variable>|
                                         <dimspec>

<dimspec>                          ::=  <array identifier>(<bound>)|
                                         <array identifier>(<bound>,<bound>)
                                         <string simple variable>(<bound>)

<bound>                            ::=  <integer> (see rule 4)

<DEF statement>                    ::=  DEF<function identifier>(<parameter>)=<expression>

<FILES statement>                  ::=  FILES<name>|FILES $<name>|
                                         FILES* name>|FILES*|
                                         <FILES statement>,<name>|
                                         <FILES statement>,$<name>|
                                         <FILES statement>,*<name>|
                                         <FILES statement>,*

<name>                             ::=  a string of up to 6 printing <character>'s
                                         except comma, and not beginning with "$" or
                                         "*".

<REM statement>                    ::=  REM<character string> (see rule 5)

<CHAIN statement>                  ::=  CHAIN<source string>|
                                         CHAIN<source string>,<expression>

<MAT statement>                    ::=  <MAT READ statement>|<MAT INPUT statement>|
                                         <MAT PRINT statement>|
                                         <MAT initialization statement>|
                                         <MAT assignment statement>

 <MAT READ statement>              ::=  MAT READ<actual array>|
                                         MAT READ<file reference>;<actual array>
                                         <MAT READ statement>,<actual array>

<actual array>                     ::=  <array identifier>|<array identifier>(<dimensions>)

<dimensions>                       ::=  <expression>|<expression>,<expression>

<MAT INPUT statement>              ::=  MAT INPUT<actual array>|
                                         <MAT INPUT statement>,<actual array>

<MAT PRINT statement>              ::=  <MAT PRINT 1>|<MAT PRINT 2>

<MAT PRINT 1>                      ::=  MAT PRINT<array identifier>|
                                         MAT PRINT<file reference>;<array identifier>|
                                         <MAT PRINT 2><array identifier>
```

```
<MAT PRINT 2>                      ::=   <MAT PRINT 1>,|<MAT PRINT 1>;

<MAT PRINT USING statement>        ::=   <MAT PRINT USING 1>|
                                         <MAT PRINT USING 1>;<array identifier list>

<MAT PRINT USING 1>                ::=   <MAT PRINT USING<">format string>"|
                                         <MAT PRINT USING><sequence number>|
                                         <MAT PRINT USING><string variable>

<array identifier list>            ::=   <array identifier list element>|
                                         <array identifier list>,
                                             <array identifier list element>

<array identifier list element>    ::=   <array identifier>|<special function>

<MAT initialization statement>     ::=   MAT<array identifier>=<initialization function>|
                                         MAT<array identifier>=<initialization function>
                                                                (<dimensions>)

<initialization function>          ::=   ZER|CON|IDN

<MAT assignment statement>         ::=   MAT<array identifier>=<array identifier>|
               (rule 6)                  MAT<array identifier>=<array identifier>
                                         <mat operator><array identifier>
                                         MAT<array identifier>=INV(<array identifier>)|
                                         MAT<array identifier>=TRN(<array identifier>)|
                                         MAT<array identifier>=(<expression>)*<array
                                                                     identifier>

<mat operator>                     ::=   +|-|*

<ASSIGN statement>                 ::=   ASSIGN<source string>,<expression>,<variable>|
                                         ASSIGN<source string>,<expression>,<variable>,
                                                                <source string>
```

STRING EVALUATION BY ASCII CODES

Each user terminal character is represented by an A.S.C.I.I. (American Standard Code for Information Interchange) number except 2741.

Strings are compared by their A.S.C.I.I. representation.

The A.S.C.I.I. sequence, from lowest to highest is:

Lowest: <u>*bell*</u>

space	5	J		u
!	6	K	a	v
"	7	L	b	w
#	8	M	c	x
$	9	N	d	y
%	:	O	e	z
&	;	P	f	{
'	<	Q	g	¦
(	=	R	h	}
)	>	S	i	~ *Highest*
*	?	T	j	
+	@	U	k	
,	A	V	l	
-	B	W	m	
.	C	X	n	
/	D	Y	o	
Ø	E	Z	p	
1	F	[	q	
2	G	\	r	
3	H	]	s	
4	I	↑	t	

MEMORY ALLOCATION BY A USER

Approximate space available for user allocation: 10,000 2-character words.

Examples of User-Determined Allocation

> *Note:* *This is variable "system overhead"; it is not included in word counts produced by the LEN command.*

a. Introduction of each simple, string, or matrix variable uses 4 words.

b. A 9 word stack is reserved for GOSUB's.

c. 6 X (maximum level of FOR...NEXT loop nesting).

d. Each file name mentioned in a FILES statement reserves as many words for buffer space as there are words in each logical record of the file. Each "*" in a FILES statement reserves 256 words of BUFFER space; each file and "*" also reserves 15 words of table space.

e. An approximate estimate of space required for a program is:

> 11 words per BASIC statement
>
> +2X(number of matrix elements dimensioned)
>
> +1/2X(number of string characters used)

Semicompiled programs require slightly more space than that shown by the LEN command. CATALOG gives the actual length of CSAVED programs.

APPENDIX A
How to Prepare A Paper Tape Off-Line

To prepare a BASIC program on paper tape for input:

1. Set terminal status to "LOCAL."

2. Press the ON button on the paper tape punch.

3. Press @^{C} (or HERE IS if available) several times to put leading feed holes on the tape.

4. Type the program as usual, following each line with *return linefeed.*

5. Press @^{C} (or HERE IS) several times to put trailing feed holes on the tape.

6. Press the OFF button on the paper tape punch.

The standard on-line editing features, such as line delete and character delete, may be punched on paper tape.

Pressing the BACKSPACE button on the paper tape punch, then the RUBOUT or DEL key on the keyboard, physically deletes the previous character from the paper tape.

Programs punched onto paper tape in the above manner, or produced by the PUNCH command, may be input to the system through the paper tape reader after typing the TAPE command. They may not be input as data using INPUT or ENTER statements. (See Section II and Appendix B.)

APPENDIX B
The X-ON, X-OFF Feature

Terminals equipped with the X-ON, X-OFF feature must be used if it is desired to input data from a paper tape while a program is running.

Data is punched on paper tape in this format:

> *line of data items separated by commas x-off return linefeed*
>
> *(x-off, return* and *linefeed* are user terminal keys.)

Remember that each line of data must end with *x-off return linefeed.*

The X-OFF character causes the paper tape reader to stop reading tape after each carriage return until more input is requested by the program. Lines output by PRINT and PRINT USING statements are terminated by the X-off character

Programs on paper tape produced by the XPUNCH command are in the correct format to be input as data strings from terminals with the X-ON, X-OFF feature. No line of such a program should exceed 72 characters (not counting X-OFF, carriage return, and linefeed), since each must fit into a single string. Programs produced by XPUNCH are not suitable for input in TAPE mode (Appendix A) from terminals with the X-ON, X-OFF feature.

APPENDIX C
Diagnostic Messages

USER COMMAND ERROR MESSAGES

Error messages are listed below with the command which may invoke them. The message ILLEGAL FORMAT may be typed in response to many commands. The message PLEASE LOG IN is typed if a command (other than HELLO) or a line of syntax is entered from a port on which no user is logged in.

APPEND

       INVALID NAME
       NO SUCH PROGRAM
       ILL-STORED PROGRAM
       ENTRY IS A FILE
       SEMI-COMPILED PROGRAM
       NO COMMON AREA ALLOWED
       PROGRAM TOO LARGE
       UNABLE TO RETRIEVE FROM LIBRARY
       SEQUENCE NUMBER OVERLAP

CATALOG

       CAN'T READ DIRECTORY

CSAVE

       See SAVE.

DELETE

       NOTHING DELETED

GET

      INVALID NAME
      NO SUCH PROGRAM
      ILL-STORED PROGRAM
      ENTRY IS A FILE
      PROGRAM TOO LARGE
      UNABLE TO RETRIEVE FROM LIBRARY

GROUP

      See CATALOG.

HELLO

      ILLEGAL ACCESS
      NO TIME LEFT

KILL

      ILLEGAL NAME
      NO SUCH ENTRY
      FILE IN USE

LIBRARY

      See CATALOG.

LIST

      RUN ONLY

LPRINTER

      LP BUSY
      LP DOWN
      LP FREE
      LP NOT AVAILABLE

MESSAGE

      CONSOLE BUSY

NAME

      ONLY 6 CHARACTERS ACCEPTED
      ILLEGAL FIRST CHARACTER

OPEN

>NAME TOO LONG
>ILLEGAL FIRST CHARACTER
>LIBRARY SPACE FULL
>SYSTEM OVERLOAD
>DUPLICATE ENTRY
>UNSUCCESSFUL; KILL AND REPEAT.

PROTECT

>PRIVILEGED COMMAND
>INVALID NAME
>NO SUCH ENTRY

PUNCH

>See LIST.

RENUMBER

>SEQUENCE NUMBER OVERFLOW/OVERLAP
>BAD PARAMETER

RUN

>See Execution Errors.

SAVE

>RUN ONLY
>NO PROGRAM NAME
>NO PROGRAM
>OUT OF STORAGE IN LINE n
>LIBRARY SPACE FULL
>SYSTEM OVERLOAD
>DUPLICATE ENTRY
>UNSUCCESSFUL; KILL AND REPEAT.

TAPE

>If entered from an IBM 2741 Selectric:
>ILLEGAL

UNPROTECT

>See PROTECT.

XPUNCH

>See LIST.

LANGUAGE PROCESSOR ERROR MESSAGES

The following messages are output by the BASIC language processor to indicate errors or possible errors in users' BASIC programs.

Syntax Errors

One of the following error messages will be typed by the system after the entry of a BASIC statement with incorrect syntax. In all cases but the last, the line will be deleted.

```
OUT OF STORAGE
ILLEGAL OR MISSING INTEGER
EXTRANEOUS LIST DELIMITER
MISSING ASSIGNMENT OPERATOR
CHARACTERS AFTER STATEMENT END
MISSING OR ILLEGAL SUBSCRIPT
MISSING OR BAD LIST DELIMITER
MISSING OR BAD FUNCTION NAME
MISSING OR BAD SIMPLE VARIABLE
MISSING OR ILLEGAL 'OF'
MISSING OR ILLEGAL 'THEN'
MISSING OR ILLEGAL 'TO'
MISSING OR ILLEGAL 'STEP'
MISSING OR ILLEGAL DATA ITEM
ILLEGAL EXPONENT
SIGN WITHOUT NUMBER
MISSING RELATIONAL OPERATOR
ILLEGAL READ VARIABLE
ILLEGAL SYMBOL FOLLOWS 'MAT'
MATRIX CANNOT BE ON BOTH SIDES
NO '*' AFTER RIGHT PARENTHESIS
NO LEGAL BINARY OPERATOR FOUND
MISSING LEFT PARENTHESIS
MISSING RIGHT PARENTHESIS
PARAMETER NOT STRING VARIABLE
UNDECIPHERABLE OPERAND
MISSING OR BAD ARRAY VARIABLE
STRING VARIABLE NOT LEGAL HERE
MISSING OR BAD STRING OPERAND
NO CLOSING QUOTE
72 CHARACTERS MAX FOR STRING
STATEMENT HAS EXCESSIVE LENGTH
MISSING OR BAD FILE REFERENCE
'PRINT' MUST PRECEDE 'USING'
ILLEGAL OPERAND AFTER 'USING'
VARIABLE MISSING OR WRONG TYPE
OVER/UNDERFLOWS—WARNING ONLY
```

Execution Errors

This section lists messages output to indicate errors detected during program execution. Such errors cause termination of the execution.

        UNDEFINED STATEMENT REFERENCE
        NEXT WITHOUT MATCHING FOR
        SAME FOR-VARIABLE NESTED
        FUNCTION DEFINED TWICE
        VARIABLE DIMENSIONED TWICE
        LAST STATEMENT NOT 'END'
        UNMATCHED FOR
        UNDEFINED FUNCTION
        ARRAY TOO LARGE
        ARRAY OF UNKNOWN DIMENSIONS
        OUT OF STORAGE
        DIMENSIONS NOT COMPATIBLE
        CHARACTERS AFTER COMMAND END
        BAD FORMAT OR ILLEGAL NAME
        MISSING OR PROTECTED FILE
        GOSUBS NESTED TEN DEEP
        RETURN WITH NO PRIOR GOSUB
        SUBSCRIPT OUT OF BOUNDS
        NEGATIVE STRING LENGTH
        NON-CONTIGUOUS STRING CREATED
        STRING OVERFLOW
        OUT OF DATA
        DATA OF WRONG TYPE
        UNDEFINED VALUE ACCESSED
        MATRIX NOT SQUARE
        REDIMENSIONED ARRAY TOO LARGE
        NEARLY SINGULAR MATRIX
        LOG OF NEGATIVE ARGUMENT
        SQR OF NEGATIVE ARGUMENT
        ZERO TO ZERO POWER
        NEGATIVE NUMBER TO REAL POWER
        ARGUMENT OF SIN OR TAN TOO BIG
        TOO MANY FILES STATEMENTS
        NON-EXISTENT FILE REQUESTED
        WRITE TRIED ON READ-ONLY FILE
        END-OF-FILE/END OF RECORD
        STATEMENT NOT IMAGE
        NON-EXISTENT PROGRAM REQUESTED
        CHAIN REQUEST IS A FILE
        PROGRAM CHAINED IS TOO LARGE
        COM STATEMENT OUT OF ORDER
        ARGUMENT OF TIM OUT OF RANGE
        BAD FORMAT STRING SUBSCRIPT
        BAD FILE READ
        BAD FILE WRITE DETECTED

CAN'T READ PROGRAM CHAINED TO
ILL-STORED PROGRAM CHAINED TO
PROGRAM BAD
MISSING FORMAT SPECIFICATION
ILLEGAL OR MISSING DELIMITER
NO CLOSING QUOTE
BAD CHARACTER AFTER REPLICATOR
REPLICATOR TOO LARGE
REPLICATOR ZERO
MULTIPLE DECIMAL POINTS
BAD FLOATING SPECIFICATION
ILLEGAL CHARACTER IN FORMAT
ILLEGAL FORMAT FOR STRING
MISSING RIGHT PARENTHESIS
MISSING REPLICATOR
TOO MANY PARENTHESIS LEVELS
MISSING LEFT PARENTHESIS
ILLEGAL FORMAT FOR NUMBER

Execution Warnings

The following messages are printed by the system to inform the user of conditions which may be unexpected or undesirable. These conditions do not terminate execution.

BAD INPUT, RETYPE FROM ITEM *XXXX*
LOG OF ZERO—WARNING ONLY
ZERO TO NEGATIVE POWER–WARNING
DIVIDE BY ZERO—WARNING ONLY
EXP OVERFLOW—WARNING ONLY
OVERFLOW—WARNING ONLY
UNDERFLOW—WARNING ONLY
EXTRA INPUT—WARNING ONLY
READ-ONLY FILES:

APPENDIX D
Additional Library Features

Normally, programs and files in a user's library are stored on a mass storage device called a disc, which is external to the computer. Only the current program and portions of currently accessed files occupy the user's "working space" in the computer. TSB also makes use of another, usually smaller, mass storage device called a drum, on which many system tables are stored. There may also be room on the drum for a limited number of user programs and files. In certain cases, programs and particularly files which reside on the drum have improved (shorter) access times over those on the disc.

The system operator has control over placement and removal of programs and files on the drum. He also has several other program and file movement capabilities of which the user should be aware. These operator commands, and their functions, are listed here.

SANCTIFY This command enables the operator to move a program (no longer than 8192 words) or a file (no longer than 32 records) from the disc to the drum. The area on the disc where it resided is retained. The entry will remain on the drum until it is removed by the operator (see below) or killed by the user who owns it. Only entries whose access times are critical should be santified.

Note: The system operator cannot SANCTIFY programs or files if the TSB system is the base system (20854A) or option 001 (20854A-001).

DESECRATE This command moves a sanctified file from the drum back to its original location on the disc, or deletes the drum copy of a sanctified program. (The disc copy of the program is retained.)

Notes: 1)   If a sanctified program cannot be retrieved from a user's library because of a data error on the drum, it may be possible to DESECRATE the program and retrieve the copy from the disc.

2)   The system operator is not able to DESECRATE programs or files if the TSB system is the base system (20854A) or option 001 (20854A-001).

BESTOW   This command enables the operator to remove a program or file from one user's library and place it in another user's library, or to transfer ownership of an entire library.

COPY   This command is used to make a duplicate copy of any user program or file in the library of any other user (or the same user). The copy may be given a new name.

LOAD
DUMP   The LOAD command enables the operator to load selected programs and files or entire user libraries from magnetic tape. DUMP allows the operator to write such programs, files or libraries onto magnetic tape. This can be done only at system start-up time (commonly once a day) and is a convenient way of transferring entries between 2000 systems, or dumping TSB files for other utility purposes.

Note: *Except as noted, any of the above may be requested using the MESSAGE command. All pertinent idcodes and program or file names must be included.*

APPENDIX E
User Terminal Interface

User terminals can be operated in either of two modes, on-line or off-line. In on-line mode, connection to the computer is established, a log on procedure is performed, and the user is in contact with the computer through the TSB system. This system accepts and executes any legal command entered by the user. Illegal commands are rejected, usually with an informative message printed or displayed on the terminal.

To enter a command, type either the short or full form of the command; if additional parameters are required or permitted, type a hyphen, then the parameters. Terminate the command by pressing *return*. Some commands cause an obvious response from the system such as a listing or punching operation. Other commands result in computer operations; the only response is the generation of a *linefeed*, indicating that the system has accepted the command and is ready for another entry.

Terminals with paper tape punching capabilities may be used to prepare paper tape in off-line mode. Off-line operation of these terminals is described in Appendix A.

Several types of user terminals can be connected to the TSB system. Most generate ASCII code and one generates CALL 360 or PTTC/EBDC (non-ASCII) code.

The following user terminals generate ASCII code:

- HP 2762A/B Terminal Printer

- HP 2600A Keyboard-Display Terminal

- HP 2640A Interactive Display Terminal

- HP 2749A/B Teleprinter Terminal

- General Electric TermiNet 300 Data Communications Terminal, Model B(10/15/30 cps transfer rates) with Paper Tape Reader/Punch, Option 2

    Note: The terminal must be strapped for "ECHO-PLEX".

- GE TermiNet 1200 Communications Terminal

- GE TermiNet 30 Matrix Impact Printer

- Memorex 1240 Communications Terminal (10/15/30 cps transfer rates)

    Note: The terminal must be equipped with the even parity
    checking option.

DEC 1975

- Texas Instruments Silent 700

- Execuport 300 Data Communications Transceiver Terminal

- ASR-37 Teleprinter Terminal with Paper Tape Reader/Punch

> *Note:* *If the terminal is equipped with the Shift Out (SO) feature,*
> *SO must be disabled because the TSB system does not*
> *allow use of this feature.*

The following user terminal generates non-ASCII code:

- IBM 2741 Communication Terminal

> *Note:* *The terminal must be connected to the system over telephone*
> *lines. In addition, the terminal must be equipped with the*
> *following features:*
>
> 1. *Interrupt, Receive (IBM #4708) and Transmit (IBM #7900)*
> *associated with the terminal's ATTN key.*
>
> 2. *Dial-Up (IBM #3255) to enable system connection through*
> *a 103A modem or acoustic coupler.*

Any terminal equipped with the automatic *linefeed* feature (operator selectable) must be operated with this feature OFF.

> *Note:* *Although cursor, form feed, horizontal and vertical tabulation,*
> *and various special function keys are provided on specific types*
> *of user terminals, these capabilities are not supported by the*
> *High Speed option. Some of these operations may be requested*
> *from the keyboard, but results are unpredictable. Features*
> *provided by the TSB system, such as the TAB, SPA, and*
> *LIN functions, and the PRINT and PRINT USING statements,*
> *should be used to control output format. However, terminals*
> *equipped with automatic linefeed after carriage return or on*
> *end of line may cause unpredictable results. These functions and*
> *statements are described in other sections of this manual.*

IBM 2741 COMMUNICATION TERMINAL INTERFACE

Because the IBM 2741 terminal generates non-ASCII code, special consideration must be given to the representation of several ASCII characters and functions which are not available in the 2741 character set.

For input from a 2741 terminal, these characters (and some of the functions) are simulated by entry of a two-character code. The first character of this code is the cent symbol (¢). The cent symbol is followed by one of several alphanumeric or special characters to compose a unique code representing one ASCII character or function.

On input from a 2741 terminal, the two-character code is translated into the internal ASCII code. On output to a 2741 terminal, ASCII code is translated into the appropriate two-character representation.

The TAPE command is not allowed from ports configured for 2741 terminals. If entered, the system responds with the message *ILLEGAL*.

The IBM 2741 Communications Terminal must be equipped with the interrupt feature associated with the *ATTN* key. This key represents the *break* function; it is used to terminate program or command execution. The underline character (_) is equivalent to a back arrow (←) and represents the delete character on the IBM 2741 Communications Terminal.

Any CALL/360 or PTTC/EBCD characters that do not have an equivalent ASCII character are ignored on input.

Table 1-1 shows 2741 terminal representation of ASCII characters and functions.

Table 1-1. IBM 2741 ASCII Character Simulation

ASCII Graphic Control Character	IBM 2741 Character Representation		User Terminal Function	IBM 2741 Character Representation	
	CALL/360	PTTC/EBCD		CALL/360	PTTC/EBCD
[	¢(	¢(	*control*②	¢C	¢C
\	¢/	¢/	*break*	ATTN key	ATTN key
]	¢)	¢)			
^	↑	¢A			
`	¢'	¢'			
{	¢0	¢0			
}	¢S	¢S			
~	¢T	¢T			
__	__ ①	__ ①			
ESC	¢E	' ¢E			
FS	¢F	¢F			
GS	¢G	¢G			
RS	¢R	¢R			
US	¢U	¢U			

① Underline character, used as delete character (←).

② Code must be followed by an appropriate alphabetic character; otherwise, it is ignored.

E-3

EXAMPLES:

Action	*Code Required*
System input request termination (control C)	*¢CC*
Input line deletion (control X)	*¢CX (See NOTE)*
Character deletion (backspace)	*_ (underline)*

Note: This entry must be followed by return. Otherwise, it is ignored.

Index

A

ABS function, 3-18
Acoustic Coupler, 1-3
Addition symbol, 2-4
Additional library features, D-1
AND operator, 2-5
APPEND, 3-8
Array, defined, 3-2
Arithmetic Evaluation, 2-3
Arithmetic Operators, 2-4
ASCII code, string evaluation, 9-10
ASSIGN, 4-6
Assignment operator, 2-10, 6-6
Assignment statement, 2-10
ATN function, 3-19

B

Backus Naur Form, BASIC language, 9-3
BASIC command, 1-7
BASIC language
   Backus Naur Form, 9-3
   defined, 1-7
   syntax, 9-1
BASIC programs, 1-9
BASIC Statements, 1-7
Bestowing files, D-2
Boolean operators, 7-2
Branching
   to statements, 2-11
   to subroutines, 3-14
break key, 2-37
BRK function, 3-24
BYE, 2-27

C

C^c , 1-2
Carriage control, 8-14
Carriage control characters, defined, 8-2
Carriage spacing, output, 2-20
CATALOG, 3-10
CHAIN, 3-20

Changing a statement, 1-9
Changing file references during execution, 4-6
Character deletion, 1-2
Character spacing, 8-2
Clearing the user work area, 2-29
COM, 3-22
Command, definition, 2-25
Command error messages, C-1
Commands, BASIC, 1-7
Communicating with system operator, 2-36
Conditional branching, 2-12, 4-12, 6-10
Connection to computer
   via telephone, 1-3
   direct, 1-4
Control characters, 1-2
control key, 1-2
Copying a file, 4-21, D-2
COS function, 3-19
Creating files, 4-3
CSAVE, 3-5

D

DATA, 2-15, 6-12
DATA, strings, 6-12
Data input, matrix, 5-6
Data set, 1-4
DEF FN, 3-16
Defining functions, 3-16
DELETE, 3-9
Deleting
   files, 4-4
   programs, 1-12, 3-4, 3-7
   statements, 1-9, 3-9
Desecrating files (Options 210/215 only), D-1
Determining file length, 4-18
Diagnostic messages, C-1
DIM, 5-2, 6-5
Dimensioning
   matrix, 5-2
   strings, 6-5
Division symbol, 2-4
Documenting a program, 1-13
DUMP, selective, D-2

E

E notation, defined, 2-2
ECHO, 2-27
END, 2-22
End-of-file, defined, 4-12
ENTER, 3-23
Erasing
    files, 4-26
    records, 4-25
Error messages, 1-8, C-1
Equality symbol, 2-4
Execution
    error messages, C-5
    warning messages, C-6
EXP function, 3-18
Exponentiation symbol, 2-4
Expression, defined, 2-3
Expression list, defined, 8-1, 8-10

F

File
    accessing errors, 4-28
    BESTOW, D-2
    COPY, D-2
    copying, 4-21
    defined, 4-1
    DESECRATE (Option 210/215 only), D-1
    erasing, 4-26
    length determination, 4-18
    matrix printing, 5-16
    matrix reading, 5-17
    pointer, 4-13
    pointer manipulation, 4-18
    SANCTIFY (Option 210/215 only), D-1
    selective LOAD/DUMP, D-2
    storage requirements, 4-17
FILES, 4-5
Fixed-point format specifications, 8-6
Floating-point format specifications, 8-7
FOR . . . NEXT, 2-13
FOR . . . NEXT with STEP, 3-16
Format characters
    defined, 8-2
    fixed-point, 8-6
    floating-point, 8-7
    integer, 8-5
    string, 8-4
Format specification
    defined, 8-1
    grouped, 8-9
Format string, defined, 8-1, 8-10
Formatted output, 8-1
Function, defined, 3-3, 3-13

G

General mathematical functions, 3-18
GET, 3-6

GO TO, 2-11
GOSUB. . .RETURN, 3-13
Greater than symbol, 2-4
Greater than or equal to symbol, 2-4
GROUP, 3-10
Group library, 3-10
Grouping format specifications, 8-9

H

Half-duplex coupler, 1-4
Hardwired connection, 1-4
HELLO, 2-26

I

Identification code, user, 1-5
Identity matrix, 5-13
IF. . .THEN, 2-12, 6-10
IF END#. . .THEN, 4-12
IMAGE, 8-13
Inequality symbol, 2-4
INPUT, 2-17, 5-5, 6-7
INPUT, matrix, 5-5
Input, program data, 2-17
INT function, 3-18
Integer format specifications, 8-5
Interface, user terminal, E-1

K

KEY, 2-33
Keyboard mode, 2-33
KILL, 3-7, 4-4

L

Language processor error messages, C-4
LEN function, 3-19, 6-11
LENGTH, 3-4
Length, string, 6-11
Less than symbol, 2-4
Less than or equal to symbol, 2-4
LET, 2-10, 6-6
LIBRARY, 3-10
LIN function, 8-2
Line deletion, 1-2
Line printer
    access, 2-33
    carriage control, 2-34
    control characters, 2-34
    messages, 2-35
Line spacing, 2-20, 8-2
linefeed, 1-2
LIST, 2-28
Listing
    file contents, 4-11
    programs, 1-10, 2-28
    record contents, 4-20
Literal string, defined, 8-2