TENET USERS MANUAL

BASIC

TENET BASIC STATEMENTS

Assignment & Sequence Control

INTEGER v_i	4-2
REAL v_i	4-2
DOUBLE v_i	4-3
COMPLEX v_i	4-3
DOUBLE COMPLEX v_i	4-3
STRING v_i	4-4
DIM a_i	4-5
[LET] sv_i = e	4-7
DO ln[:ln] [,ln[:ln]]...[,ln[:ln]]	4-10
GOTO ln	4-12
ON e GOTO ln_i	4-13
IF e THEN $\left\{\begin{matrix} s_1 \\ ln_1 \end{matrix}\right\}$ [ELSE $\left\{\begin{matrix} s_2 \\ ln_2 \end{matrix}\right\}$]	4-14
FOR sv = $e_1 \left\{\begin{matrix} \text{TO} \\ \text{WHILE} \end{matrix}\right\} e_2$ [STEP e_3]	4-15
NEXT v_i	4-15
DATA c_i	4-19
READ sv_i	4-20
RESTORE [ln]	4-21
REM text	4-22
PAUSE	4-23
END	4-24

Function & Subroutine

DEF [dt] FNname [(v_i)][=e]	5-2,4
RETURN [e]	5-4
GOSUB ln	5-6
ON e GOSUB ln_i	5-7

Terminal Input/Output

PRINT sv_i	6-2
INPUT sv_i	6-7
PRINT IN FORM x\$:sv_i	6-9

Matrix

MAT $a_1 = a_2$	7-2
MAT $a_1 = a_2 + a_3$	7-3
MAT $a_1 = a_2 - a_3$	7-4
MAT $a_1 = a_2 * a_3$	7-5
MAT a_1 = (e) * a_2	7-6
MAT a_1 = INV(a_2)	7-7
MAT a_1 = TRN(a_2)	7-8
MAT a = e	7-9
MAT a = IDN	7-10
MAT READ a_i	7-11
MAT INPUT a_i	7-12
MAT PRINT [IN FORM x\$]:a_i	7-14

File

OPEN $\left\{\begin{matrix} \text{"fn"} \\ \text{"SCR"} \end{matrix}\right\}$ $,e_1$ $\left\{\begin{matrix} \text{,SYMBOLIC} \\ \text{,BINARY} \end{matrix}\right\}$ $\left\{\begin{matrix} \text{,OUTPUT} \\ \text{,INPUT} \\ \text{,IO} \end{matrix}\right\}$ $\left\{\begin{matrix} \text{,SEQUENTIAL} \\ \text{,RANDOM}(e_2) \\ \text{,RANDOM} \end{matrix}\right\}$ $\left\{\begin{matrix} \text{,NEW} \\ \text{,OLD} \end{matrix}\right\}$	8-4
CLOSE e_i	8-6
RESTORE FILE e_i	8-7
APPEND FILE e_i	8-8
ERASE FILE e_1 FROM e_2 TO e_3	8-9
ON ENDFILE (e) GOTO ln	8-10
ON ENDREC (e) GOTO ln	8-11
[MAT] INPUT FROM e_1[AT e_2][IN FORM x\$]:v_i	8-12,4
[MAT] PRINT ON e_1[AT e_2][IN FORM x\$]:v_i	8-15;6-9

Edit

ENTER ln [STEP n]	9-3
LIST [ln[:ln][,...ln[:ln]]]	9-4
RENUMBER [ln_1,ln_2,n]	9-5
DEL[ETE] $\left\{\begin{matrix} \text{ALL} \\ \text{ln[:ln][,...[:ln]]} \end{matrix}\right\}$	9-6
ALTER ln	9-7
TABS [p_1,p_2,p_3,p_4]	9-9

Program Control

RUN	10-3
CONTINUE	10-3
SAVE "fn" $\left\{\begin{matrix} \text{,SYMBOLIC} \\ \text{,BINARY} \end{matrix}\right\}$ $\left\{\begin{matrix} \text{,NEW} \\ \text{,OLD} \end{matrix}\right\}$ $,ln_i$	10-4
LOAD "fn"	10-5
LINK "fn" [,ln]	10-6
TAPE	10-7
QUIT	10-8

TENET BASIC
Users Manual

JULY 1970

TENET
927 Thompson Place
Sunnyvale, California 94086

Pub. No. 2001

Price: $4.50

PREFACE

This document is a user's guide which explains in detail the features, vocabulary, and usage of TENET BASIC. It presents the timesharing user with the information necessary for him to fully utilize the capabilities of the TENET 210 Timesharing System. Although it is possible for the inexperienced user to learn the BASIC language from this text, it is recommended that he first refer to the TENET 210 TIMESHARING PRIMER for an introductory discussion of the BASIC language.

FIRST EDITION:

Specifications contained herein are subject to changes which will be reported in subsequent revisions. Copies of this and other TENET publications can be obtained through TENET branch offices. A form is provided at the back of this publication for readers' comments. If the form has been detached, please direct your comments to TENET, 927 Thompson Place, Sunnyvale, California 94086 (Attention: Software Publications Dept.).

CONTENTS

APPENDIXES

1. INTRODUCTION

TENET BASIC

TENET BASIC is a subsystem of the TENET 210 Timesharing System. It was designed and implemented to enable a wide variety of users to communicate with a powerful timesharing computer. The TENET BASIC system is a greatly extended version of the original Dartmouth BASIC. From the user's viewpoint, the BASIC language itself has not changed, but its flexibility and power have increased. TENET BASIC is not only a language to describe a program, but a system which has an interactive capability necessary for dynamic problem solution.

The TENET 210 Timesharing System was developed specifically for interactive timesharing operations. It consists of the TENET 210 computer system connected to a set of interactive terminals. The user views this system in two levels. The first is the small resident command processor, EXECUTIVE. A user's first and last contact with the TENET 210 System is at the EXECUTIVE level. EXECUTIVE provides the initiation and termination procedures for sessions at a terminal, performs accounting tasks, file management, and enables the user to access the second level containing the TENET BASIC subsystem. Only the EXECUTIVE commands necessary to the TENET BASIC user are discussed. For more information about EXECUTIVE and other subsystems available to the user, refer to the TENET EXECUTIVE USER'S MANUAL.

CONVENTIONS USED IN THIS MANUAL

The following conventions are used throughout this manual:

- Upper case letters, digits, and special characters must appear exactly as shown in the format representation for all statements.

- Information in lower case letters in the format representations is to be supplied by the user.

- Braces { } indicate that one of the items enclosed must be used.

- Brackets [ ] indicate that the item (or items) is optional.

- An ellipsis (a series of three periods, ...) indicates that a variable number of items may be included in the list. Variable-length lists are indicated by the subscript $_i$.

- Control characters are followed by the superscript ^{c}.

- The example preceding the discussion of each statement is a syntactic example only.

- Examples following the discussion of each statement are for the purpose of demonstrating the usage of the statements and are not necessarily examples of good programming practice.

- The word " type " indicates user input from the teletypewriter; " print " indicates computer output.

- Underlined text in the examples indicates computer output or computer requests.

- Although a " prompt " character precedes each BASIC statement during an actual programming session, it is omitted throughout this text.

2. USING TENET BASIC

Before the user can access the TENET BASIC subsystem, he must first establish a connection with the TENET 210 computer. The computer receives and transmits information through an interactive terminal such as the Model 33 Teletypewriter Terminal. For this type of terminal the teletypewriter LINE/OFF/LOCAL knob must be turned to the LINE position to establish a communication link. The connection is then confirmed by the system when it prints out a message such as:

The Model 33 Teletypewriter Terminal is discussed in Appendix F.

```
    TENET TIME SHARING  6/30/70
```

The header message may vary according to individual installations.

MODE OF OPERATION

The TENET 210 system receives information from the terminal in full duplex mode only. Characters typed into the terminal are transmitted to the computer without simultaneous printing. Once the computer receives a character, it transmits a character back to the terminal for printing. This is a full duplex operation whereby the listing documents exactly what the computer receives, thus minimizing undetected transmission errors. Therefore, a character printed at the terminal is an echo of what the computer received. Certain control characters are not echoed. As a security precaution, words denoting special permissions are never echoed.

INPUT CONVENTIONS

The following keys are used to transmit and modify information entered from the terminal:

(CR) Carriage Return Key. This key designates an end of statement. Each time this key is pressed the system echoes a Carriage Return and Line Feed, thereby positioning the teletypewriter print head at the beginning of the next line.

LF — Line Feed Key. This key designates a continuation of statement. Each time this key is pressed the system echoes a Carriage Return and Line Feed, thereby positioning the teletypewriter print head at the beginning of the next line. Since the Carriage Return echoed by the system does not act as an end of record, another Carriage Return must be issued by the user to terminate multi-line statements.

A^C — A Control Key. This key deletes the previous character entered by the user. It causes a backspace arrow to be printed (echoed) and may be used repeatedly, deleting a character each time the key is pressed. For example:

> TENT A^C ET 222 A^C A^C 10 will be printed at the terminal as:
>
> TENT ← ET 222 ←← 10
>
> The system will interpret it as:
>
> TENET 210

Q^C — Q Control Key. This key completely deletes the line currently being typed (i. e., before the Carriage Return key is pressed). A Line Feed and Carriage Return are echoed when this key is pressed. Example:

> " WHAT IS IT " Q^C will be printed at the terminal as:
>
> " WHAT IS IT " ↑

Break — The Break Key causes a transmission interrupt and effectively disconnects the terminal from the computer. Pressing this key could cause a loss of the program and its data.

ESC or ALT MODE — Escape Key or ALT MODE. This key has the same effect as the Q^C when correcting program statements. However, as this key has another significance in other situations (for example, during the LOGIN sequence), it should not be used in place of a Q^C key.

BELL — The bell sounds when the user has typed at character position 61 to warn that he is approaching the 72-character line capacity limit.

PROMPT CHARACTERS

Before any information can be entered from the terminal, it must be preceded by a signal, or prompt, from the system. This prompt signifies that the system is ready to accept input. For example, after the user logs into the system, the prompt character issued is " - ", indicating that he is at the EXECUTIVE level and that only EXECUTIVE level commands may be entered. Once the user has entered the TENET BASIC subsystem, the prompt character " > " is printed by the system before the user enters each program statement. Requests for additional or corrected information are signalled by a " ? ".

ENTERING THE TENET 210 SYSTEM

Once the terminal is connected to the computer and the header message issued, the system is automatically placed in the LOGIN mode, and the user is requested to identify himself. In the following LOGIN sequences, the underlined text indicates what the system prints.

```
    -LOGIN account; name (CR)
```

where:

```
    account = user's account number, a numeric value from 0 through 511.
    name    = user's name, from 1 through 8 characters.
```

If either or both of the items requested by the LOGIN prompt are omitted or invalid, the user is reprompted by ACCOUNT ? and/or NAME ?. The user must respond to these prompts with the appropriate entries followed by a Carriage Return. Examples:

```
    -LOGIN 210 (CR)
    NAME ? SMITH (CR)
```

or

```
    -LOGIN SMITH (CR)
    ACCOUNT ? 210 (CR)
```

The user must successfully sign in within three minutes of connect time, or else the terminal is automatically disconnected.

Passwords

If the user has designated a password to be associated with an account number, the LOGIN sequence will include a request for a password. The password prompt (PASSWORD ?) appears after the user has entered his account and name. As a security precaution, password entries are not echoed at the teletypewriter.

LOGIN Messages

The following messages and diagnostics may appear during the LOGIN sequence:

ERROR
The account number, name,or password has been entered incorrectly. The appropriate item will be requested.

NAME IN USE
Another user is using the same account number and name specified in the LOGIN attempt.

USER LIMIT EXCEEDED
The user has exceeded the maximum CPU time, disc space, or terminal time allocated to him. The user is disconnected.

THE TENET BASIC SUBSYSTEM

Once the user is at the EXECUTIVE level, he can enter the TENET BASIC subsystem by the EXECUTIVE command " BASIC ". TENET BASIC then issues its own prompt character, >.

```
-BASIC(CR)
>
```

Command Modes

The computer accepts commands in two modes: Immediate Execution and Program Execution. Immediate Execution commands are executed upon receipt. They are not actually a part of the program but control its disposition. Program Execution commands constitute programs, and thus are not executed immediately, but are deferred until program execution time (run time).

Line Numbers

All Program Execution commands begin with line numbers that indicate their position within a program. Thus, program statements need not be entered in numeric sequence as they are ultimately sorted by the computer. If duplicate line numbers are used, the later version of the statement will replace the earlier.

Line numbers must be positive integer values ranging from 1 to 99999 with no embedded blanks.

Syntax Errors

Program statements entered from the keyboard (as opposed to paper tape input) are checked for syntactic errors. If an error is found, the system prints out an appropriate message. The line in error is not deleted by the system; it must be reentered correctly or the same error message will be issued when program execution is attempted. Thus, even though incorrect statements may be entered into a program, a program with any errors (including syntax) will not be executed.

Syntax checking after each statement is optional and controlled at individual terminals. However, programs are always checked for all errors at execution time and the appropriate error messages are printed at the terminal.

Line Length

Teletype input must consist of no more than 72 characters per line, and a maximum of 256 characters per statement when the Line Feed is used.

Character Set

BASIC programs may be written using the following character set:

- Alphabet: 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
- Digits: 1 2 3 4 5 6 7 8 9 0
- Special characters:

'	single quote
"	double quote
<	less than
=	equal to
>	greater than
≠	not equal to
+	plus
-	minus
*	asterisk
/	right oblique
↑	up arrow
(	left parenthesis
)	right parenthesis
!	exclamation mark
,	comma
.	period
;	semicolon
:	colon
	blank
@	at
$	dollar
%	per cent

Any valid teletypewriter terminal character not listed above is not a BASIC character, but may be used where specifically noted.

Blanks

Since TENET BASIC allows variable names of up to four characters, it is necessary to separate reserved words, constants, and variable names with one or more blanks. A blank is necessary after any item which could have the first character of the next item as a potentially valid character. Blanks between other elements of the language are optional.

Blank spaces may not be embedded in any variable name, number, or operator.

Statement Types

The BASIC subsystem accepts three types of statements or commands: program statements, edit commands, and control commands. All three types are normally used during a session at the terminal.

- Program statements describe (to the TENET BASIC system) operations to be performed on program data. A program statement preceded by a line number is used and executed as part of a program. A program statement <u>not</u> preceded by a line number is executed immediately, whereby it is used for instant calculations or program debugging.

- EDIT commands modify and/or correct program statements. The program statements themselves are the data upon which EDIT statements operate. These commands are always executed immediately.

- Control commands specify actions which alter the status of the user and/or his program. For example, they direct the execution, saving, and retrieval of programs. These commands are always executed immediately.

THE INTERACTIVE ENVIRONMENT

TENET BASIC allows the user to execute a program and based on the results examine and or alter data, execute selected portions of the program, and create independent variables to be used on a temporary basis.

Immediate Execution program commands may create variables that exist at the Immediate Execution level only and are not saved in the original program. An Immediate Execution data type declaration statement (REAL, INTEGER, etc.) creates an explicitly defined variable existing at the

Immediate Execution level; if a variable of the same name exists at the Program level, it will not be affected or referenced by the Immediate Execution level statement.

A variable explicitly defined at the Immediate Execution level is an Immediate variable. If it is not explicitly defined at the Immediate Execution level but a variable of the same name is defined (explicitly or implicitly) at the Program level, the Program level definition is used for the variable. Finally, if no variable of the same name exists at the Program level, the system creates an Immediate Execution level variable of an implicit data type (real or string).

Example:

```
> 10 REAL BOY
> 20 INTEGER GIRL
> 30 BOY = 30
> 40 GIRL = 25
> 50 SONS = BOY
> 60 CHIL = BOY + GIRL
> 70 SING = ABS ( BOY- GIRL )
> 80 PRINT BOY, CHIL, SING
> RUN
30         55         5
> GIRL = 30
> GOTO 60
30         60         0
> PRINT SONS
30
> BABY = 5
> PRINT BABY
5
```

In the above program, statements 10 through 80 constitute the Program level. After the program is executed and the results are output, the Immediate Execution statements GIRL = 30 followed by the GOTO 60 (a return to statement 60 in the program) cause a reevaluation of the program using the new value for GIRL. After the result of this operation is printed, the Immediate Execution command PRINT SONS is evaluated. Since SONS is not defined on the Immediate Execution level but exists at the Program level, the program variable's value is used in this operation. The last Immediate Execution command specifies a variable which does not exist at the Program level. Thus, BABY is an Immediate Execution level variable.

If the program included a statement explicitly defining BABY as INTEGER, and the last two Immediate Execution statements were

```
BABY = 5. 5
PRINT BABY
```

the result would be

```
5
```

In this case BABY would be a Program level variable.

3. ELEMENTS OF BASIC

Constants, variables, and expressions are the fundamental elements of the TENET BASIC language. Used in conjunction with BASIC reserved words, they constitute the means by which all program data is manipulated.

TENET BASIC reserved words are listed in Appendix E.

CONSTANTS

Constants are program elements whose values remain unchanged through the programming process. They may be numbers or literal text (strings).

Numeric Constants

A numeric constant is a number which may be expressed in several formats: integer, real (single-precision floating-point), double (double-precision floating-point), complex and double complex.

Numeric constants may not contain embedded blanks.

INTEGER

An integer constant is expressed as a string of 1 to 9 significant digits with no exponent and no decimal point.

Integer values are stored internally as one word.

Examples:

```
234
234983943
4
```

SINGLE-PRECISION REAL

A single-precision real constant is expressed as a string of 1 to 7 significant decimal digits with a decimal point. An exponent (E format) may also be used. Single-precision real constants may be expressed in any of the following forms:

Single precision real values are stored internally as one word.

Form	Example
i.	.3
.f	.09845
i.f	888.234
i. E±e	4. E-5

Form	Example
.fE±e	.454E+3
i.fE±e	23.4E4

where i = integer part
f = fractional part
e = exponent

The range allowed for a single-precision real constant (x) is $|x| \leq 1.584 \times 10^{19}$

DOUBLE-PRECISION REAL

Double precision real values are stored internally as two words.

A double-precision real constant is expressed as a string of decimal digits with any of the following characteristics:

- More than 9 significant digits with no decimal point
- More than 7 significant digits with a decimal point
- A value greater than 1.584×10^{19}
- Double precision (D) exponentiation

Double-precision real constants may be expressed in any of the following forms:

Form	Example
i.	2098234234
.f	.00000005
i.f	4.2343434
i.D±e	6.D-11
.fD±e	.005D-6
i.fD±e	5.44D13

The range allowed for double-precision real constant (x) is $|x| \leq 6.296 \times 10^{76}$

COMPLEX AND DOUBLE COMPLEX

Complex values are stored internally as two words; double complex values are stored internally as four words.

Complex constants must be generated within a program using the complex number functions CMPLX (single-precision) and DCMPLX (double-precision). Any type of numeric constants may be used as arguments to these functions.

CMPLX (r, i) or DCMPLX (r, i)

where r = real part
i = imaginary part

Examples:

CMPLX (5.6, 1) creates a single precision complex constant using a single-precision argument for the real part and an integer argument for the imaginary part. The integer constant is converted to single-precision.

DCMPLX (.3E - 4, .5D - 9) creates a double-precision complex constant using a single-precision argument for the real part and a double-precision argument for the imaginary part. The single-precision constant is converted to double-precision.

String Constants

A string constant is any series of characters enclosed by a set of single (') or double (") quotation marks. A single quote is a valid character within a string enclosed in double quotes; a double quote is a valid character within a string enclosed by single quotes. However, both a single and double quote cannot appear as part of the same string constant.

- A Line Feed (LF) used in the generation of strings is not considered part of the string constant.
- A Carriage Return (CR) appearing as part of the string causes an error.
- The length of a string is limited to 255 characters (i.e., total line length allowed).

Examples:

" ABCDEFGHIJK "
" TIC IS (') "
' THE DOUBLE QUOTE (") IS VALID IN THIS STRING '

Reserved Constant Names

For convenience, the following commonly used numeric constants may be referenced by name. These values are initially set by the system but may be redefined by the user to represent other values.

Name	Value
EPS	10^{-10}
PI	3.14159265358979324

VARIABLES

A variable is a quantity whose value was previously defined, is not yet defined, or may change through the course of a program. There are two types of BASIC variables: scalar and subscripted.

Variable names may not contain embedded blanks or begin with the reserved letter combination FN.

Scalar Variables

A scalar variable represents a single quantity, numeric, or string. Its symbolic name can consist of up to four characters. The first character mustbe a letter, but the remaining characters can be digits, letters, or dollar signs. Blank spaces cannot be part of a variable name.
Examples:

```
AB$D
SUM
RATA
COSTA
X$X$
X
IVAL
B2
```

TENET BASIC maintains a set of variable names that are reserved words and, as such, may not be used as variable identifiers. These are listed in Appendix E . All numeric variables are initially assigned a value of zero.

Subscripted Variables

A subscripted variable name designates an element of an array (matrix). Arrays are multi-element structures whose components can be treated as single values. The content of an array is arranged according to both the size and the dimensioning defined for the array. For example, the numbers 1, 2, 3, 4, 5, 6 could be arranged as follows:

6 by 1	3 by 2	1 by 6	2 by 3
1	1 2	1 2 3 4 5 6	1 2 3
2	3 4		4 5 6
3	5 6		
4			
5			
6			

Operations which treat arrays as single values are discussed in section 7.

The user can reference individual items in an array by specifying the position of the item within the array. For example, COST (2, 2) references the item in row 2, column 2 of the array named COST. RATE (6) references the sixth element of the array named RATE.

The subscripted variable name consists of the name of the array followed by subscripts in parentheses. Multiple subscripts must be separated by commas. Each subscript consists of an arithmetic expression. (In TENET BASIC, an arithmetic expression is defined as any constant or variable, or combination of these joined by operators.)

The same name cannot be used for both a scalar and a subscripted variable in the same program.

Examples:

```
BAR ( 1,2*X )
GOOD ( 1,4,5 )
CAG$ ( 45 )
FI ( 9,8 )
```

Data Types and Variables

Program variables are assigned a data type (integer, real, string, etc.) either implicitly or explicitly. The data type of a variable name is ascertained implicitly by the content of the name itself. Conventionally, all names containing a dollar sign ($) are implicitly string variables and all others are data type real. This implicit data type convention may be overidden by explicit type declaration.

Explicit data type declaration statements are described on p.4-1.

EXPRESSIONS

An expression is any constant, variable or combination of these joined by operators and parentheses, as necessary, to denote the order in which operations are to be performed. There are four types of operators: unary arithmetic, binary arithmetic, relational, and logical.

Operators may not contain embedded blanks.

Unary Arithmetic Operators

Unary arithmetic operators operate on only one quantity (constant, variable, or the evaluated combination of these).

Symbol	Meaning	Example
+	Positive value	+4
-	Negative value	-16

Binary Arithmetic Operators

Binary arithmetic operators are used to form arithmetic expressions as in an ordinary mathematical notation.

Symbol	Meaning	Example
+	Addition	2+3
-	Subtraction	CVAL - IVAL
*	Multiplication	RATE * TIME * PRIN
↑	Exponentiation	4 ↑ 4
/	Division	16/4
MOD	Modulo	8 MOD 5

ARITHMETIC OPERATIONS ON COMPLEX VALUES

Arithmetic operations on complex values are performed on both the real and imaginary parts. For example:

```
if     A   =   CMPLX ( 3,1 )
       B   =   CMPLX ( 6,-2 )

then   A+B =   CMPLX(9,-1)
```

MIXED DATA TYPE ARITHMETIC

When variables of different data types are used in the same expression with arithmetic operators, data type of the result is determined as follows:

Result of Mixed Mode Operations Using *, -, and+

	Integer	Real	Complex	Double	Double Complex
Integer	Integer	Real	Complex	Double	Double Complex
Real	Real	Real	Complex	Double Complex	Double Complex
Complex	Complex	Complex	Complex	Double Complex	Double Complex
Double Complex	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex

Relational Operators

A relational operator is used to compare one quantity with another. They are evaluated for logical value: if the expression is true, its value is 1; if the expression is false, its value is 0.

Symbol	Meaning	Example
<	Less than	6<8 = 1 (true)
<=	Less than or equal to	6< = 5 = 0 (false)
>	Greater than	8>6 = 1 (true)
>=	Greater than or equal to	4> = 6 + 1 = 0 (false)
=	Equal to	4+3 = 6 + 1 = 1 (true)
#	Not equal to	6 # 8 = 1 (true)
#=	Approximately equal (determined by the value of EPS, p.3-3.)	.23D-11# = .24D-11 = 1 (true)

RELATIONAL OPERATIONS ON STRING VALUES

Any of the relational operators, except approximately equal to (#=) can be used to compare string values. Each character of a string has an associated numeric code. Two strings are compared character by character. The first pair of non-matching characters encountered determines the evaluation of the string. The character with the higher associated numeric code is considered the greater. Thus, the string " X " is considered greater than " AAAAAAAA ". If two strings of different lengths are identical up to the end of the shorter string, the longer string is considered greater. Thus " AAA " is considered greater than " AA ".

Code	Character	Code	Character
20	space	3A	:
21	!	3B	;
22	"	3C	<
23	#	3D	=
24	$	3E	>
25	%	3F	?
26	&	40	@
27	'	41	A
28	(	42	B
29	)	43	C
2A	*	44	D
2B	+	45	E
2C	,	46	F
2D	-	47	G
2E	.	48	H
2F	/	49	I
30	0	4A	J
31	1	4B	K
32	2	4C	L
33	3	4D	M
34	4	4E	N
35	5	4F	O
36	6	50	P
37	7	51	Q
38	8	52	R
39	9	53	S

Code	Character	Code	Character
54	T	5A	Z
55	U	5B	[
56	V	5C	\
57	W	5D	]
58	X	5E	↑
59	Y	5F	←

Examples:

The following are true (= 1):

" SUN " ≤ " SUNDAY "
" MATH " > " MANY "
" XXXX " < " XXXZ "

Strings may not be compared with numeric values.

RELATIONAL OPERATIONS ON COMPLEX VALUES

Relational operations on complex values are evaluated using the absolute value of the complex values.

$$\sqrt{r^2 + i^2}$$

where r = real part
i = imaginary part

Examples:

The following are true (=1):

CMPLX(3.4321, 5.45 CMPLX(4.324, 0)
CMPLX(4, 3) CMPLX(4, -1)

Logical Operators

Every numeric value also has a logical value. A numeric value not equal to zero has a logical value of true (one); a numeric value equal to zero has a logical value of false (zero).

NOT NOT X = 0 if X $\neq$ 0
= 1 if X = 0

AND A and B = 0 if A = 0 or B = 0
= 1 if A $\neq$ 0 and B $\neq$ 0

EQV A EQV B = 1 if A = 0 and B = 0
= 1 if A $\neq$ 0 and B $\neq$ 0
= 0 otherwise

IMP A IMP B = 0 if A $\neq$ 0 and B = 0
= 1 otherwise

```
XOR     A XOR B = 1 if A = 0 and B ≠ 0
                = 1 if A ≠ 0 and B = 0
                = 0 otherwise

OR      A OR B = 1 if A ≠ 0 or B ≠ 0
               = 0 if A = 0 and B = 0
```

Logical operations are not allowed on strings.

LOGICAL OPERATIONS ON COMPLEX VALUES

Logical operations on complex values are performed using the most significant real part of each complex value (i.e., the first word of each complex value).

Examples:

Each of the following is equal to 1:

```
CMPLX(3,4) IMP CMPLX(5,4)
CMPLX(3 234343,0) EQV CMPLX(34534343, 5)
CMPLX(5, 6) XOR CMPLX(0, 0)
```

Hierarchy of Operations

The order of performing individual operations within an equation is determined by the hierarchy of operators and the use of parentheses. Operations of the same precedence are performed from left to right in an expression. Operations within parentheses are performed before operations not in parentheses. The hierarchy (from highest to lowest) of operators is as follows:

```
unary +, unary-, NOT
↑
MOD
*,/
+,-
<, <=, >, >=, =, #, #=
AND
XOR, OR
IMP
EQV
```

Examples:

A/(-2.3 * IVAL) is evaluated:

1. -2.3
2. -2.3 * IVAL
3. A/(the result of step 2)

4* (B + RATE) ↑ (PRIN/TIME) is evaluated:

1. PRIN/TIME
2. B + RATE
3. (The result of step 2) ↑ (the result of step 1)
4. 4* (the result of step 3)

ALP*BETA-(X AND Y)/4.3*RATE+(TAG* (4.5 ↑ X)) is evaluated:

1. 4.5 ↑ X
2. TAG* (the result of step 1)
3. X AND Y
4. ALP*BETA
5. (The result of step 3)/4.3
6. (The result of step 5)*RATE
7. (The result of step 4) - (the result of step 6)
8. (The result of step 7) + (the result of step 2)

String Concatenation

The only arithmetic operation allowed on strings is concatenation - specified by the operator +. For example:

```
if  A$ = "ABCDEFG "
    B$ = "GHIJKLMN"
then  A$ + B$  =  " ABCDEFGGHIJKLMN "
```

4. ASSIGNMENT AND SEQUENCE CONTROL STATEMENTS

VARIABLE DECLARATION STATEMENTS

Because a variable name may be used to represent a variety of data types, variable names must be precisely declared as to the type of data they represent. If the user expects to perform complex data type operations and/or matrix operations, he must declare variables according to the type of values used in these operations.

In TENET BASIC all variables are considered to have two declarable aspects — data type (integer, single precision real, double-precision real, complex, double complex, or string) and structure (scalar or array). In general, the variable name itself implicitly declares the type and structure of the variable unless the user specifically declares otherwise. Unless a variable name is explicitly declared in a variable declaration statement, the following rules for implicit variable declaration are followed:

- A variable name containing the character $ is considered data type string.
- A variable name without the $ character is considered data type single-precision real.
- A variable name immediately followed by a left parenthesis is considered a one-dimensional array of 10 elements
- A variable name not immediately followed by a left parenthesis is considered scalar.

The user may override the implicit declaration conventions for variable names by explicitly declaring the type and/or structure of selected variables prior to their first use in a program. The program statements INTEGER, REAL, DOUBLE, COMPLEX, DOUBLE COMPLEX, STRING, and DIM are variable declaration statements. All but the DIM statement can be used to declare both the type and structure of program variables. DIM may be used only to declare (or redefine) structure.

In the following format specifications for variable declaration statements the element "var" designates a scalar or an array variable There is one exception: only array variables may be declared in a DIM statement.

INTEGER

INTEGER var_1 [, var_2, ... var_n]

INTEGER ISIN, ART$(10), IVAL(6:10, IVOR:12)

INTEGER may be used for Immediate or Program Execution.

The INTEGER statement declares that the variables listed represent integer values and are to be stored as signed two's-complement integer values. Each integer value requires one 32-bit computer word.

Integer values have precision for up to nine significant digits and the maximum size number allowed is 2147483647 ($2^{31} - 1$).

REAL

REAL var_1 [, var_2, ... var_n]

REAL LBS, MASS, DEN (100)

REAL may be used for Immediate or Program Execution.

The REAL statement declares that the variables listed represent single-precision real values and are to be stored as single precision floating point values with a 25-bit signed mantissa and a 7-bit signed exponent. Each single-precision real value requires one 32-bit computer word.

Single-precision real values have precision up to seven significant digits, and the maximum range allowed is $|x| \leq 1.584 \times 10^{19}$.

DOUBLE

```
DOUBLE var₁ [ ,var₂,...varₙ ]
```

DOUBLE FPS (-19:10, 2, 0:5)

The DOUBLE statement declares that the variables listed represent double-precision real values and are to be stored as double-precision floating-point values with a 55-bit signed mantissa and a 9-bit signed exponent. Each double precision real value requires two 32-bit computer words.

DOUBLE may be used for Immediate or Program Execution.

Double-precision real values have precision up to 16 significant digits, and the maximum range allowed is $| x | \leq 6.296 \times 10^{76}$.

COMPLEX

```
COMPLEX var₁ [ ,var₂,...varₙ ]
```

COMPLEX A2, B2, C2, BETA

The COMPLEX statement declares that the variables listed represent complex values and are to be stored as two single-precision floating point values. The first value is the real part and the second is the imaginary part. Each complex value to be stored requires two 32-bit computer words.

COMPLEX may be used for Immediate or Program Execution.

The range and precision for each part of a complex value is the same as for single-precision real values.

DOUBLE COMPLEX

DOUBLE COMPLEX var$_1$ [, var$_2$, ... var$_n$]

DOUBLE COMPLEX AMPS (20, 30)

DOUBLE COMPLEX may be used for Immediate or Program Execution.

The DOUBLE COMPLEX statement declares that the variables listed represent double complex values and are to be stored as two double-precision floating-point values. The first (double-precision) value is the real part; the second is the imaginary part. Each double complex value requires four 32-bit computer words.

The range and precision for each part of a double complex value is the same as for double-precision real values.

STRING

STRING var$_1$ [, var$_2$,... var$_n$]

STRING NAME (300), ADDR (300), ZIP (300):5

STRING may be used for Immediate or Program Execution.

The STRING statement explicitly declares that the variables listed represent string values. Array string variables may be of fixed or varying length; scalar string variables must be of varying length. A varying length string is one whose length is not specified in the declaration statement. For example, STRING AB (30) declares a string array of 30 elements of varying length. Space is assigned to varying length strings dynamically — according to the length of the string values assigned to them in the course of program execution.

A fixed-length string array is declared by appending to the array declaration a ":" followed by an integer constant that designates the number of characters for each element of the array. For example, STRING AS (30):12 specifies that each element of the 30-element array AB is 12 characters long.

Space is reserved for a fixed-length string array based on the number of elements and the number of characters in each element. All string values supplied for the array must be compatible with the string length specified by the user Strings that are shorter are left-justified with trailing blank fill. Strings that are longer cause a warning message to be issued when program execution is attempted and the string is truncated with loss of the rightmost characters.

Maximum string length is 255 characters for both fixed and varying length strings.

DIM

DIM $var(s_i)_1$ [, $var(s_i)_2$, ... $var(s_i)_n$]

DIM BETA (4, 6, 9), X$ (0:15), TEMP (IVAL, 80)

The DIM statement may either declare new array variables according to implicit type conventions, or for previously declared arrays, redefine the total size, range of subscripts, and number of subscripts as long as the total size of the area originally assigned to the array is not exceeded and the total number of dimensions originally specified is not exceeded. When used to redefine an explicitly declared array variable, the DIM statement does not redefine data type.

DIM may be used for Immediate or Program Execution.

When used to declare a new array variable, the DIM statement is essentially a REAL statement for variable names without the $ character, and a STRING statement for variable names with the $ character.

Variable Type Declaration Rules

- Any variable name without a subscript list appearing in a declaration statement is defined as a scalar variable. The name cannot appear in any other declaration statement and can never be used in the same program with subscripts.

- Any variable name with a subscript list appearing in a declaration statement is defined as an array The subscript list consists of the numbers of dimensions and the range for each dimension. The amount of space reserved for an array is determined by the subscript values during execution. The size and dimension range can change each time a subsequent DIM or type declaration statement is executed. However, the execution of the first type declaration statement determines the maximum space allocated for the variable. Any subsequent alterations in the range of the dimensions must not define an array whose total size exceeds that of the initial declaration.

- A dimension is declared by either a single expression indicating the upper limit of the subscript range with an implicit lower value of 1, or by a pair of expressions separated by a colon: For example: AB (3:6*Q, 10) specifies a two-dimensional array whose second subscript range is from 1 to 10 .

- The subscript range must be in increasing order, but may consist of negative elements: For example: AB(-3:6) .

- All arrays of more than one dimension must be declared in a type declaration or DIM statement before they are used in a program.

LET OR ASSIGNMENT STATEMENT

[LET] var_1 [, var_2, ... var_n] = exp

```
LET A = -2
LET VAL1, VAL2, VAL3 = 3*A/4
B = 3.454/VAL1
LET BETA ( 3, 4 ) = 3
```

The LET, or assignment statement enables the user to assign the value of an expression to the variable(s) on the left side of the equal sign.

- The assignment statement is the only BASIC statement that need not begin with a reserved word BASIC command.
- If the values on the left side of the assignment statement have different data types, the expression is evaluated and conversion is performed relative to each of the values independently.

LET may be used for Immediate or Program Execution.

Mixed Data Type Assignments

Whenever a variable of a particular data type is assigned to the value of a quantity of a different data type, the original variable is converted according to Table 4-1, where if A = B, conversions will occur according to the data type of A and B.

Variables may also be assigned by the READ (p. 4-20), DATA (p. 4-19), FOR (p. 4-15), and INPUT (p. 6-7) statements.

Table 4-1. Mixed Data Type Assignments

A	B	RESULT
INTEGER	INTEGER	B
	REAL	INT (B)
	DOUBLE	INT (B)
	COMPLEX	INT (REAL (B))
	DOUBLE COMPLEX	INT (REAL (B))
REAL	INTEGER	FLOAT (B)
	REAL	B
	DOUBLE	FLOAT (B)
	COMPLEX	FLOAT (REAL (B))
	DOUBLE COMPLEX	FLOAT (REAL (B))
DOUBLE	INTEGER	DBL (B)
	REAL	DBL (B)
	DOUBLE	B
	COMPLEX	DBL (REAL (B))
	DOUBLE COMPLEX	DBL (REAL (B))
COMPLEX	INTEGER	CMPLX (B, 0)
	REAL	CMPLX (B, 0)
	DOUBLE	CMPLX (B, 0)
	COMPLEX	B
	DOUBLE COMPLEX	CMPLX (REAL (B), IMAG (B))
DOUBLE COMPLEX	INTEGER	DCMPLX (B, 0)
	REAL	DCMPLX (B, 0)
	DOUBLE	DCMPLX (B, 0)
	COMPLEX	DCMPLX (REAL (B), IMAG (B))
	DOUBLE COMPLEX	B

Examples:

```
10 DOUBLE IVAL, XVAL
20 INTEGER BETA
25 XVAL = 1.25
30 IVAL = 6
40 ALPH, BETA = IVAL/XVAL
50 PRINT ALPH, BETA, XVAL, IVAL
RUN
4.8       4       1.25       6
```

Since the variable ALPH is not explicitly declared in the foregoing program, it is considered type real; the result of the operation in line 40 causes data type conversion, and the value of ALPH is printed out as a real value. However, as BETA was explicitly declared type integer, the result of the operation at line 40 is converted from type double-precision to integer.

```
10 INTEGER IVAL
20 REAL RVAL
30 RVAL, IVAL = 1/2
40 PRINT RVAL, IVAL
50 END
RUN
 .5  0
```

The expression in statement 30 is evaluated and converted appropriately for each of the values on the left-hand side of the (first) equal sign.

DO STATEMENT

```
DO line no. [ : line no. ] [, line no. [ :line no. ]] ... [ , line no. [ :line no. ] ]
```

```
DO 50: 100, 150, 200
DO 60: 100, 10, 50
DO 550
```

DO may be used for Immediate or Program Execution.

Other statements which enable execution sequence modification are GOTO (p. 4-12), ON (p. 4-13), FOR (p. 4-15), IF (p. 4-14), and GOSUB (p. 5-6).

The DO statement enables a statement or a set of statements to be selectively executed in its place. A single statement is specified by its line number; a set of statements is indicated by a range of two line numbers separated by a colon. Once the statements indicated by the DO statement are executed, program execution continues at the next sequential statement following the DO statement.

- Functions, GOSUB, and other DO statements may be executed within the range of a DO statement. If a function or GOSUB contains DO statements, these DO statements are terminated when the GOSUB or function is returned. In other words, execution of a RETURN statement causes premature termination of any DO statements in effect at that subprogram level.

- An attempt to execute a statement outside the range specified in a DO statement causes control to be returned to the DO statement. For example: If within the range 20: 80, statement 30 is a GOTO, it may branch only to a statement within the range 20: 80. Otherwise control returns to the DO.

- When DO statements are nested, the last DO statement within the set is the first completed.

  Examples:

  ```
  10 ...
  20 ...
  30 ...
  40 ...
  50 DO 20
  60 END
  ```

  The above statements are executed in the following sequence: 10, 20, 30, 40, 50, 20, 60.

```
10 ...
20 DO 40
30 ...
40 GOTO 100
.
.
.
100 ...
```

The above statements are executed in the following sequence: 10, 20, 40, 30, 40, 100. The statement at line 100 is outside the range of the DO statement at line 20 and hence is not executed after the first execution of statement 40.

```
10 ...
20 DO 60: 90
25 GOTO 100
30 ...
40 ...
50 ...
60 ...
70 ...
80 DO 30: 50
90 ...
100 ...
```

The most recently executed DO defines the current control range. These statements would be executed in the sequence shown below as controlled by normal execution and the two DO statements.

```
Normal
Sequence ──►10, 20,                          , 25, 100
First DO ───────►60, 70, 80,           , 90
Second DO ──────────────────►30, 40, 50
```

GOTO STATEMENT

```
GOTO line no.
```

```
GOTO 10
GOTO 45
```

GOTO may be used for Immediate or Program Execution.

The GOTO statement unconditionally transfers control from one point to another in a program.

Other statements which enable execution sequence modification are DO (p.4-10), GOSUB (p.5-6), ON (P.4-13), FOR (p.4-15), and IF (p.4-14).

- After the GOTO statement is executed, program execution continues from the line number specified.
- If the line number specified is a non-executable statement, control passes to the next executable statement.

Example:

```
20 GOTO 60
30 RATE = BETA + 1
40 GOTO 80
50 ...
60 LET IVAL = RATE
70 GOTO 30
80 END
```

The above program statements would be executed in the following sequence: 20, 60, 70, 30, 40, 80.

ON . . . GOTO STATEMENT

ON exp GOTO line no.$_1$ [, line no.$_2$, . . . line no.$_n$]

```
ON Y GOTO 200, 400, 600, 800
ON BETA + RATE GOTO 20, 30, 40, 50, 60, 70, 80
ON A > BETA GOTO 25
```

The ON statement (or multi-branch GOTO) is a variation of the GOTO statement. Whereas GOTO unconditionally transfers program control, the ON statement presents a choice of line numbers to which control may be transferred. The expression is the condition of transfer; its numeric value determines which of the line numbers in the list is to be executed next. (Line numbers are in the implicit sequence 1, 2, 3, . . .) If the expression has a value not in the range 1 to n (n being the number of line numbers in the list), program control is not transferred, and execution continues at the statement following the ON statement.

ON...GOTO may be used for Immediate or Program Execution.

Other statements which enable execution sequence modification are GOTO (p.4-12), DO (p.4-10), GOSUB (p.5-6), FOR (p.4-15), and IF (p.4-14).

- The expression is evaluated (and truncated, if necessary) to derive an integer value. If the expression is logical, only a true (i.e., non-zero) value would cause a transfer of program control.

Examples:

```
40 LET A, B = 2.2
50 ON A + B GOTO 60, 80, 110, 150
60 LET VAL = X + 1
70 GOTO 300
80 LET VAL = X + 3
90 GOTO 300
110 LET VAL = X + 4
140 GOTO 300
150 LET VAL = X
160 GOTO 300
```

The value of the expression in statement 50 is 4 (4.4 truncated) indicating that program control will be transferred to the fourth item in the line number list, statement 150.

```
90 LET BETA = 6
100 LET A = 3
110 ON A XOR B GOTO 140
130 . . .
140 . . .
```

As the value of the expression in statement 110 is false and therefore outside the range of the line number list, execution resumes at line 130.

IF STATEMENT

$$\text{IF exp THEN} \left\{ \begin{array}{l} \text{statement}_1 \\ \text{line no.}_1 \end{array} \right\} \text{ ELSE } \left\{ \begin{array}{l} \text{statement}_2 \\ \text{line no.}_2 \end{array} \right\}$$

```
IF A = B THEN 50
IF A * BETA THEN LET RATE = IVAL
IF C>3.45 THEN BETA = A ELSE GO TO 80
IF RATE = IVAL THEN 40 ELSE 100
```

IF...THEN may be used for Immediate or Program Execution.

Other statements which enable execution sequence modification are GOTO (p.4-12), DO (p.4-10), GOSUB (p.5-6), ON (p.4-13), and FOR (p.4-15).

The IF statement is used to override the normal sequence of program execution. The logical value of the expression following the IF determines subsequent operations. If the logical value is true (non-zero), the statement or line number following the THEN is executed next followed by the next sequential program statement (not the statement following ELSE, if any). If the logical value is false (zero), the statement or line number following either ELSE (if present), or the next sequential statement is executed.

- Specifying a line number after THEN or ELSE is equivalent to a GOTO line number statement.

- The following types of statement may not appear in an IF statement: DATA, DEF, DOUBLE, INTEGER, IF REAL, REM, STRING, COMPLEX, FOR, ON, NEXT, DOUBLE COMPLEX, DIM.

Examples:

```
10 A = 6
15 BETA = 0
30 IF A = 6 THEN 50
40 . . .
50 IF A * BETA THEN GOTO 70 ELSE B = B + 1
60 GOTO 50
70 END
```

As the value of the expression in statement 30 is true, control is transferred to statement 50. At this point A * BETA is zero (i.e., false). The alternate statement (B = B + 1) is executed next and then statement 60 returns control to statement 50. The value of the expression A * BETA is now true. Control is then transferred to statement 70.

FOR AND NEXT STATEMENTS

$$\text{FOR var} = \text{exp}_1 \left\{\begin{matrix}\text{TO}\\ \text{WHILE}\end{matrix}\right\} \text{exp}_2 \left[\text{STEP exp}_3\right]$$

$$\text{NEXT var}_1 \; [\; , \text{var}_2, \ldots \text{var}_n \;]$$

```
FOR BETA = 1 TO 10
•
•
NEXT BETA
•
•
FOR RATE = 5.5 * IVAL TO ADDR STEP .5/XVAL
•
•
NEXT RATE
•
•
FOR IVAL = 4 WHILE IVAL > = 143 STEP TAU`
FOR XVAL = BYO WHILE AA STEP TAU * 3.34
•
•
NEXT XVAL, IVAL
```

FOR and NEXT may be used for Program Execution only.

The FOR statement used in conjunction with the NEXT statement causes the repeated execution of a set of program statements (loop). The FOR statement defines the beginning of a loop; the NEXT statement is the terminal statement of the loop.

The FOR statement specifies the initial value (exp_1), the terminal value (exp_2) and the incremental value(exp_3) of the loop variable (var). The loop variable may be implicitly defined by its use as a loop variable. (It can be any data type except string or complex.)

The computed value of the expression exp_1 is the value to which the loop variable is set before the loop is first executed. It must always be specified. The value of exp_2 determines when the looping process will terminate.

Terminal conditions may be static or dynamic. A static terminator, indicated by " TO exp_2", is evaluated once— when the loop is initialized. When the loop variable becomes greater (or less for negative increments) than the terminal value, execution is terminated. Normal statement processing resumes at the statement following the end of the loop (after the loop's companion NEXT statement).

A dynamic terminator, indicated by WHILE, is evaluated each time the loop is executed. As long as exp_2 (following WHILE) is logically true ($\neq 0$) the looping process continues. Otherwise the loop is terminated.

The increment value exp_3 is a negative or positive value by which the loop variable is incremented when the NEXT statement is executed upon completion of each pass through the loop. If an increment value is not specified it is assumed to be 1. The loop increment is the value of the expression at initialization.

FOR loops may be nested to a level of 31.

FOR ... NEXT loops may be nested, but not overlapped. For example:

```
┌  FOR X ...
│  .
│  .
│┌ FOR Y ...
││ .
││ .
│└ NEXT Y
│  .
│  .
└  NEXT X
```

or

```
┌  FOR X ...
│  .
│  .
│┌ FOR Y ...
││ .
││ .
└└ NEXT Y, X
```

The following is illegal:

```
┌  FOR X
│  .
│  .
│┌ FOR Y
││ .
││ .
└┼ NEXT X
 │ .
 │ .
 └ NEXT Y
```

The NEXT statement may terminate more than one loop. A NEXT statement which terminates multiple loops must list loop variables in inverse order of their appearance in nested FOR statements.

- Companion FOR and NEXT statements must specify the same loop variable. There must be a one-to-one correspondence of FOR statement loop variables and NEXT statement loop variables.

- A transfer may occur out of a FOR loop only if there is a companion transfer into the range of the same FOR loop and vice versa. Thus the following is permissible:

```
    10 FOR X TO Y STEP IVAL
    .
    .
    30 GOTO 180
    40 BETA = X * ALPH
    .
    .
    70 NEXT X
    .
    .
    180 ...
    .
    .
    200 GOTO 40
```

  Caution should be exercised when branching from and to FOR loops, especially between two FOR loops with the same loop variable name. When a NEXT statement is executed a return occurs to the most recently executed FOR statement with the same loop variable.

  For example:

```
    10 FOR X TO Y
    .
    .
    40 GOTO 500
    .
    .
    60 NEXT X
    .
    .
    480 FOR X TO BETA
    500 ...
    .
    .
    550 NEXT X
```

  In the above, statement 40 causes a transfer out of the FOR loop beginning at line 10. Control is transferred to statement 500, within another FOR loop with the same loop variable name X. As statement 10 is the most recently executed FOR statement, control will be transferred from the NEXT statement at line 550 to the loop beginning at line 10.

- The value of a loop variable may be modified within the range of a FOR loop or by any program statements accessed from within the range of a FOR loop.

- The loop variable is assigned the initial value even if the terminal condition (static or dynamic) is not met on the first pass and the loop itself is not executed.

Example:

The following program prints the value and square root of BETA, where BETA takes on 100 values from 1 to 100.

```
10 FOR BETA = 1 TO 100
15 PRINT BETA, SQRT ( BETA )
30 NEXT BETA
```

The above program is equivalent to:

```
10 BETA = 0
15 BETA = BETA + 1
30 PRINT BETA, SQRT ( BETA )
40 IF BETA<=100 GOTO 15
50 END
```

Both of the above programs are equivalent to:

```
10 PRINT 1, SQRT ( 1 )
15 PRINT 2, SQRT ( 2 )
30 PRINT 3, SQRT ( 3 )
.
.
990 PRINT 99, SQRT ( 99 )
1000 PRINT 100, SQRT ( 100 )
```

DATA STATEMENT

DATA const_1 [, const_2 ... const_n]

```
DATA 4, 0.05, -6.3, 0, 0
DATA " THIS IS A STRING ", 6.374E-3, -3.14159E3
DATA 100
```

The DATA statement supplies internal data to be used in a program. Before a program is executed, all constants appearing in all of the program's DATA statements are organized into an internal data list according to their order of appearance in the program. Whenever a (non-file) READ statement is executed, data is obtained from this list. There is a one-to-one relation of constant to variable – the first constant is assigned to the first variable, and so forth. A data list pointer moves through the list sequentially as values are assigned by READ statements.

DATA may be used for Program Execution only.

The normal sequence of obtaining items from the data list can be altered using the RESTORE statement (p.4-21).

- Constants in the data list may be numeric or string.
- Complex or double complex constants are read from the internal data list as two words. The first word is the real part; the second is thc imaginary part.

The rules of mixed data type assignments apply to READ and DATA statements.

- String values in DATA statements may be continued onto the next line by enclosing the string within quotes and using the Line Feed (LF) key to continue typing on the next line. (LF) will not appear as part of the string.
- Since the function of DATA statements is to create an internal data list prior to program execution, DATA statements are ignored when a program is actually executed.

  Examples:

  Since DATA statements are used only in conjunction with READ statements, examples of their usage are found under the discussion of the READ statement.

READ STATEMENT

READ var$_1$ [, var$_2$ ... var$_n$]

READ I, J, ALL (I, J)
READ VAL1, VAL6, ANS$
READ A$, B$, C$, D$, E$ (4)

READ may be used for Immediate or Program Execution.

The READ statement specifies a list of variables to be assigned values when the program is executed; these values are obtained from the internal data list generated by the program's DATA statements.

Whole arrays can be read using the MAT READ statement (p. 7-11).

- Variables specified in READ statements may be simple or subscripted.

The rules of mixed data type assignments apply to READ and DATA statements.

- Complex or double-complex values are read as two numeric constants. The first numeric constant is considered the real part; the second is considered the imaginary part.

Part or all of the data list may be reaccessed by using the RESTORE statement (p. 4-21).

- As data list items are read the data list pointer is advanced to the next item in the list.

Another form of the READ statement may be used for reading files (p. 8-12).

- When the end of the data list is reached, an attempt to read a value for a variable causes an error message to be issued. The user should then take appropriate remedial action.

Examples:

```
10 READ I, J
20 DATA 4, 6,13.3
30 READ X ( I, J ), Z$
40 DATA " END "
```

Values are assigned as follows:

```
I = 4
J = 6
X ( I, J ) = 13.3
Z$ = END
```

RESTORE STATEMENT

```
RESTORE [ line no. ]
```

RESTORE
RESTORE 35

The RESTORE statement enables the user to change the position of the data list pointer of the internal data file created by DATA statements, thereby allowing repeated access to items within the list.

RESTORE may be used for Immediate or Program Execution.

- If the RESTORE statement does not specify a line number, the data list pointer moves to the first item in the list. The next READ is performed at this point.

- If a line number is specified, the data list pointer is reset to the first constant appearing in the specified statement. (Line numbers specified in RESTORE statements must reference DATA statements.)

DATA statements are discussed on p. 4-19.

Examples:

```
10 DATA 1, 2, 3
20 READ A, BETA, CAT
30 RESTORE
40 READ X, Y, Z
```

Values are assigned as follows:

```
A    = 1    X = 1
BETA = 2    Y = 2
CAT  = 3    Z = 3
```

```
10 DATA 10, 20, 30
20 DATA "HELLO", "BYE", "SOLONG"
30 READ VAL1, VAL2, VAL3, V$1, V$2, V$3
40 RESTORE 20
50 READ A$1, A$2, A$3
```

Values are assigned as follows:

```
VAL1 = 10      A$1 = HELLO
VAL2 = 20      A$2 = BYE
VAL3 = 30      A$3 = SOLONG
V$1 = HELLO
V$2 = BYE
V$3 = SOLONG
```

REM STATEMENT

	REM any string of characters
or	statement ! any string of characters

REM THIS PROGRAM COMPUTES YEARLY INTEREST RATES

X = W * 5.25 ! THE VALUE OF X IS COMPUTED HERE

REM may be used for Immediate or Program Execution.

The REM statement acts as a comment with which the user can annotate the listing of his program. It may be inserted at any point in the program without affecting execution. The REM statement may not be used as part of another statement. Individual statements can be annotated by adding ! followed by the comment. All characters between the ! and CR are considered to be the comment.

Examples:

```
10 REM THESE FIGURES ARE BASED ON FY'70
20 LET NET1 = GROS-BETA        ! COMPUTE NET PROFIT
30 NEW = NET1/NET0             ! COMPUTE GROWTH RATE
```

PAUSE STATEMENT

```
PAUSE
```

PAUSE

The PAUSE statement causes a suspension of program execution. When this statement is executed, the following message is issued:

PAUSE AT line no.

The status of program variables may be interrogated and/or altered during suspension of program execution. When the user enters the Immediate Execution command CONTINUE, program execution will resume at the statement following the PAUSE.

PAUSE may be used for Program Execution only.

STOP may be used in place of the word PAUSE.

END STATEMENT

```
END
```

END

END may be used for Program Execution only.

The END statement is used to terminate a multi-line function and return control to the main program. It is also optionally used to terminate a main program and return control to the BASIC subsystem.

Multi-line user-defined functions are discussed on p. 5-4.

When an END statement is executed in a multi-line function, it is equivalent to a RETURN with a zero value result. When an END is encountered in the main program, control returns to the BASIC subsystem. The END statement for the main program is optional since control will automatically return to the BASIC subsystem after the highest numbered statement is executed.

5. FUNCTION AND SUBROUTINE STATEMENTS

INTRODUCTION

In addition to using the built-in functions provided by TENET BASIC, one can write a set of statements that can serve as a user-defined function. Once defined as a function, it may be used throughout a program.

Up to 30 functions may be defined by the user in each program.

Built-in functions are described in Appendix D.

A call to an undefined function causes an error message to be issued when Program Execution is attempted.

Unique functions may be defined either as a single line function or as a multi-line function using the DEF statement. Both single and multi-line functions return a value as do the standard functions, and they must be defined before they are referenced in a program. At execution time function definitions are skipped and are not executed until they are referenced by name in an expression. The appearance of a function name in an expression is considered a function call which specifies the input arguments (values to be used when the function is executed). After a function is executed, the value of the result is returned to the expression, and evaluation of the expression continues.

SINGLE-LINE FUNCTION — DEF STATEMENT

DEF [data type] FNname [(arg_1, arg_2, ... arg_n)] = exp

```
DEF FNBETA = ( 3.14159 * D↑2 )/4
DEF FNAB ( A, B ) = ( A + B ) ↑ 2/( A-X ) ↑ 2
DEF DOUBLE COMPLEX FNNAME = ( 7874 * X ↑ 3 )
```

DEF may be used for Program Execution only.

The single-line function, when referenced in a program, uses the expression after the first equal sign to determine a value to be returned to the function call statement.

When a function call is made, arguments (if any) are checked for type compatibility. The arguments are known as argument dummies, local to the function and have no relationship to any variables of the same name outside the function. Each argument of a single-line function has the same data type as the function itself. Argument dummies have a one-to-one correspondence with the calling arguments in a function call.

Restricted function names are included in the TENET BASIC reserved word list (Appendix E).

- DEF statement arguments are optional and must be simple variables (non-subscripted).
- Each function is identified by the letters FN followed by a name of one to four characters.
- The expression in the DEF statement follows the standard rules for mixed mode operations. Any variable in the expression which is not an argument dummy refers to the non-local variable.

The non-local variable is the variable with the same name outside the function (at the main program level).

- The data type of the function result may optionally be specified in the DEF statement.
- The expression itself may contain calls to previously defined functions but cannot directly or indirectly call itself.
- The input arguments to a function call must be type compatible.

Examples:

```
20 DEF FNCIRC ( D ) = ( 3.14159 * D ↑ 2 )/4
```

Statement 20 defines a function to compute the area of a circle.

```
10 DEF FNAB ( A, B ) = ( A + B ) ↑ 2/( A-X ) ↑ 2
20 X = 3
30 Y = FNAB ( 6, 3 )
```

Statement 30 is a call to the function defined in statement 10. Y is assigned the value returned by the function, which in this case is 9.

MULTI-LINE FUNCTION — DEF STATEMENT

DEF [data type] FNname [arg_1, arg_2, ... arg_n)]
•
•
RETURN exp
END FNname

```
DEF INTEGER FNRATE ( A, B, C, D )
•
•
RETURN
END
```

DEF may be used for Program Execution only.

Restricted function names are included in the TENET BASIC reserved word list (Appendix E).

The multi-line function is defined by a set of statements independent of the main program beginning with a DEF statement and terminated by an END statement. The RETURN statement specifies the value to be returned to the main program. Statements that are part of the multi-line function definition cannot reference, or be referenced by statements outside the range of the function definition.

Up to 30 dummy arguments may be specified for each multi-line user-defined function.

- The function argument dummies are optional. If specified, they must be explicitly declared in a type statement within the function before their use.

Non-local arrays, except for dummy arguments, may not be redimensioned within a function.

- All variables used within a function are non-local to the function except those that are dummy arguments and those that are explicitly declared in a data type of DIM declaration statement within the function.

Multi-line functions may not include GOSUB statements.

- The value of a scalar variable appearing in a function call statement is not changed by a function, even though the equivalent function argument dummy has been changed by the function.

The data type and number of dimensions of the calling array and its corresponding dummy (array) argument must be consistent.

The non-local variable is the variable with the same name outside the function (at the main program level).

- A dummy argument may be an array. An array dummy must appear in a DIM or type statement with dimension information in the function even though the variable has been declared in the main program. Its declaration in the function must precede its use in the function. Arrays may be redimensioned local to the function. The dimensions of the calling array remain unchanged outside the function although values in the calling array may be changed by the function.

- When a dummy array is redimensioned, a reference to an element of the array by a subscripted variable does not necessarily address the same data as an equivalent reference by the same subscript outside the function.

The rules for dynamically redimensioning arrays are described on p. 4-5.

- When the RETURN statement is executed, the value of the expression in the RETURN statement is returned to the main program, and execution continues from the point at which the function was called. When a value is not specified, the resultant value returned by the function is 0.

- A function cannot be defined within another function.

- There can be a maximum of 30 user-defined functions in a program.

- A data type declaration may precede the function name in the DEF statement. Otherwise the implicit type rule is used.

Example:

```
200 DEF DOUBLE FNDERR ( VAL1, VAL2 )
210 DOUBLE VAL1, VAL2, TEM1, TEM2
220 TEM1 = SQRT ( VAL1 ↑ 2 )
230 TEM2 = SQRT ( VAL2 ↑ 2 )
240 IF TEM1>TEM2 THEN RETURN TEM1
250 RETURN TEM2
260 END
270 Y = 6
280 X = 7
.
.
400 PRINT FNDERR ( X, Y )
```

Statement 400 is a call to the function defined at line 200. The values of X and Y are used as input to FNDERR. (Their values are not affected in the main program.)

GOSUB STATEMENT

```
GOSUB line no.
```

GOSUB 200

GOSUB may be used for Immediate or Program Execution.

The GOSUB statement is used to direct program control to a subset of the main program. The line number specified in this statement is the first line of the subset. Execution continues from this point until a RETURN statement is encountered. Control is then returned to the statement following the GOSUB call.

- GOSUB statements may not appear within the range of multi-line functions.
- GOSUB's may be nested to any level.
- All subroutines (beginning with a GOSUB statement) must end with the execution of a RETURN statement. Unlike the RETURN statement that terminates user-defined functions, the RETURN that terminates GOSUB statement sets does not specify an expression since a value is not returned to the main program.
- An error occurs whenever a RETURN is executed without an associated GOSUB.

RETURN may be used for Program Execution only.

The RETURN statement is discussed on p.5-4.

Examples:

```
190 X = 3.4
195 Y = 2.3
200 IF X>Y THEN GOSUB 400
210 A = X+Y
.
.
400 Y = SQRT ( Y )
500 X = X * Y
600 RETURN
```

As the expression in statement 200 is true, the statements beginning at line number 400 are executed. This routine changes the value of the variables X and Y. When statement 600 turns control to line number 210, the new values of X and Y are used to determine the value of A.

ON . . . GOSUB STATEMENT

ON exp GOSUB line no.$_1$,[line no.$_2$, ... line no.$_n$]

```
ON A + B GOSUB 400, 500, 600, 700
ON Y GOSUB 60, 100
```

The multi-branch GOSUB statement is comparable to the multi-branch GOTO statement. Both specify a condition that determines a portion of the program to which control will be transferred. However, the ON ... GOSUB statement transfers control to a subset of the program, which after execution, returns control to the statement following the ON ... GOSUB.

ON...GOSUB may be used for Immediate or Program Execution.

All subroutines (beginning with a GOSUB statement) must end with the execution of a RETURN statement.

The expression in the ON ... GOSUB statement is the condition of transfer; its numeric value determines which of the line numbers in the list (each the beginning of a subset) is to be executed next. Line numbers are implicitly numbered sequentially starting with a value of 1. If the expression has a value not in the range 1 to n (n is the total number of line numbers in the list), program control is not transferred and execution continues at the statement following the ON ... GOSUB.

- The expression is evaluated (and truncated, if necessary) to derive an integer value. If the expression is logical, only a true (i.e., non-zero) value would cause a transfer of program control.
- ON ... GOSUB statements may not appear within the range of multi-line functions.

Example:

```
50 ON Y GOSUB 100, 200, 300
60...
```

Value of Y	Execution Sequence after ON ... GOSUB
0 or less	Statement 60
1	Statement 100
2	Statement 200
3	Statement 300
4 or more	Statement 60

6. TERMINAL INPUT/OUTPUT STATEMENTS

This section discusses the input/output (I/O) statements applicable to the transmission of data between a program and a terminal. I/O operations associated with matrices and disc files are discussed in sections 7 and 8, respectively. As a rule, most of the options available for terminal I/O may be used for matrix and file I/O.

All information input from or output to the terminal is in symbolic form. Information may be input or output at the terminal in standard (default) format, whereby the user has minimal control over format, or in a user-defined format which allows the user considerable format control.

STANDARD FORMAT STATEMENTS — PRINT

$$\text{PRINT } \text{exp}_1 [\left\{ \begin{matrix} , \\ ; \\ : \end{matrix} \right\} \text{exp}_2 \left\{ \begin{matrix} , \\ ; \\ : \end{matrix} \right\} \ldots \text{exp}_n [\left\{ \begin{matrix} , \\ ; \\ : \end{matrix} \right\}]]$$

```
PRINT " ANSWER = "; A * BETA
PRINT 5
PRINT X +A/BETA:
```

PRINT statements may be used for Immediate or Program Execution.

Each print line consists of 6 fields of 12 characters each.

The PRINT statement prints output data at the user's terminal. Each of the expressions (exp_i) in the PRINT statement is evaluated,and the resultant value(s) is output at the terminal. The user may control the spacing of information printed by the separating delimiters.

The TAB function is described on p. 6-5.

Negative numbers and strings are truly concatenated, but a blank space is reserved before positive numbers.

Delimiter	Meaning
,	Positions the next item of information at the next field on the print line.
;	Positions the next item of information three characters from the present position.
:	Positions the next item of information at the next character position. (Output is concatenated.)

- When the position for the next item exceeds the line width (position 72 on a teletypewriter), a Carriage Return and Line Feed is issued; and the item is printed at the beginning of the next line.

- Delimiters may appear in direct sequence (e. g. , PRINT A::B).

Terminal output spacing is approximated owing to the space limitations of this page.

- A delimiter appearing as the last element of a PRINT statement specifies the print position of the first item of information in the next PRINT statement executed. When a final delimiter is not specified, the next PRINT statement's output will be positioned at the beginning of the next line.

- Numeric values are left-justified within each field, and a maximum of six significant digits are printed. Values greater than 10×10^{66} and less than 0.1 are printed in E format. Integer values appear without a decimal point (e.g., 8.00 is printed as 8).

- The first position of a numeric field is reserved for the sign of the value; positive values are preceded by a blank space.

- String values are left-justified and may cross field boundaries. String values that exceed remaining line length are always printed at the beginning of a new line. Thus, if a string is encountered in an output list that will exceed the remaining width of the current print line, a Carriage Return/Line Feed is automatically generated and the string is printed at the beginning of the next line. String values greater than 72 characters are printed on successive lines in segments no greater than 72 characters each.

- Complex values are always printed out such that both the real and imaginary parts are on the same print line. Thus, if a print line cannot accommodate the imaginary as well as real part of a complex value, a Carriage Return/Line Feed is automatically generated and both parts are printed at the beginning of the next line.

- A PRINT statement without an output list is executable and generates a blank line (regardless of the terminating delimiter, if any, of the most recently executed PRINT statement).

- The TAB function may be used as an expression in the PRINT statement list. (The TAB function positions the teletypewriter print head at an exact position specified by the user.)

Examples:

```
100 READ A, BETA, C, D, E$
200 DATA 8, 4.300, -8.000, 0.0000000001
300 DATA " THIS STRING EXCEEDS AN OUTPUT FIELD "
400 X = A + C
500 PRINT A, BETA, C, D, E$
•
•
```

When the above is executed, the results are

```
8              4.3                -8                1.E-09

THIS STRING EXCEEDS AN OUTPUT FIELD
```

If statement 500 were written as

```
500 PRINT A ;BETA ;C ;D ;E$
```

the results would be

```
8   4.3   -8   1.E-09 THIS STRING EXCEEDS AN OUTPUT FIELD
```

If statement 500 were written as

```
500 PRINT A :BETA :C :D :E$
```

the results would be:

```
8 4.3-8 1.E-09THIS STRING EXCEEDS AN OUTPUT FIELD
```

If the statements shown below were used instead of statement 500,

```
500 PRINT A
600 PRINT BETA
700 PRINT C
800 PRINT D
900 PRINT E$
```

the results would be

```
8
4.3
-8
1.E-09
THIS STRING EXCEEDS AN OUTPUT FIELD
```

TAB Function

```
TAB ( exp )
```

```
PRINT A:TAB(30) : BETA
PRINT TAB(10), "THERE IS NO WAY"
```

The TAB function may be used only in a PRINT statement. The expression specifies the exact position at which an item of information will be printed. Blank spaces are output up to the desired print location. The expression (exp) is evaluated and truncated if necessary to an integer value. It must be within the range 1-72.

The TAB function is analogous to the TAB key on a standard typewriter.

The output delimiters described for the PRINT statement precede and follow the TAB function with the following effects:

Preceding Delimiter	Effect
,	Advances carriage to the next field, then fills in blanks up to tab position specified. If the next field is past the tab position specified, the TAB function is ignored.
;	Advances the carriage forward three character positions, then fills in blanks up to the specified tab position. If the three-character forward move passes the location specified, the TAB function is ignored.
:	Advances the carriage to the specified tab position. If the carriage is past the specified tab position, the TAB function is ignored.

The TAB function is conventionally preceded by a colon delimiter.

Following Delimiter	Effect
,	The next output item is printed at the first field following the tab position specified.
;	The next output item is printed three characters after the tab position specified.
:	The next output item is printed at the tab position specified.

The TAB function is conventionally followed by a colon delimiter.

Terminal output spacing is approximated owing to the space limitations of this page.

Examples:

```
100 B,  A = 6.324
200 PRINT A:  TAB(20): B
```

The results are

```
6.324                          6.324
                   TAB 20 ---->
```

If statement 200 were written

```
200 PRINT A:  TAB(20); B
```

the results would be:

```
6.324                             6.324
                   TAB 20 + 3 ---->
```

If statement 200 were written

```
200 PRINT A:  TAB(20),  B
```

the results would be

```
6.324                             6.324
                   TAB 20 + 4 ---->
```

If statement 200 were written

```
200 PRINT A,  TAB(6): B
```

the results would be

```
6.324          6.324
         TAB 12 -->
```

Tab 6 is ignored since the sixth position is passed when the preceding comma causes the carriage to be moved to the next field.

INPUT

INPUT var_1 [, var_2, ... var_n]

```
INPUT A, B$, CVAL, IVAL, F$
INPUT RATE
```

The INPUT statement allows the user to supply data to a program directly from the terminal at execution time instead of reading from a previously created data list.

INPUT statements may be used for Immediate or Program Execution.

When the INPUT statement is executed,the system prints out the character ? at the terminal. The user should then respond by typing in appropriate values corresponding to the appearance of variable names in the program's INPUT statement. As with the READ and DATA statements, there is a sequential one-to-one correspondence between variable names and values.

If the user does not respond with a sufficient number of values from the terminal to satisfy the INPUT statement's list of variables, the system prints out the characters ?? requesting more data for subsequent items in the variable list.

The user may selectively supply data by responding to the ? request with the special character \. This causes the current unsatisfied variable to be skipped in the variable list. The value of the variable does not change if the \ option is used.

- Data input from the terminal may be numeric or string. Values are checked for type compatibility and converted if necessary according to the rules specified in Section 3. If a variable in the input list is type string, all input for that variable is accepted as type string. However, if a variable is type numeric and a string constant is supplied, an error condition is generated.

To supply data for complex variables, input two numeric constants. The first is considered the real part; the second is considered the imaginary part.

INPUT - DATA TRANSFER CONVENTIONS

An input (non-formatted) request from a symbolic file or the teletypewriter is terminated as follows:

- Numeric data is terminated by a blank, comma, or Carriage Return.
- String data with a leading single or double quote is terminated by a quote of the same type.
- String data without a leading quote is terminated by a comma or Carriage Return.

When the end of an item of data is reached, the input pointer is positioned to the next character that is not a blank, comma, or Carriage Return. Line Feed characters used for editing and writing multi-line statements are ignored and do not appear in the resultant data item.

- Data supplied from the terminal should be separated by commas or blanks and should be in the same sequence as their corresponding variable names in the INPUT statement. Since a Carriage Return signals the end of input of a single item, the Line Feed is used to continue the input of an item on the next line.
- Quotation marks are not necessary when supplying string input unless the string contains commas or blank spaces (trailing or leading).

Example:

```
05 STRING  FOM
10 INPUT A, B, C
20 A = A * B+C
30 B = B - C
40 C = C+A - B
50 INPUT XYZ$, FOM, NUM$
100 END
RUN
? 2, 4, 8
? "STRING1"
? "10#.3#"
? "5123"
16       -4          28
STRING1
10#.3#5123
```

USER-CONTROLLED FORMAT STATEMENT — PRINT IN FORM

PRINT IN FORM string: exp_1 [, exp_2, ... exp_n]

```
PRINT IN FORM "%%.%%":IVAL,QVAL,JVAL
PRINT IN FORM A$: AVAL,BVAL,CVAL
PRINT IN FORM "**** BBBB $$$$.$$ BBBB %%.%%": IVAL, AMT, RATE
```

By defining output fields, the PRINT IN FORM statement enables the user to specify the exact format of program output. An output field is defined by a form definition string which serves as a model for the items in the PRINT IN FORM statement list. An output field may be defined at any point in a program and its (string) variable name used in the PRINT IN FORM statement, or the form definition string may be directly specified in the statement enclosed in quotation marks.

PRINT statements may be used for Immediate or Program Execution.

The special format characters described below are used to define the type and format of output fields. Field length is determined by the number of contiguous characters that constitute the field definition (including decimal points). Multiple fields can be defined within the form definition string by delimiting each field with one or more blanks. These blanks do not appear as part of the output image.

Blank spaces may not be embedded within field definitions.

Format Character	Output Field Format
multiple or single %'s	Data is right-justified with leading blank fill within a field of the length specified by the number of characters in the field definition string. If a decimal point is embedded in the field definition string, data is rounded to the specified number of fractional digits. At least one % is required before the decimal point, if any, and one additional % is required for negative values. Up to 16 significant digits may be output.

Examples:

Data	Field Definition	Output Image
387647	"%%%%%%%%"	387647
39847	"%.%%%"	3.985
-9.3	"%%%.%%%%"	- 9.3000
DATA	"%%%%%%"	DATA

multiple #'s

Data is expressed in scientific (E) notation within a field of the length specified by the number of characters in the field definition string. This format requires at least seven characters in the field definition string including a # for the sign of the value, if negative, a # for the decimal point, a # for E, a # for the sign of the exponent, and two #'s for the two-digit exponent. Seven or more #'s without a decimal point imply a decimal point preceding the first significant digit. An embedded decimal point specifies the position of the decimal point when the item is printed. The exponent value is altered accordingly. Numbers are rounded to the specified number of decimal digits. Up to 16 significant digits may be output.

Examples:

Data	Field Definition	Output Image
-3.E06	"#######"	-.3E+06
4234320	"#.#######"	4.234E+06
600	"#######"	.6E+03
800	"##.#####"	80.E+01

single #

If a single # is specified as the output field definition, data is output according to standard PRINT statement conventions.

multiple $'s

Data is output as for a % field except that a $ is printed as the first significant character.

Examples:

Data	Field Definition	Output Image
23493	"$$$$$$.$$"	$ 234.93
23493	"$$$$.$$"	$234.93

multiple *'s

Data is output as for a % field except that leading blanks are filled with * characters. The field definition must include an extra * character for at least one preceding *.

Examples:

Data	Field Definition	Output Image
234	"*********"	******234
23.454	"*****.**"	***23.45

multiple or single B's

Blanks in the output image may be specified by inserting the letter B for each space between field definition strings.

Examples:

Data	Field Definition	Output Image
23.4,6	"$$$.$$ BB *****"	$23.40 ****6
45, 23, 4	"%% BB %% BB %"	45 23 4

A blank space must precede and follow B fields and Pxx specifications within a field definition string.

Pxx

Positions the carriage at print position xx (1 to 72) of the print line. If the print position is already past location xx of the teletypewriter print line, Pxx is ignored.

Example:

Data	Field Definition	Output Image
2,3	"%.%% P6 %.%%"	2.00 3.00

P6

- For each of the above field format types, allowance in field length must be made for the minus sign of negative numbers.

- Variables in the output list are associated with the field definitions in the string on a one-to-one basis in order of appearance.

- Any of the special characters %, *, $, # can be used to output a string. The string is printed left-justified. If the string exceeds the number of characters defined for the field, an error message is issued. Blanks are extended to the end of the field if the string is smaller than the field specified.

- Literal text may be included in the field definition by inserting a string enclosed by single quotes within a field definition bounded by double quotes, and vice versa.

- Standard PRINT statement output list delimiters do not apply when using the PRINT IN FORM statement, as the field definition specifies variable positioning.

- If there are more output items than field definitions, the form definition strings will be used repeatedly (from the beginning) until the output list is exhausted.

- When the variable list is completed, a Carriage Return and Line Feed are generated.

- A Carriage Return and Line Feed can be generated between output items by inserting a /. Consecutive /'s generate blank lines between output items.

  Examples:

  ```
  5 A,B = 632
  10 A1$ = "%%% BBB %%%"
  20 A2$ = "%%%       %%%"
  30 PRINT IN FORM A1$: A,B
  40 PRINT IN FORM A2$: A,B
  50 RUN
  632    632
  632632
  100 COST = 36.30
  110 PRINT IN FORM " 'COST=' $$$.$$": COST
  120 END
  RUN
  COST = $36.30
  ```

FORMAT REPLICATION

Field definition characters and entire field definitions can be repeated by prefacing the item to be repeated by a numeric quantity (repeat count). Parentheses are necessary for clarification when multiple fields are to be repeated.

- Parentheses can be nested only to a level of 5.
- Repeat counts may be used with all of the field definition characters except P.

Examples:

```
10 A$ = "%%% BB %%"
20 B$ = "3% 2B 2%"
```

Statements 10 and 20 are equivalent field definitions.

```
30 X$ = "$$$.$$ B $$$.$$ B"
40 Y$ = "2(3$.2$ B)"
```

X$ and Y$ are equivalent field definitions.

7 MATRIX STATEMENTS

TENET BASIC provides a special set of statements to facilitate matrix operations. All statements which control matrix operations begin with the word MAT and allow entire arrays as arguments. (Throughout this manual the terms matrix and array have identical meanings.)

All array variables must be explicitly declared in either a data type or DIM statement before they appear in a matrix statement. There are two exceptions: both the MAT READ and MAT INPUT statements allow subscript specification which either explicitly declares an array for the first time in a program, or dynamically redimensions the array at execution time.

The limitations on dynamically redimensioning matrices are described on p. 4-5.

When an array is dynamically redimensioned, the contents of the area assigned to the array are neither destroyed or altered. Redimensioning affects only the arrangement of the space assigned to the array when it is used. For example:

```
    DIM A(10, 10)
    MAT READ A
```

This combination of statements assigns the same information to A as

```
    MAT READ A(20, 5)
```

In the first version, A is a 10 x 10 matrix; in the second, A is a 20 x 5 matrix.

Most matrix statements can be used with arrays of any number of dimensions. However, operations involving more than one matrix require compatibility as to number of dimensions and subscript range for each dimension for each of the arrays used in the statement.

MAT ASSIGNMENT STATEMENT

$$\text{MAT array}_1 = \text{array}_2$$

MAT A = BETA

MAT IVAL = JVAL

MAT statements may be used for Immediate or Program Execution.

The rules for mixed data type assignments are described on p.4-7.

The matrix assignment statement copies the contents of the array variable on the right-hand side of the equal sign into the array variable on the left. Both matrices must currently have the same number of dimensions and subscript range for each dimension. If the arrays are not of the same type, data type conversion is performed.

Examples:

```
    10 REAL AAA(5,5)
    20 DOUBLE BBB(5,5)
    30 MAT AAA = BBB
```

Statement 30 copies the contents of the two-dimensional array BBB into AAA. Since AAA is type REAL, the contents of BBB are converted from double precision to single precision and then placed in AAA.

MAT ADDITION STATEMENT

MAT array$_1$ = array$_2$ + array$_3$

MAT A = B + C

MAT MAT1 = MAT2 + MAT3

The matrix addition statement calculates, element by element, the sum of two arrays and stores the result in the array on the left-hand side of the equal sign. All arrays must be the same data type and have the same number of dimensions and subscript range for each dimension.

MAT statements may be used for Immediate or Program Execution.

- Only one arithmetic operation is allowed in a MAT statement.
- The same array name may appear on both sides of the equal sign.

Examples:

```
10 COMPLEX VAR1(4,4), VAR2(4,4), ANS1(4,4),
   ANS2(4,4)
.
.
.
60 MAT VAR1= VAR1 + VAR2
70 MAT ANS1 = VAR1 +ANS2
```

MAT SUBTRACTION STATEMENT

MAT array_1 = array_2 - array_3

MAT A = A - BETA
MAT IVAL = ANS - RATE

MAT statements may be used for Immediate or Program Execution.

The matrix subtraction statement calculates, element by element, the difference of two arrays and stores the result in the array on the left-hand side of the equal sign. All arrays must be the same data type and have the same number of dimensions and subscript range for each dimension.

- Only one arithmetic operation is allowed in a MAT statement.
- The same array name may appear on both sides of the equal sign.

Example:

```
10 DIM SAM(5,6), BETA(5,6), LOT(5,6)
20 MAT BETA = SAM - LOT
```

MAT MULTIPLICATION STATEMENT

MAT array_1 = array_2 * array_3

MAT A = A * BETA
MAT RATE = INS * IVAL

The matrix multiplication statement calculates the matrix product of two arrays and stores the result in the array on the left-hand side of the equal sign. All arrays must be the same data type and currently have two dimensions. The subscript range for each dimension must correspond as follows:

MAT statements may be used for Immediate or Program Execution.

$\text{var}_1(i, j) = \text{var}_2(i, k) * \text{var}_3(k, j)$

where identical subscripts indicate identical subscript ranges.

- Only two-dimensional, numeric arrays may be used in this statement.
- Only one arithmetic operation is allowed in a MAT statement.
- The same array name may not appear on both sides of the equal sign.

Example:

```
20 INTEGER MAT1(10,10), LOST(10,1), TOTE(1,10)
30 MAT MAT1 = LOST * TOTE
```

MAT SCALAR MULTIPLICATION STATEMENT

```
MAT array1 = (exp) * array2
```

MAT A = (3) * BETA

MAT XVAL = (SQRT(BETA)) * YVAL

MAT statements may be used for Immediate or Program Execution.

The rules for mixed data type conversion are described on p. 4-7.

The scalar multiplication MAT statement multiplies each element of an array by the value of the expression and stores the result into the array on the left-hand side of the equal sign. Both arrays must have the same number of dimensions and subscript range for each dimension. If the arrays are not of the same type, data-type conversion is performed.

- The expression must be enclosed in parentheses.
- Only one matrix operation is allowed in the statement.
- The same array name may appear on both sides of the equal sign.

Example:

```
5 DIM A(5,5), B(5,5)
10 MAT A = (1) * B
15 COMPLEX IVAL (5,5), JVAL(5,5), KVAL(5,5)
20 MAT IVAL = (2) * IVAL
30 MAT IVAL = (1/4) *JVAL
40 MAT KVAL = (SQR(2.35)) * IVAL
```

Statement 10 is equivalent to

```
MAT A = B
```

MAT INVERSION STATEMENT

MAT array_1 = INV (array_2)

MAT A = INV(BETA)

The matrix inversion statement stores the (matrix) inverse of an array on the left-hand side of the equal sign. Both arrays must have the same number of elements and dimensions and identical structures.

MAT statements may be used for Immediate or Program Execution.

- The same array name may appear on both sides of the equal sign.
- The matrix or array to be inverted is considered ill-conditioned, i.e., difficult to invert accurately, if the pivot element exceeds the smallest element by more than the value of EPS (1 x 10^{-6}) during the inversion process. This condition produces the message

  NEARLY SINGULAR MATRIX
- Mixed data type assignments are not allowed with this statement.

Matrices are inverted using the Gauss-Jordan method with matrix pivoting.

EPS is a predefined value which may be redefined by the user.

MAT TRANSPOSITION STATEMENT

MAT array_1 = TRN (array_2)

MAT BETA = TRN (PHI)

MAT statements may be used for Immediate or Program Execution.

The matrix transposition statement stores the transpose of an array into the array on the left-hand side of the equal sign. Both arrays must be of the same data type and have the same number of dimensions and subscript range for each dimension. However, the subscript range for each dimension of one array must be in reverse order of the ranges for the other array, i.e., array_1 (i, j) vs. array_2 (j, i).

- Only two-dimensional, numeric arrays may be used in this statement.
- The same array name may not appear on both sides of the equal sign.

Example:

```
10 DIM IVAL (5,2), JVAL (2,5)
20 MAT IVAL = TRN(JVAL)
IVAL            JVAL

1 6             1 2 3 4 5
2 7             6 7 8 9 0
3 8
4 9
5 0
```

MAT INITIALIZATION STATEMENT

```
MAT array = (exp)
```

```
MAT BETA = (4*IVAL)
MAT ZYG = (0)
```

The matrix initialization statement sets all elements of the array specified to the value of the expression onthe right-hand side of the equal sign.

MAT statements may be used for Immediate or Program Execution.

- The array must have been previously declared in the program before its appearance in this statement.
- The data type of the evaluated expression is converted, if necessary, to that of the array.

The rules for mixed data type assignments are described on p.4-7.

Example:

```
40 INTEGER BYO(2,2)
.
.
80 XVAL = 4 * 1.3
.
.
100 MAT BYO = (XVAL)
.
.
```

When statement 100 is executed each element of the array BYO is set to the value of XVAL. Although the computed value of XVAL is 5.2, each element of the array is given a value of 5 since the array was explicitly declared data type integer.

MAT IDENTITY STATEMENT

```
MAT var = IDN
```

MAT SAM = IDN

MAT RAGE = IDN

MAT X = IDN

MAT statements may be used for Immediate or Program Execution.

If the IDN function is used in a MAT statement, an identity matrix is created (containing zero values with a diagonal of 1's). The array specified in this statement must be a square, two-dimentional array.

- Only two-dimensional, numeric arrays may be used in this statement.

Examples:

```
10 DIM VAT(7,7)
20 MAT VAT = IDN produces VAT as follows:
1 0 0 0 0 0 0
0 1 0 0 0 0 0
0 0 1 0 0 0 0
0 0 0 1 0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 1 0
0 0 0 0 0 0 1
```

MAT READ STATEMENT

MAT READ array_1 [, array_2, ... array_n]

```
MAT READ IVAL, JVAL, KVAL
MAT READ S$M(8), BETA(5,6), RATE(2,2,2)
```

The MAT READ statement enables the user to assign values for entire arrays from the data list generated by the program's DATA statements. The user may redimension an array when the MAT READ statement is executed by specifying subscript values in parentheses following the name of the array. If no subscripts are specified, the array must have been previously declared, and its most recent declared dimensions are used.

MAT statements may be used for Immediate or Program Execution.

DATA statements are described on p.4-19.

The limitations on dynamically redimensioning arrays are described on p.4-5.

- Arrays are supplied values with the most rapidly changing (rightmost) subscript supplied first; e.g., the nine-element array X(3,3) would be supplied values as follows: X(1,1), X(1,2), X(1,3), X(2,1), X(2,2), X(2,3), X(3,1), X(3,2), X(3,3).
- As data list items are read, the data list point is advanced to the next item in the list.
- When the end of the data list is reached, an attempt to read a value for a variable causes an error message to be issued.
- Complex or double complex values are read from the internal data list in two parts: first the real part, then the imaginary part.

Example:

```
10 DIM A(5), B(20), C(3,3)
20 DATA 1,2,3,4,5,6,7,8,9
30 MAT READ A, B(3)
40 RESTORE
50 MAT READ C
```

The array A is filled with the values 1,2,3,4,5. The array B, after being redimensioned to three elements, is filled with the values 6,7,8. The two-dimensional array C is read by rows, resulting in the following matrix:

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

MAT INPUT STATEMENT

MAT INPUT array_1 [, array_2,... array_n]

```
MAT INPUT BETA, PHU, ZOO
MAT INPUT A(4,5), B(7,8), F$(21), SAM(J,4,5)
```

MAT statements may be used for Immediate or Program Execution.

The INPUT statement is discussed on p.6-7.

The MAT INPUT statement is identical to the MAT READ statement except that data is supplied for the elements of an array by the user from the terminal at execution time (just as for the terminal INPUT statement).

When the MAT INPUT statement is executed, the system prints out the character ? at the terminal. The user should respond by typing in appropriate values for array elements in an order corresponding to their position in the array, and corresponding to the appearance of array names in the MAT INPUT statement.

If the user does not respond with a sufficient number of values from the terminal to satisfy the MAT INPUT statement's list of arrays, the system prints out ?? requesting more data for subsequent items in the variable list.

The user may selectively supply data by responding to the ? request with the special character \ . This causes the current unsatisfied subscripted variable to be skipped in the variable list. The value of the variable does not change if the \ option is used.

- Data input from the terminal may be numeric or string. Values are checked for data type compatibility and converted if necessary according to the rules specified in section 3. If a variable in the input list is type string, all input to that array is accepted as type string. However, if an array is type numeric and a string constant is supplied, an error condition is generated.

- The Input-Data Transfer Conventions specified in section 6 apply to supplying values in response to the request generated by a MAT INPUT statement.

Example:

```
10 DIM X(2), Y(2,3), ZED(1,4)
.
.
80 MAT INPUT X, Y, ZED (1,2)
.
.
190 END
```

When statement 80 is executed, the system requests that the user type in the appropriate values for the MAT INPUT array list:

```
? 1,2,4,5,6,7,8,9,3.456,5.678
```

Internally, the arrays are filled as follows:

```
X   = 1 2
Y   = 4 5 6
      7 8 9
ZED = 3.456  5.678
```

MAT PRINT STATEMENT

MAT PRINT array$_1$ [$\left\{\begin{matrix} , \\ ; \\ : \end{matrix}\right\}$ array$_2$ $\left\{\begin{matrix} , \\ ; \\ : \end{matrix}\right\}$ . . . array$_n$ [$\left\{\begin{matrix} , \\ ; \\ : \end{matrix}\right\}$]]

```
MAT PRINT BETA, RATE, YEAR,
MAT PRINT Z : Y ; X
MAT PRINT ALPH
```

MAT statements may be used for Immediate or Program Execution.

The PRINT statement is described on p.6-2.

The MAT PRINT statement is similar to the PRINT statement for terminal output except it prints out entire arrays at the terminal. MAT PRINT statement delimiters affect <u>preceding</u> array variables as follows:

Delimiter	Format
,	The elements of each row are printed in fields of 12, allowing 6 elements per print line.
;	The elements of each row are printed with three spaces between each element.
:	The elements of each row are concatenated. (Negative numbers and strings are truly concatenated, but a blank space is reserved before each positive number.)

The final delimiter is optional. If the delimiter is omitted, a comma is assumed.

- The rightmost subscript is the most rapidly changing subscript and it determines the number of items of data to be printed across each print line. Each row of an array is printed separately, with double spacing between array rows. Each row of the array will begin at a new line. Thus, a one-dimensional array is printed across one or more lines, and the subscript notation (x) is equivalent to (1, x).

Examples:

```
5 DIM A (2,3)
10 DIM BETA (9)
20 DIM ZOO (4)
30 DIM IVAL (8)
35 MAT A = (4)
40 DATA 1,2,3,4,5,6,7,8,9
50 DATA 11,22,33,44
60 DATA 21,22,23,24,25,26,27,28
80 MAT READ BETA, ZOO, IVAL
90 MAT PRINT A, BETA, ZOO, IVAL
RUN
4  4  4
4  4  4
1  2  3  4  5  6
7  8  9
11 22 33 44
21 22 23 24 25 26
27 28
```

The array BETA was printed as a 9-element row, ZOO as a 4-element row, and IVAL is an 8-element row. As BETA and IVAL contained more elements than would fit on a print line using the comma delimiter their row output is continued onto another line. However, if statement 90 specified concatenated output, the following would be printed:

```
4  4  4  4
4  4  4  4
1  2  3  4  5  6  7  8  9
11 22 33 44
21 22 23 24 25 26 27 28
```

If statement 30 declared IVAL as IVAL(4,2,1), IVAL would be printed as

```
21
22
23
24
25
26
27
28
```

```
10 DIM ACE(3,7)
20 MAT ACE = (5)
30 MAT PRINT ACE
RUN

5  5  5  5  5  5
5
5  5  5  5  5  5
5
5  5  5  5  5  5
```

The occurrence of a single item of data at the beginning of alternate output lines is caused by the seventh element of each row which will not fit on the print line when the comma delimiter is used.

MAT PRINT IN FORM STATEMENT

MAT PRINT IN FORM string: array$_1$ [, array$_2$, . . . array$_n$]

```
MAT PRINT IN FORM   A$: IVAL, JVAL
MAT PRINT IN FORM "***** BBB $$$$.$$" : ANS, AMNT
```

MAT statements may be used for Immediate or Program Execution.

The PRINT IN FORM statement is discussed on p. 6-9.

The MAT PRINT IN FORM statement specifies the exact format in which elements of an array are to be printed at the terminal. This statement is identical to the PRINT IN FORM statement for standard terminal output, except that entire arrays are specified in the statement's variable list.

The special characters used to define output fields for arrays are the same as for the PRINT IN FORM statement. However, since the MAT PRINT IN FORM statement prints multiple items for each variable name, it is recommended that the slash character (/) is inserted within the field definition string to generate a Carriage Return/Line Feed at the end of each row.

Example:

```
10 DATA 6,34.56,4,34.95,5,67.45,12,23.54
20 DATA2,4,5,6,7,8,4,5,6,9
30 MAT READ A(2,4), B(5,2)
40 MAT PRINT IN FORM "***** BB $$$.$$ /" : A,B
RUN
****6   $34.56
****4   $34.95
****5   $67.45
***12   $23.45
****2   $ 4.00
****5   $ 6.00
****7   $ 8.00
****4   $ 6.00
****6   $ 9.00
```

8. FILE STATEMENTS

INTRODUCTION

A file is a structured set of data maintained on a mass storage device external to the central computer memory. The structure and content of files vary to facilitate file usage under different circumstances.

A file is addressed by a unique file name. A directory of all files belonging to a particular user is maintained by the TENET 210 System. Though the total number of files belonging to a particular user is essentially unlimited, the BASIC user may access (read and/or write) up to eight files at one time. Each active file may contain up to eight million characters.

FILE CONTENT

Data stored in a file may be symbolic or binary. The contents of a symbolic file appear identical to teletype input and output (including the Carriage Return and Line Feed characters). Symbolic input is always compared with input variable requests for type compatibility. Unformatted symbolic files have the same input conventions as standard teletype input.

Data stored in binary files appears in internal machine format. Binary files are more compact and efficiently stored than symbolic files. They must be read in the same manner as they are written and any output to a binary file will have the following data type/number of words correspondence:

Data Type	Number of Words
Integer	1
Real	1
Double	2
Complex	2
Double Complex	4
String (n characters)	$1 + \text{INT}\left[\frac{n+3}{4}\right]$

String values in binary files use an extra word to specify string length (in characters).

Input variable requests from binary files cannot be checked for type compatibility. The data type/number of words correspondence as listed above is used to determine the amount of information transferred relative to the data type requested in binary files.

Binary files, as such, cannot be listed on the teletype.

FILE STRUCTURE

A file has an internal organization specified by the user according to the way the file will be used. The structure of a file is fixed when it is first created and cannot be changed.

Sequential Access Files

A sequential file is a single, continuous set of symbolic or binary information. Information is arranged on the file sequentially starting at the beginning of the file. A write-to or read-from operation on a binary file uses n number of words where n is based on the variable type to be written or read. For a symbolic file, a read or write operation reads or writes the data exactly as it would appear on the teletype using the PRINT or INPUT positioning conventions. A sequential file can only be read or written sequentially from beginning to end; thus individual items in a sequential file cannot be accessed randomly. File breakdown by records is meaningless for sequential files.

Random Access Files

A random file is a set of independently addressable subsets of data called records. By specifying the number of a record, the user can access an arbitrary location in a large file. A record may be accessed and/or altered without affecting the rest of the random file. Either fixed or variable-length records may be used in a random file. Fixed-length records allow quicker access to any part of the file, whereas variable-length records are more flexible to use though slower to access. On random files with fixed-length records, the length of the record is specified by the user (in number of characters for symbolic files and in number of words for binary files). A read or write operation on a fixed-length record always starts at the beginning of a record. If the operation does not reference all the data available in the record, the remainder of the record is ignored. Attempting to write a record longer than the length specified in an OPEN statement is illegal. Random fixed-length records can be read or written in any order. Writing random fixed-length records causes the allocation of disc space for all records from one through the record number specified. Reading a record which has been allocated disc space but not explicitly written will not necessarily result in an error message.

A variable-length random record can be any length. Its size is the total length of the data specified in the WRITE statement that created the record. A write of a variable-length record, followed by another write of the same record, destroys the previous data.

When writing (i.e., replacing) a record which already exists, whether fixed or variable-length, the new record size must not exceed that of the old. Variable-length records must originally be written in sequence. However, once the file has been created, variable-length records may be rewritten in any order.

OPEN STATEMENT

$$\boxed{\text{OPEN}\left\{\begin{array}{l}\text{"fname"}\\ \text{"SCR"}\end{array}\right\},\text{exp}_1\left[\begin{array}{l},\text{BINARY}\\ ,\text{SYMBOLIC}\end{array}\right]\left[\begin{array}{l},\text{INPUT}\\ ,\text{OUTPUT}\\ ,\text{IO}\end{array}\right]\left[\begin{array}{l},\text{RANDOM(exp}_2\text{)}\\ ,\text{RANDOM}\\ ,\text{SEQUENTIAL}\end{array}\right]\left[\begin{array}{l},\text{NEW}\\ ,\text{OLD}\end{array}\right]}$$

```
OPEN "MYFILE",7
OPEN "XFILE",4,INPUT,RANDOM(100),OLD
OPEN "ADFILE",6, SYMBOLIC,IO,OLD, RANDOM
OPEN "BIFILE",8,BINARY,IO,RANDOM,NEW
OPEN "SCR",3,SYMBOLIC,OUTPUT,SEQUENTIAL
```

OPEN may be used for Immediate or Program Execution.

Before a file can be used in a program, it must be activated by the OPEN statement. Opening a file tells the system that the user is about to perform some operations on the file and automatically positions the file at its beginning-of-information ready to be read or written. The required parameters for the OPEN statement are the file name and the file number. The default values for the other, optional parameters are SYMBOLIC, OUTPUT, SEQUENTIAL, and NEW.

Files designated INPUT have read-only permission; i.e., the user may only read information from the file. A file remains read-only until the file is closed and reopened with another permission. A file opened for OUTPUT is a write-only file, i.e., the user may only write information onto the file. If a file opened for OUTPUT does not exist in the user's directory, a new file is created. A file designated IO can be read or written. INPUT or OUTPUT permission is generally more efficient than IO. Since a random IO file allows writing on any place in the file, it must be used cautiously.

For information about shared files and passwords see the EXECUTIVE Users Manual.

The user may access files in other users' directories if the files have been designated as sharable and the user has the correct file name, account number, and user name.

- If the file to be opened is owned by the user, the name consists of a string of 1 to 8 alphanumeric characters, including the symbols % and $, enclosed in quotation marks. The first character must be alphabetic, %, or $.

- If the file to be opened is not owned by the user but shared, the name of the file must be of the form:

  file name = " a; un; fn "

  where

  a = account number (0-511) of file's creator.

  un = name of file's creator (1-8 alphanumeric characters, including % and $; the first character must be alphabetic, %, or $.)

  fn = name of file (1-8 alphanumeric characters, including % and $; the first character must be alphabetic, %, or $.)

  Example: "63;JOEDOE;$FUND$"

- If SCR is used as a file name, a temporary (scratch) file is created which will cease to exist after the file number is closed or the user leaves the BASIC subsystem. SCR can be used with more than one file number simultaneously, each being a different temporary file.

SCR files are NEW only.

- exp_1 is a value from 0 to 8 identifying one of the active files available to the user. File 0 is permanently assigned as the terminal.

Only eight disc files may be active (accessed) at one time.

- OLD indicates that the file to be opened already exists in the user's directory. NEW indicates that the file is being created.

NEW can be used with OUTPUT or IO files only.

- INPUT permission is allowed on OLD files only; OUTPUT and IO are permitted on both OLD and NEW.

- SYMBOLIC and BINARY indicate the type of information contained in the file.

- RANDOM and SEQUENTIAL indicate the structure of the file. (exp_2) is required if a file of fixed-length records is desired and indicates record length (in characters for symbolic files, and in words for binary files).

- SEQUENTIAL files cannot be opened for both read and write permission (IO).

CLOSE STATEMENT

CLOSE exp_1 [$,\text{exp}_2$, ... exp_n]

```
CLOSE 1,2,3,4,5,6,7,8
CLOSE 5
CLOSE 2,4,5
```

CLOSE may be used for Immediate or Program Execution.

After the user has completed work on a file, he may deactivate it by using the CLOSE command. Once a file is closed the number of the file may be assigned to another file. The CLOSE statement does not destroy or delete the information in the deactivated file unless it is a temporary (SCR) file. (CLOSE requests on file 0 (terminal) are ignored.)

A file opened for output may be closed and reopened as an input file and vice versa. A symbolic file, however, cannot be reopened as a binary file. Random fixed, random variable, and sequential files must always be referenced in the manner specified when the file was originally created.

- exp_i indicate file numbers currently active, which are to be closed.

Example:

```
 10 OPEN "MYFILE", 3, INPUT, RANDOM, OLD
 60 OPEN "ADFILE", 5, OUTPUT, SEQUENTIAL, NEW
 .
 .
100 CLOSE 3, 5
200 OPEN "ADFILE", 5, INPUT, OLD
```

RESTORE FILE STATEMENT

RESTORE FILE exp_1 [,exp_2, ... exp_n]

RESTORE FILE 6

The RESTORE FILE statement enables the user to position a sequential file at its beginning-of-information. If the file is opened with read-only permission, the file may be reread only. If the file is opened with write-only permission, it may be rewritten only. I/O files may either be written, restored, and read, or read, restored, and written (rewritten).

RESTORE FILE may be used for Immediate or Program Execution.

- exp_i are currently active file numbers

Example:

```
10 OPEN "MYFILE", 5, SYMBOLIC, INPUT, SEQUENTIAL, OLD
20 INPUT FROM 5 : A,B,C,D,E,F
25 A,C,D,F = A*D+C+D/E+F
30 RESTORE FILE
40 INPUT FROM 5 : A,B,C,D,E,F
•
•
```

APPEND FILE STATEMENT

APPEND FILE exp_1 [,exp_2, ... exp_n]

APPEND FILE 7

APPEND FILE may be used for Immediate or Program Execution.

The APPEND FILE statement enables the user to write additional information onto a sequential file. The file is positioned past its end-of-information ready for the next set of information.

- The APPEND FILE statement may be used only on sequential output files.
- exp_i are currently active file numbers.

Example:

```
10 OPEN "MYFILE", 4, SYMBOLIC, OUTPUT, SEQUENTIAL, OLD
20 APPEND FILE 4
30 WRITE ON 4:A, B
```

After MYFILE is opened with write-only permission, it is positioned past the information already existing on the file. The next statement appends information to the end of the file.

ERASE FILE STATEMENT

ERASE FILE exp_1 FROM exp_2 TO exp_3

ERASE FILE 6 FROM 4 TO 6

The user may selectively delete records from a random access file by using the ERASE statement. The actual number of records in the random access file is not altered unless the exp_3 is the last record in the file. Deleted records are filled with null characters regardless of the original data type of the file contents.

ERASE FILE may be used for Immediate or Program Execution.

- exp_1 is a currently active file number.
- exp_2 and exp_3 are record numbers.

Example:

```
10 OPEN "AFILE", 3, SYMBOLIC, OUTPUT, SEQUENTIAL, OLD
20 OPEN "BFILE", 4, SYMBOLIC, INPUT, RANDOM, OLD
30 INPUT FROM 4 AT 3 : IVAL, KVAL, JVAL
40 INPUT FROM 4 AT 4 : LVAL, MVAL, NVAL
50 INPUT FROM 4 AT 5 : OVAL, QVAL, RVAL
60 APPEND FILE 3
70 WRITE ON 3 : IVAL, KVAL, JVAL, LVAL, MVAL, NVAL, (LF)
OVAL, QVAL, RVAL
80 ERASE FILE 4 FROM 3 TO 5
```

The random access file BFILE is opened for read access only. After values are read from records 3, 4 and 5 of BFILE, they are written onto AFILE. Records 3, 4 and 5 are then deleted from BFILE.

ON ENDFILE STATEMENT

```
ON ENDFILE (exp) GOTO line no.
```

ON ENDFILE (4) GOTO 300
ON ENDFILE (J) GOTO 150

ON ENDFILE may be used for Immediate or Program Execution.

The ON ENDFILE statement specifies a line number to which program control will be transferred when the end of information on the file specified is reached by any subsequent read of the file.

The ON ENDFILE statement must be executed after the file is opened, but before it is used. Closing a file number and reopening it requires another ON ENDFILE statement. If multiple ON ENDFILE statements appear for the same file number, the most recent is used.

- (exp) is a currently active file number.
- An error message is issued if an end of file is encountered and there is no ON ENDFILE statement for the file number.
- The ON ENDFILE statement is executable and thus, to be effective, must be executed each time the file number it references is opened.

Example:

```
 50 OPEN "MY FILE", 3, INPUT, NEW
 60 ON ENDFILE (3) GOTO 100
 70 INPUT FROM 3 : A
 .
 .
100 CLOSE 3
 .
 .
```

ON ENDREC STATEMENT

ON ENDREC (exp) GOTO line no.

ON ENDREC(3)GOTO 100
ON ENDREC(1) GOTO 40

The ON ENDREC statement is identical to the ON ENDFILE statement except that it applies to the end of record condition for random access files. If an attempt is made to read past the end of a record, or write past the end of a fixed-length record, the ON ENDREC statement causes program control to be transferred to the line number specified.

ON ENDREC may be used for Immediate or Program Execution.

- (exp) is a currently active file number.
- An error message is issued if an end of record is encountered prematurely and there is no ON ENDREC statement for the file number.
- The ON ENDREC statement is executable and thus, to be effective, must be executed each time the file it references is opened.

INPUT (SEQUENTIAL FILE) STATEMENT

[MAT] INPUT FROM exp : var_1 [, var_2, ... var_n]

```
MAT INPUT FROM 6: A(3,3),B,C,D$,E$
INPUT FROM 6: IVAL, JVAL, KVAL(3,3),KVAL(3,4)
```

INPUT statements may be used for Immediate or Program Execution.

The IN FORM option may be used for symbolic file input/output.

The INPUT FROM file statement is used for sequential files and causes information from the specified file to be supplied for the variables in the list. This statement may be used for symbolic or binary files. Reading values from a file is comparable to reading values from the internal data set created by a program's DATA statements. The variable list follows the same conventions as terminal input/output.

The MAT READ statement is discussed on p. 7-11.

- exp is a currently active file number.
- Entire arrays may be read from a file by specifying MAT at the beginning of the statement and specifying array names in the variable list as for the MAT READ statement.

Example:

```
10 OPEN "MY FILE", 2, SEQUENTIAL, INPUT, OLD
 .
 .
70 READ FROM 2: RATE, TIME, IVAL, KVAL
 .
 .
80 PRINT RATE, TIME, IVAL, KVAL
 .
 .
```

INPUT (RANDOM FILE) STATEMENT

[MAT] INPUT FROM exp_1 AT exp_2 : var_1 [, var_2, ... var_n]

```
INPUT FROM 6 AT 2: A,B,C,D,E$,T(2,3),X
MAT INPUT FROM 8 AT 3: IVAL, KVAL, LVAL (9)
```

The INPUT FROM ... AT statement is used for random files and causes information from the file number specified to be supplied for the variables in the list. A record number must always be specified following the AT.

INPUT statements may be used for Immediate or Program Execution.

Reading values from a file is comparable to reading values from the internal data set created by the program's DATA statements. The variable list follows the same conventions as terminal input/output.

The IN FORM option may be used for symbolic file input/output.

- This statement may be used for symbolic or binary files.
- exp_1 is a currently active file number; exp_2 is a record number of of the random access file.
- Entire arrays may be read from a file by specifying MAT at the beginning of the statement and specifying array names in the variable list, as is done for the MAT READ statement.

The MAT READ statement is discussed on p.7-11.

Example:

```
10 OPEN "MY FILE", 2, RANDOM, INPUT, OLD
.
.
70 INPUT FROM 2 AT 6: RATE, TIME, IVAL, KVAL
.
.
140 PRINT RATE, TIME, IVAL, KVAL
.
.
```

INPUT IN FORM STATEMENT

[MAT] INPUT FROM exp_1 [AT exp_2] IN FORM string : var_1 [, var_2, ... var_n]

```
INPUT FROM 6 IN FORM FF$: IVAL, KVAL, JVAL, S$E$
MAT INPUT IN FORM "######.## BB" FROM 2 AT 5: A,B,C,D
INPUT IN FORM H$ FROM 6 AT 1: FAT$
```

INPUT statements may be used for Immediate or Program Execution.

The INPUT IN FORM statement must be used to read data from a file previously written using the PRINT IN FORM statement. The field definition strings specified in this statement should be identical to those used when the items in the variable lists were originally written onto the file. Thus the values for the variables may be read accurately by skipping over information such as fields of *'s and $'s which may occur between items of data on the file.

The PRINT IN FORM statement is discussed on p.6-9.

- The special field definition characters for the INPUT IN FORM statement are identical to those of the PRINT IN FORM statement.

Example:

```
10 OPEN "XFILE", 4, OLD, INPUT
20 OPEN "YFILE", 7, NEW
.
.
80 INPUT FROM 4 IN FORM "%%%%% BB": A,B,C,D,E,F,G
.
.
100 PRINT IN FORM "***** BB" ON 7: A,B,C,D,E,F,G
.
.
```

Statement 80 inputs a set of variables from XFILE in the format in which they were written onto the file. The same variables are written onto the file YFILE in a different format. Any subsequent reads of YFILE will require the same field definition as specified in line 100.

PRINT (SEQUENTIAL FILE) STATEMENT

$$[\ \text{MAT}\ ]\ \ \text{PRINT ON exp : var}_1\ [\ \left\{\begin{matrix} , \\ ; \\ : \end{matrix}\right\}\ \text{var}_2\ \left\{\begin{matrix} , \\ ; \\ : \end{matrix}\right\}\ \ldots\ \text{var}_n\ [\ \left\{\begin{matrix} , \\ ; \\ : \end{matrix}\right\}\ ]\ ]$$

```
PRINT ON 8: E$
PRINT ON 6: A:B:C:D:E;F,
PRINT ON 7: IVAL: JVAL; FVAL, MVAL:
```

The PRINT ON statement is used with symbolic or binary sequential files and causes information specified by the variable list to be written on the file named. This statement is identical to terminal PRINT statements except that values are written on a file and not printed at the terminal. The variable list follows the same conventions as terminal input/output.

PRINT statements may be used for Immediate or Program Execution.

The IN FORM option for input and output may be used for symbolic file input/output.

- The delimiters for an output command to a binary sequential file do not have meaning since the binary data has a specified word size/ data type association.
- The delimiters for a symbolic sequential file introduce spacing and a Carriage Return as with normal terminal output. A sequential file may be written without any Carriage Returns, but it would be impossible to list such a file since the line would be too wide for the terminal print line.
- Sequential files are always written starting at their current location.
- A special non-printing delimiter follows all string output to a sequential file. When such data is read, this character acts as an input delimiter as does a comma.
- Entire arrays may be written onto a file by beginning the statement with the word MAT and specifying array names in the variable list, as is done for the MAT PRINT statement.

The MAT PRINT statement is discussed on p.7-14.

Example:

```
5 OPEN "MY FILE", 24, SEQUENTIAL
10 DATA 4.56, 7.89
20 READ IVAL, KVAL
30 FOR X = 1 TO 100
40 MVAL = IVAL * KVAL
60 PRINT ON 4: IVAL, KVAL, MVAL;
65 IVAL = IVAL + 2.31
70 KVAL = KVAL + 4.32
80 NEXT X
.
.
.
```

The values generated by the program loop in statements 30 through 80 are written onto the file MYFILE.

PRINT (RANDOM FILE) STATEMENT

[MAT] PRINT ON exp$_1$ AT exp$_2$: var$_1$ [, var$_2$, ... var$_n$]

```
PRINT ON 5 AT 4:F$
PRINT ON 6 AT X-1: A: D: C,B;
PRINT ON 1 AT 12:IVAL, JVAL, KVAL;
```

The PRINT ON ... AT statement is used with random files and writes information specified by the variable list on the record number named. This statement is identical to terminal print statements except that values are written on a file and not printed at the terminal. The variable list conventions are the same as terminal input/output.

PRINT statements may be used for Immediate or Program Execution.

The IN FORM option for input and output may be used for symbolic file input/output.

- Random files are written beginning at the record specified by exp$_2$.
- Entire arrays may be written onto a record by beginning the statement with the word MAT and specifying array names in the variable list, as is done for the MAT PRINT statement.

The MAT PRINT statement is discussed on p.7-14.

Example:

```
5 OPEN "MY FILE", 5, RANDOM, OLD
10 DATA 3.24, 7.89
20 READ IVAL, KVAL
30 FOR X = 1 TO 100
40 MVAL = IVAL * KVAL
50 PRINT ON 5 AT X: IVAL, KVAL, MVAL
60 IVAL = IVAL + 2.34
70 KVAL = KVAL + 4.32
80 NEXT X
•
•
```

The values generated by the program loop in statements 30 through 80 are written onto the file MYFILE in records 1 to 100.

9. EDIT STATEMENTS

ELEMENTARY EDITING FEATURES

Standard Editing

The special teletype keyboard characters given below are recognized by the system as editing commands. While each line of a program is input to the computer, alterations and corrections can be made using these keys.

$\textcircled{A^C}$ This key deletes the previous character input. The key may be used repeatedly and erases a character each time the key is repeated. A backward arrow (←) is echoed for each $\textcircled{A^C}$. For example, BATH←←SC←IC is interpreted as BASIC.

$\textcircled{Q^C}$ This key completely deletes the line being input (i.e., before the Carriage Return key is pressed). An upward arrow (↑) is echoed for each $\textcircled{Q^C}$. The prompt character is not reissued.

For example:

>30 FOR I = 1 TO $\textcircled{Q^C}$ causes the entire line to be erased.

Inserting Statements

A statement can be inserted between two existing statements by typing the new statement with a line number within the bounds of the existing statements.

For example:

```
>10 A = 4.56
>20 C = A+B
>30 D = 6
>40 E = C+D
>15 B = 2.34
```

Statement 15 will automatically be inserted in sequence

```
10, 15, 20, 30, 40.
```

Replacing Statements

Any statement may be replaced or changed from any point in the program by entering its line number and retyping the entire statement.

For example:

```
>10 LET A = 3
>10 LET B = 5
```

The first line 10 is replaced by the second.

ENTER STATEMENT

ENTER line no. [STEP exp]

```
ENTER 10
ENTER 10 STEP 5
```

Normally the user must type in a line number for each statement when creating a program. However, the ENTER command can be used to direct the system to generate line numbers automatically as the user creates a program. The ENTER command specifies the initial line number and the increment value desired between generated line numbers.

ENTER may be used for Immediate Execution only.

- If an increment value (line no.) is not specified, a value of 10 is assumed.
- The initial line number must be higher than any of the current program statements. If while in the ENTER mode, a syntactically incorrect statement is typed in, automatic line number generation is terminated.
- Automatic line number generation can be terminated by pressing D^C immediately after a line number.

Example:

```
ENTER 5 STEP 10
5 READ A,B,C,D
15 DATA 1,2,3,4
25 A = B+C*A
35 PRINT A
45 END
55 D^C
```

LIST STATEMENT

```
LIST [ line no. [ : line no. ] [ , ... line no. [ : line no. ] ] ]
```

```
LIST 40
LIST 40:70, 100, 160:200
LIST
```

LIST may be used for Immediate Execution only.

The LIST statement enables the user to specify that an entire program, a portion of a program, or a single statement be printed out at the terminal. If no line numbers are specified, the entire program is listed. Commas separate individual statements or statement groups. A colon designates a range of statements.

- A line number may also be expressed as FIRST or LAST, specifying the first or last line of the program.

Example:

```
10 BETA 3,4,5,9 A^C 6
20 READ A,B,C,D
30 LET A = B*D
30 LET A = B*C
40 B = B/C
50 PRINT A
60 A = A*B
70 PRINT A,B
LIST 10:30, LAST

10 DATA 3,4,5,6
20 READ A,B,C,D
30 LET A = B*C
70 PRINT A,B
```

RENUMBER STATEMENT

RENUMBER [line no.$_1$, line no.$_2$, increment]

```
RENUMBER
RENUMBER 50, 100, 10
```

The RENUMBER command enables the user to assign new line numbers to all or part of a program. All of the program's statements which reference line numbers (GOTO, GOSUB, IF, etc.) are similarly modified according to the new line numbers.

RENUMBER may be used for Immediate Execution only.

- The part of the program to be renumbered begins at line no.$_1$ and ends with the last program statement. The first statement to be renumbered is assigned line no.$_2$. Subsequent line numbers are incremented by the increment value specified.
- The RENUMBER command cannot generate line numbers which cause intermixing of the new line numbers and lines not renumbered.
- If not specified, the following values are assumed for the RENUMBER statement parameters:

  line no.$_1$ - new line numbers start at 100.
  line no.$_2$ - renumbering begins at the first line of the program.
  increment - the increment value is 10.

DELETE STATEMENT

```
{ DELETE } { ALL                                                     }
{ DEL    } { line no. [ :line no. ][ , ... line no.[ : line no. ] ] }
```

DELETE ALL

DELETE FIRST : 100

DELETE 10:50, 80

DEL 25, 10:18, 41

DELETE may be used for Immediate Execution only.

The DELETE statement deletes a single statement, a range of statements, or a whole program. Commas separate individual statements or statement groups; a colon designates a range of statements.

- If several lines are to be deleted, the numbers must be in ascending order.
- FIRST and LAST may also be used to indicate the first and last statements of the program to be deleted.
- A DELETE statement specifying a non-existent line causes an error.

Example:

```
10 READ A,B,C,D
20 DATA 2,3,4,5
30 LET A = B*C
40 LET D = A*B
50 C = C*D
60 PRINT A,B
70 PRINT C,D,E
DELETE 30:50, 70
LIST
10 READ A,B,C,D
20 DATA 2,3,4,5
30 PRINT A,B
```

ALTER STATEMENT

```
ALTER line no.
```

ALTER 10
ALTER FIRST
ALTER LAST

The ALTER statement prints out any specified statement at the terminal and enables the user to change its content by means of a set of special editing control characters.

ALTER may be used for Immediate Execution only.

The following control characters may be used in conjunction with the ALTER command in addition to the standard editing characters discussed in the beginning of this section.

Elementary editing techniques are discussed on p. 9-1.

Control Character	Function
C^c	Copies the next character from the old line and echoes to the new line.
S^c	Skips the next character in the old line and echoes a % in the new line.
D^c	Copies and echoes the remainder of the old line to the new line and stops editing the old line.
E^c	Enter/Exit insert mode. Echoes '<' and '>', respectively. The text typed in this mode is inserted into the new line and does not affect the old line.
Other	Other characters typed in replace corresponding characters in the old line.

- Multi-line statements are edited on a segment basis, i.e., between Line Feeds to a terminating Carriage Return. The control, " copy remainder of edited line," will copy only up to the next LF or CR.

- FIRST or LAST may also be used to indicate the first or last statement of the program to be edited.

Example:

10 VDOT=FNX(YPOS +PI-2)*FNDELT(YPOS)
ALTER 10
10 VDOT=FNX(YPOS +PI-2)*FNDELT(YPOS)

C^c	C^c	C^c	C^c	C^c	C^c	C^c	C^c	C^c	C^c	C^c	C^c	C^c	C^c	C^c	C^c	S^c	S^c	S^c	S^c	D^c	(as echoed)
1	0		V	D	O	T	=	F	N	X	(	Y	P	O	S	%	%	%	%	)*FNDELT(YPOS)	

ALTER 10
10 VDOT=FNX(YPOS)*FNDELT(YPOS)

C^c	C^c	C^c	C^c	C^c	C^c	C^c	C^c	E^c	SQRT(YPOS↑ 2)+	E^c	D^c	(as echoed)
1	0		V	D	O	T	=	<	SQRT(YPOS↑ 2) +	>	FNX(YPOS)*FNDELT(YPOS)	

LIST 10
10 VDOT=SQRT(YPOS↑ 2) +FNX(YPOS)*FNDELT(YPOS)

TABS STATEMENT

TABS [pos_1, pos_2, pos_3, pos_4]

```
TABS 10, 30, 50, 60
TABS
TABS 23, 45
```

The TABS statement enables the user to establish tab positions on the terminal analogous to setting tabs on a standard typewriter. By specifying tab positions for the terminal using this statement, the user can easily indent portions of his program to improve its readability. The tabs have no effect on the execution of the program. The I^C control character is equivalent to the tab key on a conventional typewriter and advances the print position of the terminal to the next tab position. The appropriate number of blanks is inserted into the data buffer.

TABS may be used for Immediate Execution only.

The TABS statement also affects the function of the U^C and M^C control characters used with the ALTER command.

- Up to four tab positions may be specified in the TABS statement.
- The TABS statement removes previous tabs and establishes new tab positions as specified.
- A TABS statement with no tab positions specified removes any previous tabs.

10. PROGRAM CONTROL STATEMENTS

INTRODUCTION

Once the BASIC system is in control, the user may direct the disposition of a BASIC program by using the control commands discussed in this section. These commands differ from BASIC language commands in that they are not a part of the program, but control the program's activities.

RUN STATEMENT

```
RUN
```

RUN

RUN may be used for Immediate Execution only.

The RUN command causes the initialization and execution of the program currently in the program source area.

- Information regarding previous program execution is lost.
- All previously opened files are automatically closed when the RUN command is executed.

Example:

```
10 READ A,B,C
20 DATA 2,3,4
30 PRINT A*B; B/A; C/A+B
40 END
RUN
```

CONTINUE STATEMENT

```
CONTINUE
```

CONTINUE

If the user has suspended program execution by pressing the ESC key or by executing the PAUSE statement, and intermediate activities have not destroyed the original program and data, execution can be resumed by using the CONTINUE statement.

CONTINUE may be used for Immediate Execution only.

Execution continues at the statement following the line where execution was suspended.

Example:

```
10 LET A = 5.67
20 B, C = 3.45
30 ANS = A*SQRT(A/B)
40 PAUSE
50 IVAL = C*SQRT(C/ANS)
.
.
.
220 END
RUN
PAUSE AT 40
PRINT ANS
7.2688389
CONTINUE
.
.
.
```

SAVE STATEMENT

$$\boxed{\text{SAVE "file name"} \begin{bmatrix} \text{, SYMBOLIC} \\ \text{, BINARY} \end{bmatrix} \begin{bmatrix} \text{, NEW} \\ \text{, OLD} \end{bmatrix} [\text{line no.}_1, \text{ line no.}_2, \ldots \text{ line no.}_n]}$$

```
SAVE "FILEA", SYMBOLIC, NEW
SAVE "BIN1", BINARY, OLD, 100, 400
SAVE "MYFILE"
```

SAVE may be used for Immediate Execution only.

The SAVE command stores the current program (symbolic form) or the compiled equivalent (binary form) on the file specified. The file specified will be created if designated as NEW, or a previously generated file will be used if designated as OLD. The only required parameter for this statement is the file name. The default values for the other optional parameters are SYMBOLIC and NEW.

LOAD is discussed on p. 10-5; LINK is discussed on p. 10-6.

- The file created may be used later with the LOAD (symbolic file only) and LINK (binary file only) commands.
- An error message is printed if the parameter NEW is used with a file already in the user's directory, or if OLD is specified with a file name that does not exist in the directory.
- If a binary file is saved which will subsequently be used in a LINK statement requiring entry to the file at a point other than its first statement, the SAVE ... BINARY command must specify a line number or set of line numbers which may be specified as an entry point to the file in the LINK statement.

LOAD STATEMENT

```
LOAD "file name"
```

LOAD "MYFILE"

The LOAD statement enables the user to retrieve a saved file from storage for use in the program source area. Loading a saved file updates (by replacement) any source statements already in the program source area.

LOAD may be used for Immediate Execution only.

- Statements in the source area are replaced by statements from the file that are loaded with the same line numbers. Other statements are inserted according to line number sequence.

- The LOAD statement may specify a symbolic file only.

Example:

```
LOAD "FILEA"
LOAD "FILEB"
LIST
.
.
.
SAVE "MYFILE", NEW
```

FILEB is loaded into the program source area to edit FILEA by replacing duplicate statements. The LIST command causes the entire contents of the source area (i. e., FILEA as modified by FILEB) to be printed at the terminal. The SAVE command causes entire content of source to be stored as the new file MYFILE. The files FILEA, FILEB remain unchanged in the user's directory.

LINK STATEMENT

```
LINK "file name" [ , line no. ]
```

LINK "BIFILE"
LINK "BF%%%" , 300

LINK may be used for Immediate or Program Execution.

The LINK statement enables the user to retrieve and execute a previously saved binary file. When the LINK statement is executed, any current program execution is terminated and execution of the contents of the file specified is initiated. If a line number is specified in the LINK statement, execution will begin at that line number, otherwise, program execution will begin at the first statement of the file.

Since the file activated by the LINK statement is not available in symbolic form, no interactive processing is allowed during execution caused by the LINK statement. If errors occur during execution, the user can only attempt to issue the CONTINUE statement.

Once program execution is completed (or abnormally terminated) the BASIC prompt character will be issued.

Example:

```
10 PRINT "ORDER INVENTORY SYSTEM"
20 PRINT "TO INPUT AN ORDER, TYPE 'ORDER'"
30 PRINT "TO INPUT A SHIPPING INVOICE, TYPE 'SHIPPED'"
40 INPUT A$
50 IF A$ = "ORDER" THEN LINK "ORDFILE" ELSE LINK "SHPFILE"
60 END
```

TAPE STATEMENT

TAPE

TAPE

Instead of entering BASIC program statements directly from the terminal, the user may prepare his program on paper tape. The TAPE command enables BASIC to accept paper tape input.

TAPE may be used for Immediate Execution only.

- The TAPE command must be used prior to paper tape input. Information on the tape must be terminated by the control character D^c on the tape or a D^c must be entered from the keyboard after the tape has been read in.

Paper tape preparation is discussed in Appendix F for the Model 33 Teletype Terminal.

- The syntax of statements on paper tape is not checked when the tape is input. Syntax errors are detected following paper tape input. The incorrect line and the error message are then printed.
- Statements already in the source area are replaced by those from paper tape input with the same line numbers. Other statements are inserted according to line number sequence.

Example:

```
TAPE
( read in paper tape program )
D^c
LIST
.
.
.
RUN
```

Once the paper tape program is entered, the LIST command may be used to display the contents of the program source area on the terminal.

QUIT STATEMENT

QUIT

QUIT

QUIT may be used for Immediate Execution only.

The QUIT statement is the only statement which enables the user to leave the BASIC subsystem and return to EXECUTIVE. When the program is returned to EXECUTIVE, the contents of the current source area (unless SAVE is used) and data area are lost, and all open files are closed.

LEAVING THE SYSTEM — LOGOUT

Once the user has left the BASIC subsystem, he can issue the EXECUTIVE level LOGOUT command to terminate a terminal session. At this time, accounting statistics pertaining to computer time and storage are recorded by the system and a permanent record maintained for each account and user name.

```
-LOGOUT CR
tttt   mm/dd/yy
CPU MINS-   xx.xx
TERMINAL MINS-   xx.xx
FILE MODULES-   zzzz
```

where:

tttt = time of day in 24-hour clock units
mm = month
dd = day
yy = year
xx.xx = time total in minutes
zzzz = disc space units associated with the user

After the system prints the above information, the terminal is returned to LOGIN command mode. If there is no response within three minutes, the terminal is disconnected.

APPENDIX A. ANSI CHARACTER SET

ANSI Hex Code	Character	Teletype Key	Teletype/ Printer Graphic	Hollerith Card Code
00	NUL	P^{cs}		12-0-9-8-1
01	SOH or DEL	A^c		12-9-1
02	STX	B^c		12-9-2
03	ETX	C^c		12-9-3
04	EOT	D^c		9-7
05	ENQ	E^c		0-9-8-5
06	ACK	F^c		0-9-8-6
07	BEL	G^c		0-9-8-7
08	BS	H^c		11-9-6
09	HT	I^c		12-9-5
0A	LF	Line Feed		0-9-5
0B	VT	K^c		12-9-8-3
0C	FF	L^c		12-9-8-4
0D	CR	Return		12-9-8-5
0E	SO	N^c		12-9-8-6
0F	SI	O^c		12-9-8-7
10	DLE	P^c		12-11-9-8-1
11	DC1	Q^c		11-9-1
12	DC2	R^c		11-9-2
13	DC3	S^c		11-9-3
14	DC4	T^c		9-8-4
15	NAK	U^c		9-8-5
16	SYN	V^c		9-2
17	ETB	W^c		0-9-6
18	CAN	X^c		11-9-8
19	EM	Y^c		11-9-8-1
1A	SUB	Z^c		9-8-7
1B	ESC	K^{cs}		0-9-7
1C	FS	L^{cs}		11-9-8-4
1D	GS	M^{cs}		11-9-8-5
1E	RS	N^{cs}		11-9-8-6
1F	US	O^{cs}		11-9-8-7
20	Blank	Space Bar		
21	!	1^s	!	12-8-7
22	"	2^s	"	8-7
23	#	3^s	#	8-3
24	$	4^s	$	11-8-3
25	%	5^s	%	0-8-4
26	&	6^s	&	12
27	'	7^s	'	8-5
28	(	8^s	(	12-8-5
29	)	9^s	)	11-8-5
2A	*	$:^s$	*	11-8-4
2B	+	$;^s$	+	12-8-6
2C	,	,	,	0-8-3
2D	-	-	-	11
2E	.	.	.	12-8-3
2F	/	/	/	0-1
30	0	0	0	0
31	1	1	1	1
32	2	2	2	2
33	3	3	3	3
34	4	4	4	4
35	5	5	5	5
36	6	6	6	6
37	7	7	7	7
38	8	8	8	8
39	9	9	9	9
3A	:	:	:	8-2
3B	;	;	;	11-8-6
3C	<	$,^s$	<	12-8-4
3D	=	$-^s$	=	8-6
3E	>	$.^s$	>	0-8-6
3F	?	$/^s$	?	0-8-7
40	@	P^s	@	8-4
41	A	A	A	12-1
42	B	B	B	12-2
43	C	C	C	12-3
44	D	D	D	12-4
45	E	E	E	12-5
46	F	F	F	12-6
47	G	G	G	12-7
48	H	H	H	12-8
49	I	I	I	12-9
4A	J	J	J	11-1

ANSI Hex Code	Character	Teletype Key	Teletype/ Printer Graphic	Hollerith Card Code
4B	K	K	K	11-2
4C	L	L	L	11-3
4D	M	M	M	11-4
4E	N	N	N	11-5
4F	O	O	O	11-6
50	P	P	P	11-7
51	Q	Q	Q	11-8
52	R	R	R	11-9
53	S	S	S	0-2
54	T	T	T	0-3
55	U	U	U	0-4
56	V	V	V	0-5
57	W	W	W	0-6
58	X	X	X	0-7
59	Y	Y	Y	0-8
5A	Z	Z	Z	0-9
5B	[	K^S	[	12-8-2
5C	\	L^S	\	0-8-2
5D	]	M^S	]	11-8-2
5E	↑	N^S	↑	11-8-7
5F	←	O^S	←	0-8-5
60	` or ¢		@	8-1
61	a		A	12-0-1
62	b		B	12-0-2
63	c		C	12-0-3
64	d		D	12-0-4
65	e		E	12-0-5
66	f		F	12-0-6
67	g		G	12-0-7
68	h		H	12-0-8
69	i		I	12-0-9
6A	j		J	12-11-1
6B	k		K	12-11-2
6C	l		L	12-11-3
6D	m		M	12-11-4
6E	n		N	12-11-5
6F	o		O	12-11-6
70	p		P	12-11-7
71	q		Q	12-11-8
72	r		R	12-11-9
73	s		S	11-0-2
74	t		T	11-0-3
75	u		U	11-0-4
76	v		V	11-0-5
77	w		W	11-0-6
78	x		X	11-0-7
79	y		Y	11-0-8
7A	z		Z	11-0-9
7B	{			12-0
7C	|			12-11
7D	}			11-0
7E	¬ or ~			11-0-1
7F	RUBOUT			12-9-7

APPENDIX B. MESSAGES

This appendix contains a complete list of all messages issued by the BASIC subsystem at the terminal. While most of the messages are error notifications, a few are informative only and require no response from the user. Each error message is discussed in terms of probable cause and remedial action. There are three classes of error messages: syntax (S), compilation (C), and execution (E).

SYNTAX ERRORS

Syntax error messages are issued immediately following the input of a BASIC statement whose form is incorrect (unless automatic syntax error checking is deactivated at the terminal). These messages always pertain to the most recent statement entered by the user. Likewise, they can be corrected immediately by typing in the correct statement with the same line number as the line in error. (If desired, the error message may be ignored at this time, and remedial action deferred until program execution is attempted and the message is issued again.) For example:

```
>100 DEF ABCD (K,J,L)
            ↑
FUNCTION IDENTIFIER MISSING
>100 DEF FNAB (K,J,L)
> ...
```

COMPILATION ERRORS

Compilation error messages are caused by syntax errors which were not corrected as the statement was input, and errors caused by statements whose meaning is incorrect within the rules of the BASIC language. These errors must be corrected within the current program before the program can be executed. Since these messages are issued after the user completes the program and enters the RUN command, the system prints out the statement in error with an upward arrow (↑) indicating the probable point of the error. The user must enter the RUN command after the statement is corrected. For example:

```
>RUN
60 GOTO 500
          ↑
LINE NUMBER DOES NOT EXIST
>60 GOTO 100
>RUN
```

EXECUTION ERRORS

Execution error messages are caused by errors in program content, whereby the computer cannot carry out the instructions generated by the program. Execution error messages are generated as the program is executed. When an error is encountered, the system prints out the line number of the statement with the appropriate message. The user has two courses of action: correct the current program source and reattempt execution, or correct the program interactively, whereby program execution will continue, but the current program source is not corrected. For example:

```
>RUN
ERROR IN LINE 100
ILLEGAL SQRT ARGUMENT
>LIST 100
100 A = SQRT(B)
>PRINT B
-36
>100 A = SQRT(-B)
>RUN
```

The above corrects the statement in the current program. However, if the user is interested only in the results of the program without correcting the source program immediately, he may do the following:

```
>RUN
ERROR IN LINE 100
ILLEGAL SQRT ARGUMENT
>LIST 100
100 A = SQRT(B)
>PRINT B
-36
>B = 36
> GOTO 100
```

	S	C	E
ATTEMPT TO CHANGE FILE STRUCTURE Cause: In an OPEN statement any of the following may have been specified (possibly by default): - binary file as SYMBOLIC - symbolic file as BINARY - random file as SEQUENTIAL - sequential file as RANDOM - fixed-length for a variable length record file Action: Correct the appropriate OPEN statement.			X
ATTEMPT TO INPUT FROM WRITE FILE Cause: The user has attempted to read a file which is open for writing. Action: Close the appropriate file and reopen for read access.			X
ATTEMPT TO WRITE ON INPUT FILE Cause: The user has attempted to write on a file which is open for reading. Action: Close the appropriate file and reopen for write access.			X
BASIC ERROR Cause: A BASIC system error was encountered in a BASIC program. Action: Attempt to rerun the program or call operator.	X	X	X
CONCATENATION STRING LENGTH ERROR Cause: String length has exceeded maximum of 255 characters as result of string concatenation. Action: Correct current program.			X
CORRECT SYNTAX IN FORM Cause: An INPUT statement containing the IN FORM option was syntactically incorrect. Action: Correct current program.	X	X	
DATA COMPATIBILITY ERROR Cause: An attempt was made to use a string value in an operation using numeric data or vice versa. Example: A$ = B when A$ is string and B is real. Action: Correct current program.		X	
DIRECT MODE STATEMENT ONLY Cause: An attempt was made to use an Immediate Execution command for Program Execution. For example, the command RUN was preceded by a statement number. Action: Delete Immediate Execution commands from program.	X	X	

	S	C	E
DUPLICATE DEFINITION Cause: 1. A scalar variable appeared in two or more type-conflicting statements. 2. A scalar variable was used in a program according to implicit type conventions and was subsequently explicitly declared in a type statement. Action: 1. Delete appropriate explicit declarations from current program. 2. Renumber explicit type statement so that it is encountered before the variable is used in the program.		X	
DUPLICATE FILE NAME Cause: An OPEN attempt was made with the NEW option where a file of the same name already exists in the user's file directory. Action: 1. The old file of the same name must be deleted (at the EXECUTIVE level). 2. The new file must be given another name and the program's OPEN statement corrected to reflect the new file name.			X
DUPLICATE FUNCTION PARAMETERS Cause: A variable name was used more than once as a dummy argument for a particular function. Action: Change the duplicate dummy argument name in the DEF statement.		X	
END OF DATA Cause: The internal data list produced by a program's DATA statements did not satisfy the program's READ statement's variable list. Action: 1. Correct the DATA statements to include more data items. 2. Insert a RESTORE statement at an appropriate line number to reset the data list pointer.			X
END OF FILE Cause: An end of file was processed on a disc file without an ON ENDFILE statement existing for that file. Action: Correct current program to include an ON ENDFILE statement for the appropriate file. The ON ENDFILE must be executed after each opening of the file number to which it refers.			X
END OF RECORD Cause: An end of record was processed on a random file record without an ON ENDREC statement existing for that record. Action: Correct current program to include an ON ENDREC statement for the appropriate file record.			X

	S	C	E
EXPRESSION TOO LARGE Cause: The internal memory requirements of an expression exceeded available space. Action: Subdivide the appropriate statement in the current program containing the expression into statements which contain expressions which perform the same operations as the larger expression.		X	
FILE ALREADY CLOSED Cause: An attempt was made to CLOSE a file which was already closed in the program. Action: Delete the redundant CLOSE statement.			X
FILE ALREADY OPEN Cause: An attempt was made to OPEN a file which was already open in the program. Action: If the file was already OPEN for the desired access, remove redundant OPEN statement. If not, CLOSE the file and reopen for the desired access .			X
FILE DOES NOT EXIST Cause: 1. An operation was attempted on a file not open. 2. A file specified as OLD in an OPEN statement does not exist in the user's directory. Action: 1. Insert OPEN statement for appropriate file. 2. Check name of files in user's directory and correct OPEN statement if appropriate to specify NEW file.			X
FILE NOT SHARABLE Cause: An attempt was made to OPEN a private (non-shared) file. Action: None.			X
FLOATING DIVIDE BY ZERO Cause: An attempt was made to divide a floating-point number by zero. Action: Correct the appropriate program statement to eliminate division by zero.			X
FLOATING OVERFLOW Cause: An arithmetic operation caused a result too large to be represented in single or double precision floating point format. Action: Adjust the appropriate values in the current program.			X
FLOATING UNDERFLOW Cause: An arithmetic operation caused a result too small to be represented in single or double-precision floating point format. Action: Adjust the appropriate values in the current program.			X

	S	C	E
FOR LOOPS NESTED OVER 31 Cause: The number of loops in a FOR loop nested set has exceeded the limit of 31. Action: Correct the current program to reduce the number of loops within the same nesting to ≤ 31.		X	
FORM OPERATION ATTEMPT ON BINARY FILE Cause: An attempt was made to use the IN FORM option in a READ or WRITE binary file statement. Action: Delete the IN FORM option from the appropriate READ or WRITE statement.			X
FORWARD REFERENCE UNSATISFIED Cause: 1. A FOR statement does not have a corresponding NEXT statement. 2. There is no END statement for a user-defined multi-line function. Action: Insert the NEXT or END statement into the current program.		X	
FUNCTION DEFINED WITHIN FUNCTION Cause: A DEF statement was encountered within the range of a DEF... END statement group. Action: Unless an attempt was deliberately made to nest (illegally) multi-line function definitions, an END statement may be missing for a DEF encountered. Insert an END statement into the current program if appropriate.		X	
FUNCTION IDENTIFIER MISSING Cause: An attempt was made to define a multi-line function whose name does not begin with the letters FN. Action: Prefix the function name with the letters FN in the current program.	X	X	
FUNCTION NOT DEFINED Cause: 1. A reference was made to a function name prior to the definition of the function in the program. 2. A variable name begins (illegally) with the letters FN. Action: 1. Rearrange current program contents so that the function definition precedes any usage of the function. 2. Correct the variable name so that it does not begin with FN.		X	
FUNCTION PREVIOUSLY DEFINED Cause: The same function name was used more than once for different functions. Action: Modify redundant function names.		X	
GOSUB WITHIN FUNCTION Cause: A GOSUB statement was used in a multi-line function definition. Action: Correct current program to eliminate any GOSUB statements from multi-line function definitions.		X	

	S	C	E
ILLEGAL ASIN/ACOS ARGUMENT Cause: An argument greater than 1.0 was used in either the ASIN or ACOS function. Action: Adjust argument value in current program.			X
ILLEGAL ASSIGNMENT Cause: A constant or function identifier appeared on the left side of the equal sign in an assignment statement. Action: Correct assignment statement in current program.	X	X	
ILLEGAL ATAN ARGUMENT Cause: An attempt was made to use the ATAN function with the values x = 0, y = 0. Action: Adjust argument values in current program to be non-zero.			X
ILLEGAL CHARACTER FOR NUMERIC CONVERSION Cause: A non-numeric character was used as a numeric value. Action: Correct current program.			X
ILLEGAL CHARACTER INPUT Cause: 1. An unrecognizable (non-BASIC) character was received as input. 2. Transmission error. Action: 1. Enter correct characters. 2. Reenter input.	X	X	
ILLEGAL EXP ARGUMENT Cause: An exponential argument was greater than the limit of 176.752. Action: Adjust argument value in current program to be ≤ 176.752.			X
ILLEGAL EXPRESSION Cause: An expression is missing an operator or operand. Action: Insert appropriate expression in current program.		X	
ILLEGAL FILE DESIGNATOR Cause: A file number specified in an OPEN, CLOSE, READ, INPUT, or WRITE statement specified a value not in the range 0 - 8. Action: Correct the current program.			X
ILLEGAL FILE STRUCTURE Cause: An illegal file structure was specified in an OPEN statement. One of the following illegal combinations has been specified: NEW and INPUT SCR and INPUT SEQUENTIAL and IO Action: Correct the OPEN statement to reflect a legal combination of parameters.			X

	S	C	E
ILLEGAL FIXED RECORD LENGTH Cause: The length of a fixed-length record specified in an OPEN statement was not in the range 1 to 1024. Action: Correct the record length specification.			X
ILLEGAL FOR LOOP NESTING Cause: The ranges of the program's FOR loops overlap. Action: Correct the FOR loop nesting structure in the current program.		X	
ILLEGAL FOR STATEMENT Cause: FOR statement syntactically incorrect. Action: Correct FOR statement.	X	X	
ILLEGAL FORM SPECIFICATION Cause: A non-field definition character was found in an input/output statement using the IN FORM option. Action: Correct the field definition to include only the characters allowed.			X
ILLEGAL INPUT DELIMITER Cause: A character other than a comma or blank was used to delimit items entered in response to requests generated by INPUT statements. Action: Reenter data using a comma or blank spaces to separate items.			X
ILLEGAL LINE NUMBER REFERENCE Cause: A reference was made to a statement within a multi-line function from outside the function or vice versa. Action: Correct line number reference.		X	
ILLEGAL LOG ARGUMENT Cause: An argument of zero or less was used with a logarithmic function. Action: Correct function argument to a positive, non-zero value.			X
ILLEGAL LOOP VARIABLE Cause: A string, array, or complex value was used as a loop variable. Action: Correct loop variable to be scalar (integer, real, or double) variable.		X	
ILLEGAL MAT OPERATOR Cause: An attempt was made to perform an operation other than the following: addition, subtraction, scalar multiplication, multiplication by matrix. Action: Correct current program.	X	X	

	S	C	E
ILLEGAL OPERATOR Cause: An attempt was made to perform an operation using operators other than those specified as valid operators in section 3 or this manual. Action: Correct appropriate operator.		X	
ILLEGAL RECORD IDENTIFIER Cause: A number specified as a record identifier was not a positive integer value. Action: Change record identifier to positive value.			X
ILLEGAL SIN/COS ARGUMENT Cause: An argument in a SIN or COS function was greater than 2^{52}. Action: Adjust argument value in current program to be $\leq 2^{52}$.			X
ILLEGAL SQRT ARGUMENT Cause: A SQRT argument was less than zero (negative value). Action: Correct SQRT argument to a non-negative value.			X
ILLEGAL STRING FUNCTION ARGUMENT Cause: The order and type of arguments to a string function were incorrect. Action: Specify appropriate arguments in their proper order for a string function.			X
ILLEGAL SUBSCRIPT Cause: A string value was used as a subscript identifier. Action: Use only numeric values for subscript specification.		X	
ILLEGAL TAN ARGUMENT Cause: A TAN function argument was greater than 2^{52}. Action: Adjust argument in current program to be $\leq 2^{52}$.			X
ILLEGAL USE OF 0 Cause: An attempt was made to do a binary operation on file 0, reserved as the teletype file. Action: Correct file number reference or delete from program.			X
ILLEGAL USE OF RANDOM FILE Cause: A record number was not specified for a random file. Action: Include record number specification in appropriate program statement.			X
ILLEGAL USE OF SEQUENTIAL FILE Cause: A record number was specified for a sequential file in a file input/output statement. Action: Delete record number specification for a sequential file in the file input/output statement.			X

	S	C	E
INCOMPLETE FORM SPECIFICATION Cause: An I/O statement with the IN FORM option did not specify format of input/output data. Action: Correct the appropriate statement.			X
INCORRECT STATEMENT FORM Cause: The statement is syntactically incorrect. Action: Correct statement.	X	X	
INDIRECT MODE STATEMENT ONLY Cause: A statement that may be used for Program Execution was entered for Immediate Execution, i.e., without a statement number. Action: Prefix the statement with an appropriate statement number.		X	
INPUT RECORD LENGTH OVERFLOW Cause: Input to a fixed-length record exceeded the length specified by the user in an OPEN statement for the appropriate file. Action: Reduce input or increase the length of the record in the OPEN statement.			X
INPUT STRING LENGTH ERROR Cause: A string exceeded the 255 character string limit. Action: Shorten string or create multiple strings.			X
INTEGER OVERFLOW Cause: An integer variable was assigned a floating point value which cannot be expressed as an integer without losing meaning; i.e., the floating point number is too large to be represented as an integer value. Action: 1. Adjust value in current program. 2. Redefine integer variable to a real or double-precision type.			X
INVALID VARIABLE NAME Cause: An attempt was made to use a BASIC reserved word as a variable name to begin a variable name of more than two characters with the letters FN, or to use a variable name with more than four characters. (The latter may be caused by missing blanks after a variable name.) Action: Change the appropriate variable name.		X	
I/O WITHIN I/O Cause: A function call during an output operation caused an additional input or output request. Action: 1. Delete input/output operation from function definition. 2. Remove function call from input/output statement.			X
LAST LINE IS Cause: Informative message caused by operation of LOAD or TAPE command informing user of last line input. Action: None.		X	

S C E

INTERRUPT AFTER COMPLETION LINE xxxx — E: X

Cause: This is issued in response to the user pressing the ESC or ALT MODE key during program execution. xxxx is the line number at which execution was interrupted.

Action:
1. Interrogate variable values in current program and/or alter them.
2. Reenter program at an alternate line using a GOTO, DO, etc.
3. Enter a CONTINUE command to resume execution at the line following the line number specified in the message.

INTERRUPT DURING LINE xxxx — E: X

Cause: The user has pressed the ESC or ALT MODE key while output or a matrix inversion operation was in process. xxxx is the line number at which execution was interrupted.

Action:
1. If interrupt occurred during an output operation, output may not be continued unless the user reenters the program at a point prior to the output operation, whereby the output operation will be repeated. A CONTINUE command causes execution to be resumed at the line following the line number specified.
2. If interrupt occurred during a matrix inversion, the user is notified of the inversion; he may continue the inversion by entering the CONTINUE command immediately after the message is printed. Otherwise the matrix inversion is terminated and a subsequent CONTINUE command causes execution to be resumed at the line following the line number specified in the message.
3. Interrogate variable values in current program and/or alter them.
4. Reenter program at an alternate line using a GOTO, DO, etc.

INVALID ACCOUNT NUMBER — E: X

Cause: An attempt was made to open a shared file with an incorrect account number.

Action: Correct account number in the OPEN statement.

INVALID DATA TYPE — C: X

Cause: A value was used whose data type is not allowed with the operator used.

Example: B$ - B$ the minus operator cannot be used with strings.
A ↑ CMPLX (6, -3) — The ↑ operator cannot have a complex exponent.

Action: Correct current program.

INVALID RESTORE — E: X

Cause: A RESTORE statement did not reference a DATA statement.

Action: Modify RESTORE statement to reference a DATA statement.

	S	C	E
LINE NUMBER MISSING Cause: 1. A program statement was entered from paper tape without a line number. Statement cannot be issued from paper tape for Immediate Execution. 2. A non-existent line. number was referenced in a program. Action: 1. Correct paper tape input. 2. Check if reference is in error and correct if necessary, or else insert a dummy statement such as REM with line number desired.	X	X	
LOOP TERMINATION NOT SPECIFIED Cause: A terminal expression was not specified in a FOR statement which includes a step or increment value. Action: Delete increment value phrase or add terminal condition.	X	X	
LOOP VARIABLE ALREADY ACTIVE Cause: The same loop variable name appears in two FOR statements without a separating NEXT statement. Action: Insert a NEXT statement at end of the first FOR loop.		X	
MATRICES NOT CONFORMABLE Cause: Two matrices with non-compatible subscript ranges were used in the same matrix operation. Action: Redimension matrices prior to use in matrix operation to make them conform to meet the requirements of the particular matrix operation.			X
MATRIX NEARLY SINGULAR Cause: Inversion was attempted on an ill-conditioned matrix. Action: 1. Change values in matrix. 2. Change value of EPS .			X
MATRIX NOT SQUARE Cause: A non-square matrix was used in a matrix inversion operation. Action: Redimension the matrix prior to use in the identity operation.			X
MATRIX NOT TWO DIMENSIONAL Cause: A non two-dimensional matrix was used in a matrix inversion operation. Action: Redimension the matrix prior to use in the inversion operation.			X
MATRIX TOO SMALL Cause: The result of a matrix operation was too large to be contained in the resultant matrix. Action: Change data type declaration of resultant matrix to double precision.			X

	S	C	E
MEMORY EXCEEDED Cause: An array as dimensioned exceeded memory capacity. Action: Reduce the size of the array.		X	X
MISSING GOTO OR GOSUB Cause: A statement beginning with the word ON did not contain a GOTO or GOSUB specification. Action: Insert GOTO or GOSUB in appropriate ON statement.	X	X	
MISSING THEN Cause: A statement beginning with the word IF did not contain THEN. Action: Insert THEN into the appropriate IF statement.		X	
MISSING QUOTE Cause: 1. A string value beginning with a single quote did not terminate with a single quote. 2. A string beginning with a double quote did not end with a double quote. Action: Insert appropriate quotation mark.			X
NO CONTINUE Cause: Operations were performed which alter the way in which a program is stored in memory. Action: Issue the RUN command.			X
NO EXECUTION POSSIBLE Cause: An Immediate Execution command such as GOTO was issued before program execution is attempted. Action: Issue the RUN command before any Immediate Execution commands are executed which reference line numbers of program to be executed.			X
NON ARITHMETIC IN MAT STATEMENT Cause: An attempt was made to use string matrices in a matrix arithmetic operation. Action: Correct current program according to operations allowed on numeric matrices.		X	
NON ARRAY IN MAT STATEMENT Cause: A scalar variable name was used in a matrix statement as an array name. Action: Define the variable as an array.		X	
PARAMETER INCOMPATIBILITY Cause: The argument dummies of a multi-line function definition were not compatible as to data type with their calling arguments. Action: Explicitly declare either dummy or calling arguments for type compability.		X	

	S	C	E
PARAMETER NOT DEFINED Cause: An argument to a multi-line function was not explicitly declared before its use in the function. Action: Declare the function parameter within the function definition before it is used.		X	
PARENTHESIS NESTING OVERFLOW Cause: The number of parenthetical sets of an expression in a FORM string exceeded the limit of five levels. Action: Redefine FORM string.			X
PAUSE AT LINE xxxx Cause: This message is issued when a program's PAUSE statement is executed. Action: The user may then issue an Immediate Execution statement to interrogate the state of the program. The CONTINUE command can be used to cause program execution to resume if possible.			X
RECORD DOES NOT EXIST Cause: The record number specified for a random file did not exist. Action: Check for appropriate record number and modify OPEN or file I/O statement appropriately.			X
RECORD TRUNCATED Cause: Output record was too long to fit in fixed-length record. Action: Modify PRINT statement field definition to accommodate actual output record length.			X
RECURSIVE FUNCTION CALL Cause: A multi-line function definition contained a call to itself. Action: Alter the function call.			X
RETURN WITHOUT GOSUB Cause: A RETURN statement appeared in a program without a companion GOSUB statement. Action: 1. Check to see if RETURN intended for multi-line function. 2. Insert a GOSUB statement at the appropriate location within the program.			X
STATEMENT DRIVER NOT FOUND Cause: A BASIC reserved word command was spelled incorrectly or was not set off by a leading and following blank space. Action: Correct the statement.	X	X	

	S	C	E
STATEMENT NOT VALID WITH IF Cause: Any of the following statements appeared within an IF statement: DATA, DEF, DOUBLE, INTEGER, REAL, COMPLEX, STRING, DOUBLE COMPLEX, IF, FOR, ON, or NEXT. Action: Remove the appropriate illegal statement from the IF statement.	X	X	
STRING CONSTANT OR VARIABLE ONLY Cause: An expression was used where only a scalar variable or constant is allowed. Action: Declare variable name as string or enclose constant in quotation marks.		X	
STRING TOO BIG FOR FORM Cause: An attempt was made to input or output a string using the IN FORM option where the field definition could not accommodate the size string specified for input or output. Action: 1. Shorten string length. 2. Redefine field in the input/output statement.			X
SUBSCRIPT ON SIMPLE VARIABLE Cause: A previously declared (implicitly or explicitly) scalar variable name appeared with subscript specification. Action: 1. Declare the appropriate variable name as an array name prior to usage in program. 2. Remove subscript specification from variable name.		X	
SUBSCRIPT OUT OF RANGE Cause: One of the subscripts specified for an array points to a non-existent location within the array. Action: Correct subscript specification.			X
SYNTAX ERROR Cause: A BASIC statement was syntactically incorrect. This is a general message covering all types of syntax errors. In some cases of syntax error, the message is more specific. Action: Correct statement syntax.	X	X	
TABS INPUT INCORRECT Cause: 1. More than four tab positions were specified in a TAB statement. 2. The tab positions specified in a TAB statement were not in increasing order. Action: Correct the TABS statement.		X	

	S	C	E
TERMINATE TAPE MODE WITH CONTROL D Cause: An attempt was made to follow paper tape input without the user inserting the control character D^c before subsequent operations. Action: Press the D^c key.		X	
TOO MANY FUNCTION DEFINITIONS Cause: More than 30 user-defined multi-line function definitions appeared in the same program. Action: Reduce the number of user-defined multi-line functions within the program to 30 or less.		X	
TOO MANY FUNCTION PARAMETERS Cause: 1. More than 30 argument dummies were used for the same function definition. 2. More calling arguments were specified than there were dummy arguments for a multi-line function definition. Action: Reduce the number of function parameters to 30 or less, or make the number of calling arguments consistent with the number of dummy arguments.		X	
TOO MANY SUBSCRIPTS Cause: More subscripts than are currently dimensioned for an array variable were specified in an array reference. Action: Correct the program so that all subscript specifications for an array have the same number of dimensions.			X
UNBALANCED PARENTHESES Cause: 1. A right parenthesis exists for which there is no companion left parenthesis. 2. A left parenthesis exists for which there is no companion right parenthesis. Action: Add the left or right parenthesis to the statement as appropriate.			X
USE OF UNINITIALIZED ARRAY Cause: No information has been stored in the space reserved for an array. Action: Insert MAT read or MAT initialization statement so that it occurs before use of the array in the program.		X	
USER FILE OVERFLOW Cause: Attempted output operation caused more data to be written than the file could hold. Action: Reduce the amount of data for output or write multiple files.			X
USER NAME NOT DEFINED Cause: An attempt was made to open shared file with an incorrect user name. Action: Correct user name in OPEN statement.			X

APPENDIX C. DIRECTORY OF BASIC STATEMENTS

This appendix contains a complete list of all the TENET BASIC statements discussed in this manual. Statements are described by format, function, execution mode (Immediate (I) or Program (P)) and the page on which they appear in this manual.

Throughout this appendix the following abbreviations are used:

a	array variable
c	constant (numeric or string)
dt	data type
e	expression
fn	file name
ln	line number
n	numeric constant
p	tab position
s	BASIC statement
sv	scalar variable
v	variable (scalar or array)
x$	string

ASSIGNMENT AND SEQUENCE CONTROL STATEMENTS

	Execution	Page No.
INTEGER v_i Declares the variables named as type integer.	I/P	4 - 2
REAL v_i Declares the variables named as type real.	I/P	4 - 2
DOUBLE v_i Declares the variables named as type double.	I/P	4 - 3
COMPLEX v_i Declares the variables named as type complex.	I/P	4 - 3
DOUBLE COMPLEX v_i Declares the variables named as type double complex.	I/P	4 - 3
STRING v_i Declares the variables named as type string.	I/P	4 - 4
DIM a_i Declares the dimensions of an array variable.	I/P	4 - 5
[LET] $sv_i = e$ Assigns a value to a variable.	I/P	4 - 7
DO ln [:ln] [, ln[:ln]] ... [,ln [:ln]] Specifies a statement and/or range of statements to be executed.	I/P	4 - 10
GOTO ln Unconditionally transfers control to a specified point in a program thus overriding sequential statement processing.	I/P	4 - 12

	Execution	Page No.
ON e GOTO ln_i Conditionally transfers program control to one of the specified line numbers.	I/P	4 - 13
IF e THEN $\left\{\begin{matrix} s_1 \\ ln_1 \end{matrix}\right\}$[ELSE $\left\{\begin{matrix} s_2 \\ ln_2 \end{matrix}\right\}$] Conditionally transfers program control or conditionally executes a statement.	I/P	4 - 14
FOR sv = e_1 $\left\{\begin{matrix} \text{TO} \\ \text{WHILE} \end{matrix}\right\}$ e_2[STEP e_3] Directs the repeated execution of a series of statements (program loop) terminated by a NEXT statement.	P	4 - 15
NEXT v_i Terminates a program loop and increments the value tested in the companion FOR statement.	P	4 - 15
DATA c_i Creates internal to the BASIC program a data list to be read by the READ statement.	P	4 - 19
READ sv_i Reads data specified by DATA statements.	I/P	4 - 20
RESTORE [ln] Resets DATA list pointer to a previous position.	I/P	4 - 21
REM text Inserts comments into a program listing.	I/P	4 - 22
† PAUSE Temporarily suspends program execution.	P	4 - 23
END Signals termination of a program or user defined multi-line function.	P	4 - 24, 5 - 4

† STOP may be used instead of PAUSE.

FUNCTION AND SUBROUTINE STATEMENTS

	Execution	Page No.
DEF[dt] FNname[(v_i)] [= e] Defines a function in a single line if expression specified; otherwise, begins a user defined multi-line function.	P	5 - 2, 5 - 4
RETURN [e] Returns the value produced by a multi-line function or subroutine to the main program.	P	5 - 4, 5 - 6
GOSUB ln Transfers program control to a set of statements constituting a subroutine.	I/P	5 - 6
ON e GOSUB ln_i Conditionally transfers program control to one of a selection of subroutines.	I/P	5 - 7

TERMINAL INPUT/OUTPUT STATEMENTS

INPUT sv_i Reads input data from the terminal.		
PRINT sv_i Writes output data on the terminal.	I/P	6 - 2
PRINT IN FORM x$:$sv_i$ Writes output data on the terminal in the specified format.	I/P	6 - 9

MATRIX STATEMENTS

MAT $a_1 = a_2$ Copies contents of one matrix into another matrix.	I/P	7 - 2
MAT $a_1 = a_2 + a_3$ Copies result of matrix addition into a matrix.	I/P	7 - 3
MAT $a_1 = a_2 - a_3$ Copies result of matrix subtraction into a matrix.	I/P	7 - 4

	Execution	Page No.
MAT $a_1 = a_2 * a_3$ Copies result of matrix multiplication into a matrix.	I/P	7 - 5
MAT $a_1 = (e)* a_2$ Copies the result of matrix multiplication by a scalar value into a matrix.	I/P	7 - 6
MAT a_1 = INV (a_2) Copies the result of a matrix inversion into a matrix.	I/P	7 - 7
MAT a_1 = TRN (a_2) Copies the result of a matrix transposition into a matrix.	I/P	7 - 8
MAT a = (e) Initializes all elements of a matrix.	I/P	7 - 9
MAT a = IDN Creates an identity matrix.	I/P	7 - 10
MAT READ a_i Assigns values from a DATA statement to an entire array,	I/P	7 - 11
MAT INPUT a_i Reads input data for entire arrays from the terminal.	I/P	7 - 12
MAT PRINT a_i Prints output of entire arrays at terminal.	I/P	7 - 14
MAT PRINT IN FORM x\$:a_i Writes array data on the terminal in the specified format.	I/P	7 - 16

FILE STATEMENTS

	Execution	Page No.
$\text{OPEN}\begin{bmatrix}\text{"fn"}\\ \text{"SCR"}\end{bmatrix}, e_1 \begin{bmatrix}\text{, SYMBOLIC}\\ \text{, BINARY}\end{bmatrix} \begin{bmatrix}\text{, OUTPUT}\\ \text{, INPUT}\\ \text{, IO}\end{bmatrix} \begin{bmatrix}\text{, SEQUENTIAL}\\ \text{, RANDOM}(e_2)\\ \text{, RANDOM}\end{bmatrix} \begin{bmatrix}\text{, NEW}\\ \text{, OLD}\end{bmatrix}$ Creates or activates a file.	I/P	8 - 4
CLOSE e_i Terminates file usage.	I/P	8 - 6
RESTORE FILE e_i Positions a file at its beginning of information.	I/P	8 - 7

	Execution	Page No.
APPEND FILE e_i Positions a file beyond last data item, ready to write next information set.	I/P	8 - 8
ERASE FILE e_1 FROM e_2 TO e_3 Deletes specified records from a random file.	I/P	8 - 9
ON ENDFILE (e) GOTO ln Specifies a statement to receive control if an end-of-file for file e is subsequently encountered.	I/P	8 - 10
ON ENDREC (e) GOTO ln Specifies a statement to receive control if a premature end-of-record for file e is encountered.	I/P	8 - 11
† [MAT] INPUT FROM e:v_i Reads [array] information from a sequential file.	I/P	8 - 12
† [MAT] INPUT FROM e_1 AT e_2:v_i Reads [array] information from a random file.	I/P	8 - 13
† [MAT] INPUT FROM e_1 [AT e_2] IN FORM x$:$v_i$ Reads [array] input data from the terminal in the format specified by the string x$.	I/P	8 - 14
†† [MAT] PRINT ON e:v_i Writes [array]output data onto a sequential file.	I/P	8 - 15
†† [MAT]PRINT ON e_1 AT e_2:v_i Writes [array]output data onto a random file.	I/P	8 - 17
†† [MAT]PRINT ON e_1 [AT e_2] IN FORM x$:$v_i$ Writes [array]output data onto a sequential (or random) file.	I/P	6 - 9

† READ may be used instead of INPUT.

†† WRITE may be used instead of PRINT.

EDITING STATEMENTS

ENTER ln [BY n] Causes automatic prompting of line numbers when creating or updating a source program.	I	9 - 3
LIST [ln [:ln] [, ... ln [:ln]]] Prints a listing of the program, a statement, or set of statements at the terminal.	I	9 - 4

	Execution	Page No.
RENUMBER [ln_1, ln_2, n] Renumbers program statements.	I	9 - 5
DEL [ETE] $\begin{matrix}\text{ALL}\\ \text{ln[:ln] [, ... ln [:ln]]}\end{matrix}$ Deletes all statements, single statements, or sets of statements.	I	9 - 6
ALTER ln Enables modification of a statement using special control characters.	I	9 - 7
TABS [p_1, p_2, p_3, p_4] Sets tab stops on terminal print line.	I	9 - 9

PROGRAM CONTROL STATEMENTS

RUN Causes program compilation and execution.	I	10 - 2
CONTINUE Resumes program execution at point of interruption.	I	10 - 3
SAVE "fn" $\begin{bmatrix}\text{,SYMBOLIC}\\ \text{,BINARY}\end{bmatrix}$ $\begin{bmatrix}\text{,NEW}\\ \text{,OLD}\end{bmatrix}$, ln_i Saves the program in the source area on a file for later use.	I	10 - 5
LOAD "fn" Retrieves a saved program (symbolic only) from storage and places it into the program source area.	I	10 - 5
LINK "fn" [, ln] Retrieves a saved program in binary from storage, places it in the program source area and begins executing it.	I/P	10 - 6
TAPE Signals paper tape input.	I	10 - 7
QUIT Causes an exit from the BASIC subsystem and return to EXECUTIVE control.	I	10 - 8

APPENDIX D. FUNCTIONS

TENET BASIC provides several pre-defined functions which enable the user to specify complex mathematical operations using simple expressions. A function is essentially a request for a routine or procedure to compute a value. A function may appear as part of an expression within a program statement.

Each standard function has the same format: the name of the function followed by one or more arguments (a number or arithmetic expression) separated by commas and enclosed in parentheses. The following abbreviations are used to define data types permissible with the function parameters:

Parameter	Data Type
I	Integer
R	Real
D	Double
C	Complex
DC	Double Complex
S	String

In some functions, two algorithms are given for simple and complex numbers. Data type conversion is specified by notation such as:

I	R	D	C	DC
R	R	D		

If integer or real type arguments are used in the function call, the value returned will be type real. A type double argument will return a type double value. A blank signifies that the function or individual function algorithm does not use the particular data type.

Frequently used mathematical constants:

π	3.1415 92653 58979 324
Degrees per radian	57.295 77951 30823 209
Radians per degree	.01745 32925 19943 2958
Ln2	.69314 71805 59945 309
Ln10	2.3025 85092 99404 568
LOG2	.30102 99956 63981 195
LOGe	.43429 44819 03251 828
e	2.7182 81828 45904 524
$\sqrt{2}$	1.4142 13562 37309 505
$\sqrt{10}$	3.1622 77660 16837 933

GENERAL MATHEMATICAL

FUNCTION	VALUE RETURNED	ALGORITHMS or EXAMPLES	I	R	D	C	DC
ABS (x)	Absolute value of x	ABS (x) = $\lvert x \rvert$ ABS (z) = $\sqrt{x^2 + y^2}$ z = x + iy	R	R	D	R	D
DEG (x)	Number of degrees equivalent to x (in radians)	DEG (x) = (x * 180/π)	R		D		
INT (x)	Greatest whole number $\leq$ x	INT (-6.35) = -7.0 INT (5.9) = 5.0 INT (0) = 0.0		R	D		
FIX (x)	Truncates fractional part of floating point x	FIX (-6.35) = -6.0 FIX (5.99) = 5.0 FIX (.8) = 0.0		R	D		
FP (x)	Fractional part of x	FP (x) = x - FIX (x) FP (-6.35) = -0.35 FP (5.99) = 0.99		R	D		
FRACT (x)	Absolute values of the fractional part of x	FRACT (-x) = ABS (FP(x)) FRACT (-6.35) = 0.35 FRACT (5.99) = 0.99		R	D		
SGN (x)	Sign of x	SGN (x) = 1 if x > 0 = 0 if x = 0 = -1 if x < 0	I	I	I		
SQRT (x)	Square root of x	SQRT (x) = $\sqrt{x}$ SQRT (z) = u + iv z = x + iy u = SQRT ($\frac{1}{2}$(r + x)) v = y / 2u r = ABS (z)	R	R	D	C	DC
COMP (x,y)	Comparison of x with y. (Complex values are compared on basis of vector magnitude as for ABS.) String comparison is discussed under string functions.	COMP (x,y) = 1 if x > y = 0 if x = y = -1 if x < y	I	I	I	I	I
ROUND (x)	x rounded to nearest whole	ROUND (x) = FIX (ABS(x) + 0.5) *SGN (x) ROUND (-6.5) = -7.0 ROUND (-7.3) = -7.0 ROUND (3.2) = 3.0		R	D		
MIN (x_1, x_2, ... x_n)	Minimum value of x_i	MIN (1,2,3.0) = 1.0 MIN (1,2,3) = 1 MIN (-1.1,2,3.01) = -1.1	I	R	D		

FUNCTION	VALUE RETURNED	ALGORITHM or EXAMPLES	I	R	D	C	DC
MAX (x_1, x_2, ... x_n)	Maximum value of x_i	MAX (1,2,3.0) = 3.0 MAX(-4.0,2,3) = 3.0	I	R	D		
RAD (x)	Radians equivalent to x (in degrees)	RAD (x) = (x * π)/180	R	R	D		
DBL (x)	Double precision equivalent of x	DBL (0.6E - 06) = 0.6D - 06	D	D	D	D	D
FLOAT (x)	Real (single precision floating point) equivalent of x	FLOAT (6) = 6.0	R	R	R	R	R

LOGARITHMIC AND EXPONENTIAL

LOG (x)	Natural log of x	LOG (x) = Ln_e (x) $x \geq 0$ LOG (z) = $\mathrm{Ln}_e\ r + i\theta$ $z = x + ry$ r = ABS (x) θ = ATAN (iy,x)	R	R	D	C	DC
LOG 10 (x)	LOG to the base 10 of x	LOG 10 (x) = LOG (x) * $\mathrm{Ln}_{10}e$	R	R	D	C	DC
EXP (x)	Natural exponential of x; e^x	EXP (x) = e^x where $\lvert x \rvert < 176.75$ EXP (z) = EXP (x) * (COS(y) + SIN (y)) $z - x + iy$	R	R	D	C	DC

CIRCULAR — TRIGONOMETRIC

ASIN (x)	Arcsine of x in radians	ASIN (x) = SIN^{-1} (x) $\lvert x \rvert \leq 1.0$ ASIN (z) = π-ASIN β -i LOG$\sqrt{\alpha + (\alpha^2 - 1)}$ $z = x + iy$ NOTE: See ACOS for β and α	R	R	D	C	DC
ACOS (x)	Arcosine of x in radians	ACOS (x) = COS^{-1} (x) $\lvert x \rvert \leq 1.0$ ACOS (z) = 2π- ACOS (β) + i LOG $\alpha + (\alpha^2 - 1)$ $z = x + iy$ $\alpha = \theta + \phi$ $\beta = \theta - \phi$ $\theta = \frac{1}{2}\sqrt{(x+1)^2 + y^2}$ $\phi = \frac{1}{2}\sqrt{(x-1)^2 + y^2}$	R	R	D	C	DC
ATAN (x)	Arctangent of x in radians	ATAN (x) = TAN^{-1} (x) ATAN (z) = $\pi + \frac{1}{2}$ ATAN(2x /(1-x^2 - y^2)) +$\frac{1}{2}$iLOG(θ / ϕ) $z^2 \neq -1$ $z = x + iy$	R	R	D	C	DC

FUNCTION	VALUE RETURNED	ALGORITHM or EXAMPLES	I	R	D	C	DC
ATAN (y, x)	Arctangent of y/x in radians	$\text{ATAN}(y,x) = \text{ATAN}(y/x)$ if $y>0,\ x>0$ $= \pi/2 - \text{ATAN}(y/x)$ if $y>0,\ x<0$ $= -\pi/2 - \text{ATAN}(y/x)$ if $y<0,\ x<0$ $= \text{ATAN}(y/x)$ if $y<0,\ x>0$ $=$ ERROR if $y=x=0$	R	R	D		
		$\text{ATAN}(y,x) = \text{ATAN}(z)$ $z = x + iy$				C	DC
COS (x)	Cosine of x in radians	$\text{COS}(x) = \text{COS}(x)$ $\lvert x \rvert \leq 1.0 \times 10^6$	R	R			
		$\text{COS}(x) = \text{COS}(x)$ $\lvert x \rvert \leq 1.1 \times 10^{15}$			D		
		$\text{COS}(z) = \text{COS}(x)\,\text{COSH}(y) + i\text{SIN}(x)\,\text{SINH}(y)$ $z = x + iy$				C	DC
SIN (x)	Sine of x in radians	$\text{SIN}(x) = \text{SIN}(x)$ $\lvert x \rvert \leq 1.0 \times 10^6$	R	R			
		$\text{SIN}(x) = \text{SIN}(x)$ $\lvert x \rvert \leq 1.1 \times 10^{15}$			D		
		$\text{SIN}(z) = \text{SIN}(x)\,\text{COSH}(y) + i\text{COS}(x)\,\text{SINH}(y)$ $z = x + iy$				C	DC
TAN (x)	Tangent of x in radians	$\text{TAN}(x) = \text{TAN}(x)$	R	R	D		
		$\text{TAN}(z) = \left(\frac{\text{SIN}(2x) + i\text{SINH}(2y)}{\text{COS}(2x) + \text{COSH}(2y)}\right)$ $z = x + iy$				C	DC

HYPERBOLIC

FUNCTION	VALUE RETURNED	ALGORITHM or EXAMPLES	I	R	D	C	DC
COSH (x)	Hyperbolic cosine of x	$\text{COSH}(x) = \frac{1}{2}(\text{EXP}(x) + \text{EXP}(-x))$	R	R	D		
		$\text{COSH}(z) = \text{COS}(iz)$ $z = x + iy$				C	DC
SINH (x)	Hyperbolic sine of x	$\text{SINH}(x) = \frac{1}{2}(\text{EXP}(x) - \text{EXP}(-x))$	R	R	D		
		$\text{SINH}(z) = -i\text{SIN}(iz)$ $z = x + iy$				C	DC
TANH (x)	Hyperbolic tangent of x	$\text{TANH}(x) = \frac{\text{SINH}(x)}{\text{COSH}(x)}$	R	R		D	
		$\text{TANH}(z) = -i\text{TAN}(iz)$ $z = x + iy$				C	DC
ACOSH (x)	Hyperbolic arccosine of x	$\text{ACOSH}(x) = \text{LOG}(x + (x^2 - 1)^{\frac{1}{2}})$ when $x \geq 1$	R	R	D		
		$\text{ACOSH}(z) = i\text{ACOS}(z)$ $z = x + iy$				C	DC
ASINH (x)	Hyperbolic arcsine of x.	$\text{ASINH}(x) = \text{LOG}(x + (x^2 + 1)^{\frac{1}{2}})$	R	R	D		
		$\text{ASINH}(z) = i\text{ASIN}(iz)$ $z = x + iy$				C	DC

COMPLEX

COMPLX(x, y)	Complex result formed from two real arguments	$\text{COMPLX}(x, y) = x + iy$				C	DC

FUNCTION	VALUE RETURNED	ALGORITHM or EXAMPLES	I	R	D	C	DC
IMAG (x)	Imaginary part of complex argument	IMAG (z) = y z = x + iy				R	D
REAL (x)	Real part of complex argument	REAL (z) = x z = x + iy				R	D
CONJ (x)	Complex conjugate of complex argument	CONJ (z) = x - iy z = x + iy				C	DC
PHASE (x)	Angle between $+\pi$ and $-\pi$	PHASE (z) = ATAN (y,x) z = x + iy				R	D
POLAR (x)	Polar form of complex argument	POLAR (z) = θ + iR z = x + iy θ = PHASE (z) R = ABS (z) Example: A = POLAR (B) ALPH = REAL (A) RADIUS = IMAG (A) The type of A and B is complex.				C	DC

STRING

FUNCTION	VALUE RETURNED
INDEX (s_1, s_2, e)	Starting position of the string s_2 within s_1. If not found, 0 is returned. The expression e, if specified, gives the position at which to start the search.
LEFT (s, e)	Substring of s starting from left and (the value of the expression) e characters in length. If e specifies a length greater than s, s is returned.
RIGHT (s, e)	Substring of s starting from the right and e characters in length.
LENGTH (s)	Number of characters in the string s.
VAL (s)	Numeric value of the string s. Any valid numeric form is acceptable. [VAL (6E2) is 600]
STR (e)	String equivalent of the numeric expression e.
SUBSTR (s, e_1, e_2)	String composed of a substring of s starting at the character specified by the numeric expression e_1 and for the number of characters specified by e_2. If the latter is omitted, a substring consisting of the remainder of s after the character. specified by e_1 will be generated.
SPACE (e)	String of blanks the length of the numeric expression e.
ASC (e)	Character equivalent of the expression e in ANSI code; generates the ANSI code number of the input character.

FUNCTION	VALUE RETURNED
CHAR (s)	Hexadecimal equivalent of the string s (1 character).
COMP (s_1, s_2)	Compares string_1 with string_2; the following values are returned: 1 if $s_1 > s_2$ 0 if $s_1 = s_2$ -1 if $s_1 < s_2$

MISCELLANEOUS

TREC (file no.)	Number of records written for the file specified.
TCHAR (file no.)	Total number of characters assigned to the file specified.
DET	Value of the determinant of the most recently inverted matrix.
POS	Current position of the teletype head.
TEL	0 if no input waiting at terminal. 1 if input waiting.

RANDOM NUMBER GENERATION (RND)

RND (e)

The RND function is a pseudo random number generator which produces a random number between 0 and 1 exclusive.

- The RND function requires a single argument that may be a positive, negative or zero value.
- If the argument is zero, the same number will be produced whenever the RND function is first executed in any program. Subsequent uses in the same program produce the next sequential random numbers.
- If the argument is a positive value, the same random number is generated each time the function is called using the same positive value. Different positive values generate different random numbers. A RND (+ e) followed by RND (0) will produce a series of random numbers starting with the random number associated with the numeric value of + e.
- If the argument is negative, a random number is generated from a value derived by combining the date and the internal clock of the computer. As the value of the negative argument has no bearing on the number it generates, all negative arguments are essentially equivalent. Whenever an RND function call appears with a negative argument, a different number is generated.

- Random numbers are expressed as 7-digit floating point values between 0 and 1 exclusive. Other ranges of random numbers may be generated by combining the RND statement in an arithmetic expression.

  Example:

  ```
      10 FOR I = 1 TO 10
      20 X = RND ( 0 )
      .
      .
      70 NEXT I
  ```

  Ten random values are generated for X within the range 0 to 1. If the RND function had a positive (non-zero) argument, the same single value would be repeatedly generated by X.

APPENDIX E. RESERVED WORDS

The following words are reserved by the BASIC subsystem and cannot be used as variable names.

ABS
ACOS
ACOSH
ALL
AND
APPEND
AS
ASC
ASIN
ASINH
ATAN
ATANH
BINARY
CHAR
CLOSE
COMP
COMPLEX
CONJ
CONTINUE
COS
COSH
DATA
DEF
DEG
DEL
DELETE
DET
DIM
DO
DOUBLE
EDIT
ELSE
END
ENDFILE
ENDREC
ENTER
EPS
EQU
ERASE
EXP
FIRST
FIX
FLOAT
FOR
FORM
FP
FRACT
FROM
GOSUB
GOTO
IF
IMAG
IMP
INDEX
INPUT
INT
INTEGER
INV
IO
LAST
LEFT
LENGTH
LET
LINK
LIST
LOAD
LOG
LOG10
MAT
MAX
MIN
MOD
NEXT
NOT
ON
OPEN
OR
OUTPUT
PAUSE
PHASE
PI
POLAR
POS
PRINT
QUIT
RAD
RANDOM
READ
REAL
REM
RENUMBER
RESTORE
RETURN
RIGHT
ROUND
RUN
SAVE
SEQUENTIAL
SIGN
SIN
SINH
SPACE
SQRT
STEP
STOP
STR
STRING
SUBSTR
SYMBOLIC
TAB
TABS
TAN
TANH
TAPE
TCHAR
TEL
THEN
TO
TREC
TRN
VAL
WHILE
WRITE
XOR

APPENDIX F. MODEL 33 TELETYPEWRITER TERMINAL

The Model 33 Teletypewriter Terminal used by the TENET Timesharing System consists of a control unit, keyboard, paper tape punch, and paper tape reader mechanism.

CONTROL UNIT

The configuration of the control unit (see Figure F-1) depends on whether the terminal is direct or acoustically coupled to the computer. The control unit on a direct-coupled terminal consists of only a LINE/OFF/LOCAL knob.

LINE If the control is in the LINE position, the terminal is on and connected to the computer, i.e., on-line.

OFF The terminal is off and incapable of communicating with the computer.

LOCAL The terminal is on but not connected to the computer. When the terminal is in this mode (off-line), operations such as punching paper tape may be performed.

Acoustically coupled terminals require a headset mechanism which is used to hold a telephone receiver. The LINE/OFF/LOCAL knob for acoustically coupled terminals is the same as for direct-coupled terminals except that when the knob is in the LINE position, the terminal is not automatically connected to the computer, but is capable of being connected to the computer. To establish a connection, the user must first turn the knob to the LINE position, phone the computer site, wait for a high-pitched tone, and place the telephone receiver into the headset mechanism.

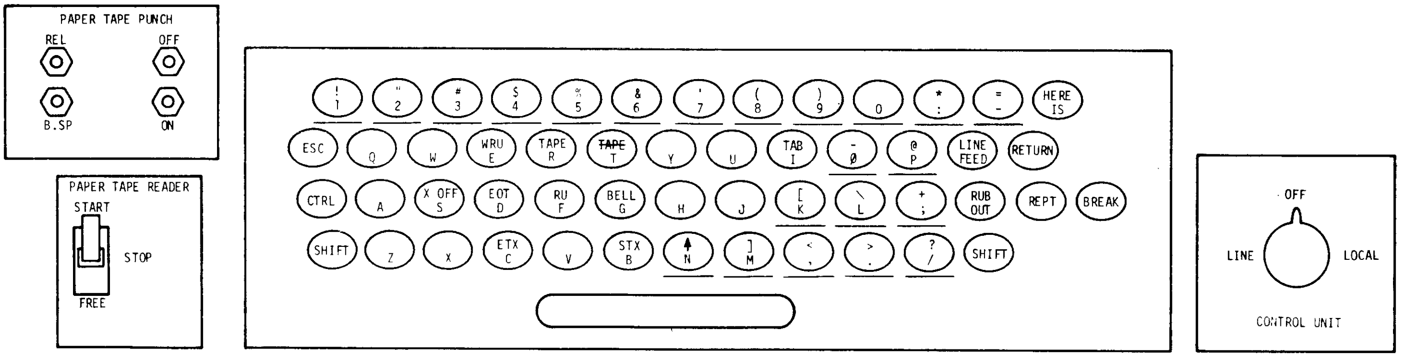

Figure F-1. Model 33 Teletypewriter Terminal Keyboard and Controls

THE KEYBOARD

The teletype keyboard (see Figure F-1) is used as a standard typewriter keyboard with the exception of the keys described below.

SHIFT

Only those keys underlined in Figure F-1 have a shift position. The shift key is non-locking and must be depressed when typing. Characters are printed as they appear on the upper half of the key. However, on some terminals:

K shift is not marked but appears as a [
L shift is not marked but appears as a \
M shift is not marked but appears as a]

The keyboard locks whenever an attempt is made to use the shift with a key with no shift position.

CTRL (Control)

Any alphabetic character may be pressed in conjunction with CTRL. (CTRL is non-locking.) The resulting control character is not always printed at the terminal. Characters used with the CTRL are discussed in section 9 of this manual. They are designated by the subscript ^{c}. Control characters not recognized by the system are ignored but cause the bell to ring once for each ignored character.

ESC or ALT MODE

This key terminates any input/output operation in progress and causes a program interrupt.

LINE FEED

Each time the Line Feed key is pressed, the paper is advanced one line. (When the terminal is connected to the computer, the system automatically generates a Carriage Return for each Line Feed.)

RETURN or
CARRIAGE RETURN

This key positions the print head at the beginning of a line. When the terminal is connected to the computer (on-line), the system automatically generates a Line Feed for every Carriage Return.

RUBOUT

This key is used to delete characters on paper tape. It is always ignored, but does not ring the bell.

REPT (Repeat)

This key causes any character key pressed while the REPT is pressed to be repeated for as long as the REPT key is pressed.

HERE IS

Transmits and prints whatever is on the answerback drum.

BREAK

On a direct coupled terminal this key is ignored; on an acoustically coupled terminal, pressing this key disconnects the terminal from the computer.

PAPER TAPE PUNCH

The paper tape punch is used to produce a perforated tape which can be used as input to the computer instead of input from the teletypewriter terminal keyboard.

OFF and ON — The ON button initiates and continues paper tape punching until the OFF button is pressed. Information punched on paper tape is also printed at the terminal.

REL — The release button frees the paper tape so that the user can manually pull blank tape through the punch mechanism.

BKSP — The backspace button moves the paper tape backwards one frame each time the button is pressed. It is used in conjunction with the RUBOUT key to delete paper tape entries.

PREPARING PAPER TAPE OFF-LINE

The user can save information in paper tape form. The TAPE command, for example, is used by the BASIC subsystem to access programs saved on paper tape. Similarly, the contents of data files may be punched on paper tape and later read into a file by the EXECUTIVE. To prepare paper tape off-line, turn the terminal control dial to LOCAL, depress the Punch ON button, and enter data from the keyboard. Since this is an off-line operation, the user will find it convenient to follow a Line Feed with a Carriage Return and vice versa. When paper tapes are read by the system, Line Feed /Carriage Return and Carriage Return /Line Feed combinations are treated as Line Feed and Carriage Return respectively.

The control keys $\textcircled{A}^{c}$ (delete previous character) and $\textcircled{Q}^{c}$ (delete current line) may be used to edit paper tape entries.

PAPER TAPE READER

START — This control initiates and continues paper tape reading.

STOP — This key terminates paper tape reading.

FREE — This control frees the reader mechanism so that the tape can be pulled through the reader manually.

GLOSSARY

ALGORITHM A set of prescribed rules or procedures for the solution of a problem.

ALPHANUMERIC Pertaining to the character set which includes the 26 alphabetic characters, the 10 numeric characters, and any special characters ($, % and @ in TENET BASIC).

ANSI An abbreviation for the American National Standards Institute.

ARRAY A multi-element variable. Same as matrix.

BINARY OPERATOR An operator which operates on two quantities (constants, variables, or the evaluated combination of these), e.g., * , / .

CENTRAL PROCESSOR The unit of a computer system which controls the interpretation and execution of instructions, exclusive of peripheral devices such as teletype terminals.

COMPILE To translate a symbolic or source program such as BASIC into the binary machine language for execution by the central processor.

CONCATENATION Joining sets of information end-to-end.

CONSTANT A program element whose value remains unchanged through the programming process. A constant may be a numeric constant or literal text (string).

DATA A general term pertaining to any numeric values or character strings which constitute information which can be processed or generated by a program.

DEBUG To detect, locate, and correct program errors.

DOUBLE PRECISION Pertaining to the use of two computer words instead of one to represent a real value. This yields increased precision (16 significant digits instead of 7 digits) and increased range (up to $10^{\pm 76}$ instead of $10^{\pm 19}$).

EXECUTE To carry out an instruction or run a program.

EXPRESSION Any variable, constant, or combination of these joined by operators.

FILE A structured set of information maintained on a mass storage device external to computer memory.

FIXED-LENGTH RECORD A record whose length is prescribed by the user before the record is created.

FULL DUPLEX
Pertaining to a method of transmitting information from a teletype terminal whereby a character is entered from the teletype keyboard (but not printed), transmitted to the computer, and transmitted back to the terminal from the computer for printing. This mode of transmission ensures that the character printed is that which the computer received.

IMMEDIATE EXECUTION
Pertaining to statements which are not prefaced by line numbers and are executed immediately upon input.

INPUT
To transfer information from an external source (such as the teletype, paper tape, or disc file) to the computer's memory.

I/O
An abbreviation for input/output.

LEAST SIGNIFICANT DIGIT
The rightmost digit of a number.

LOGICAL OPERATOR
An operator which operates on logical values (AND, OR, EOR, NOT, etc).

LOOP
A sequence of instructions that are executed repeatedly until some terminal condition is met.

MATRIX
A multi-element variable. Same as an array.

MOST SIGNIFICANT DIGIT
The leftmost (non-zero) digit of a number.

OFF LINE
Pertaining to peripheral devices not under direct control or connection to the computer.

ON LINE
Pertaining to peripheral devices under direct control or connection to the computer.

OUTPUT
To transfer information from the computer's memory to an external source (such as the teletype terminal, paper tape, or disc file).

PRIVATE FILES
Files which are saved, but not accessible to any users other than the file's creator.

PROGRAM
A set of statements or instructions which direct the solution of a problem.

PROGRAM EXECUTION
Pertaining to statements which are prefaced by line numbers and are executed only within the context of a program.

PROGRAM FILE
A file containing a saved program used as input to a compiler or to the computer for processing.

(PUNCHED) PAPER TAPE
A paper tape on which a pattern of holes is used to represent data.

RANDOM ACCESS FILES
Files whose records may be accessed in any order.

RECORD
Any unit of data; in TENET BASIC, information bounded by a prompt character and carriage return.

RELATIONAL OPERATOR

An operator that compares one quantity with another, e.g., >, <, =. Evaluation is true (=1) or false (=0).

RUN

To execute a program.

SAVED FILES

Files which are stored on disc by explicit user command. They are maintained until explicitly removed by command of the file's creator.

SCALAR VARIABLE

Representing a single quantity, numeric value,or string.

SEQUENTIAL ACCESS FILES

Files which may be accessed only in the way in which they were written, i.e., sequentially.

SHARED FILES

Files which are saved and may be accessed by users other than the file's creator.

SOURCE LANGUAGE

A symbolic language input to a translation process, such as the BASIC subsystem.

STRING

A constant consisting of any series of characters enclosed by a set of single or double quotation marks.

SUBROUTINE

A subset of the main program which may be called repeatedly from the main program to perform a task.

SUBSCRIPTED VARIABLE

Representing a single element of a multi-element structure (array or matrix).

SYNTAX

Pertaining to the structure of a language statement.

TIMESHARING SYSTEM

A system which supports multiple users, as though each user were the only one using the computer.

UNARY OPERATOR

An operator which operates on only one quantity (constant, variable, or evaluated combination of these).

VARIABLE

A quantity whose value was previously defined, is not yet defined, or may change through the course of a program.

VARIABLE-LENGTH RECORD

A record whose length is not prescribed by the user, but determined by its actual content.

INDEX

DOCUMENT REVIEW FORM

Your comments concerning this document help us produce better documentation for you.

General Comments

	Yes	No
Is the material easy to read ?	□	□
well organized ?	□	□
accurate ?	□	□
complete ?	□	□
well illustrated ?	□	□
suitable for your needs ?	□	□

How do you use this document ?

□ As an introduction to the subject
□ For additional knowledge
□ For continual reference
□ Other

Specific Clarifications and/or Corrections

Reference	Page No.

This form should not be used as an order blank. Requests for copies of publications should be directed to the TENET sales office serving your locality.

FOLD

FIRST CLASS
PERMIT NO. 480
SUNNYVALE, CA.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A

POSTAGE WILL BE PAID BY

TENET
927 THOMPSON PLACE
SUNNYVALE, CA. 94086

ATTENTION: SOFTWARE PUBLICATIONS

FOLD

FROM: NAME______________________________

POSITION______________________________

ADDRESS______________________________

FUNCTIONS

General Mathematical

ABS(x)	D-2
DEG(x)	D-2
INT(x)	D-2
FIX(x)	D-2
FP(x)	D-2
FRACT(x)	D-2
SGN(x)	D-2
SQRT(x)	D-2
COMP(x,y)	D-2
ROUND(x)	D-2
MIN(x_1,x_2,...x_n)	D-2
MAX(x_1,x_2,...x_n)	D-3
RAD(x)	D-3
DBL(x)	D-3
FLOAT(x)	D-3

Logarithmic

LOG(x)	D-3
LOG10(x)	D-3

Exponential

EXP(x)	D-3

Circular –Trigonometric

ASIN(x)	D-3
ACOS(x)	D-3
ATAN(x)	D-3
ATAN(x,y)	D-4
COS(x)	D-4
SIN(x)	D-4
TAN(x)	D-4

Hyperbolic

COSH(x)	D-4
SINH(x)	D-4
TANH(x)	D-4
ACOSH(x)	D-4
ASINH(x)	D-4

Complex

COMPLX(x,y)	D-4
IMAG(x)	D-5
REAL(x)	D-5
CONJ(x)	D-5
PHASE(x)	D-5
POLAR(x)	D-5

String

INDEX(s_1,s_2,e)	D-5
LEFT(s,e)	D-5
RIGHT(s,e)	D-5
LENGTH(s)	D-5
VAL(s)	D-5
STR(e)	D-5
SUBSTR(s,e_1,e_2)	D-5
SPACE(e)	D-5
ASC(e)	D-5
CHAR(s)	D-6
COMP(s_1,s_2)	D-6

Miscellaneous

TREC(file no.)	D-6
TCHAR(file no.)	D-6
DET	D-6
POS	D-6
TEL	D-6
RND(e)	D-6

TENET, Inc. / 927 Thompson Place / Sunnyvale, California 94086 / (408) 245-8751