Price: $4.75

TYMSHARE MANUALS

REFERENCE SERIES

SUPER BASIC

May 1971

TYMSHARE, INC.
525 UNIVERSITY AVENUE, SUITE 220
PALO ALTO, CALIFORNIA 94301

REGIONAL OFFICES

Los Altos, California ■ Inglewood, California ■ Chicago, Illinois ■ Englewood Cliffs, New Jersey

DISTRICT OFFICES

District sales offices located in major cities across the United States.

RM3

CONTENTS

Page

Page

TYMSHARE REFERENCE MANUAL
SYMBOL CONVENTIONS

The symbols used in this manual to indicate Carriage Return, Line Feed, and ALT MODE/ESCAPE are as follows:

Carriage Return: ↵

Line Feed: ↴

ALT MODE/ESCAPE: ⊕

Action At The Terminal

To indicate clearly what is typed by the computer and what is typed by the user, the following color code convention is used.

Computer: Black **User: Red**

Control Characters

Control characters are denoted in this manual by a superscript c. For example, D^c denotes Control D. The method for typing a control character depends on the type of terminal used. Consult the literature for your particular terminal.

Note On Spacing In Examples

Because this manual is set in type with characters of varying width, the spacing in some of the examples may not appear exactly as on the terminal, where all characters are the same width. If the spacing in an example appears misleading, this general rule will be helpful:

The number of blanks or spaces printed can usually be determined by counting the print positions (characters) in the line or lines above.

INTRODUCTION

Tymshare SUPER BASIC represents for the first time a truly conversational language incorporating features for both business and scientific applications.

SUPER BASIC provides a powerful yet simple set of commands and diagnostics that allow the new user to learn the language in a few hours and yet give the experienced programmer the most extensive list of features ever included in a single language.

A few of the outstanding features of SUPER BASIC are:

- String manipulation.
- Matrix arithmetic.
- Complex, logical, and double precision variables.
- Octal constants, binary operators and functions.
- Formatted input and output, including picture formatting as well as more advanced formatting for maximum flexibility.
- Conditional modifiers.
- Direct commands.
- Complete built-in editing.
- Random files.
- Binary program files.
- Program optimizer to allow fastest possible execution speed.

This manual contains a complete description of all the features of SUPER BASIC. Section 1 describes a subset of commands which, once learned by the beginning user, will enable him to write complete SUPER BASIC programs and run them on the Tymshare system.

Though written in a tutorial manner, Section 1 is well suited for reference. The rest of the commands are included in Sections 2 through 10 which, while written more as reference material than as a tutorial guide, explain each feature in such a way that the user will be able to learn easily any new and unfamiliar material.

Section 2 contains a description of the arithmetic and logical features of SUPER BASIC.

The ability to manipulate text with string variables and functions is an outstanding feature of SUPER BASIC. Section 3 describes these string manipulation features.

Replacement and control statements are described in Section 4; declaration statements in Section 5. Input and output statements are discussed in Section 6. Files — sequential, random, and command — are described in Section 7. Subroutines and programmer-defined functions are detailed in Section 8.

Section 9 describes how SUPER BASIC programs are entered, stored, and executed on the Tymshare system. This section also includes the commands for editing SUPER BASIC statements. Section 10 describes tools to assist the SUPER BASIC programmer in debugging his programs.

Section 11 contains some sample programs written in SUPER BASIC and executed on the Tymshare system.

The appendices provide a summary of the entire SUPER BASIC language, an alphabetic list of all SUPER BASIC statements and their characteristics, and a summary of the EXECUTIVE.

We recommend that the new user of SUPER BASIC complete the *Tymshare SUPER BASIC Instant Series Manual* and appropriate workbook in full before proceeding beyond Section 1 of this manual.

SECTION 1
A SUPER BASIC PRIMER

AN EXAMPLE

Suppose you want to write a SUPER BASIC program that will:

1. Request that you type in five numbers,
2. Add the numbers (if the result is zero, print the message SUM IS ZERO and stop),
3. Find the average (or mean) of the numbers,
4. Print out the sum and the mean,
5. Ask for five new numbers and repeat the cycle.

The simple program that solves this problem illustrates several elementary features and commands of SUPER BASIC which will be explained below. This is the program:

```
10 PRINT "TYPE FIVE NUMBERS"
20 INPUT A,B,C,D,E
27 S=A+B+C+D+E
32 IF S=0 THEN 70
45 M=S/5
50 PRINT S,M
60 GO TO 10
70 PRINT "SUM IS ZERO"
```

Before explaining this program step by step, we should first note some general SUPER BASIC conventions.

LINE NUMBERS

All lines in the program begin with a number. These numbers identify the lines in the program, which are called statements, and specify the order in which the statements are to be executed. You can therefore type the program in any order provided that the statements are numbered in the order in which they are to be executed. Before the program is run SUPER BASIC sorts the statements into the order specified by their line numbers. *NOTE: Line numbers must be integers from 0 to 999999.*

LINE LENGTH

All statements in the sample program contain fewer than 72 characters (the maximum number of characters that may be typed across the page on most terminals). Pressing the Line Feed key while a statement is being typed will continue the statement on the next line. A statement may be continued for several lines provided that the maximum limit of 256 characters is not exceeded. At the end of each entire statement, a Carriage Return must be typed. For example, the last statement in the program could have been typed as:

```
>70 PRINT "SUM IS ↵
 ZERO" ↵
```

NOTE: Spaces have no significance in SUPER BASIC except when they are included within quote marks (as in the above statement). Thus, spaces may be eliminated from all but these instances if you are not concerned with the readability of the printed copy.

STATEMENTS

Indirect Statements

All the statements in the program above are called indirect statements. Any statement that begins with a line number is indirect; that is, the instruction or command in such a statement is not executed when it is typed, but is executed when the running program reaches the statement in normal sequence.

Direct Statements

Direct statements do not begin with line numbers and are executed as soon as they are typed in. Direct statements cannot be saved as part of a program as indirect statements can.

Some commands can be executed indirectly only, some directly only, and others may be used either way. For example, if GO TO 10 had been typed in our sample program without the line number 60, SUPER BASIC would have executed the command immediately by transferring to statement 10 and continuing execution from there. All this would have happened before you could have typed in any more statements. To find out if a command can be executed indirectly, directly, or both, see *Appendix A*.

We are now ready to explain this program step-by-step.

PRINT "text"

When SUPER BASIC encounters the first statement,

```
10 PRINT "TYPE FIVE NUMBERS"
```

the text included within the double quote marks is printed on the terminal. In this case the text is an in-

struction to the person who is running the program: he is instructed to type five numbers. The text also could have been enclosed in single quote marks.

INPUT variable list

The INPUT command in the second statement,

20 INPUT A,B,C,D,E

will, when executed, cause SUPER BASIC to print a question mark followed by a space and wait for five numbers to be typed in.[1] The letters A through E in this statement are called variables. Their purpose is to store values that will be used later in a computation. The first number typed will be stored in A, the second in B, and so on. Just as the comma is used to separate variables in the INPUT statement, it is also used to separate the values when they are typed in.[2] This will be shown later in an illustration of the actual run of the sample program.

ASSIGNMENT STATEMENT

Statement 27,

27 S = A+B+C+D+E

is called an assignment statement. This statement is similar to the others in the program which begin with a command word, except that in this case the word need not be typed. The optional word which may be included in an assignment statement is LET. For example, statement 27 could have been typed as

27 LET S = A+B+C+D+E

The function of the assignment statement is to compute the value of the expression on the right of the = and assign that value to the variable on the left. *NOTE: An expression may not be typed to the left of the =; for example, A+B = C is not a valid statement.*

Since the = means "is assigned the value of" rather than "is equivalent to", the following is a valid assignment statement:

15 X = X+1

If the value of X were 5 before this statement was executed, the statement would set X to 6.

The rules which govern naming variables and typing expressions correctly are included under *FUNDAMENTAL CONCEPTS OF SUPER BASIC*, page 5.

IF condition THEN line number

In statement 32,

32 IF S = 0 THEN 70

we test to see if the value of S is zero. If it is, then this statement will cause SUPER BASIC to go to line 70 where it prints the message SUM IS ZERO and stops since there are no more statements to execute. If S is not zero, SUPER BASIC will continue to the next statement in sequence.

45 M = S/5

This assignment statement calculates the mean and assigns the result to M. Note that since the sum of the five numbers has been calculated previously and assigned to the variable S, we do not need to repeat the computation of S in this statement.

PRINT variable list

When SUPER BASIC encounters the next statement,

50 PRINT S,M

the values which were computed and assigned to S and M are printed. A comma is used to separate the variable names.

Since any number or expression also may follow the PRINT command, we could have omitted assignment statement 45 and typed the PRINT statement 50 as:

50 PRINT S, S/5

If SUPER BASIC were to encounter this statement, it would print S, compute S/5 and then print that result.

GO TO line number

The IF statement in this program causes a conditional transfer; that is, SUPER BASIC will transfer to another part of the program provided that a certain condition is true. The GO TO command however, transfers to another statement unconditionally. Thus, when

60 GO TO 10

is executed, SUPER BASIC goes to line 10 and requests new values for A,B,C,D, and E.

Note the importance of certain statements in the program.

- What would happen if we were to omit statement 50? SUPER BASIC would solve for S and M but would never print the results; the solution would remain the secret of the computer.
- Suppose we omitted statement 32. Then, if S were zero, SUPER BASIC would not print SUM IS ZERO and stop as we had specified. Instead, the mean would be calculated (also as zero), the sum and mean would be printed, and five more numbers would be requested.
- If we were to omit statement 60 (the unconditional transfer), SUPER BASIC would, after printing the values of S and M, print the message SUM IS ZERO and stop.

1 - If you type fewer numbers than required, SUPER BASIC types another question mark and waits for the rest of the input.

2 - A Carriage Return, Control D, or spaces may also be used to separate the values when typed.

RUNNING THE EXAMPLE PROGRAM

The complete procedure for entering and running the example program, then leaving the Tymshare system is illustrated and explained below.

```
-SBASIC ↵
```
After the log in procedure is completed, the system is ready. The dash indicates that you are in the EXECUTIVE and can call SUPER BASIC by typing SBASIC↵. The > indicates that SUPER BASIC is ready and you may begin to type the program statements.

```
>10 PRINT "TYPE FIVE NUMBERS" ↵
>20 INPUT A,B,C,D,E ↵
>27 S = A+B+C+D+E ↵
>32 IF S = 0 THEN 70 ↵
>45 M = S/5 ↵
>50 PRINT S,M ↵
>60 GO TO 10 ↵
>70 PRINT "SUM IS ZERO" ↵
>RUN ↵
```
The direct command RUN ↵ causes SUPER BASIC to execute the program.

```
TYPE FIVE NUMBERS
? 10,20,30,40,50 ↵
```
Five numbers are typed, separated by commas. A Carriage Return is typed after the last number.

```
 150          30
```
The sum is 150; the mean is 30.

```
TYPE FIVE NUMBERS
? 13,-7,-23,19,8 ↵
```
SUPER BASIC again requests five numbers.

```
 10           2
```
This time the results are 10 and 2.

```
TYPE FIVE NUMBERS
? 40,25,-50,15,-30 ↵
SUM IS ZERO
```
The sum of the next five numbers is zero. SUPER BASIC prints the specified message and stops.

```
>QUIT ↵
-LOGOUT ↵
```
The QUIT command (which may be abbreviated as Q) returns you to the EXECUTIVE where you can leave the system by typing LOGOUT ↵.

NOTE: We could have interrupted the execution of the program at any time by pressing the ALT MODE/ESCAPE key in reply to the INPUT question mark.

FUNDAMENTAL CONCEPTS OF SUPER BASIC

NUMBERS

How To Type Numbers Into SUPER BASIC

Numbers may be typed into SUPER BASIC in three ways:

- Integer format (whole numbers without a decimal point).
- Decimal format (numbers containing a decimal point).
- E format. The letter E means "times ten to the power of". For example, -53×10^9 can be typed as -53E9, and the number .00000000000063 (in which twelve zeroes follow the decimal point) can be typed as .63E-12. The E notation cannot stand alone; thus, 1000 may be typed as 10E2 or 1E3 but not as E3.

SUPER BASIC will accept up to eleven significant digits; any number containing more significant digits will be rounded to eleven. *EXCEPTION: A variable declared DOUBLE (double precision) can store up to seventeen significant digits. This feature is discussed in* **DOUBLE PRECISION ARITHMETIC**, *page 26.*

The largest number that SUPER BASIC will accept is .57896044618E77.

Note that the following are not numbers in SUPER BASIC: 1/2, $\sqrt{4}$, $(5/6)^{17}$. They are expressions which SUPER BASIC must compute into a number of acceptable form. Such expressions may not be used as data input to a program.

How SUPER BASIC Prints Numbers

SUPER BASIC ordinarily will print numbers as follows:[1]

- Numbers are stored internally in SUPER BASIC with eleven significant digits (seventeen, for double precision) but are rounded to eight digits when printed.
- If the absolute value of the number[2] is less than .1 or greater than or equal to 100,000,000, the number will be printed in E format. Otherwise, it will be printed as an integer or decimal number.
- Trailing zeroes after a decimal point are not printed.

To illustrate these rules, we will use the PRINT command directly, that is, without a line number so that SUPER BASIC will execute the command immediately.

```
>PRINT .076, -568905117↵
  7.6E-02        -5.6890512E+08

>PRINT -.600174172, 63.810↵
-.60017417          63.81

>PRINT 6E7, 6E8↵
 60000000          6E+08
```

VARIABLES

The purpose of a variable is to be assigned or to store a single value that will be used in some computation or will be printed as a solution. A variable is so called because its value may be changed.

Variable Names

A variable can be named in one of three ways:[3]

- Any letter from A to Z.
- Any letter followed by any digit from 0 to 9.
- Any letter followed by the dollar sign, $.

Some acceptable variable names are:

```
Z    B2    M4    I$
```

and some unacceptable names are:

```
1C    PC    A27    INT
```

The VAR = ZERO Command

A variable ordinarily must be defined (assigned a value either by appearing on the left side of an assignment statement or in an INPUT[4] statement) before it can be referred to in a SUPER BASIC statement. Referring to an undefined variable will cause an error message to be printed unless the VAR = ZERO command has been executed previously. This command automatically assigns the initial value of zero to all variables which would otherwise be considered as undefined. For example:

```
>10 VAR = ZERO↵
>20 PRINT "TYPE A"↵
>30 INPUT A↵
>40 PRINT A,B↵
>RUN↵
TYPE A
? 6↵
 6              0
```

The user typed in the value of 6 for the variable A. B was never defined, but because of the VAR = ZERO command in line 10, B's initial value was set to zero. If line 10 had been omitted, the PRINT A,B statement would have caused SUPER BASIC to print A and then an error message indicating that B was never defined.

The VAR = ZERO command also can be executed directly. Note that the RUN command ordinarily ignores any direct commands that might have been given previously and executes only those statements preceded by line numbers. The direct VAR = ZERO command is an exception; it will not be ignored when the RUN command is given. For example:

```
>10 X = 15↵
>20 PRINT X,Y↵
>RUN↵
 15
ERROR IN STEP 20:
VARIABLE HAS NO VALUE
>VAR = ZERO↵
>RUN↵
 15              0
```

Only the value of X was assigned in line 10. The direct VAR = ZERO command, since it was given before the RUN, caused the value of Y to be set to 0.

1 - You can control the format in which SUPER BASIC will print numbers. See *FORMATTING WITH IMAGE*, page 51, and *FORMATTING WITH FORM*, page 56.
2 - Absolute value simply means: for positive numbers, the number itself, for negative numbers, the number without its minus sign.
3 - Subscripted variables are discussed under *MATRIX ARITHMETIC*, page 20.
4 - Or READ statement, page 10.

The VAR = UNDEF Command

This command nullifies the VAR = ZERO command. It affects only those variables which would be undefined if the VAR = ZERO command had never been given by once again declaring those variables to be undefined. For example:

```
>10 VAR = ZERO↵
>20 C = 12↵
>30 PRINT C,D↵
>40 PRINT "NOW, 'VAR = UNDEF'"↵
>50 VAR = UNDEF↵
>60 PRINT C,D↵
>RUN↵
  12                0
NOW, 'VAR = UNDEF'
  12
ERROR IN STEP 60:
VARIABLE HAS NO VALUE
```

After the VAR = UNDEF command, C is still 12 but D is undefined, as though the VAR = ZERO command had never been given.

If we had assigned any value to D before giving the VAR = UNDEF command, D would not have been undefined by this command. Thus, if we were to insert 35 D = 5 into the above program, VAR = UNDEF would have no effect and 5 would print as the value of D. Similarly, 35 D = 0 would cause 0 to print as the value of D (since VAR = UNDEF undefines only those variables that are zero because of the VAR = ZERO command).

ARITHMETIC EXPRESSIONS

Arithmetic expressions are formed by combining numbers and/or variables with arithmetic operators as in ordinary mathematical formulas.

There are eight arithmetic operators in SUPER BASIC:

Symbol	Meaning	Example
↑	Exponentiation	X↑3 (=X^3)
-	Unary minus	-2↑2 (=-(2^2)=-4)
MOD	Modulo[1]	9 MOD 7 (=2)
*	Multiplication	3*B (=3 X B)
/	Division	3/2 (=1.5)
DIV	Division (integer result)[2]	3 DIV 2 (=1)
+	Addition	8+F1
-	Subtraction	C$-5

Parentheses often are required in SUPER BASIC arithmetic expressions where they might not be needed in ordinary mathematical notation. For example, if you type $\frac{A+B}{C}$ as A+B/C in SUPER BASIC, the expression will be interpreted as $A+\frac{B}{C}$. This is because SUPER BASIC performs division before addition, unless parentheses are used to denote otherwise. Thus, $\frac{A+B}{C}$ must be typed as (A+B)/C.

The order in which SUPER BASIC will perform arithmetic operations is as follows:[3]

1. **Whatever is enclosed in parentheses** will be computed first according to rules 2 through 6 below. When sets of parentheses appear within other sets of parentheses, the innermost set is evaluated first, then the next set, and so on.
2. **Exponentiation.**
3. **Unary minus.** Thus, if the expression is -2↑2, 2↑2 is computed first, and the value of the expression is -4.
4. **Modulo operator.** Thus, 15 MOD -6/2 is interpreted as (15 MOD -6)/2 and not 15 MOD -3.
5. **Multiplication and division.** If *, /, and DIV appear in the same expression, SUPER BASIC calculates from left to right; that is, 3/B↑2*C is equivalent to $(3/B^2)\times C$.
6. **Addition and subtraction.** If + and - appear in the same expression, SUPER BASIC calculates from left to right (same as *, /, and DIV above).

MATHEMATICAL FUNCTIONS

A number of standard mathematical functions are available in SUPER BASIC. Each one has the same form: the name of the function followed by the argument (a number or an arithmetic expression) enclosed in parentheses. Some of these functions are listed in the chart below.[4]

Function	Value Returned
ABS(X)	Absolute value of X
SQR(X) *or* SQRT(X)	Positive square root of X
Trigonometric (all angle arguments and results are in radians).	
SIN(A)	Sine of A
COS(A)	Cosine of A
TAN(A)	Tangent of A
ASIN(X)	Angle whose sine is X

Table continued on next page.

1 - This standard mathematical operator is defined as follows: Y MOD Z = Y-Z*FIX(Y/Z). FIX is explained on page 17.
2 - DIV is defined as Y DIV Z = INT(Y/Z). INT is explained on page 17.
3 - See page 31 for a complete table of precedence for SUPER BASIC operators.
4 - Additional mathematical functions of SUPER BASIC are described under *ARITHMETIC FUNCTIONS*, page 17.

Function	Value Returned
ACOS(X)	Angle whose cosine is X
ATN(X) *or* ATAN(X)	Angle whose tangent is X (range $-\pi/2$ to $+\pi/2$)
ATN(Y,X) *or* ATAN(Y,X)	Angle whose tangent is Y/X (range $-\pi$ to $+\pi$)
SINH(A)	Hyperbolic sine of A
COSH(A)	Hyperbolic cosine of A
TANH(A)	Hyperbolic tangent of A
Logarithmic	
LOG(X)	Natural (base e) logarithm of X
LGT(X) *or* LOG10(X)	Common (base 10) logarithm of X
LOG2(X)	Base 2 logarithm of X
Exponential	
EXP(X)	Natural exponential of X, e^X
EXP2(X)	2^X
Functions with no argument.	
PI	π, 3.1415926535
DPI	Double precision[1] π, 3.1415926535897932

These functions may be included in any expression; for example, all of the following are acceptable in SUPER BASIC:

```
Z$-EXP (X1+LOG(5/X1))
SQR (SIN(R)↑2+COS(Q)↑2)
LOG(N*X-SIN(PI/N))
```

RELATIONAL EXPRESSIONS

A relational expression is one which compares one value to another (where the values may be represented by variables or expressions) using relational operators. A complete list of these operators can be found on page 29; the most commonly used are listed below.

Symbol	Meaning
<	Less than
<=	Less than or equal to
=	Equal to
>=	Greater than or equal to
>	Greater than
# or <>	Not equal to

A relational expression commonly occurs in an IF statement (where the THEN part of the statement will be executed only if the specified relation is true). For example,

```
32 IF S = 0 THEN 70
```

causes a transfer to statement 70 only if the value of S is zero; that is, if the expression S = 0 is true. If S = 0 is false, SUPER BASIC will continue to the next statement.

The following are acceptable relational expressions:

```
X>5
A#B
Z$<=Y↑K+EXP(Z)
ABS(C3)=1
```

USING LOOPS IN A PROGRAM: THE FOR AND NEXT STATEMENTS

Perhaps the single most important programming idea is the loop. While we can write useful programs in which each statement is performed only once, such programs do not make use of the full power of the computer. Therefore, we prepare programs having parts which are performed not once but many times, perhaps with slight changes each time.

For example, suppose we want to write a program which will print out a table of the first 100 positive integers and their square roots. Without a loop, our program would be 100 lines long and would read as follows:

```
10 PRINT 1,SQR(1)
20 PRINT 2,SQR(2)
30 PRINT 3,SQR(3)
...
990 PRINT 99,SQR(99)
1000 PRINT 100,SQR(100)
```

1 - Double precision is discussed on page 26 under *DOUBLE PRECISION ARITHMETIC.*

Notice that the instruction is the same in every statement; only the number to which the instruction refers has changed from one line to the next.

Here is the same program written with a loop which uses the IF statement:

```
10 N = 0
20 N = N+1
30 PRINT N, SQR(N)
40 IF N<100 THEN 20
```

Each statement in this program represents one of the four characteristics of every loop:

- **Initialization (Statement 10 above).** The variable N is assigned the initial value of zero. If this step were omitted, SUPER BASIC would not be able to compute the N+1 in the next statement, since N would be undefined.
- **Modification each time through the loop (Statement 20).** The value of N is increased by 1. Without this statement, SUPER BASIC would execute the following instruction continually for zero and no other value.
- **Body of the loop (Statement 30).** This is the actual instruction which we want to be executed repeatedly. The body of the loop may consist of any number of statements.
- **Exit from the loop (Statement 40).** As long as N is less than 100, SUPER BASIC will go to statement 20 and once again pass through the modification and body of the loop. The last pass will be made when N is equal to 99; statement 20 will then set the value of N to 100, and statement 30 will print 100 followed by 10 (the square root of 100). Then the exit is made. N is **not** less than 100, so SUPER BASIC stops. If there were more statements in this program, the next statement in sequence then would be executed.

FOR AND NEXT

Since loops are so important and are used so often in programming, SUPER BASIC provides the two indirect commands FOR and NEXT to simplify loop specification. The program above can be written as follows with these two commands:

```
10 FOR N = 1 TO 100
20 PRINT N,SQR(N)
30 NEXT N
```

Statement 10 specifies that N is initialized to the value 1 and that N should not be set to a value greater than 100.[1] The modification, an increase of 1 each time through the loop, is implied in this statement. The body of the loop is statement 20. The NEXT command in statement 30 instructs SUPER BASIC to return to the FOR statement for the next value of N. When the body of the loop has been executed for every specified value of N, SUPER BASIC will go to the statement following the NEXT. *NOTE: The value of N after exit from the loop is the final value assigned to N, 100.*

In the following example, statement 10 specifies that K should not be set to a value greater than 7.5. The final value assigned to K is 7.

```
> 10 FOR K = 5 TO 7.5↵
> 20 PRINT K↵
> 30 NEXT K↵
> RUN↵
  5
  6
  7
>
```

THE STEP OR BY CLAUSE

N could have been increased to 100 in steps of any size other than the implied 1. To do this, we must specify the step size in a STEP or BY clause. For example, suppose we want to print the square roots of the first 50 even integers. The program would be written as the one above with statement 10 replaced by:

```
10 FOR N = 2 TO 100 STEP 2
```

There are three equivalent forms of this statement:

```
10 FOR N = 2 TO 100 BY 2
10 FOR N = 2 STEP 2 TO 100
10 FOR N = 2 BY 2 TO 100
```

The specified step size may be negative. For example, if we want to print the square roots of the first 100 integers in descending order, statement 10 would be:

```
10 FOR N = 100 TO 1 STEP -1
```

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

```
> 10 FOR Y = 3 TO 4.5 BY .5↵
> 20 PRINT Y↵
> 30 NEXT Y↵
> RUN↵
  3
  3.5
  4
  4.5
>
```

1 - N could have been replaced by any other acceptable variable name, but could not have been subscripted. Subscripted variables are discussed under *MATRIX ARITHMETIC*, page 20.

SUPPLYING DATA WITHIN THE PROGRAM: THE READ AND DATA STATEMENTS

We have already seen that assignment statements or INPUT statements may be used to assign values to variables. A second method, the combined use of the READ and DATA statements, will be shown here. A third method, input of variable values from a file, will be shown on page 74.

Consider the following program:

```
10 READ N
20 S = 0
30 FOR I = 1 TO N
40 READ X
50 S = S+X
60 NEXT I
70 M = S/N
80 PRINT M
90 DATA 5,60,-10
100 DATA 40,-2,11
```

READ

The READ command is always followed by a variable name or a list of variable names separated by commas. When SUPER BASIC executes a READ statement, the first variable listed in the statement is assigned the first value in the collection of DATA statements (the "data block"), the next variable is assigned the next value, and so on.

Thus, when SUPER BASIC executes statement 10 of our sample program, N is assigned the value of 5. The next READ statement in the program is inside a FOR loop and is executed N (that is, 5) times. This statement causes X to be assigned the next available value in the data block at each pass through the loop. Therefore, when I = 1 (the first pass through the loop), X is set to 60 and, in statement 50, added to S (which is initially zero). During each of the five times through the loop, a new value is assigned to X and added to S. The result is that when the exit from the FOR loop is made, S will be equal to the sum of the X's.

The program finally calculates M, the mean of the numbers, in statement 70.

DATA

The numeric values which are listed in DATA statements must be numbers, not expressions, and must be separated by commas.

The location of DATA statements in a program is arbitrary, although the usual procedure is to place them in a group at the end of the program. The only requirement is that the statements be numbered in the order in which the data is to be read.

The distribution of the elements of data among DATA statements also is arbitrary. For example, we could have typed, in place of statements 90 and 100 in our sample program, either

```
90 DATA 5,60,-10,40,-2,11
```

or

```
90 DATA 5
100 DATA 60,-10
110 DATA 40,-2,11
```

RESTORE

Once all the data has been read from a data block, another READ request will cause an error message telling you that you are out of data. However, if you wish at any time during the program to reread all or part of the data block, you can do this with a RESTORE command. When this command is executed, the next READ command will start reading data from the beginning of the data block; that is, from the first value in the first DATA statement. RESTORE can be executed either directly or indirectly.

For example, if now we wanted to use the formula

$$D = \sqrt{\frac{\sum (X_i - M)^2}{N}}^{1}$$

to calculate the standard deviation of the X's, we could add the following statements to our sample program:

```
110 RESTORE
120 READ N
130 A = 0
140 FOR I = 1 TO N
150 READ X
160 A = A+(X-M)↑2
170 NEXT I
180 D = SQR(A/N)
190 PRINT D
```

1 - The numerator of this fraction uses the mathematical symbol Σ meaning "the sum of". We want to find $(X-M)^2$ for every X and sum the results.

Statement 120 is necessary even though N already has the value of 5 at this point in the program. If this statement were omitted, the first X read by statement 150 would be 5, which is incorrect.

NOTE: If a program containing DATA statements is run more than once, the data block will be restored automatically.

ENTERING AND USING A SUPER BASIC PROGRAM

Before you can call SUPER BASIC and run any of the sample programs or your own programs, you must enter the Tymshare system. The proper procedure is described in the *Tymshare Executive Reference Manual*.

To call SUPER BASIC, type SBASIC followed by a Carriage Return. SUPER BASIC will reply with a > when ready to receive a command.

We will once again review the procedure shown on Page 5. Once SUPER BASIC is ready, start typing your program. Each statement must be terminated by a Carriage Return. Only after the Carriage Return is typed does SUPER BASIC analyze the statement and print an error message if the syntax is incorrect, that is, if the statement does not conform to the rules of SUPER BASIC's grammar. After an error message prints, retype the line correctly.[1]

Remember that an indirect statement (one with a line number) is executed only when the running program reaches the statement in normal sequence; while a direct statement (without a line number) is executed immediately after you type the terminating Carriage Return.

THE ENTER COMMAND

When the ENTER command is used, SUPER BASIC will automatically supply line numbers for statements typed at the terminal. ENTER specifies the first line number at which statements will be entered and the amount by which the line numbers will be incremented. The command takes the form

ENTER line number BY increment

Follow each line with a Carriage Return as usual, and terminate the ENTER command with a Control D. For example,

```
>ENTER 10 BY 5
10 INPUT A,B
15 X=A↑B
20 Y=SIN(X)-A*B
25 PRINT X,Y
30D^C
>
```

NOTE: If anything is typed before the D^C, the preceding line will be copied.

If the starting line number is omitted, it is assumed to be 0. Thus, the command

```
>ENTER BY 5
```

is equivalent to

```
>ENTER 0 BY 5
```

If the increment is omitted, it is assumed to be 10 unless some other increment has already been specified, in which case the last specified increment is assumed. For example,

```
>ENTER 10                    BY 10 is assumed
.
.
.
>ENTER 45 BY 5
.
.
.
>ENTER 100                   BY 5 is assumed
```

If you have already entered statements into SUPER BASIC and you add or insert additional statements with the ENTER command, you are protected against deleting or interleaving statements.[2] For example, if a statement numbered 10 has been typed into SUPER BASIC and later the command

```
>ENTER 10 BY 5
```

is given, the message

NEXT ENTRY WOULD CAUSE DELETION OR INTERLEAVING OF STATEMENTS

and another > will be printed, since typing another statement 10 will delete the old statement 10.

1 - SUPER BASIC's extensive editing features, which allow you to correct errors either before or after you type the Carriage Return at the end of an incorrect statement, will be described later in this manual.

2 - See *SIMPLE EDITING IN SUPER BASIC*, page 14, for methods of deleting and inserting program statements.

READING A PROGRAM FROM PAPER TAPE

Another way to enter a program into SUPER BASIC is by reading the statements from paper tape previously punched "off line", that is, when the terminal is not connected to the computer. For details, see see the *Tymshare Paper Tape Package Manual*.

RUNNING A PROGRAM

A SUPER BASIC program is executed with either of the direct commands RUN, GO TO, or START.

- RUN begins execution at the lowest numbered statement of the program. Any direct or indirect statements previously executed are ignored.[1]
- GO TO followed by a line number begins execution at the statement specified. Any direct or indirect statements previously executed are not ignored; all information is retained.[2]
- START is equivalent to GO TO followed by the line number of the lowest numbered statement. The program is executed from the beginning, but all previous information is retained. *NOTE: As RUN, the START command can be direct only.*

If the program can be executed, the results will be given quickly. This does not necessarily mean that the program is free from error and the answers are correct. There might be a logical error that SUPER BASIC cannot find. Or, there might be an error (other than a syntax error) which prevents execution. If this is so, SUPER BASIC will print a message indicating why it cannot execute the program. Correct your error and try again.

SAVING A PROGRAM

Once you have a program that is running correctly, you may want to save it on a file (a storage area set aside for you in the Tymshare computer). To do this, type the direct command SAVE followed by the name of the file and a Carriage Return. *NOTE: The file name typed after SAVE can be any of the valid file names allowed by the Tymshare EXECUTIVE[3].*

SUPER BASIC replies with NEW FILE if you do not already have a file with that name, and OLD FILE if you do have a file with that name.

In reply to NEW FILE or OLD FILE, you either:

- Confirm the command by typing a Carriage Return. *NOTE: A Carriage Return after OLD FILE causes the contents of the old file to be replaced. Or,*
- Abort the command by pressing the ALT MODE/ESCAPE key.

Example

```
>SAVE KL22 ↵
NEW FILE ↵
>
```

NOTE: Only indirect statements (those with line numbers) will be saved on the file.

To save part of your program, type SAVE followed by the file name and a comma. Then type the line numbers of the statements you wish to save. Separate the numbers with commas and use the dash (-) to indicate a range. Thus,

```
>SAVE INT, 1-15,30,70-100 ↵
OLD FILE ↵
```

replaces the former contents of the file INT with statements 1 through 15, 30, and 70 through 100. *NOTE: A maximum of four line numbers and/or line ranges may be used in a single SAVE command.*

In the following example, a short program is read from paper tape, loaded into SUPER BASIC, corrected, executed, and saved on a file.

```
-TAPE ↵                    The TAPE program reads the
                           paper tape into the file AREA.

:RUN ↵

INPUT FROM T ↵

OUTPUT TO AREA ↵
 NEW FILE ↵

ECHO? YES ↵

TURN ON READER.

10 PRINT "TYPE THE BASE AND THE
HEIGHT"
20 IMPUT B,H          The file contains an error.
30 A=1/2*B*H
40 PRINT "THIS IS THE AREA:"
50 PRINT A
```

1 - With the exception of VAR = ZERO and VAR = UNDEF.

2 - When a program containing the READ command is executed more than once, the data is reread from the beginning of the data block even if a direct GO TO was given to execute the program.

3 - See *Appendix C, THE EXECUTIVE*, for more information.

```
NUMBER OF CHARACTERS WRITTEN IS 107

:QUIT ↵

-SBASIC ↵

>LOAD AREA ↵          The file AREA is loaded.

20 IMPUT B,H          SBASIC identifies the error.
SYNTAX ERROR
>20 INPUT B,H ↵       The error is corrected.
>RUN ↵
TYPE THE BASE AND THE HEIGHT
? 10,6 ↵
THIS IS THE AREA:
 30

>SAVE AREA ↵          The corrected program is saved
 OLD FILE ↵           on the file AREA.
>QUIT ↵

-
```

REUSING A SAVED PROGRAM

To reenter a program saved on a file, type LOAD followed by the file name and a Carriage Return. The file may be in your own directory or in another user's directory.

Examples

```
>LOAD THIS ↵          Loads THIS from the user's
                      own directory.
>LOAD (BOB)@THAT ↵
                      Loads @THAT from the direc-
                      tory of user BOB.
>LOAD PROB1 ↵         Loads PROB1 from the user's
                      own directory.
```

In these examples, LOAD is used as a direct command. However, LOAD may also be used indirectly, in which case special rules for specifying the file name must be followed. These and other features of the LOAD command are discussed under *LOAD and LINK,* page 100.

LOOKING AT A PROGRAM

At any time you may have part or all of your program printed by typing the direct command LIST.

Typed alone, LIST causes the entire program to be listed. When LIST is followed by a line number or numbers, only the statements specified are listed. For example,

```
>LIST 4,10,20-30,65 ↵
```

will print lines 4, 10, 20 through 30, and 65.

You can stop the printing at any time by pressing the ALT MODE/ESCAPE key. *NOTE: A maximum of four line numbers and/or line ranges may be used in a single LIST command.*

COMMENTS IN A PROGRAM

Either an exclamation point (!) or the word REM is used to insert remarks or comments as direct or indirect statements. For example;

```
>REM NOW WE WILL TYPE "RUN" ↵
>! FOLLOWED BY "LIST" ↵
>
```

Since these remarks are direct statements, they will not be saved with the program. The following remarks

```
>10 ! THIS PROGRAM CALCULATES THE ↵
>20 ! AREA OF A TRIANGLE ↵
>
```

will be saved because they are indirect statements. They will be listed along with the rest of the program, but will not be printed out when the program is run. Any characters can be typed after ! or REM.

In addition, ! can be used to insert comments at the end of direct or indirect statements. For example,

```
>45 M = S/5          !CALCULATES THE MEAN ↵
>GO TO 20            !OBSERVE THE RESULTS ↵
```

SELF-STARTING PROGRAMS

A program which has been saved on a file may begin to execute automatically as soon as it is loaded. To accomplish this, you may store a RUN, START, or direct GO TO command on the file immediately following the program. You cannot do this in SUPER BASIC because direct commands execute as soon as they are typed and cannot be saved with the program when the SAVE command is given. However, the Tymshare editing language, EDITOR, allows you to read in the SUPER BASIC program from a file, append the desired direct command and then write the program back on the file.[1] When the program is loaded into SUPER BASIC, it will begin to execute immediately. *NOTE: Programs loaded with the LINK command begin execution immediately also. See* **LOAD and LINK,** *page 100.*

1 - For more information, see the *Tymshare EDITOR Manual, Reference Series.*

SIMPLE EDITING IN SUPER BASIC

This section describes only the simplest editing features of SUPER BASIC. The advanced editing features — those which SUPER BASIC shares with EDITOR — are explained under *EDITING FEATURES*, page 103.

INSERTING STATEMENTS

To insert one or more lines between two existing statements in your program, simply type the new statements with line numbers that lie between the numbers of the existing statements. For example, if you have left out a statement between statements 40 and 50, type the additional statement with any number from 41 to 49. SUPER BASIC will list and execute your program in numerical sequence. *NOTE: Statements can also be inserted with the ENTER command, discussed on page 11.*

DELETING STATEMENTS

To delete a statement from your program, either type the line number of the statement followed by a Carriage Return or use the direct command DELETE (may be shortened to DEL). DELETE followed by a line number or numbers will delete the specified statements. For example, either DELETE 10 ↵ or 10 ↵ will delete statement 10. The command

```
> DEL 5,10-35,70 ↵
```

will delete lines 5, 10 through 35, and 70. *NOTE: A maximum of four line numbers and/or line ranges may be used in a single DELETE command.*

To delete the entire program, type DELETE ALL ↵. This command also deletes the values of all variables. Remember to give this command whenever you are finished with one program and wish to load another; SAVE will **not** remove a program from SUPER BASIC.

In addition to deleting existing lines in your program, you may delete an incorrect statement (direct or indirect) at any time before typing the terminating Carriage Return. To do this, type a Control Q (Q^C). An ↑ will print on the terminal and the line will be deleted. Then retype the entire statement.

In the example below, the user deletes 40 FOR I=1 TO with a Q^C and retypes the statement correctly:

```
> 40 FOR I = 1 TO Q^C ↑
40 FOR J = 1 TO 3 ↵
>
```

CHANGING STATEMENTS

To change any statement in your program, simply retype it with the same line number. Whenever you enter a new statement with the same number as a line already in the program, the old statement is replaced by the new one.

If you make an error while typing a statement, you may delete the incorrect character immediately. To do this, type a A^C after the incorrect character (a ← will print on the terminal). Use A^C repeatedly to delete as many characters as necessary.

Example

```
> 10 PRIMA^C←NT "TYPE X" ↵
> 20 X=A^C←A^C←INPUT X ↵
> LIST ↵
10 PRINT "TYPE X"
20 INPUT X
>
```

REVIEW OF COMMANDS IN SECTION 1

The following commands have been discussed thus far in this manual:

Command	Example	Purpose
Assignment Statement	45 M = S/5	Assigns values to variables
DATA	90 DATA 5,60,-10	Stores data in a program
DELETE *or* DEL	DEL 5,10-35,70	Deletes all or part of a program

Commands Review (Continued)

Command	Example	Purpose
ENTER	ENTER 10 BY 5	Supplies line numbers automatically
FOR *and* NEXT	10 FOR N = 1 TO 100 20 PRINT N, SQR(N) 30 NEXT N	Repeats execution of a line or lines for specified values
GO TO . . .	60 GO TO 10 GO TO 10	Unconditional transfer
IF. . . THEN. . .	32 IF S=0 THEN 70	Conditional transfer
INPUT	20 INPUT A,B,C,D,E	Accepts data input from the keyboard
LIST	LIST 4,10,20-30	Lists all or part of a program
LOAD	LOAD KL22	Enters program statements from a file
PRINT	70 PRINT "SUM IS ZERO" PRINT X,Y	Prints text and values of variables
QUIT *or* Q	QUIT	Returns to the EXECUTIVE
READ	10 READ N	Accepts input from DATA statements
REM *and* !	REM PRINT A 55 A=A+1 !ADD 1	For comments or remarks
RESTORE	110 RESTORE	Allows rereading of DATA statements from the beginning
RUN	RUN	Starts execution at the lowest numbered statement
SAVE	SAVE KL22	Saves all or part of a program
START	START	Same as GO TO followed by line number of lowest numbered statement
VAR=UNDEF	70 VAR=UNDEF	Nullifies the effect of VAR = ZERO
VAR=ZERO	10 VAR=ZERO	Initializes variables to zero

Many useful SUPER BASIC programs can be written and used with these few commands. We conclude Section 1 with two more examples. Try these on the terminal, together with some programs of your own. The fastest and easiest way to learn the Tymshare system is to use it!

SAMPLE PROGRAMS

PRODUCT OF A SET OF NUMBERS

This program will read up to 1000 numbers from DATA statements and print the product of the numbers. The last number typed in the data block is to be 5E55. This makes it unnecessary for the user to count how many data items he types, as will be explained below.

```
10 P = 1
20 FOR I = 1 TO 1000
30 READ X
40 IF X = 5E55 THEN 80
50 P = P*X
60 IF P = 0 THEN 80
70 NEXT I
80 PRINT P
90 DATA 15,-9,1.5,33,6,-4,22,9,5E55
```

Each number that is read is compared to what we know is the last data item, 5E55. If the number read is not equal to 5E55 (that is, we have not yet reached the end of the data block), the number will be accepted as one which should be multiplied. The product is stored in the variable P. P is initialized to 1 in line 10 so that the first time through the FOR loop, the first data item (1*X) will be stored in P. Each subsequent time through the loop, the product calculated thus far will be multiplied by the number just read. When 5E55 is read, SUPER BASIC will go immediately to line 80 and print the product, P.

Line 60 states another condition under which SUPER BASIC should print the product calculated thus far, that is, if this product is 0. In this case there is no reason to continue multiplying, since the result will be 0 regardless of what numbers follow. *NOTE: This statement is optional; it merely saves calculation time if one of the data items is 0.*

Try this sample program with any set of numbers. If you use the data provided in the above example, the answer should be 31755240. You can substitute any number in place of 5E55 in this program, as long as the number you choose appears only at the end of the data block.

DOUBLE DECLINING BALANCE DEPRECIATION

This program calculates and lists the depreciation and book value of an asset at the end of every year of its useful life.

The original cost (C) and the estimated useful life (L) of the asset are used to calculate the depreciation (D). At the end of the first year,

$$D = \frac{2 \cdot C}{L}$$

The book value at the end of the first year is C-D (original cost less depreciation). For each subsequent year, the depreciation and book value are calculated by the same formulas as above, substituting for C the book value at the end of the previous year.

The user is asked to type in the original cost and the estimated useful life. Following the listing of the program is a sample run for an asset which costs $7,000 and is depreciated over 15 years.

```
>LIST
0 ! DOUBLE DECLINING BALANCE DEP.
10 PRINT "TYPE COST OF ASSET AND"
20 PRINT "ESTIMATED USEFUL LIFE"
30 INPUT C,L
40 PRINT "YEAR","DEP.","BOOK VALUE"
50 FOR X = 1 TO L
60 D = 2*C/L
70 C = C-D
80 PRINT X,D,C
90 NEXT X
>RUN
TYPE COST OF ASSET AND
ESTIMATED USEFUL LIFE
? 7000,15
YEAR            DEP.            BOOK VALUE
 1              933.33333       6066.6667
 2              808.88889       5257.7778
 3              701.03704       4556.7407
 4              607.56543       3949.1753
 5              526.55671       3422.6186
 6              456.34915       2966.2695
 7              395.50259       2570.7669
 8              342.76891       2227.9979
 9              297.06639       1930.9316
 10             257.45754       1673.474
 11             223.12987       1450.3441
 12             193.37922       1256.9649
 13             167.59532       1089.3696
 14             145.24928       944.12032
 15             125.88271       818.23761
>
```

The commas in statement 40 caused spaces to be printed between the column headings. All of the PRINT statement forms and rules are discussed in detail under *PRINTING FEATURES*, page 49.

SECTION 2
ARITHMETIC AND LOGICAL FEATURES

ARITHMETIC FUNCTIONS

Some of the standard mathematical functions available in SUPER BASIC were described in Section 1. Described below are the additional built-in mathematical functions.

Function	Returns
SGN(X)	1 if X > 0 0 if X = 0 -1 if X < 0
INT(X) or IP(X)	Greatest integer less than or equal to X
FP(X)	X-INT(X)
FIX(X)	X truncated
ROUN(X)	X rounded
COMP(X,Y)	1 if X > Y 0 if X = Y -1 if X < Y
PDIF(X,Y)	X-Y if X-Y is positive; otherwise, 0
MAX(X_1,X_2,...,X_n)	Value of largest argument
MIN(X_1,X_2,...,X_n)	Value of smallest argument

NOTE: Three of the above functions, COMP, MAX, and MIN, can be used not only with numeric arguments but also with string arguments, as explained on page 40.

SGN(X)

The sign function SGN(X) yields 1 if the value of the argument X is positive, 0 if X is equal to 0, and -1 if X is negative. Thus,

SGN(31) = 1
SGN(0) = 0
SGN(-.2387) = -1

INT(X) OR IP(X)

The integer function is INT(X) or IP(X) where, as with other functions, X can be any expression. This function yields the greatest integer less than or equal to X. Thus,

INT(7.8) = 7
INT(-2.4) = -3

FP(X)

The fractional part of X is defined as follows:

FP(X) = X-INT(X)

Thus,

FP(8) = 0
FP(123.456) = .456
FP(-1.8) = .2 **[-1.8-(-2)]**

FIX(X)

This function is defined as SGN(X)*INT(ABS(X)). It truncates the value of the expression X as follows:

FIX(7.8) = 7
FIX(-2.4) = -2

Whatever follows the decimal point is dropped. Note that FIX(X) is equivalent to INT(X) for positive numbers, but not for negative numbers; for example, FIX(-2.4) = -2, but INT(-2.4) = -3.

ROUN(X)

The function ROUN(X) rounds the value of the expression X to the nearest integer. ROUN(X) is equal to INT(X+.5).

Example

ROUN(7.2)=7 **ROUN(7.5)=8**
ROUN(-1.3)=-1 **ROUN(-1.8)=-2**

COMP(X,Y)

The function COMP(X,Y) compares the values of X and Y and returns -1 if X is less than Y, 0 if X and Y are equal, and 1 if X is greater than Y.

Example

COMP(15,3)=1
COMP(3,15)=-1
COMP(-4.5,-4.5)=0

PDIF(X,Y)

PDIF(X,Y) returns the positive difference of X and Y, that is, X-Y, if X-Y is positive; otherwise, it returns 0.

Example

PDIF(15,3)=12 **PDIF(3,15)=0**
PDIF(-7.9,-8)=.1 **PDIF(-8,-7.9)=0**

MAX(X_1,X_2,...,X_n)

The MAX function takes two or more arguments and returns the value of the largest argument.

Example

MAX(16,0,-5,11)=16
MAX(-.17,0)=0
MAX(3.2,3.5,3.31)=3.5

MIN(X_1,X_2,...,X_n)

The MIN function takes two or more arguments and returns the value of the smallest argument.

Example

MIN(16,0,-5,11)=-5
MIN(-.17,0)=-.17
MIN(3.2,3.5,3.31)=3.2

UTILITY FUNCTIONS

The utility functions are summarized in the table below and then described in the following paragraphs.

Function	Returns
RND(X)	Random number
POS	Terminal print position
POS(N)	File print position
TAB(X)	Spaces to print position X on terminal.
TAB(X,N)	Spaces to print position X on file N.
DATE	Date and time of day
TIME	Reads internal clock

RND(X)

The RND function is a pseudo random number generator. It may have either one argument or no argument. If an argument is given, it must be an integer and may be zero, positive, or negative. RND without an argument is equivalent to RND(0). The random number will be between 0 and 1 exclusive.

If RND(0) is used alone, the first use of the function in a program will always produce the same number. When RND(0) is used again in the same program, the next random number in sequence is given.

RND(0) can be used repeatedly to generate a sequence of random numbers once that sequence has been initiated by RND with a non-zero argument.

RND with a positive argument returns a random number generated from the argument. Thus, RND(16) will always produce the same number, which will be different from the number RND(30). A sequence of random numbers can be initiated by RND with a positive argument and then RND(0) (or RND) can be used repeatedly to generate the next random numbers in the sequence.

RND with a negative argument returns a random number generated from a reading of the internal clock of the computer. The value of the negative argument has no bearing on the random number it generates; for example, RND(-1) used twice in a program will yield different random numbers which depend only on reading the internal clock. Thus, using RND with a negative argument to initiate a sequence of random numbers will produce a different sequence of numbers each time the program is run.

Example

```
10 PRINT RND(-1);
20 FOR I = 1 TO 9
30 PRINT RND;
40 NEXT I
```

If this program is run twice, two different sequences of random numbers will be printed. However, if the argument of the RND function in line 10 were changed to 0 (or no argument) or to a positive number, running the program twice would yield the same sequence of random numbers.

CAUTION: If RND with a negative argument is used repeatedly to generate a sequence of random numbers rather than simply to initiate the sequence, the result may not be satisfactory. Because the internal clock is incremented only 60 times per second, the numbers generated will not be truly random. See **TIME**, *page 20, for a description of the internal clock.*

POS AND POS(N)

The function POS can have either no argument or one argument. When no argument is given, the function specifies the position on the terminal at which the print head is located.

Example

```
>10 FOR I = 1 TO 10↵
>20 READ X↵
>30 PRINT X:      !CONCATENATED ZONES↵
>40 IF POS>15 THEN PRINT↵
>50 NEXT I↵
>60 DATA 10,20,30,40,50,60,70,80,90,100↵
>RUN↵
 10  20  30  40  50
 60  70  80  90  100

>
```

As specified in line 40, a Carriage Return is printed after the print head passes position 15.

The POS function is used with an argument only when writing on files.[1] The argument is the file number of the output file. When writing on a sequential symbolic file, the value of the function is one greater than the number of characters written since the last Carriage Return.

Example

```
>10 OPEN "XX", OUTPUT, 2↵
>20 FOR I = 1 TO 10↵
>30 READ X↵
>40 PRINT ON 2: X:↵
>50 IF POS(2)>15 THEN PRINT ON 2:↵
>60 NEXT I↵
>70 DATA 10,20,30,40,50,60,70,80,90,100↵
>RUN↵

>QUIT↵

-COPY XX TO TEL↵

 10  20  30  40  50
 60  70  80  90  100

-
```

NOTE: When POS(N) is used with random files, its meaning is different than described here. See **THE POS(n) FUNCTION WITH RANDOM FILES,** *page 85.*

TAB(X) AND TAB(X,N)

The TAB function can have either one or two arguments. TAB(X) is used in the PRINT statement to move the print head to the X^{th} print position on the terminal. The function is used with a colon if the number or text which follows it is to be printed at the specified position. For example,

```
>PRINT TAB(20):B↵
                    -456
                     ↑
                 20th position

>PRINT A:TAB(12):B↵
 18          -456
              ↑
          12th position

>
```

If a semicolon is used after the TAB function, the print head will move beyond the specified print position; a comma causes it to move to the next PRINT zone of 15 spaces.

If the semicolon or comma which precedes the TAB function causes the print head to move beyond the position specified by the TAB, the TAB will be ignored. For example,

```
>PRINT A,TAB(12):B↵
 18             -456
                 ↑
             16th position

>
```

The comma caused the print head to move past the first field of 15, so TAB(12) was ignored.

TAB with two arguments is used when writing on files. The form is

TAB(X,N)

where X specifies the print position to which SUPER BASIC should tab, and N specifies the file number. For example, the statement

```
>PRINT ON 2: A: TAB(12,2):B↵
```

prints the value of A, tabs to position 12 and prints the value of B on the file that has been opened for output as file 2.

DATE

DATE is a function with no argument that returns a string of twelve characters indicating the date and the time of day. For example,

```
>PRINT "DATE AND TIME: ": DATE↵
DATE AND TIME: 01/27  10:36
>
```

1 - When POS(N) is used in writing on a sequential binary file, the value is one greater than the number of words written on the file. (A word is considered to be three characters.)

TIME

TIME is a function with no argument. It reads the internal clock of the computer and returns an integer value that is increased by 1 every 1/60 of a second. For example, if the statement

0 A=TIME

were at the beginning of a program, and

950 Z=TIME

were at the end of the program, the total elapsed seconds could be calculated and printed as follows:

```
960 T=(Z-A)/60
970 PRINT T:" TOTAL ELAPSED SECONDS"
```

MATRIX ARITHMETIC

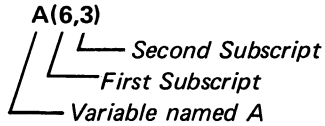
Variables were described in Section 1 as being able to store one value. There are times, however, when the user will want to store a set of values in a list or table which he can refer to by a single name. This is done by using subscripted variables to designate elements of such lists or tables, which are called arrays. A variable may have any number of subscripts; in other words, SUPER BASIC allows arrays of any dimension (each subscript representing a dimension).

Subscripts are typed in parentheses after the variable name. For example, A(7) refers to the seventh item in a list (a one-dimensional array, or vector) named A, and B(3,7) denotes the element in the third row and seventh column of a table (a two-dimensional array, or matrix) named B.

SUBSCRIPTED VARIABLE NAMES

The name of a subscripted variable must be a single letter or a single letter followed by a $ (dollar sign). The variable name used for a subscripted variable also may be used to denote a simple variable in the same program. For example, A and A(1) are considered to be separate variables. However, the same name cannot be given to arrays of different dimensions in the same program; for example, A(1) and A(3,7) may not be used simultaneously in a program.

Example



SUBSCRIPTS

Subscripts may be variables (including other subscripted variables) or expressions of any complexity, The following subscripted variables are acceptable:

A(5) C(1+K) F(5,30) R(B(3,J),C-D) X(A*B,20)

Subscripts may have any value, including negative and non-integer. If the value of a subscript is non-integer, SUPER BASIC will truncate the value.

SIZE OF ARRAYS

SUPER BASIC automatically supplies space for subscripts 1 to 10 for arrays of one or two dimensions. Therefore, a vector named A containing 10 elements could be entered simply with the statements

```
10 FOR I = 1 TO 10
20 READ A(I)
30 NEXT I
40 DATA 2,3,-5,7,2.2,4,-9,123,4,-4
```

THE DIM COMMAND

If an array is to have a subscript greater than 10 or have more than two dimensions, the size of the array must be specified by the DIM command which can be executed directly or indirectly. This command instructs SUPER BASIC to reserve a specified amount of space for array elements. For example,

10 DIM A(15)

will reserve 15 locations for elements A(1) to A(15). The DIM statement does not define any array elements; it simply allows a certain number of values to be accepted as input to the array.

Any number of arrays can be dimensioned in a single DIM statement as follows:

60 DIM K(20),L(3,3,1),M(A*B),N(X,3,3,2)

The user may save storage space by dimensioning arrays with subscripts less than 10, even though such dimensioning is not required. Thus, DIM E(3,5) will reserve space for exactly 15 elements, whereas without the DIM statement, 100 (10X10) spaces would be reserved for the array E. *NOTE: Whatever the maximum subscript value, arrays of three or more dimensions require a DIM statement.*

Subscripts start from 1 unless otherwise specified. One way to specify a different subscript base is with the following form of the DIM command:[1]

10 DIM A(0:15)

This statement will reserve space for elements A(0) to A(15). The user may specify that a subscript start from any number. For example,

DIM B(5:10) *Reserves space for B(5) to B(10).*

DIM B(-6:10,-2:4) *Starts subscripts at negative values; the 0th elements are included.*

The user may redimension an array at any time by using another DIM statement. Note however, that redimensioning (or executing the same DIM statement a second time) causes any existing elements in an array to be cleared (that is, be undefined). For example,

```
> 10 DIM X(20)↵          The array X is dimensioned.
> 20 X(1) = 3,X(2) = 7↵  Two elements are defined.
> 30 DIM X(0:20)↵        X is redimensioned to include the 0th element.
> RUN↵                   The above statements are executed.
> PRINT X(1),X(2)↵
VARIABLE HAS NO VALUE
                         X(1) and X(2) have been undefined.
>
```

THE BASE COMMAND

Another method of specifying that subscripts start with some number other than 1 is by using the BASE command which can be executed directly or indirectly. The form of this command is

BASE n

where n can be any numeric expression. BASE applies only to arrays which have not yet been dimensioned, and will cause the subscripts of those arrays to begin from n unless:

- The lower limit of a subscript is specified in a DIM statement, such as DIM A(-2:5), or
- Another BASE command is given which specifies a different base.

Example 1

```
5 BASE 0
10 DIM A(15),B(-2:2,10)
```

will cause the A subscript and the second B subscript to start at 0. Suppose that the following statements were added to the above:

```
15 BASE 1
20 DIM C(3)
```

The dimensions of arrays A and B would not be affected; the subscript of array C would begin at 1 and not 0.

Example 2

```
10 BASE -10
20 FOR I = -10 TO 10
30 P(I) = I↑3
40 NEXT I
```

The array P was never dimensioned, so space was automatically supplied for the array up to a subscript value of 10. If the subscript value were to exceed 10, P would need to be dimensioned explicitly in a DIM statement.

NOTE: RUN nullifies the effect of any previous BASE command.

MATRIX OPERATIONS

Although the user may write his own routines for matrix operations, SUPER BASIC contains a set of commands which make calculations involving matrices or vectors considerably easier. All of these commands begin with the word MAT, and many of them are similar in form to the ordinary SUPER BASIC instructions. *NOTE: The MAT commands apply only to arrays of one or two dimensions. Any attempt to use them with multi-dimensional arrays will cause an error message to be printed.*

Input Of Matrix Data

The following input commands do not require that the specified matrices or vectors be dimensioned before the commands are given. A matrix or vector that has not been dimensioned previously, however, must be dimensioned in the MAT command itself (see details below). *NOTE: This rule applies in all cases, even if the subscript value will not exceed 10. SUPER BASIC must know when to stop accepting data for input.*

MAT READ

MAT READ A,B,C

will read values into the previously dimensioned matrices (or vectors) A, B, and C from the data block defined in the DATA statements of a program. Any number of matrices can be read with a single MAT READ instruction.

1 - A second method of specifying a base other than 1 uses the BASE command.

It is possible to use the MAT READ statement itself to dimension a matrix or vector which has not been dimensioned previously (or to redimension one which already has). In this case, simply type the dimensions of the arrays just as they would be typed in a DIM statement. For example,

```
65 MAT READ K(15),L(-1:1,3),M
```

will read values into a 15 element vector K, a 3 by 3 matrix L (with the first subscript ranging from -1 to 1), and a previously dimensioned matrix M.[1] This statement is exactly equivalent to

```
65 DIM K(15),L(-1:1,3)
70 MAT READ K,L,M
```

Matrices are read in row order; that is, the second subscript varies more rapidly. For example,

```
10 MAT READ A(4,3)
```

is equivalent to

```
10 FOR I = 1 TO 4
20 FOR J = 1 TO 3
30 READ A(I,J)
40 NEXT J,I
```

In both cases, values will be read from the DATA statements in the following order: A(1,1),A(1,2), A(1,3),A(2,1),. . . ,A(4,2),A(4,3).

NOTE: A matrix statement should not be used to dimension a matrix unless DIM dimensioning is desired. For example,

```
10 COMPLEX A(20,20)
 .
 .
 .
70 MAT INPUT A(20,20)
```

is equivalent to

```
10 COMPLEX A(20,20)
 .
 .
 .
70 DIM A(20,20)
71 MAT INPUT A
```

The DIM statement supersedes the COMPLEX declaration. Therefore, the elements of the matrix are real numbers, not complex numbers.

MAT INPUT

The MAT INPUT command performs the same function for matrices and vectors as the INPUT command does for variables; SUPER BASIC prints a question mark and waits for the data to be typed from the keyboard. Matrix values should be typed in the same order that they would be read by a MAT READ statement; that is, in row order (with the second subscript varying more rapidly).

The form of the MAT INPUT command is similar to MAT READ in that the matrices or vectors may be dimensioned either previously or in the MAT statement itself.

Also included in SUPER BASIC is a MAT INPUT FROM command corresponding to the INPUT FROM command for reading data from a file.

Example 1

```
MAT INPUT A(2,3)
```

will cause SUPER BASIC to wait for six values to be typed, in the order: A(1,1),A(1,2),A(1,3),A(2,1), A(2,2),A(2,3).

Example 2

```
10 OPEN "MATDATA",INPUT,1
20 MAT INPUT FROM 1: A(2,3,4)
```

accepts 24 values from MATDATA as input to the matrix A.

Example 3

```
10 DIM F(5),G(4,4)
 .
 .
 .
95 MAT INPUT F,G(4,X),H(7,7)
```

Vector F and matrix G are dimensioned in line 10. Statement 95 redimensions matrix G, dimensions a new matrix H, and requests data for F, G, and H.

Output Of Matrix Data

A command of the form

```
MAT PRINT A,B,C
```

can be executed directly or indirectly to print the matrices (or vectors) A, B, and C. Every element of A, B, and C must have a value.

Matrices are printed row by row. The elements of each row are printed in normal (15 space) print zones unless the matrix name is followed by a semicolon or a colon in the PRINT statement. A semicolon **after** a matrix name will cause the elements of each row to be printed in packed zones; a colon will cause concatenated print zones. Each row is separated from the next by a blank line.

1 - It is assumed here, and in the remaining examples in this section, that no BASE command has been given previously, so that subscripts start from 1 unless otherwise specified.

Example 1

```
>10 MAT INPUT F(2,3)↵
>20 PRINT↵
>30 MAT PRINT F;↵
>RUN↵
? 1,2,3,4,5,6↵

 1   2   3

 4   5   6
```

Example 2

```
MAT PRINT R;S,T;
```

will print R and T in packed zones and S in normal zones. *NOTE: If the semicolon after T were omitted, a comma would be understood and T would be printed in normal zones also.*

SUPER BASIC one-dimensional arrays are row vectors and therefore will be printed horizontally. A column vector, consisting of one column instead of one row, can be dimensioned as, for example, V(N,1), which would set up a 1 column, N row array and therefore print the elements vertically.

The MAT PRINT ON (or MAT WRITE ON) command corresponds to the PRINT ON (or WRITE ON) command for writing data on a file.

Mathematical Operations With Matrices

All of the following operations require that the solution matrix or vector be dimensioned properly before the operation is performed. For example, the statement MAT C = A+B will add the matrices A and B and store the result in matrix C; C **must** be dimensioned properly before this statement is executed (even if neither subscript exceeds 10).

Only one mathematical operation with matrices may be performed per statement. Thus, MAT X = R+S+T is not allowed, but can be achieved by two MAT instructions.

Each of the following statements can be executed directly or indirectly.

Matrix Addition

MAT C = A+B

A statement of this form adds the matrices (or vectors) A and B and stores the result in C. A, B, and C all must be of the same dimensions for this statement to be executed.

Matrix Subtraction

MAT C = A-B

This statement subtracts the matrix (or vector) B from the matrix (or vector) A and stores the result in C. A, B, and C must have the same dimensions.

Matrix Multiplication

MAT C = A*B

In order for this statement to be executable, A and B must be "conformable"; that is, they must be of such dimensions that their product is defined. In addition, C must be dimensioned properly to contain the result. This instruction applies to matrices only. Multiplying vectors is not permitted in SUPER BASIC. However, vectors effectively can be multiplied if they are dimensioned as matrices. For example, the vector A(N) could be dimensioned as A(N,1).

Scalar Multiplication

MAT C = (n)*A

This statement performs scalar multiplication; that is, each element of the matrix (or vector) A is multiplied by the number (or numeric expression) n (which must be enclosed in parentheses) and stores the result in C. C must be the same dimension as A. *NOTE: The instruction MAT C = (1)*A may be typed simply as MAT C = A.*

Matrix Transposition

MAT C = TRN(A)

This statement transposes the rows and columns of A and places the result in C; it is equivalent to letting C(I,J) = A(J,I) for all values of I and J. C and A need not be square; an M by N matrix will be transposed into an N by M matrix. *NOTE: This instruction applies to matrices only. Vector transposition is not permitted in SUPER BASIC. However, vectors effectively can be transposed if they are dimensioned as matrices. For example, the column vector A(N,1) will be transposed into the row vector C(1,N).*

Matrix Inversion

MAT C = INV(A)

This statement inverts the square matrix A (using the Gauss-Jordan method with complete matrix pivoting) and stores the result in C. The matrix will not be inverted if it is singular or nearly so (that is, "ill-conditioned", so that it is difficult to invert accurately). SUPER BASIC will print an error message if it encounters a pivot element less than EPS, a special

variable that is preset internally to 10^{-10}. EPS can be used as any other variable, and therefore can be set to any value by the user. Changing its value from 10^{-10} would thus modify the definition of a matrix ill-conditioned for inversion.

SUPER BASIC also includes DET, a function with no argument, which returns the determinant of the last matrix inverted.

NOTE: The same matrix may appear on both sides of a MAT statement for addition, subtraction, scalar multiplication, or inversion, but not in any of the other instructions. Thus,

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

are all legal, while use of

*MAT A = B*A*
MAT A = TRN(A)

will result in nonsense.

Matrix Initialization

Setting All Elements To Zero

MAT C = ZER

This instruction sets all elements of the previously dimensioned matrix (or vector) C to zero. It can be used also to dimension (or redimension) a matrix or vector and initialize all elements to zero. Thus, the statement

MAT C = ZER(M,N)

sets up an M by N matrix C, where C need not be dimensioned previously, and fills the matrix with zeroes. An instruction of the form

MAT C = ZER(M)

performs a similar function for an M element vector.

Setting All Elements To One

MAT C = CON

This instruction is similar in form and function to MAT C = ZER, except that the matrix (or vector) is filled with ones instead of zeroes. It can be used also to dimension (or redimension) a matrix or vector, in the form

MAT C = CON(M,N) *or*
MAT C = CON(M)

Setting An Identity Matrix

MAT C = IDN

This statement sets the previously dimensioned square matrix C equal to an identity matrix, that is, a matrix with ones on the main diagonal and all other elements equal to zero. It can be used also to dimension (or redimension) a matrix, in the form

MAT C = IDN(M,M)

Example Of Matrix Operations

This program reads the dimensions and values of matrices A and B from DATA statements. A, B, and A*B are printed, then A*B with one element changed.

```
>LIST
10 READ M,N
20 MAT READ A(M,N),B(N,N)
30 MAT PRINT A:B;     !NOTE THE FORMATS
40 DIM C(M,N)
50 MAT C = A*B
60 MAT PRINT C;
70 C(1,3) = 99        !ONE ELEMENT CHANGED
80 MAT PRINT C
90 DATA 2,3
100 DATA 1,2,3,4,5,6
110 DATA 1,0,1,-2,1,-1,0,2,3
>RUN
 1 2 3

 4 5 6


 1    0    1

-2    1   -1

 0    2    3


-3    8    8

-6    17   17


-3          8          99

-6          17         17
```

COMPLEX ARITHMETIC

COMPLEX VARIABLES

Complex arithmetic can be performed easily in SUPER BASIC by using complex variables. A variable that is to be assigned a complex value must first be declared complex. To do this, type the variable name (or names, separated by commas) in a COMPLEX statement which can be executed directly or indirectly.

In the following example A and B are declared complex, assigned values by means of the INPUT command, and printed on the terminal.

```
> 10 COMPLEX A,B↵
> 20 INPUT A,B↵
> 30 PRINT "A =":A,"B =":B↵
> RUN↵
? 5.6,-1.78,-300,15↵
A = 5.6,-1.78    B = -300, 15

>
```

Two numbers are required as input for each complex variable, namely, the real part and the imaginary part of the variable. When the value of a complex variable is printed, the real and imaginary parts are separated by a comma. The above example set A to 5.6-1.78i and B to -300+15i.

The COMPLEX statement can also be used to declare that an array will store complex values. For example,

```
10 COMPLEX R(0:20),S(M,N)
```

reserves space for a 21 element complex array R and an M by N complex array S. Each element of a complex array consists of two numbers, the real and the imaginary parts of the complex number.

The form of a complex number in a DATA statement is A,B where A and B are the real and imaginary parts of the complex number respectively. Both parts of the number must be typed; zero values may not be omitted from the DATA statement. For example,

```
> 10 COMPLEX X(3)↵
> 20 READ X(I) FOR I = 1 TO 3↵
> 30 PRINT "X(1) = ":X(1);"X(2) =":X(2);↵
                        "X(3) =":X(3)↵
> 40 DATA 5,4,5,0,-4,1.7↵
> RUN↵
X(1) = 5, 4  X(2) = 5, 0  X(3) = -4, 1.7

>
```

When relational operators are used with complex values, only the real parts of the values are compared. Thus, if X(1) to X(3) have the values that were assigned in the above example, the following expressions are true:

```
X(1)>X(3)  }
X(2)>X(3)  }   Since 5>-4.
X(1) = X(2)    Since 5 = 5.
```

COMPLEX FUNCTIONS

CMPLX(X,Y)

CMPLX(X,Y) creates a complex value whose real part is equal to X and whose imaginary part is equal to Y, where X and Y can be any real or integer numerical expression. This function must be used to include a complex number in an assignment statement. For example,

```
> 10 COMPLEX R,S↵
> 20 R = CMPLX(1,5)↵
> 30 N = 4↵
> 40 S = R+CMPLX(N+1,2)↵
> 50 PRINT "R =":R,"S =":S↵
> RUN↵
R =  1, 5          S =  6, 7

>
```

If R and S had not been declared in the above example, only the real parts of their values would have been stored; the result would have been R = 1 and S = 6. *NOTE: CMPLX may not be used with a complex argument.*

REAL(C)

This function returns the real part of a complex variable or expression.

```
> 10 COMPLEX X,Y↵
> 20 X = CMPLX(6,-1.1)↵
> 30 Y = CMPLX(2.3,5)↵
> 40 PRINT REAL(X),REAL(X+Y)↵
> RUN↵
  6                 8.3

>
```

IMAG(C)

This function returns the imaginary part of a complex variable or expression.

```
>10 COMPLEX X,Y
>20 X = CMPLX(6,-1.1)
>30 M = IMAG(X)
>40 PRINT "M=":M
>RUN
M = -1.1
>
```

ABS(C)

The ABS function, when used with a real argument, returns the absolute value of the argument. However, this function can also be used with a complex argument to return its magnitude,

$|a + bi| = \sqrt{a^2 + b^2}$

```
>PRINT ABS(CMPLX(3,4))
 5
```

CONJ(C)

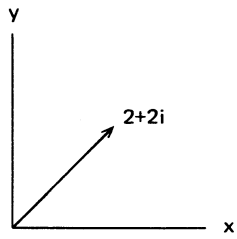
This function returns the conjugate of its complex argument; that is, if C=a+bi, the conjugate of C is a-bi.

```
>5 COMPLEX Z
>10 Z=CMPLX(1,2.5)
>15 PRINT "CONJUGATE OF Z IS":CONJ(Z)
>RUN
CONJUGATE OF Z IS 1,-2.5

>
```

PHASE(C)

This function returns the phase of its complex argument, that is, the angle (in radians) that the vector defined by the argument makes with the positive x axis. For example, consider the vector




```
>10 COMPLEX V
>20 V=CMPLX(2,2)
>30 P=PHASE(V),A=P*180/PI
>40 PRINT "THE PHASE OF ":V:" IS":P:
                              " RADIANS,"
>50 PRINT "OR":A:" DEGREES."
>RUN
THE PHASE OF  2, 2 IS .78539816 RADIANS,
OR 45 DEGREES.

>
```

POLAR(X,Y)

The POLAR function takes real arguments and returns the vector (complex number) with magnitude equal to the first argument and phase equal to the second argument in radians. Thus,

if **C=POLAR(X,Y)**
then **X=ABS(C)**
and **Y=PHASE(C)**

The complex result will be returned in the usual internal form, that is, the real and imaginary parts of the number. For example, if we know that vector Q has magnitude 3 and phase .314 radians, and vector R has magnitude 2 and phase .63 radians, we can calculate and print the product of Q and R as follows:

```
>10 COMPLEX Q,R,S
>20 Q=POLAR(3,.314),R=POLAR(2,.63)
>30 S=Q*R
>40 PRINT "PRODUCT IS":REAL(S):" +":
IMAG(S):" I,"
>50 PRINT "OR POLAR (":ABS(S):",":
PHASE(S):" )"
>RUN
PRODUCT IS 3.5193185 + 4.8594647 I,
OR POLAR ( 6, .944 )

>
```

DOUBLE PRECISION ARITHMETIC

THE DOUBLE DECLARATION STATEMENT

SUPER BASIC will normally store a maximum of eleven significant digits in a variable or array element. A variable that is declared DOUBLE, however, can store a double precision value; that is, up to seventeen significant digits will be retained. For example,

```
30 DOUBLE A,X(N,20)
```

declares that A is a double precision variable and X is an N by 20 double precision array. Three words of

storage are reserved for each double precision variable or array element. This is one more word per variable or array element than reserved by a REAL declaration statement.[1]

ASSIGNING VALUES TO DOUBLE PRECISION VARIABLES

In the following example, values with more than eleven significant digits are assigned with INPUT and READ to both single precision and double precision variables. To show how many digits are retained in each case, the values are printed using PRINT IN FORM, a feature discussed under *FORMATTING WITH FORM*, page 56. This feature lets the user specify the exact format of the values to be printed.

```
>10 DOUBLE A,C
>20 INPUT A,B
>30 READ C,D
>40 PRINT
>50 PRINT "DOUBLE PRECISION:":TAB(24):
                "SINGLE PRECISION:"
>60 PRINT IN FORM "%.17% 4B %.17%/":A,B
>70 PRINT IN FORM "20# 3B 20#":C,D
>80 DATA 1234.5678901234E10,
          1234.5678901234E10
>RUN
?  1.23456789012345678,  1.23456789012345678

DOUBLE PRECISION:      SINGLE PRECISION:
1.23456789012345670    1.23456789010000000
 .12345678901234E+14    .12345678901000E+14
```

The results will be the same if we replace lines 20 and 30 above by:

```
20 INPUT A
30 READ C
35 B=A, D=C
```

as long as B and D are single precision variables.

To assign a value to a double precision variable in an assignment statement, type the number in D format. This format is the same as E format except that the letter D is used instead of E. The D indicates that the number is double precision. For example,

Number	E Format	D Format
-53×10^9	-53E9	-53D9
.00000000000063	.63E-12	.63D-12
7	7E0	7D0

D format must be used when the value to be assigned to a double precision variable is non-integer or an integer of more than eleven digits. Integers of eleven or fewer digits need not be typed in D format. For example,

```
10 DOUBLE W,X,Y,Z,A
20 W=5.362D-6
30 X=.123456789012345D13
40 Y=123456789012D1
50 Z=50          !OR 50D0
60 A=W*X+Y*Z
.
.
.
```

DOUBLE PRECISION FUNCTIONS

DBL(X)

The DBL function returns the double precision value of its argument. For example,

```
>5 R=1.456346
>6 E1=5.002307E12
>10 X=8746867586.5
>20 Y=6589564509.6
>30 Z=6723219087.6
>40 R2=X↑2+Y↑2+Z↑2+R*E1
>50 R1=R2↑.5
>60 PRINT IN FORM "11%/":R1
>70 R1=(DBL(R2))↑.5
>80 PRINT IN FORM "11%/":R1
>RUN
  12850359339     Using single precision.
  12850359341     Using the DBL function.

>
```

In this example, the user requires eleven places of accuracy for R1. The eleventh digit of R1 is inaccurate unless the value of R2 is computed to double precision accuracy when it is raised to the .5 power.

DPI

DPI is a function with no argument. It is equal to double precision π, 3.1415926535897932.

1 - See *USING DECLARATION STATEMENTS*, page 47.

OCTAL CONSTANTS AND BINARY OPERATIONS

OCTAL CONSTANTS

Octal constants are denoted by a leading O. For example, to specify octal 41, type O41. Single digit octal constants must be specified by O00 through O07 so that they can be distinguished from the variables O0 through O7. For example,

```
>PRINT O41;O05↵
 33    5

>
```

Since SUPER BASIC automatically prints numbers to the base 10, octal 41 is printed in the above example as its decimal equivalent, 33.

Octal constants can be used anywhere decimal constants are used (such as in assignment statements or as function arguments), but will not be recognized in DATA statements or in file or terminal input.

If more than eight digits are specified after O, the last eight will be accepted. Thus, O312345671 is equivalent to O12345671.

BINARY OPERATORS

The operators BAN, BOR, and BEX (Binary AND, Binary OR, and Binary exclusive OR) operate on and return integer values. Any attempt to use these operators with non-integer operands will generate meaningless results.

BAN, BOR, and BEX perform logical operations on the digits of the binary equivalents of their operands as shown in the following table:

	Binary Digits			
Operands / Operation	1	1	0	0
	1	0	1	0
AND	1	0	0	0
OR	1	1	1	0
EXCLUSIVE OR	0	1	1	0

For example, consider

A BOR B where A=28, B=6 (or, expressed as octal constants, A=O34, B=O06)

The binary equivalents of A, B, and the result of A BOR B are

A	011100
B	000110
A BOR B	011110

The result, binary 11110, is equal to octal 36, or decimal 30. Thus,

```
>PRINT 28 BOR 6↵
 30

>PRINT "OK" IF O34 BOR O06=O36↵
OK

>
```

BAN, BOR, and BEX all have the same priority as MOD, that is, just below unary minus but above *, /, and DIV.

BINARY FUNCTIONS

The LSH function performs a logical left shift of the digits of the binary equivalent of a number; the RSH function performs a logical right shift. These functions take the form

LSH(*v,n*) and **RSH(*v,n*)**

where v is an integer variable or expression, and n is the shift count. If the first argument is non-integer, the result will be meaningless. If the second argument is non-integer, it will be truncated.

For example, consider

LSH(X,Y) where X=53 (octal constant O65) and Y=1 to 3.

X and the results of the left shifts are listed below in binary, octal, and decimal representation.

	Binary	Octal	Decimal
X	000110101	65	53
LSH(X,1)	001101010	152	106
LSH(X,2)	011010100	324	212
LSH(X,3)	110101000	650	424

NOTE: The functions LSH and RSH are shifts rather than cycles of the registers.

Thus,

```
>10 X=53          !OR X=O65↵
>20 PRINT LSH(X,Y) FOR Y=0 TO 3↵
>30 !NOTE THAT LSH(X,0)=X↵
>RUN↵
 53
 106
 212
 424

>
```

The shift count may be positive or negative.

LSH(*v*,–*n*)=RSH(*v*,*n*)
RSH(*v*,–*n*)=LSH(*v*,*n*)

A shift count greater than 24 is equal to a shift count of 24.

```
>PRINT "VERIFIED" IF RSH(O63,24)=RSH↵
(O63,26)↵
VERIFIED

>
```

LOGICAL VARIABLES, EXPRESSIONS, AND OPERATORS

LOGICAL VARIABLES AND EXPRESSIONS

Every numeric variable or expression in SUPER BASIC is considered to have, in addition to a numeric value, a logical value which is either TRUE or FALSE. The logical value of a variable or expression is defined as TRUE if the numeric value is not zero, and FALSE if the numeric value is zero.[1] For example,

Expression	Numeric Value	Logical Value
A	0	FALSE
B	18	TRUE
C+2	-7	TRUE
R*S-4	0	FALSE

Thus, a numeric variable or expression can be used as the condition in an IF statement as follows:

10 IF X THEN 200

This statement specifies that if X is TRUE (not zero) the program will transfer to line 200. If X is FALSE (zero), the program will continue with the next statement in order.

More commonly used in the IF. . . THEN. . . statement to specify a condition, is an expression containing one of the relational operators. Note that a relational expression must have one of the logical values TRUE or FALSE and can, therefore, be considered as a logical expression. For example,

30 IF S = 0 THEN 70

causes a transfer to line 70 if the expression S = 0 is TRUE, and no transfer if S = 0 is FALSE.

The relational operators included in SUPER BASIC are:

Symbol	Meaning
<	less than
<=	less than or equal to
=	equal to
>=	greater than or equal to
>	greater than
# or <>	not equal to
<<	very much less than
>>	very much greater than
=#	approximately equal to

The last three operators listed require some explanation.

The logical value of an expression using << or >> depends on the internal sum of the values of the operands; that is,

A<<B is true if, internally, A+B=B
A>>B is true if, internally, A+B=A

The =# operator uses in its definition the special variable EPS, which SUPER BASIC presets to 10^{-10}.

A=#B is true if ABS(A/B-1)<EPS

EPS can be used in SUPER BASIC as any other variable and therefore can be set to any value by the user. Changing its value from the initial 10^{-10} would thus modify the definition of =#.

1 - If the variable or expression is complex, its logical value is set to the logical value of its real part. Complex arithmetic is discussed on page 25.

SUPER BASIC stores the logical value of an expression as either 1 or 0. A TRUE expression is set to 1 and a FALSE expression is set to 0. For example,

Expression	Logical Value
A = B	1 (for TRUE) if A = B, 0 (for FALSE) if A # B
C<D↑2	1 if C<D↑2, 0 if C>=D↑2

Thus

PRINT A = B *Prints 1 if A = B, prints 0 if A # B.*

Z = C<D↑2 *Sets Z = 1 if C<D↑2, sets Z = 0 if C>= D↑2.*

X = Y = 5 *Sets X = 1 if Y = 5, sets X = 0 if Y # 5.*

The Kronecker delta may be written with logical expressions. The Kronecker delta is defined

$$\delta_{ij} = \begin{cases} 1 \text{ if } i=j \\ 0 \text{ otherwise} \end{cases}$$

The Kronecker delta may be written in SUPER BASIC as

```
D=(I=J)
```

DECLARING LOGICAL VARIABLES

If a variable is declared to be a logical variable, it will be set to its logical value (1 for TRUE or 0 for FALSE) when a numeric value is assigned to it. To perform such a declaration, type the variable name (or names, separated by commas) in a LOGICAL statement, which can be executed either directly or indirectly. For example,

```
> 10 LOGICAL A,B↵
> 20 A = 18,B,C = 6↵
> 30 PRINT "A =":A:" AND B =":B↵
> 40 PRINT "BUT C =":C↵
> RUN↵
A = 1 AND B = 1
BUT C = 6

>
```

Since A and B were declared logical, their logical values were printed. Because 18 and 6 are non-zero (that is, TRUE), the logical value of both A and B was printed as 1.

The LOGICAL statement also can be used to declare that an array will store logical values. As it is declared, the array is dimensioned exactly as it would be in a DIM statement. No previous dimensioning is necessary. For example,

```
10 LOGICAL X(10),Y(4,N)
```

reserves space for a 10 element logical array X, and a 4 by N logical array Y.

The LOGICAL statement differs from DIM and other declaration statements in that the elements of an array declared LOGICAL are initialized to zero.[1] Thus, when statement 10 (the previous example) is executed, the elements of arrays X and Y are set to zero. This is true even if X and Y already have some elements defined when the LOGICAL statement is executed.

LOGICAL OPERATORS

In SUPER BASIC there are six logical operators which operate on logical variables and expressions. The result of a logical operation is a logical expression which is either TRUE (1) or FALSE (0).

The results of using logical operators where A and B are logical variables or expressions are shown in the following table:

T = True F = False

Operator	A B	T T	T F	F T	F F
AND	A AND B	T	F	F	F
OR	A OR B	T	T	T	F
EXCLUSIVE OR	A XOR B	F	T	T	F
EQUIVALENCE	A EQV B	T	F	F	T
IMPLICATION	A IMP B	T	F	T	T
NOT	NOT A:	If A is True, then NOT A is False. If A is False, then NOT A is True.			

Some examples of logical expressions containing logical operators are:

```
A AND NOT B
X = 3 OR X = 5
E*5>A-B XOR E<= 100
A#2*EXP(5) AND I=J
```

Relational and logical operations should be used with caution. The following examples illustrate the use of these operators.

1 - The INTEGER declaration statement also initializes array elements to zero.

```
>PRINT 5 AND 6↵
 1
>
```
The logical expression 5 AND 6 is evaluated. The value, 1, is printed.

```
>PRINT 5;6↵
 5      6
>
```
The numbers 5 and 6 are printed.

```
IF A=X OR Y THEN 50
```
Control transfers to statement 50 if A=X or if Y is unequal to zero.

```
IF A=X OR A=Y THEN 80
```
Control transfers to statement 80 if A=X or if A=Y.

```
>10 A=5, B=7↵
>20 PRINT A=B↵
>RUN↵
 0
>
```
The logical expression A=B is evaluated and printed.

```
95 PRINT SUBSTR(S,(A>B)-(A<B)+2,1)
```
The values, zero or one, of the logical expressions A>B and A<B are used to compute the second argument of the SUBSTR function.

```
PRINT L(N,(X>A)+(X>B)+(X>C))
```
The three logical expressions are evaluated, then added. For example, if X>A is false, and the other expressions are true, the statement is equivalent to PRINT L(N,2).

```
50 A=B=5
```
This is a logical statement. The expression B=5 is evaluated first. If the expression is true, A is set to 1. If the expression is false, A is set to 0.

```
50 A,B=5
```
This statement sets the variables A and B equal to 5.

Note that a logical operator works only with the **logical** value of what is on either side of it. Thus, X = 3 OR X = 5 may **not** be typed as X = 3 OR 5. The 5 will be considered to be true, since it is a non-zero value. Therefore, whatever the value of X, the expression X = 3 OR 5 always will be true. The correct form of the expression will operate as follows:

```
75 IF X = 3 OR X = 5 THEN NEXT X
```

If the value of X is 3 or 5, the expression is true and the THEN statement will be executed. If the value of X is neither 3 nor 5, the expression is false and the program will go on to the next line.

PRECEDENCE OF OPERATORS

The order of priority among the different types of operators in SUPER BASIC is as follows, in descending order:

Expressions in parentheses
Evaluation of functions
Exponentiation (↑)
Unary minus (-)
MOD, BAN, BOR, BEX
Multiplication and division (*, / and DIV)
Addition and subtraction (+ and -)
Relational operators (<, <=, =, >, >=, < > or #, <<, >>, =#)
NOT
AND
OR, XOR
IMP
EQV

For example, the following logical expressions are evaluated in the indicated order.

Example 1

```
A>B AND NOT R OR S
 1      2
 └──3──┘
 └─────4─────┘
```

1. Relational operator >
2. Logical operator NOT
3. Logical operator AND
4. Logical operator OR

Example 2

```
A AND C<D↑3 = B
        1
      2
    └─3─┘
  └─────4─────┘
```

1. Exponentiation ↑
2. First relational operator <
3. Second relational operator =
4. Logical operator AND

SECTION 3
STRINGS

STRING VARIABLES

Instead of assigning a numeric value to a variable, the SUPER BASIC user may set a variable equal to a string of characters. String variables make it possible to accept names, addresses, mixed alphabetic and numeric identification, and similar data as input from files or from the terminal. SUPER BASIC accepts strings of any length up to the limits of core at the time the string statement is executed.

A variable that is to be assigned a string value can be named in the same three ways as numeric variables: a single letter, a letter followed by a single digit, or a letter followed by $. Variable names for string arrays and arrays storing both strings and numbers can be, as for numeric arrays, a single letter or a letter followed by a $.

All forms of the PRINT command can be used to print strings. The effect of the comma, semicolon, and colon are the same for printing string variables as for printing any text enclosed in quote marks (explained in *PRINTING FEATURES*, page 49).

ASSIGNING STRING VALUES

A string value, like a numeric value, can be assigned to a variable with either an assignment statement, an INPUT statement or a READ statement (including INPUT FROM a file, and matrix input instructions).

When a variable is assigned a string value in an assignment statement, the string must be enclosed in single or double quote marks. For example,

```
> 10 S="ABC XYZ"↵
> 15 T='STRING2'↵
> 20 PRINT S,T↵
> RUN↵
ABC XYZ          STRING2

>
```

Every character inside the quote marks is accepted as part of the string. Remember that a Line Feed is used only for continuation in SUPER BASIC and therefore cannot be included in a string. Thus,

```
> T="STRING↓
S"↵

> PRINT T↵
STRINGS

>
```

Only the characters STRINGS are accepted as the value of T.

When a variable is assigned a string value with INPUT, INPUT FROM, or READ, and the variable has not been declared a string variable (explained below), the string must be enclosed in quote marks only when

- the first character of the string is numeric, that is, any of the digits 0-9, a decimal point, or a plus or minus sign;
- the string contains a comma, since a comma usually indicates the end of the value typed;
- the string contains leading spaces, since leading spaces are usually ignored on input.

Example 1: INPUT

```
> 10 INPUT A,B,C,D↵
> 20 PRINT↵
> 30 PRINT A;B;C;D↵
> RUN↵
? STRING3,"12345Q",XXX,"LA,999"↵

STRING3  12345Q  XXX  LA,999

>
```

The string values assigned to A and C did not have to be enclosed in quote marks. However, the leading digit of the string 12345Q and the comma included in LA,999 required quotes around those strings.

Example 2: READ

```
> 10 READ A,B,C,D↵
> 20 PRINT A;B;C;D↵
> 30 DATA ' SP',ZZ,'162',".DD"↵
> RUN↵
  SP   ZZ    162   .DD

>
```

The string value assigned to A had to be enclosed in quotes because of the leading space. The strings 162 and .DD required quotes because of the leading numeric character. Note that although the string assigned to C looks like a number, SUPER BASIC will not consider it as such. C will be treated as a group of characters having no numeric value.

DECLARING STRING VARIABLES

The user may declare that variables or arrays will be assigned string values. This declaration is accomplished by means of a STRING or TEXT statement, which may be executed either directly or indirectly. Although declaring string variables and arrays is not necessary, doing so will provide more efficient memory utilization and facilitate input of string values (as will be shown below).

Both arrays and non-subscripted variables can be declared in a STRING statement. As they are declared, the arrays are dimensioned exactly as they would be in a DIM statement. No previous dimensioning is necessary. For example,

10 STRING X,Y,A(5)

reserves space for array elements A(1) to A(5) and declares that the values assigned to X, Y and the array A will be strings.

A TEXT statement is used to declare arrays only. For each array declared in a TEXT statement, the maximum number of characters of an element is specified for all elements. This maximum number may be a variable or an expression. For example,

20 TEXT A(12):10,B(3,5):M*N

reserves space for a 12 element string array A, each element of which can contain up to 10 characters, and a 15 element array B with maximum string length equal to the value of M*N.

Since dimensioning arrays declared in the STRING or TEXT statement is the same as dimensioning in a DIM statement, the following is permitted:

- Dimensions may be variables or expressions

  50 TEXT J(Z):15,K(N+1,M+1):10

- The subscript base may be specified

  70 STRING C(-1:1),D(0:5,20)

An array can contain both numbers and strings. In this case the array would be dimensioned in a DIM and not in a STRING or a TEXT statement since the latter declare that all data will be string values.

ASSIGNING STRINGS TO DECLARED VARIABLES

INPUT And READ Statements

When string variables or arrays are declared, data assigned to them by means of an INPUT, INPUT FROM, or READ statement need not be surrounded by quote marks, with only two exceptions; the following strings **always** must be surrounded by quote marks, even if the variable has been declared:[1]

- A string containing a comma, such as "HART,S."
- A string containing leading spaces, such as " YES".

Example 1

```
> 10 STRING Q,R,S,T↵
> 20 READ Q,R,S,T↵
> 30 PRINT Q:R:S:T↵
> 40 DATA STRING,A23," SPACES "↵
> 50 DATA "MAY 3,1972"↵
> RUN↵
STRINGA23 SPACES MAY 3,1972

>
```

Quotes were typed around the string " SPACES " so that its leading space would be accepted. Without the quote marks, the space would have been ignored. "MAY 3,1972" was enclosed in quotes so that the embedded comma would be accepted as part of the string. Without the quote marks, SUPER BASIC would have stopped reading the value of T when it reached the comma; T would thus have been assigned the value MAY 3.

NOTE: Only commas and Carriage Returns (and **not** *spaces) may be used to separate string values that are not surrounded by quote marks.*

Example 2

```
> 10 TEXT A(3):15↵
> 20 INPUT A(I) FOR I = 1 TO 3↵
> 30 PRINT↵
> 40 PRINT A(I) FOR I = 1 TO 3↵
> RUN↵
? SMYTHE,ACCT. NO. 63794,"$1,630.75"↵
```

1 - If the R format is used, quote marks may be omitted, even in the following cases. See *The Single R*, page 59.

```
SMYTHE
ACCT. NO. 63794
$1,630.75

>
```

In the above example, array A is declared in a TEXT statement. The data need not be enclosed in quote marks. Quote marks were typed around the string "$1,630.75" to accept the embedded comma.

An array may be used to store both numeric and string data. In the following:

```
> INPUT S(I) FOR I = 1 TO 5↵
?  250,A STRING,3.75,XXX,"13.69"↵
>
```

S(1) and S(3) are numeric variables; S(2), S(4), and S(5) are string variables.

Assignment Statement

Strings in an assignment statement must be surrounded by quote marks whether or not the string variables have been declared. For example,

```
> 10 STRING A,B↵
> 20 A = "ONE"↵
> 30 B = "TWO"↵
> 40 C = "THREE"↵
> 50 D = A↵
> 60 PRINT A;B;C;D↵
> RUN↵
ONE   TWO   THREE    ONE

>
```

THE NULL STRING

While manipulating strings, a null string can be formed. This is the string "", which contains no characters.

If the VAR = ZERO command has been executed, the null string will be supplied for any undefined string variable. See *The VAR = ZERO Command*, page 6.

STRING CONCATENATION

Strings can be concatenated (joined together to form a new string) with a + sign, as illustrated below.

```
> 10 X = "XXX"↵
> 20 Y = "YYYY"↵
> 30 A = X+Y↵
> 40 B = X+"DEF"+Y↵
> 50 PRINT A;B↵
> RUN↵
XXXYYYY XXXDEFYYYY

>
```

Strings cannot be concatenated with numeric expressions; an error message will result.

STRING FUNCTIONS

To aid the user in manipulating strings, SUPER BASIC contains a number of standard functions that operate on or return string values. Three of these functions, COMP, MAX, and MIN, are used for string comparison and are explained on page 40. The functions described below are LENGTH, SPACE, VAL, STR, LEFT, RIGHT, SUBSTR, INDEX, ASC, and CHAR.

LENGTH(string)

This function returns a number equal to the number of characters in the specified string. For example,

```
> A = "JONES"↵
> PRINT LENGTH(A)↵
  5
>
```

SPACE(numeric expression)

This function returns a string consisting of as many spaces as specified by the argument. For example,

```
> 10 X = "XX",Y = "YYY"↵
> 20 A = X+SPACE(3)+Y↵
> 30 PRINT A↵
> 40 M = 2,N = 4↵
> 50 B = SPACE(M*N)+X↵
> 60 PRINT B↵
> RUN↵
XX     YYY
             XX

>
```

VAL(string)

This function takes a string of numeric information and returns a numeric value. For example,

```
> J = "1234"↵
> K = VAL(J)↵
```

would set K to the numeric value 1234. The string used as an argument of this function can contain numeric information only. X = VAL ("6E2") sets X to the value of 600, but X = VAL ("A123") would cause an error message to be printed. In addition, spaces within the argument string are ignored; thus, Y = VAL ("1.0 4") would set Y to the value of 1.04.

NOTE: If the argument of VAL is already a number, the same number is returned.

STR(numeric expression)

This function takes a numeric value and returns a string of numeric characters. For example, T = STR (99.6) sets T equal to a string variable with a string value of " 99.6". This string contains a leading space because of the omission of the + sign.

NOTE: If the argument of STR is already a string, the same string is returned.

LEFT(string, numeric expression)

This function takes the number of characters specified by the numeric expression, starting from the left side of the given string, to form another string. For example,

```
> T = "ABCDE"↵
> N = LEFT(T,2)↵
```

would give N the value of AB.

RIGHT(string, numeric expression)

This function takes the number of characters specified by the second argument starting from the right side of the given string to form another string. For example,

```
> PRINT RIGHT ("ABCDE",3)↵
CDE
>
```

When using LEFT and RIGHT, if the value of the numeric expression is less than or equal to zero, the result is the null string. If the value of the numeric expression is greater than the length of the string, the result is the string itself. For example,

```
> LIST↵
10 S="ABCDE"
20 R=RIGHT(S,0)
30 L=LEFT(S,-4)
40 G=LEFT(S,8)
50 PRINT LENGTH(R)
60 PRINT LENGTH(L)
70 PRINT G
> RUN↵
 0
 0
ABCDE

>
```

SUBSTR(string, numeric expression, numeric expression) Or SUBSTR(string, numeric expression)

This function extracts a substring from the string given as the first argument. The function can have either two or three arguments. The number given as the second argument specifies which character of the string is the first character to be extracted. The number given as the third argument specifies how many characters of the string are to be extracted. If the third argument is omitted, the substring starts with the character specified by the second argument and continues to the end of the string. For example,

```
10 X = "ABCDE"
20 Y = SUBSTR(X,2,3)
30 Z = SUBSTR(X,2)
```

will assign BCD to Y and BCDE to Z.

In the two argument form, if the second argument is less than or equal to zero, the result is the string. If the second argument is greater than the length of the string, the result is the null string. For example,

```
> S = "ABC"↵
> X = SUBSTR(S,9)↵
> Y = SUBSTR(S,-1)↵
```

sets X equal to the null string and Y equal to S.

In the three argument form, if the third argument is less than or equal to zero, the result is the null string. If the third argument is greater than the number of characters possible to be extracted, this argument is ignored, and the function is equivalent to the two argument form. For example,

```
> S = "ABCDE"↵
> X = SUBSTR(S,3,0)↵
> Y = SUBSTR(S,3,6)↵
```

sets X to the null string and Y to the string CDE.

INDEX(string, string) Or INDEX(string, string, numeric expression)

INDEX(string,string) searches the first argument for the string given as the second argument. If the second argument is a substring of the first, INDEX returns the character position of the second argument within the first; otherwise, it returns 0. When the optional third argument is not given, the search will start at the beginning of the first argument. Thus,

```
>10 X="ABCDE"↵
>20 Y=INDEX(X,"BCD")↵
>30 Z=INDEX(X,"E")↵
>40 W=INDEX(X,"F")↵
>50 PRINT Y;Z;W↵
>RUN↵
 2     5     0

>
```

Note that if X were changed to "ABCDEEEE" in the above example, Z would still be set to 5, since INDEX returns the position of the first occurrence of the second argument. But the search need not start at the beginning of the first argument. The third argument may be used to specify any desired position in the first argument where the search is to begin.

The third argument must be greater than zero. If the third argument is greater than the length of the string being searched, the result is zero.

Example 1

```
    INDEX("AXXA","A")    is 1
but INDEX("AXXA","A",2)  is 4
```

since the search beginning at position 2 does not find the A in position 1.

Example 2

```
>10 S="POSI TIONS OF THE SPACES",X=0↵
>20 X=INDEX(S," ",X+1)↵
>30 IF X#0 THEN PRINT X: ELSE STOP↵
>40 GO TO 20↵
>RUN↵
 5  11  14  18
>
```

This program searches the string S for spaces and prints the character position of each space. The first time line 20 is executed, S is searched from the beginning. Each subsequent time line 20 is executed, S is searched from one position after the last space found. When the INDEX is 0, there are no more spaces in the string and the program stops.

ASC(string)

This function returns the ASCII code of the first three characters of its argument. If the argument has fewer than three characters, the function returns the ASCII code of all the characters.

A table of these codes for all printing characters is given on the following page. The codes are listed in decimal as well as their usual octal representation.

Note in the examples below that the codes are returned in decimal rather than their usual octal representation, since SUPER BASIC automatically assumes the base 10.

Example 1

The ASCII code for % is 5; the code for A is 33 decimal (41 octal). Thus,

```
>PRINT ASC("%");ASC("A");ASC("ALPH")↵
 5     33     2174000

>
```

Example 2

```
>10 S=0↵
>20 STRING D$↵
>30 INPUT D$↵
>40 FOR I=1 TO LENGTH(D$)↵
>50 C=ASC(SUBSTR(D$,I,1))↵
>60 IF C>32 AND C<59 THEN S=S+1↵
>70 NEXT I↵
>80 PRINT "NUMBER OF ALPHA CHARS.=":S↵
```

This program accepts a string as input and considers the ASCII code of each character of the string. Any character whose code lies within the range 33-58 (41-72 octal, or A-Z) is alphabetic. The total number of such characters is computed and printed.

Octal ASCII Code	Decimal Equivalent	Character
0	0	SPACE
1	1	!
2	2	"
3	3	#
4	4	$
5	5	%
6	6	&
7	7	'
10	8	(
11	9	)
12	10	*
13	11	+
14	12	,
15	13	-
16	14	.
17	15	/
20	16	0
21	17	1
22	18	2
23	19	3
24	20	4
25	21	5
26	22	6
27	23	7
30	24	8
31	25	9
32	26	:
33	27	;
34	28	<
35	29	=
36	30	>
37	31	?
40	32	@
41	33	A
42	34	B
43	35	C
44	36	D
45	37	E
46	38	F
47	39	G
50	40	H
51	41	I
52	42	J
53	43	K
54	44	L
55	45	M
56	46	N
57	47	O
60	48	P
61	49	Q
62	50	R
63	51	S
64	52	T
65	53	U
66	54	V
67	55	W
70	56	X
71	57	Y
72	58	Z
73	59	[
74	60	\
75	61	]
76	62	↑ *or* ^
77	63	← *or* _
100	64	\
101-132	65-90	lower case[1]
134	92	|
136	94	~
137	95	rubout (end of file character)
140	96	null (Control @)
141-172	97-122	control letters[2,3]
173	123	ALT MODE/ ESCAPE
174	124	Control \
175	125	Control]
176	126	Control ↑
177	127	Control ← (abort)

NOTE 1) The internal code for lower case can be obtained by adding the decimal number 32 or the octal number 40 to the code of the corresponding upper case character. For example, lower case A is represented by decimal 65 or octal 101.

2) The internal codes for control characters can be obtained by adding the decimal number 64 or the octal number 100 to the appropriate representation for the specific alphabetic character. For example, since the code for A is decimal 33 or octal 41, the code for Control A is decimal 97 or octal 141.

3) A Line Feed followed by a Carriage Return may be generated by a Control J. A Carriage Return followed by a Line Feed may be generated by a Control M.

CHAR(numeric expression)

This function takes a numeric argument and returns the string character whose ASCII code is equal to the argument. The argument can be given as a decimal number or as an octal constant.[1] For example,

```
>PRINT CHAR(5),CHAR(33)  !DECIMAL NOS.
%                A

>PRINT CHAR(O05),CHAR(O41) !OCTAL
                                CONSTS.
%                A

>
```

The ASCII codes below can be used with CHAR to provide interesting printed results on some terminals:

Octal	Decimal	
102	66	Line Feed without a Carriage Return
105	69	Carriage Return without a Line Feed

Note the order in which the output of the following statements is printed.

```
>PRINT "FIRST":CHAR(66):"SECOND":
                        CHAR(69):"THIRD"
FIRST
THIRDSECOND

>PRINT I:CHAR(O102): FOR I=1 TO 5
 1
   2
     3
       4
         5

>
```

FILE NAMES AS STRING EXPRESSIONS

The name of the file specified in the OPEN statement and in the indirect LOAD and LINK statements[2] may be typed as a string variable or expression. In this way the file name can be assigned at the time the statement is executed. For example, if the beginning statements of a program are

```
10 STRING A
20 PRINT "TYPE THE INPUT FILE NAME"
30 INPUT A
40 OPEN A,INPUT,1
```

the following will occur:

```
TYPE THE INPUT FILE NAME
? XDATA
```

and the file XDATA will be opened for input as file 1 according to line 40.

Other Examples

```
55 LINK "FILE" + C$      Where C$ = "X", links
                         FILEX.

20 LOAD '(A2H)/@JOB'+STR(I)+ '/'
                         Where I = 3, loads
                         (A2H)/@JOB 3/.

60 OPEN B, OUTPUT, 3     Where B = "'SQR'", opens
                         'SQR' for output as file 3.
```

1 - Octal constants are explained on page 28.

2 - Explained in *LOAD and LINK*, page 100.

COMPARING STRINGS

Any of the relational operators <, < =, =, > =, >, < >, or # can be used to compare strings. A character by character, left to right comparison is done. The characters are compared according to the sequence listed above in the description of the ASC function.

Example

```
>A = "JUNE",B = "JULY"↵

>IF A>B THEN PRINT A:" > ":B↵
JUNE > JULY

>
```

The first two characters of the string values of A and B match, but since the letter N has a greater numeric code than the letter L, the string "JUNE" is greater than "JULY".

If the strings are of different lengths, the shorter string and the same number of characters from the longer string will be compared. If they match, the shorter string is taken to be the lesser of the two.

Example

```
>10 A = "SUN"↵
>20 PRINT "VERIFIED" IF A<"SUNDAY"↵
>RUN↵
VERIFIED
>
```

Some other examples of statements using string comparison are:

```
15 IF A#"PAID" THEN NEXT I
70 IF Z> = "SMITH" THEN PRINT TAB(15):Z
130 PRINT "XXX" IF A+B<"MR. JONES"
GO TO 95 UNLESS RIGHT(X,2) = "NG"
```

Comparing a string to a numeric value is permitted with the relational operators = and #. For example, if

```
A=33
```

the statement

```
IF A="END" THEN 160 ELSE 100
```

will cause a transfer to line 100.

STRING COMPARISON FUNCTIONS

The functions COMP, MAX, and MIN can take numeric arguments (as explained on page 17) or string arguments, as explained below.

COMP(S_1,S_2)

The function COMP(S_1,S_2), where S_1 and S_2 are string expressions, compares S_1 and S_2 and returns:

-1 if $S_1 < S_2$
 0 if $S_1 = S_2$
 1 if $S_1 > S_2$

Example

```
>10 X="DOG",Z="CAR"↵
>20 A=COMP(X,Z)↵
>30 B=COMP(Z,X)↵
>40 C=COMP(Z+"E","CARE")↵
>50 PRINT A;B;C↵
>RUN↵
  1    -1     0
>
```

MAX(S_1,S_2,...,S_n)

The MAX function can be used with two or more string arguments. The strings are compared and the value of the greatest argument is returned.

Example

```
>10 Q="STRING",R="STRING1"↵
>20 D(1)="DOBBS",D(2)="YOU"↵
>30 PRINT MAX(Q,R)↵
>40 PRINT MAX(D(1),D(2),Q,R)↵
>RUN↵
STRING1
YOU
>
```

MIN(S_1,S_2,...,S_n)

The MIN function can be used with two or more string arguments. The strings are compared, and the value of the least argument is returned.

Example

```
>10 Q="STRING",R="STRING1"↵
>20 D(1)="DOBBS",D(2)="YOU"↵
>30 PRINT MIN(Q,R)↵
>40 PRINT MIN(D(1),D(2),Q,R)↵
>RUN↵
STRING
DOBBS
>
```

SECTION 4

ASSIGNMENT AND CONTROL STATEMENTS[1]

THE MULTIPLE ASSIGNMENT STATEMENTS

More than one variable can be assigned the same value in one statement. The variables to be assigned must be separated by commas. For example,

```
10 X,Y = 5
70 LET A,B,C(2),D(1,1) = 0
X(1),Y,Z = 15*S/R
```

More than one assignment can be made in a single statement, as follows:

```
15 LET Q = 4, S = 16
30 A = 3,M,N = 5,W = COS(15)
100 J = SQR(X), K = J+3,H,G(1) = 0
```

The assignments are made from left to right; thus, in statement 100 above, the value of K is set to SQR(X)+3.

As shown above, use of the word LET is optional.

Be careful to note that each of the examples above is a **single statement.** Two separate statements cannot be typed on one line and separated by commas. For example, PRINT A, PRINT B is not acceptable, nor is B = C*EXP(C), PRINT A+B.

ADDITIONAL IF STATEMENT FEATURES

IF condition THEN statement

In addition to line numbers, SUPER BASIC statements may be typed after the word THEN in an IF-THEN statement. If the IF condition is false, the THEN statement will not be executed, and the program will go to the next statement in sequence.

Examples

```
70 IF X>4 THEN A = B
```
If X is greater than 4, A will be set to the value of B.

```
70 IF A = B THEN PRINT "A EQUALS B"
```
The message A EQUALS B will be printed only if A and B are equal.

THE IF-THEN-ELSE SEQUENCE

The word ELSE followed by a statement can be added to the IF-THEN sequence. This form allows the THEN statement to be executed if the condition is true, but executes the ELSE statement if the condition is false. The program continues to the next statement in order unless the THEN or ELSE clause it executes is one which transfers to another line.

Examples

```
70 IF X = .5 THEN 200 ELSE 300
```
If X is .5, the program will go to line 200; otherwise, it will go to line 300.

```
70 IF N = 0 THEN 50 ELSE C = T,D = T/N
```
If N is 0, the program will go to line 50; otherwise, the assignment statement in the ELSE clause will be executed, setting C to T and D to T/N.

```
70 IF A = B THEN PRINT "A EQUALS B"
   ELSE PRINT "A AND B NOT EQUAL"
```
If A and B are equal A EQUALS B will print; if not, A AND B NOT EQUAL will print.

Any indirect statement (except DATA, REM, or !) can be included in a THEN or an ELSE clause.

COMBINING IF STATEMENTS

Any number of IF-THEN and/or IF-THEN-ELSE sequences may be used together, such as:

```
IF X = 4 THEN IF P = L THEN R = 80 ELSE 300
                                ELSE X = X*Y
```

In this example, if X is not 4 (a false condition), the ELSE clause will set X to X*Y and the program

1 - The standard form of the assignment statement is discussed on page 4. The fundamental control statements are IF-THEN and GO TO, discussed on page 4, and FOR loops on page 8. This section contains extensions of these fundamental statements.

will continue with the next statement in order. If X is 4 (a true condition), the THEN clause will be executed to check to see if P is equal to L. If so, the value of R will be set to 80 and the program will continue; otherwise, the program will transfer to line 300.

The rule for matching THEN and ELSE clauses is similar to the rule for evaluating expressions with more than one set of parentheses. For example, the THEN and ELSE clauses in the previous example were matched from the inside out. Since a THEN clause does not require a matching ELSE, constructions such as the following are possible:

IF . . . THEN IF . . . THEN . . . ELSE . . .

Here the ELSE and the second THEN are matched with the inner IF. The outer IF has no ELSE clause.

IF . . . THEN IF . . . THEN IF . . . THEN . . . ELSE . . . ELSE . . .

IF . . . THEN IF . . . THEN . . . ELSE IF . . . THEN . . . ELSE . . .

In each of these examples the first IF in the statement has no ELSE clause, but if the ELSE were to be included, it would be added at the end of the statement.

ADDITIONAL FOR LOOP FEATURES

FOR value list

The FOR command can also be followed by a list of values for which the body of the loop is to be executed. For example, the following program prints the square roots of 2, 3, 8, 10, 12, 14, and 50:

```
10 FOR N = 2, 3, 8 TO 14 STEP 2, 50
20 PRINT N,SQR(N)
30 NEXT N
```

CALCULATION IN FOR LOOPS

If the FOR statement specifies an impossible range, that is, if the initial value is greater than the final value (less than the final value, for negative steps), the body of the loop will not be executed. SUPER BASIC will go to the statement following the corresponding NEXT. The loop variable retains the previous value, if any, assigned to it. For example,

```
>LIST ↵
100 FOR I=1 TO 10
110 NEXT I
120 PRINT I
130 FOR I=7 TO 2      This FOR loop contains an
140 NEXT I            invalid range.
150 PRINT I
>RUN ↵
  10
  10                  I retains its previous value,
                      10, and is not initialized
>                     to 7.
```

Once a loop is entered, if the NEXT statement has been omitted, SUPER BASIC will execute the body of the loop once (for the initial value) and then execute the rest of the program which follows the loop.

More complicated FOR statements are allowed. The initial value, the final value, and the step size may be expressions of any complexity. For example, if N and Z have been assigned values earlier in the program, we could write:

```
55 FOR X = N+7*Z TO (Z-N)/3 STEP N
```

Note however, that a change in the values of N and Z within the loop will change neither the final value of X nor the step size. Variables and expressions in a FOR statement are evaluated only once; namely, the first time the statement is encountered. The final value and step size will not change once the loop has been entered.

If the value of X in line 55 above were changed within the loop, this change would be accepted. For example, the following statements could be typed after line 55 to change the value of X to the value of N if X equals zero.

```
60 IF X = 0 THEN 70
65 GO TO 75
70 X = N
75   Body of loop
```

WHILE and UNTIL are often used in a FOR statement in place of the TO clause as a means of specifying the final value. For example,

```
50 FOR X = 1 WHILE X<=Y
```

which is equivalent to

```
50 FOR X = 1 UNTIL X>Y
```

The FOR loop will be executed from the initial value of X in steps of 1 as long as X is less than or equal to Y. Note that X always will be compared to the current value of Y, even if the value of Y should change within the loop; this is not true when the more common form of the FOR statement is used. For example, when

```
50 FOR X=1 TO Y
```

is encountered for the first time, the final value of X is set permanently to the value of Y at that time. Any changes of Y within the loop will not change this final value.

When a WHILE or UNTIL modifier is used with a FOR statement, the loop variable is first set to the initial value. The final value of the loop variable is that value which first causes the WHILE or UNTIL modifier to be false. For example,

```
>LIST
10 FOR I=3 WHILE I<9
20 NEXT I
30 PRINT I
40 FOR I=6 WHILE I<0
50 NEXT I
60 PRINT I
>RUN
  9
  6

>
```

In the first FOR loop, the value 9 is the first value which causes the WHILE condition to be false. In the second loop, I is set to 6 before the WHILE modifier is evaluated. Thus, the value 6 is the first value which causes the WHILE condition to be false.

NESTED LOOPS

It is often useful to have loops within loops. The order in which SUPER BASIC must execute these nested loops is illustrated in the following skeleton examples:

Allowed

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

Not Allowed

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

Allowed

```
FOR X
  FOR Y
    FOR Z
    NEXT Z
    FOR W
    NEXT W
  NEXT Y
    FOR Z
    NEXT Z
NEXT X
```

Nested FOR loops of any complexity are allowed, but crossed FOR loops are not allowed.

Note that the following construction is permitted:

```
110 FOR K . . .

140 . . .

170 NEXT K

200 FOR K . . .

220 GO TO 140
```

SUPER BASIC will execute the loop statements in the order

```
FOR K
NEXT K
FOR K
NEXT K
```

Thus, the way the statements appear in the program does not matter; only the order in which they are executed is important.

If the inclusion of nested FOR loops in a program results in two or more sequential NEXT statements, the NEXT statements may be combined and typed on one line as follows:

```
NEXT X
NEXT Y    } NEXT X,Y
```

COMPUTED GO TO STATEMENT

A computed GO TO statement, which may be executed directly or indirectly, causes transfer to one of several different parts of a program, depending on the value of a specified expression.

The form of the computed GO TO statement is

ON expression GO TO $line_1$,$line_2$,...

where $line_1$,$line_2$,... is a sequence of line numbers to which the program will transfer depending on the value of the expression. If the value of the expression is 1, the program will transfer to $line_1$; if the value of the expression is 2, the program will transfer to $line_2$, and so on. For example,

ON I*J GO TO 60,70,85

will transfer to lines 60, 70 or 85 depending on whether the value of the expression I*J is 1, 2, or 3, respectively.

If the value of the expression is less than one or greater than the number of line numbers, an error message will be printed. If the value of the expression is not an integer, the value will be truncated.

STATEMENT MODIFIERS

The number of statements in a program can be reduced greatly by using statement modifiers. One or more modifiers may be appended to most direct statements and to all indirect statements except DATA.

The statement modifiers are IF, UNLESS, FOR, WHILE, and UNTIL. For a complete list of those statements which can be modified, see *Appendix A.*

IF AND UNLESS

The IF modifier followed by a logical expression causes the command to which the IF clause is appended to be executed if the logical expression is **true**. The command is not executed if the logical expression is false. For example,

PRINT X IF X>0 *The value of X will be printed only if X is positive.*

GO TO 100 IF B *If B is not equal to zero (that is, true) the program will transfer to line 100. If B is zero (that is, false) no transfer will be made.*

Other examples using the IF modifier are:

```
30 INPUT N IF M<= SQR(7)
55 BASE I IF I#1
100 NEXT X IF G2 = 0
R = S IF Q>100
```

The UNLESS modifier followed by a logical expression causes the command to which the UNLESS clause is appended to be executed if the logical expression is **false**. The command is not executed if the logical expression is true. For example,

PRINT X UNLESS X>0 *The value of X will be printed only if X is not positive.*

Other examples using the UNLESS modifier are:

```
15 GOSUB 100 UNLESS X = 0
130 A = B↑2 UNLESS A = C
200 PRINT ON 2:Z UNLESS I<J
GO TO 55 UNLESS V*W = 1
```

FOR

FOR causes the command to which it is appended to execute repeatedly over a range of values. The FOR clause takes the same form as the FOR statement used in defining loops. For example,

```
>INPUT A(I) FOR I = 1 TO 4↵
? 6,-4,3,2↵
```

The command INPUT A(I) is executed repeatedly from the initial value of I to the final value of I (in steps of 1, since there is no STEP or BY clause).

```
>PRINT X-2 FOR X = 5,15,-9↵
 3
 13
-11
```

The command PRINT X-2 is executed for each value of X listed.

```
>PRINT X FOR X = 1 TO 6 STEP 2↵
 1
 3
 5
>
```

The command PRINT X is executed repeatedly from the initial value of X to the final value of X, in steps of 2.

Note that when FOR modifies an input or output statement, it modifies only the variable list and not the entire statement. Thus,

PRINT ON I: 16,FOR I=1 TO 3

is **not** the same as

```
FOR I=1 TO 3
PRINT ON I: 16,
NEXT I
```

Instead, it is the same as

PRINT ON I: 16,16,16,

The variable I must be defined previously for the statement to execute.

This rule should be kept in mind when using formatted input and output (which is discussed on page 51). For example, N must be previously defined if

PRINT IN IMAGE A(N): X FOR N=1 TO 4

is to execute.

WHILE AND UNTIL

WHILE followed by a logical expression causes the command to which the WHILE clause is appended to be executed repeatedly as long as the logical expression is **true**. For example,

X = 2*X WHILE X<Y

*X is reset to the value of 2*X repeatedly as long as X is less than Y. For example, if X were 1 initially and Y were 17, X would be reset to 32, since the last value of X to be multiplied by 2 would be 16.*

UNTIL followed by a logical expression causes the command to which the UNTIL clause is appended to be executed repeatedly as long as the logical expression is **false**. For example,

X = 2*X UNTIL X>= Y

*X is reset to the value of 2*X repeatedly until X is greater than or equal to Y. This statement is equivalent to the first example of WHILE above.*

UNTIL may be used with FOR in a similar manner as WHILE. For example,

```
>PRINT A↑2 FOR A = 1 STEP 2 UNTIL A>5↵
 1
 9
 25
```

The command PRINT A↑2 is executed repeatedly from the initial value of A (in steps of 2) as long as the UNTIL condition (A>5) is false.

The WHILE or UNTIL modifier must follow the complete FOR statement. For example,

50 PRINT A↑2+A FOR A=1 STEP 2 UNTIL A>7

is valid, but

50 PRINT A↑2+A FOR A=1 UNTIL A>7 STEP 2

is an unacceptable statement.

A modified indirect statement can be included in a THEN or an ELSE clause as any other indirect statement. For example,

```
IF Z THEN A(I) = B(I) FOR I = 1 TO 10 ELSE
                        J = J↑3 WHILE J<N
```

For modifies only the statement A(I) = B(I) in the THEN clause; WHILE modifies only the statement J = J↑3 in the ELSE clause, not the entire statement. An entire IF . . . THEN or IF . . . THEN . . . ELSE statement cannot be modified; neither can the IF part of these statements.

More than one modifier can be used to modify a single statement. The last modifier will be considered first, the next to the last modifier will be considered next, and so on.

Example 1

85 GO TO 105 IF A = B UNLESS N = 0

When this statement is executed, the condition N = 0 is checked first. If N is zero, the command GO TO 105 will not be executed. If N is not zero, the condition A = B is considered. If A and B are equal, the program will transfer to line 105.

Example 2

```
PRINT Y(I) FOR I = 1 TO 10 IF C(I) = P
PRINT Y(I) IF C(I) = P FOR I = 1 TO 10
```

These two statements are not equivalent. The first statement first checks to see if C(I) = P with I previously defined. If this is true, the values of Y(1) to Y(10) will be printed. The second statement checks for each value of I whether C(I) is equal to P. Those values of Y(I) for which C(I) = P will be printed.

Example 3

```
50 READ A(I,J) FOR I = 1 TO 3 FOR J = 1 TO 5
```

This statement is equivalent to

```
50 FOR J = 1 TO 5
60 FOR I = 1 TO 3
70 READ A(I,J)
80 NEXT I,J
```

First, J is set to 1 and values are read for A(1,1), A(2,1), and A(3,1), that is, for the first column of the array. Then J is set to 2 and so on, until finally, the last column is read in when J = 5. If the values were to be read in row order instead of column order, the statement would be typed as

```
50 READ A(I,J) FOR J = 1 TO 5 FOR I = 1 TO 3
```

MODIFIERS IN INPUT/OUTPUT STATEMENTS

Modifiers used in input/output statements control only the variable list — **not** the form, image, location, file number, or text. The form, image, location, file number, and text are evaluated before the modifier is executed. For example,

```
> LIST ↵
10 B(1) = "'B(1) '#/"
20 B(2) = "'B(2) '#/"
30 B(3) = "'B(3) '#/"
40 J=56.4
50 PRINT IN FORM B(I):J FOR I=1 TO 3
 > RUN ↵

ERROR IN STEP 50:
VARIABLE HAS NO VALUE
>
```

When step 50 is executed, the variable I has no value because the FOR modifier has not been executed at the time the form is evaluated. The program is corrected and executed below.

```
> LIST ↵
10 B(1) = "'B(1) '#/"
20 B(2) = "'B(2) '#/"
30 B(3) = "'B(3) '#/"
40 J=56.4
50 FOR I= 1 TO 3
60 PRINT IN FORM B(I):J
70 NEXT I
  > RUN ↵
B(1)        56.4
B(2)        56.4
B(3)        56.4

>
```

SECTION 5

USING DECLARATION STATEMENTS

Variables and arrays can be declared complex, double precision, real, integer, logical, or string. Declaration statements are sometimes necessary (for example, when complex or double precision variables are to be used) but can also be included in programs to save storage space and speed execution time.

The following chart lists DIM and the seven type declaration statements in SUPER BASIC.

Statement	Declares	Words Of Storage Per Variable Or Per Element Of An Array	Remarks
COMPLEX A,B,C(12)	Complex variables or arrays	4	Each complex variable or array element has a real and an imaginary part.
DOUBLE W(30),Z,D	Double precision variables or arrays	3	Variables or arrays declared DOUBLE can store up to 17 significant digits rather than only 11.
DIM Z(5),A(2,3)	Real, integer, string variables or arrays	2	It is always faster to declare the variable or array to be of a specific type where possible.
REAL X,Y(10),Z(N)	Real variables or arrays	2	Storage is the same as DIM for real variables or arrays, but REAL is faster.
INTEGER A,B(0:100)	Integer variables or arrays	1	The maximum number of words (elements) for an integer array is about 8000 in a program. Integer arrays are initialized to 0.
LOGICAL D,G,F(50)	Logical variables or arrays	1/24	Logical arrays are initialized to 0.
STRING M,N,A(2,3)	String variables or arrays	1/3 per character with a minimum of 2 words	If the number of characters of a STRING or TEXT element is not evenly divisible by 3, the remaining characters of the string occupy one full word. Use TEXT where possible; TEXT is always faster than STRING.
TEXT A(20):15, B(3,2):12	String arrays; specifies maximum element length	1/3 per character	

The way in which a variable is declared determines the type of value that the variable can store. The following example includes variables declared real, integer, and logical. Notice that the same value is assigned to each variable (lines 40 and 70), but a different value is stored in each, depending on the variable type.

```
>10 REAL R
>20 INTEGER I
>30 LOGICAL L
>40 R,I,L = PI
>50 PRINT R,I,L
>60 A=5, B=2
>70 R,I,L = A/B
>80 PRINT R,I,L
>RUN
  3.1415927      3          1
  2.5            2          1
>
```

real *integer* *logical*

Each variable is assigned independently of the other variables.

The result of A/B would be the same if A and B were declared INTEGER.

NOTE: When using binary data files, both single variables and arrays **must** *be declared before they are read or written, using the appropriate type declaration (INTEGER, REAL, DOUBLE, COMPLEX, LOGICAL, STRING, or TEXT). When reading values from a binary file, the variables and arrays should be declared to be of the same type as they were when written. Since the amount of storage used for a value stored on a binary file depends on its value type, SUPER BASIC cannot know how many words to read unless this is specified in a declaration statement.*

SECTION 6
INPUT AND OUTPUT STATEMENTS [1]

PRINTING FEATURES

PRINTING BLANK LINES

The PRINT command typed alone causes a Carriage Return to be printed. This form of the command is useful in making terminal output more readable by inserting blank lines. For example,

```
30 PRINT "LINE 1"
40 FOR I = 1 TO 4
50 PRINT
60 NEXT I
70 PRINT "LINE 2"
```

will cause four blank lines to be printed between LINE 1 and LINE 2.

THE PRINT ZONES

Separate PRINT commands cause the specified printout to be on separate lines. Thus,

```
100 PRINT "BOOK VAL"
110 PRINT X
```

prints BOOK VAL at the beginning of one line and the value of X at the beginning of the next line. The following program:

```
15 FOR I = 1 TO 12
20 PRINT I
25 NEXT I
```

will print the first twelve integers, each at the beginning of a line.

SUPER BASIC does, however, provide ways to print more than one number and/or string of text on one line. The characters to be printed fall into "zones", the length of which depends on whether the comma, semicolon, or colon is used in the PRINT statement.

Normal PRINT Zones

The width of the terminal paper is normally divided into five zones of fifteen spaces each. A comma is used in the PRINT statement to instruct SUPER BASIC to go to the beginning of the next zone. Thus, PRINT A,B,C,D,E will print the values of those five variables across the page. Each number will be left justified in a field of fifteen spaces. Any positive number will be preceded by a space due to the omission of the plus sign.

If there are more commas in a PRINT statement after the fifth zone is printed, printing will continue from the first zone on the next line. Thus,

```
10 FOR I = 1 TO 12
20 PRINT I,
30 NEXT I
```

will print the first five integers on one line, the second five on the next line, and 11 and 12 on a third line.

If another PRINT statement were added to this example, the first value or text listed in the additional statement would be printed in the zone immediately following the 12 (the third zone on the line). Thus,

```
> 10 FOR I = 1 TO 12 ↵
> 20 PRINT I,↵
> 30 NEXT I ↵
> 40 PRINT "XXX" ↵
> RUN ↵
 1          2          3          4          5
 6          7          8          9          10
 11         12         XXX
>
```

Inserting the statement 35 PRINT in the above example would have caused the XXX to print at the beginning of the next (fourth) line.

If text to be printed contains more than fifteen characters, it will extend into the next zone, and the next value or text to be printed will occupy the following zone. For example,

```
> PRINT "CURB WEIGHT (LBS) = ",A ↵
CURB WEIGHT (LBS) =                111
```

Zone 1 *Zone 2* *Zone 3*

The first string of text contains 19 characters. The value of A is printed in the third zone.

If the length of the text exceeds the length of the remaining zones, the text will be printed at the beginning of the next line.

1 - All input and output statements may be modified using the WHILE, FOR, and UNTIL modifiers described on page 44.

Packed PRINT Zones

A packed form of terminal output is available by using the semicolon in the PRINT statement. The semicolon instructs SUPER BASIC to skip from two to five spaces before printing the next number or text. The exact number of spaces depends on the last position in which SUPER BASIC printed before it encountered the semicolon.[1] For example,

```
> PRINT "CURB WEIGHT (LBS) = ";A↵
CURB WEIGHT (LBS) =      111
> PRINT "THIS IS";1;"EXAMPLE"↵
THIS IS      1     EXAMPLE
>
```

Concatenated PRINT Zones

To print numbers and/or text with no separating spaces, use the colon in the PRINT statement. Remember that positive numbers will be preceded by one space because of the missing plus sign. Thus,

```
> PRINT "CURB WEIGHT (LBS) =":A↵
CURB WEIGHT (LBS) = 111

> PRINT "B IS NEGATIVE":B:A↵
B IS NEGATIVE-76.3  111

> PRINT "CONCAT":"ENAT":"ED"↵
CONCATENATED

>
```

The following is not permitted in SUPER BASIC:

```
> PRINT "CURB WEIGHT (LBS) ="A↵
```

A comma, semicolon or colon must be inserted after the text.

Modified PRINT Zones

The LOL command allows the user to set the length of line desired and thus to modify the number of zones into which the display medium is divided. This command allows him to take advantage of wide carriage terminals and to produce output specially tailored to his needs.

The LOL command may be used as a direct command only. Its form is

LOL n

where n is an integer which specifies the length of the line in characters. The Carriage Return is included in the number of characters. LOL affects all output, whether to a file or to the terminal. Within each line, LOL does not affect the normal operation of the colon, semicolon, or comma in an output statement.

LOL remains in effect until another LOL command is executed.

Example

```
> 10 PRINT I: FOR I = 1 TO 14↵
> RUN↵
  1  2  3  4  5  6  7  8  9  10  11  12  13  14
> LOL 10↵
> RUN↵
  1 2 3 4
  5 6 7 8
  9 10 11
  12 13 14
> 10 PRINT I; FOR I= 1 TO 14↵
> RUN↵
  1          2          LOL 10 remains in effect.
  3          4
  5          6
  7          8
  9          10
  11         12
  13         14

>
```

CONCATENATION OF PRINT AND INPUT

When text is printed immediately before an INPUT command, the INPUT question mark need not appear on a separate line. A comma, semicolon, or colon at the end of the preceding PRINT statement will move the question mark to the end of that line. SUPER BASIC will wait there for the input. For example,

```
> 10 PRINT "WHAT IS X":↵
> 20 INPUT X↵
> 30 PRINT "X SQUARED= ":X↑2↵
> RUN↵
WHAT IS X?  15↵
X SQUARED =  225
>
```

SUPER BASIC provides another control to concatenate input with printed text. Instead of a Carriage Return, a D^C may be typed after the last item of data typed in reply to an INPUT command. The input will be accepted as usual, but the carriage will not be returned. Thus, any more text to be printed will appear on that same line rather than on the next line.

1 - The paper is divided into zones of three spaces each. SUPER BASIC first skips two spaces and then, if not positioned at the beginning of a zone, will move to the beginning of the next zone.

Example 1

```
>10 PRINT "B = ":↵
>20 INPUT B↵
>30 PRINT " (THIS IS THE BASE)"↵
>RUN ↵
B = ? 13D^C (THIS IS THE BASE)
>
```

In this example, the user typed a D^C instead of a Carriage Return after the requested input. SUPER BASIC then printed the text (THIS IS THE BASE) on the same line.

Example 2

```
>10 PRINT "WHAT IS R":↵
>20 INPUT R ↵
>30 PRINT "  S":↵
>40 INPUT S ↵
>50 PRINT "  T":↵
>60 INPUT T↵
>RUN ↵
WHAT IS R? -6D^C  S? 4D^C  T? 3↵

>
```

FORMATTING WITH IMAGE

In addition to the conventional ways that SUPER BASIC accepts input and prints output, the user can specify his own format for input and output by using "picture formatting". Format specifications can easily be made in PRINT IN IMAGE and INPUT IN IMAGE statements, as explained below. A more flexible, advanced method of formatting which uses FORM rather than IMAGE is explained under *FORMATTING WITH FORM* on page 56.

PRINT IN IMAGE STATEMENTS

The user may specify the exact format of his output by typing special characters in a string and using a PRINT IN IMAGE statement, as illustrated in the following example.

```
>10 INPUT A,B↵
>20 S ="E FORMAT #######, INTEGER %"↵
>30 PRINT IN IMAGE S:A,B ↵
>RUN ↵
?  200,5.67↵
E FORMAT  .2E+03, INTEGER 6

>
```

In this example, S is a string variable which specifies the picture format to be used. The # signs in the string caused A to be printed in E format; the % signs caused the value of B to be rounded and printed as an integer. All other characters in the string (including spaces) were printed as specified. The format symbols # and %, which are explained below, cannot be printed as part of the picture format because of their special significance.

A picture format also may be used to write on a data file. For example,

PRINT ON 3 IN IMAGE S:X*Y,Z,W *or*
WRITE ON 3 IN IMAGE S:X*Y,Z,W

will print the values of X*Y,Z and W on file 3 in the format specified by S.

The format in the above example is specified by a string variable, S. A string expression or the format string itself may also be typed after IN IMAGE. For example, all of the following are acceptable:

PRINT IN IMAGE "%%%%%":A

PRINT IN IMAGE LEFT(S,3):B

PRINT IN IMAGE S+T:C *where S = "%%%%%" and T = "%%.%"*

The picture format string can include any of the specifications listed below. The numeric fields will allow up to eleven significant digits of a number to be printed (or seventeen digits, if the number is double precision), depending on the number of symbols used in the format string. If the specified format cannot be used for the number to be printed (for example, if an insufficient number of places is specified), an error message will be printed.

Integer Field

One or more % signs denote an integer field. One % sign must be typed for each digit of the number to be printed. Negative numbers require an additional % sign because of the preceding minus sign. A non-integer value will be rounded if an integer field is specified for it. For example,

```
> A = 24, B = 174.78 ↵
>PRINT IN IMAGE "%% %%% %%%":A,-A,B ↵
24  -24  175
>
```

Integer fields are right justified; that is, if more % signs are specified than are necessary, leading spaces will be printed before the number. For example, the format "%%%" would cause 24 to be printed with one space before it, and 4 to be printed with two spaces before it.

Decimal Field

One or more % signs with an embedded decimal point denote a decimal field. The number to be printed will be rounded to the specified number of decimal places. If the number is an integer or has fewer decimal places than the format specifies, trailing zeroes will be printed. Negative numbers require an additional % sign because of the preceding minus sign.

Example 1

```
> 10 X = 175.65,Y = 11 ↵
> 20 D = "%%%.%% %%%%.%% %%.%" ↵
> 30 PRINT IN IMAGE D:X,-X,Y ↵
> RUN ↵
175.65 -175.65 11.0
```

Example 2

```
> 10 COMPLEX B ↵
> 20 B = CMPLX (.216,-.43) ↵
> 30 PRINT IN IMAGE ".%%% %.%%":B ↵
> RUN ↵
.216  -.43
```

Since B is a complex number, two fields are required for output.

Decimal fields are right justified; that is, if more % signs before the decimal point are specified than are necessary, leading spaces will be printed before the number.

NOTE: Whatever type of numeric field is specified in SUPER BASIC picture formatting, no more than eleven significant digits of a single precision value can be printed (no more than seventeen digits for double precision). If a single precision number containing more than eleven significant digits is printed with a field of more than eleven symbols, the following will occur:

- *Integer places past the eleventh significant digit will be filled with zeroes. For example, fourteen %'s will print the number 12345678901234 as 12345678901000.*
- *Decimal places past the eleventh significant digit will be replaced by blanks; for example, the field "%%%%%%%%.%%%%%" (in which eight %'s precede the decimal point and five follow it) will print the number 12345678.90123 as 12345678.901 followed by two blanks.*

The results are similar for printing a double precision value using a format of more than 17 symbols.

E Format Field

There are two forms for an E format field:

1. A series of seven or more # signs.
2. One or more # signs, followed by a decimal point and a series of five or more # signs.

If the first form is used, the number printed begins with a decimal point. The second form allows the user to specify the number of digits before the decimal point. This is shown as follows:

```
> 10 C = 500 ↵
> 20 PRINT IN IMAGE "#######":C ↵
> 30 PRINT IN IMAGE "##.#####":C ↵
> 40 PRINT IN IMAGE "##.#####":-C ↵

> RUN ↵
  .5E+03
  50.E+01
 -50.E+01
>
```

In the first form of the E format field, a minimum of seven # signs is needed.

a) The first # is for the leading space or minus sign of the mantissa (the number to the left of E).

b) The second # is for the decimal point of the mantissa.

c) The third # is for the minimum of one digit for the mantissa.

d) The fourth # is for the character E.

e) The fifth # is for the plus or minus sign of the exponent.

f) The sixth and seventh #'s are for the two digit integer exponent.

In the second form of the E format field, the # signs are used as follows:

a) A minimum of one # before the decimal point is for the mantissa.

b) Four #'s after the decimal point are for the exponential part.

c) The last # is for the leading space or minus sign of the mantissa.

Notice that in the case of a positive number in E format, the leading space must be accounted for and always will be printed, while the integer and decimal fields allow this space to be suppressed.

String Field

One or more % signs or seven or more # signs may be used to denote a string field. The number of symbols specified in the format determines how many characters of the string will be printed. For example, if A = "STRING", the format "%%%%%%" may be used to print A. The format "######" cannot be used, however, since it contains only six # signs. In the following example

```
>10 T = "CODE XY"↵
>20 PRINT IN IMAGE "%%%%%%%":T↵
>30 PRINT IN IMAGE "%%%%":T↵
>RUN↵
CODE XY
CODE
```

the entire string is printed first, then only four characters of the string are printed.

A string field is left justified; that is, if more % or # signs are specified than the number of characters in the string, trailing spaces will be printed.

Text In A Format

Any literal text may be included in the picture format string. Every character is printed exactly as it appears in the format, except for %, #, and a decimal point that is

- immediately following or preceding a % or a #, since then it is considered to be part of a field, or
- part of a $ or * field. These fields and a restriction on printing $ and * symbols as literal text are explained below.

For example, the results of a program calculating the perimeter P and the area A of a triangle may be printed as follows:

```
110 S = "PERIMETER IS %%.%, AREA IS %%%.%"
120 PRINT IN IMAGE S:P,A
```

Floating $ Field

This field is used to specify that a $ is to be printed immediately preceding an integer or decimal value (or a string). For example,

```
>R = "$$$.$$  $$$.$$  $$$$"↵
>PRINT IN IMAGE R:2.045,.7,300↵
  $2.05    $.70  $300
>
```

These formats printed the specified values as the % formats would have, except that the last of the preceding spaces is replaced by a $. The $ always floats to the position before the first digit. If the $ field is specified so that there are no preceding spaces (that is, no room for the $), SUPER BASIC prints an error message. For example, 23.06 cannot be printed with the format "$$.$$".

The $ field must consist of four or more $ signs. For example, "$$$" is not a legal field, nor is "$$.$", since each of these contains only three $ signs. If these illegal fields were included in a format string, the characters would be taken as literal text and not as field designators. For example,

```
>PRINT IN IMAGE "$%.%%":2.334↵
$2.33
>
```

Field designators

Text to be printed

The * Field

The * field is used to specify that * symbols are to appear before the number (or string) in place of the usual preceding spaces. For example,

```
>S = "****  **.**  ***.**"↵
>PRINT IN IMAGE S:23,8.625,3.2↵
**23  *8.63  **3.20
>
```

These formats printed the specified values as the % formats would have, except that each preceding space is replaced by a *. If the * field is specified so that there are no preceding spaces (no room for a *), SUPER BASIC prints an error message. For example, 19.72 cannot be printed with the format "**.**".

The * field has the same restriction as the $ field. A minimum of four symbols is necessary. In the following example, "***" is interpreted as literal text rather than a field specification and is printed as specified:

```
>PRINT IN IMAGE "***%%":"NOTE"↵
***NO
>
```

The * field is useful for check protection; that is, preceding *'s instead of spaces will prevent anyone from adding to the beginning of the dollar amount on a check.

Image Repetition

Since the "picture" specified in an IMAGE format is the image of a line, a Carriage Return is supplied when the format is exhausted. Thus, if more values are to be printed than the number of fields specified, more than one line of the same image will result.

Example 1

```
>PRINT IN IMAGE "%%":16.3,19↵
16
19
>
```

Example 2

```
>10 W="%   %   %.%%"↵
>20 PRINT IN IMAGE W:I FOR I = 1 TO 8↵
>RUN↵
1     2     3.00
4     5     6.00
7     8
>
```

The Single # Field

A single # may be used to specify what is known as "free form" output. Any number or string may be printed with this field. Up to eleven significant digits of single precision numbers (and up to fifteen of double precision) will be printed. If the single # field is used to print a string, the entire string will be printed. For example,

```
>10 A="STRING",B=68.9,C=666↵
>20 PRINT IN IMAGE "#":A,B,C,PI,DPI↵
>RUN↵
STRING
 68.9
 666
 3.14159265
 3.14159265358979

>PRINT IN IMAGE "#":123456789012345↵
   .123456789E+15

>
```

NOTE: The single # field may not be used for output to a fixed record length random file. See **Fixed Record Length File Input and Output Formats** *on page 81.*

INPUT IN IMAGE STATEMENTS

SUPER BASIC includes INPUT IN IMAGE statements as well as PRINT IN IMAGE statements. The field symbols used for formatted input are the same as those used for output.

Free Form Input Field

A single # may be used with INPUT IN IMAGE to specify what is known as a "free form" input field. This field will accept any string or numeric input.

All characters in the format string are printed except those which may be interpreted as IMAGE fields. For example, the # causes input to be accepted just as it would be from an unformatted INPUT statement, except that the INPUT question mark is suppressed. For example,

```
>10 S="TYPE A: #"↵
>20 INPUT IN IMAGE S:A↵
>RUN↵
TYPE A: 5↵
>
```

The text TYPE A: and a blank are printed. Notice that just as in unformatted input, leading blanks, commas, Line Feeds, or Carriage Returns typed before the 5 would have been ignored, and A^C, W^C, or Q^C could have been used to edit the input.

Three terminating characters are effective during free form input of string and numeric data. They are the comma, Carriage Return, and Control D. A fourth terminating character, a space, is valid for numeric input.

When more than one # field is included in a format, intervening text between fields prints as soon as a comma, Carriage Return, space, or D^C is typed. Input to a # field appearing at the end of an IMAGE must be terminated by a Carriage Return. For example,

```
>10 INPUT IN IMAGE "A=# B=# C=#":A,B,C↵
>!FIRST WE USE A COMMA TO TERMINATE↵
>RUN↵
A=7.3, B=6, C=19↵

>! NOW, REMOVE SPACES FROM THE↵
>! FORMAT AND TERMINATE WITH A↵
>! CARRIAGE RETURN↵
>10 INPUT IN IMAGE "A=#B=#C=#":A,B,C↵
```

```
> RUN
A=7.3
B=6
C=19
>
```

If the spaces had not been removed when the Carriage Return was used to terminate each field, a space would have been printed before both B= and C=.

As with unformatted terminal input, Control D can be used to terminate both string and numeric data. For example,

```
> 10 S="TYPE  X:# Y:# Z:#"
> 20 INPUT IN IMAGE S:X,Y,Z
> RUN
TYPE X:15Dc Y:A STRINGDc Z:2
```

End Of Image

Just as the Carriage Return is printed whenever an output IMAGE is exhausted, the end of an input IMAGE causes SUPER BASIC to seek a Carriage Return in the input; that is, all characters up to the next Carriage Return are ignored, or skipped.[1]

It is for this reason that a format such as

```
"A=# B=# C=#"
```

requires that the input to the last field be terminated by a Carriage Return. If any other terminator is used, the input will be accepted correctly, but since the end of the IMAGE is then encountered, SUPER BASIC looks for a Carriage Return and will not continue until one is typed. Using the Carriage Return to terminate the last field serves an additional purpose besides termination; that is, it satisfies the end of IMAGE scan.

Image Rescan

If an input format contains fewer fields than the number of variables listed in the INPUT statement, the format will be rescanned until all the variables have been given values. This means that each time the end of the IMAGE is encountered, SUPER BASIC will seek a Carriage Return as usual, and then rescan the input format from the beginning.

Example 1

```
> INPUT IN IMAGE "MONTH #DAY #":A,B,C,D
MONTH 6,DAY 17
MONTH 12,DAY 5

>
```

Example 2

This example uses the % input field, explained under *Fixed Length Input Fields*. If the data

```
12 16 43
36 99 20
```

is read from a file with the statement

```
30 INPUT FROM 2 IN IMAGE "%%":A,B
```

the value of A will be read as 12, all characters up to the next Carriage Return will be skipped, B will be read as 36, and another Carriage Return will be sought. The format will be satisfied once the second Carriage Return is seen, and SUPER BASIC will then proceed with the next statement in the program.

Fixed Length Input Fields

The free form field, a single #, can be used for output or input of any number or string. The other IMAGE fields, which use the symbols %, #(E format), $, or *, are more restrictive; they will print or accept only a specified number of characters. When these fields are used, the width of each field is of prime importance. No more (and no fewer) than the exact number of characters specified in each field will be printed or accepted as input.

Example

```
– COPY TEST TO T

12,34
THIS IS A TEST

– SBASIC

> 10 STRING S,T
> 20 OPEN "TEST",INPUT,2
> 30 INPUT FROM 2 IN IMAGE "%%%":S,T
> 40 PRINT S,T
> RUN
12,              THI

>
```

Since the field %%% is used to read S and T, exactly 3 characters are read from each line in the file. First, the characters 12, are read and assigned to S; then the end of the IMAGE causes SUPER BASIC to seek a Carriage Return. The IMAGE is then rescanned and the characters THI are read and assigned to T.

1 - During either input from or output to a fixed record length random file, the end of an IMAGE format corresponds to the end of a record rather than to a Carriage Return. See *Fixed Record Length File Input and Output Formats*, page 81.

Each of the fixed length fields can accept as input any characters that it is able to print as output. Since the numeric output fields print leading spaces if the value printed is shorter than the field, the numeric input fields will accept leading spaces. Any embedded or trailing spaces read will be converted to 0. Thus, the field %%% reads 12 preceded by a space as 12, and 12 followed by a space as 120. Also note that when the % field is used, non-numeric input is considered an error unless the variable has been declared string. Any character is accepted as input to a string variable when the % field is used.

When the fixed length fields are used to accept input from a data file, each non-field character in the format causes whatever occupies the next position in the file to be skipped during input. Only those character positions that are indicated as part of a field will be examined. For example, if A=123 is read from a data file with the format A=%%%, the value accepted will be 123. The leading non-field characters in the format cause the first two character positions of the input (the A and the =) to be disregarded. The purpose of this feature is to enable data written on a file with a particular IMAGE to be read with that same IMAGE.

NOTE: The free form (single #) field is recommended for IMAGE input from the terminal. Terminal input with fixed length IMAGE fields should be avoided.

FORMATTING WITH FORM

In addition to the line image type of picture format produced by IN IMAGE, SUPER BASIC provides a second type of format that uses IN FORM. The structure of the statements is similar, that is,

```
PRINT IN FORM S:A,B
PRINT ON 3 IN FORM S:X*Y,Z
WRITE ON 3 IN FORM S:SQRT(A),B+C
INPUT IN FORM S:M,N
INPUT FROM 6 IN FORM S:D,Z
```

However, the format is field-oriented rather than line-oriented. The picture format string will not be an image of the printed line, but will specify fields for whatever will be printed, whether numbers, strings, descriptive text, or blanks.

PRINT IN FORM STATEMENTS

Numeric Fields, String Fields, And Blanks

The symbols used to specify numeric and string fields are identical for IN FORM and IN IMAGE statements. However, the IN FORM decimal field requires at least one % before the decimal point. Thus, %.%% and %%. are valid fields, but .%%% is not.

Another major difference between the two types of format statements is that when IN FORM is used, blanks typed between fields in the format string serve to separate the fields but will not be printed. For example, if M = 12 and N = 56.88, the statement

```
PRINT IN FORM "%% %%.%%":M,N
```

will print the values of M and N with no spaces between them. The blank in the above format serves only to separate the field for M from the field for N. To print blanks between numbers, use one or more B's to denote blanks. Thus,

```
PRINT IN FORM "%% BBB %%.%%":M,N
```

will print the values of M and N with at least three spaces between them.

$ And * Fields

These fields used with PRINT IN FORM yield the same results as when used with PRINT IN IMAGE except that the sign of negative numbers is not printed. For example,

Field	Prints	IN IMAGE	IN FORM
$$$$	-16	$-16	$16
*****.**	-4.029	***-4.03	****4.03

Character And Field Replication

When IN FORM is used, the picture format can be written in a "shorthand" notation; that is, replication of characters and fields is permitted by using a multiplier. The following chart gives several examples of IN FORM character replication:

The Format	May Be Typed As
"%%%"	"3%"
"%%%%.%%%"	"4%.3%"
"#######"	"7#"
"##.#####"	"2#.5#"
"%% BBBB %%.%"	"2% 4B 2%.%"
"**********.**"	"10*.2*"

The user also may specify the number of times a field is to be used. The form of this field replication is

N(format field)

where N is the number of times the field is to be used.

Example 1

The format

"2(3%.2% B)"

is equivalent to

"%%%.%% B %%%.%% B"

Example 2

```
>10 A = 543.66,B = 78.743,C = 345.788
>20 G = "2(3%.3% 4B)  %%%"
>30 PRINT IN FORM G:A,B,C
>RUN
543.660        78.743        346
```

In this example, the field 3%.3% 4B is used twice (to print A and B); then the field %%% is used to print C.

Example 3

```
>PRINT IN FORM "3(3%)":16,5,-1
  16  5  -1
```

This statement specifies three integer fields of three symbols each, with no blanks between the fields, and therefore is equivalent to

```
>PRINT IN FORM "3% 3% 3%":16,5,-1
```

Example 4

The format

"20(4%.2% B 4(3% B)/)"

illustrating two levels of field replication, may be used to print twenty lines, each with a decimal number and four integer numbers. A / generates a Carriage Return (see below). *NOTE: Up to four levels of field replication are allowed in a format.*

Descriptive Text

When IN FORM is used, any literal text that is to be printed must be enclosed in single quote marks to denote a text field.[1] For example,

```
>10 D = " 'X EQUALS' B %.6%"
>20 X = PI/180
>30 PRINT IN FORM D:X
>RUN
X EQUALS .017453
>
```

Carriage Return In A Format

Unlike a format used in an IN IMAGE statement, no Carriage Return is given when the IN FORM format is exhausted. Thus if fewer fields are specified than the number of values to be printed, the format will be repeated on the same line as shown below.

```
>10 T ="% 2B %.% 2B"
>20 PRINT IN FORM T:I FOR I = 1 TO 5
>RUN
1   2.0   3   4.0   5
>
```

A slash (/) can be used in a format to generate a Carriage Return. Consecutive slashes may be used to generate blank lines. Note the results when the format above is modified to end with a / instead of 2B:

```
>10 T = "% 2B %.%/"
>20 PRINT IN FORM T:I FOR I = 1 TO 5
>RUN
1    2.0
3    4.0
5
>
```

When printing a matrix IN FORM, use the / to generate a Carriage Return at the end of each row. For example,

```
>MAT INPUT A(3,3)
?  1,3,-6,8,11,9,4,2,1

>MAT PRINT IN FORM "3(%% 2B)/":A
  1    3  -6
  8   11   9
  4    2   1

>
```

The Single

A single # may be used with PRINT IN FORM to specify what is known as "free form" format. Any number or string may be printed with this field. Up to eleven significant digits of a number will be printed. If the free form format is used to print a string, the entire string will be printed. For example,

1 - If the format string is enclosed in single rather than double quote marks, the literal text to be printed is enclosed in double quotes.

```
> 10 A = "STRING"↵
> 20 B = 68.9 ↵
> 30 C = 666 ↵
> 40 PRINT IN FORM "#":A,B,C,PI ↵
> RUN ↵
STRING 68.9 666. 3.14159265
> PRINT IN FORM "#":123456789012345 ↵
   1.23456789E+14
>
```

The Single R

On output, the single R field prints the string value in its entirety, followed by a Carriage Return. For example,

```
> 10 A="GGG", B="2XY", C="TESTC"↵
> 20 S="R R R"          !SAME AS 3(R)↵
> 30 PRINT IN FORM S:A,B,C ↵
> RUN ↵
GGG
2XY
TESTC

>
```

Note that the format S cannot be typed as "3R", since this is interpreted as "RRR", which is illegal. S can be typed as "3(R)" since this means 3 separate fields of one R each. In fact, in this example S in statement 30 can be replaced by "R", since this format will be used repeatedly until all variable values have been printed; that is, three times.

NOTE: During output to a fixed record length random file, the single R field writes one record on the specified file; it does not follow the record with a Carriage Return. Similarly, during input from a fixed record length random file, the single R field reads exactly one record. See **Fixed Record Length File Input and Output Formats**, *page 81.*

INPUT IN FORM STATEMENTS

SUPER BASIC includes INPUT IN FORM statements as well as PRINT IN FORM statements. The field symbols used for formatted input are the same as those for output.

Literal Text In The Format

No matter which input field is used, text in a FORM input format is **not** printed on the terminal; instead, each text character (or B, denoting a blank) causes the next input character to be skipped. In other words, only those character positions that are indicated as fields will be examined. Intervening text will be ignored. For example, if the format

```
"'A=' 3%"
```

is used to read the input A=123, the value accepted will be 123.

Output printed with a particular FORM can be read with that same FORM. Keep this in mind while learning formatted input: any input format will assume it is reading data created by an identical output format. Thus,

```
"'A=' 3%"
```

ignores two characters initially because it assumes that the two characters that appear first (A and =, or any other two characters) are not part of the actual input value. The next three characters (no more, no fewer) are assumed to constitute the input value, since exactly three characters would be printed using that same field for output.

End Of Form

Unlike the end of an IMAGE input format, the end of a FORM input format does not cause SUPER BASIC to seek a Carriage Return. This is analogous to the difference between an output IMAGE and an output FORM: the end of IMAGE always causes a Carriage Return to be printed, while the end of FORM does not.

To illustrate this, an example using the FORM single # field (explained below) will be given.

```
> 10 PRINT "B=":↵
> 20 INPUT IN FORM "#":B↵
> 30 PRINT " THIS IS THE BASE"↵
> RUN ↵
B=17, THIS IS THE BASE

>
```

A comma is used to terminate the input to the # field (17, the value of B). The end of the FORM is then encountered, but a Carriage Return is not sought. SUPER BASIC proceeds with the next statement in the program.

Form Rescan

If an input format contains fewer fields than the number of variables listed in the INPUT statement, the format will be rescanned from the beginning until all the variables have been given values. For example, the statement

```
INPUT FROM 2 IN FORM "4% 3%":A,B,C
```

accepts the value of A using the field 4% and B using the field 3%, encounters the end of the FORM, and then rescans the format from the beginning, using 4% to accept the value of C.

Carriage Return In A Format

When a / symbol is used to designate a Carriage Return in an input FORM, SUPER BASIC will seek a Carriage Return when the / is encountered. All input characters up to the next Carriage Return will be ignored.[1] This is done automatically at the end of every input IMAGE. Examples using the / will be shown below in the description of the various fields.

Free Form Input Field

The single # may be used with INPUT IN FORM to specify a free form input field. This field will accept any string or numeric input. Input is accepted just as it would be from an unformatted INPUT statement except that the question mark is suppressed. The same terminating characters are effective.

Example 1

```
>10 PRINT "VALUE OF X=":↵
>20 INPUT IN FORM "#":X↵
>30 PRINT " Y=":↵
>40 INPUT IN FORM "#":Y↵
>RUN↵
VALUE OF X=32, Y=-7.51↵
```

The colon after the text in each PRINT statement suppresses the Carriage Return that would otherwise be printed there. A comma is used to terminate the value of X; a Carriage Return terminates the value of Y. After either value, any of the following characters would be effective: comma, Carriage Return, space, or D^C.

Example 2

```
>10 PRINT "R,S= ":↵
>20 INPUT IN FORM "#":R,S↵
>RUN↵
R,S= 6,STR3↵
```

The input FORM contains only one field, but two variables are listed, so the format is used twice. The result would be the same if this FORM were replaced by "# #". Remember that blanks between FORM fields serve only to separate fields.

Example 3

In this example, the FORM is rescanned because only one field is specified but two variables are listed.

```
>5 PRINT "INPUT:"↵
>10 F="#"↵
>20 INPUT IN FORM F:M,N↵
>30 PRINT↵
>40 PRINT "USING ":F↵
>50 PRINT " M: ":M↵
>60 PRINT " N: ":N↵
>RUN↵
INPUT:
6 YEARS↵

USING #
  M:  6
  N: YEARS
```

Suppose line 10 is changed to

```
>10 F="#/"↵
```

Now the program is run for the new FORM F.

```
>RUN↵
INPUT:
6 YEARS↵
11 MONTHS↵

USING #/
  M:  6
  N:  11

>
```

After 6 is read as the value of M, terminated by a space, the / is encountered, causing all characters up to the next Carriage Return to be skipped. When the format is rescanned to read a value for N, the value is read, starting from the beginning of the second line of input, as 11.

NOTE: Either the free form (single #) field or the single R field (a string input field discussed below) is recommended for input in FORM from the terminal. Terminal input with fixed length FORM fields should be avoided.

The Single R

On input, the single R field reads all characters (including spaces, commas, quotes, and Line Feeds) up to the next Carriage Return. The Carriage Return is read but is not included in the string. Control characters are accepted as legal input except for A^C, W^C, and Q^C, which will perform their usual editing functions.

1 - During input from or output to a fixed record length random file, a / in a FORM format corresponds to the end of the record rather than to a Carriage Return. See *Fixed Record Length File Input and Output Formats*, page 81.

For example, if the data input file numbered 1 contains the entries

```
FIRST LINE ↵
2ND LINE OF INPUT ↵
   LAST,LINE ↵
```

the statement

```
40 INPUT FROM 1 IN FORM "R":A(I) FOR
                                I=1 TO 3
```

will set A(I) equal to the I^{th} line in the data file.

To illustrate and compare R and #, consider the input

```
THIS IS ↵
A TE,ST ↵
X Y Z ↵
```

The statement:	Assigns:
INPUT IN FORM "R":A,B,C	A="THIS IS" B="A TE,ST" C="X Y Z"
INPUT IN FORM "#":A,B,C *or* INPUT A,B,C	A="THIS IS" B="A TE" C="ST"
INPUT IN FORM "#/":A,B,C *or* INPUT IN IMAGE "#":A,B,C	A="THIS IS" ,C B="A TE" C="X Y Z"

PRECISE FORM CHARACTERS

The characters explained here provide the greatest flexibility possible in formatting. The user can have more precise control over input or output than the % and # fields allow. A single field can use several different kinds of characters. In addition to simple designations such as integer, decimal, and exponential, the precise FORM characters make possible such specifications as integer with sign following, or exponential with E suppressed and text embedded. Readability of output can be enhanced, and data file storage minimized.

Some of the characters described here have been mentioned previously in connection with the simple FORM characters, % and #, but the rules for using % and # have not been modified. The new characters explained here cannot be used in the same field as % and #, but a FORM can be constructed which contains both simple fields and precise fields.

A summary of the two types of characters, field and utility, is given below with an example of the use of each field character. More detailed information follows the summary.

Summary Of Field Characters

Each of the following numeric field characters can be used to comprise a field by itself, such as 6S or 13D, but can also be concatenated with other characters in the list to form a field, such as SDDD. *NOTE: Q can be used for output only.*

Character	Specifies	Example	
		Field	Prints 1004 As (*or* Reads 1004 From)
D	Digit; 0 (leading or embedded) printed	6D	001004
Y	Digit; 0 replaced by blank	6Y	1 4
Z	Leading digit; leading 0 replaced by blank	6Z	1004
Q	Left adjusted number; unneeded character positions suppressed	6Q	1004 *Output only*
*	Filling check protect	6*	**1004
$	Floating $	6$	$1004
S	Floating sign	6S	+1004
+	Sign if positive; floats	6+	+1004
–	Sign if negative; floats	6–	1004

The following characters can be used only in conjunction with other field characters. V and K can help save storage space in data files. *NOTE: There can be only one decimal point or V and one E or K per field.*

Character	Effect	Example	
		Field	Prints 1004 As (*or* Reads 1004 From)
.	Directs decimal scaling; decimal point printed	4D.DD	1004.00
V	Same as above, but decimal point suppressed	4DVDD	100400
E	Directs exponential interpretation of rest of field; E printed	3D.DESDD	100.4E+01
K	Same as above, but E suppressed	3D.DKSDD	100.4+01

None of the following can be concatenated with any other field character. In order for the decimal equivalent of a number to be read from hexadecimal, octal, or binary input using these fields, the variable must be declared integer.

Character	Specifies	Example	
		Field	Prints 1004 As (*or* Reads 1004 From)
H	Hexadecimal conversion	6H	0003EC
O	Octal conversion	8O	00001754
W	Binary conversion	24W	000000000000001111101100

String

The string field characters are listed in the table below.

Character	Prints or Reads	Example		
		Field	Prints	As
X	Any string character	6X	"A2B4"	A2B4 followed by two blanks
A	A-Z and blanks only	6A	"ABCD"	ABCD followed by two blanks
D	String characters 0-9; blank replaced by 0	6D	"1004"	100400
Y	String characters 0-9; 0 replaced by blank	6Y	"1004"	1 4 followed by two blanks

Summary Of Utility Characters

The characters listed in the following table do not print or accept values as do the field characters. These utility characters affect the format rather than the actual transmission and conversion of data. They can be used with any of the field characters summarized above.

Character(s)	Use	Field Terminator?
space	No use other than field termination	Yes
B	Prints blank on output; skips character on input	No
'text' *or* "text"	Prints enclosed text on output; on input, skips as many characters as are enclosed	No
/	Prints Carriage Return on output; seeks Carriage Return on input	Yes
\	Prints Line Feed on output; skips character on input	No
( )	Allow field replication by integer preceding the (. For example, 2(XA) is two fields, XA XA. Four levels of nesting allowed.	Yes
[ ]	Allow character replication by integer preceding the [. For example, 2[XA] is one field, XAXA. No nesting allowed.	No
,	Conditionally prints comma on output;[1] skips character on input	No

Further explanation of the precise FORM characters will concentrate on their use in output formats. The action of these characters when used to accept input can be implied from this explanation, since any input format will assume it is reading output created by an identical output format.

Field Termination

A space, /, (,), or the end of the FORM must follow each field to separate it from the next field, that is, to terminate each field. This is extremely important since concatenation of field characters is permitted. For example,

```
>10 PRINT IN FORM "4Q.DD/":16,25↵
>15 PRINT↵
>20 PRINT IN FORM "4Q. DD/":16,25↵
>RUN↵
16.00
25.00        4Q.DD field used twice.

16.25        4Q. used to print 16; DD used to print 25.

>
```

Embedded Text And Blanks

Since neither quotes enclosing text nor B's denoting blanks cause field termination, text characters and B's can be embedded in fields.

Example 1

```
>10 PRINT IN FORM "ZZV4BDD/":15.3,2.64↵
>15 PRINT↵
>20 PRINT IN FORM "ZZV 4B DD/":15.3,2.64↵
>RUN↵
15      30
 2      64        ZZV4BDD field used twice.

15      03        ZZV used to print 15.3; DD used to
                  print 2.64 (rounded to 3).
>
```

The first FORM used in this example contains one numeric field with embedded B's. The FORM in line 20 contains two numeric fields: ZZV, terminated by a space, and DD. The 4B causes four blanks to be printed between the output of these two fields.

1 - See *Conditional Field Characters*, page 65, Rule 4.

Example 2

```
>10 F="DD.DDK' X10↑'SQQ/"↵
>20 !ONE FIELD WITH EMBEDDED TEXT↵
>30 PRINT IN FORM F:1.56↵
>RUN↵

15.60 X10↑-1

>
```

Rounding Of Output

Consider the following example:

```
>PRINT IN FORM "DD/DD.D":12.65,12.65↵
13
12.7
>
```

The value to be printed by each field is 12.65. The DD field specifies no decimal places, so 12.65 is rounded to the integer 13 when printed. The DD.D field specifies one decimal place, so 12.65 is rounded to 12.7 when printed.

All SUPER BASIC fields will round values on output. If the integer part of a number has more digits than specified in the field used to print it, an error message will be given. Thus, DD cannot print 123. But if there are more digits to the right of the decimal point than there are field characters to print the decimal part, the number will be rounded. Thus, in the above example, DD specifies no decimal part, so 12.65 is printed as 13. Similarly, DD.D specifies only one decimal digit, so 12.65 is printed as 12.7.

Subfields

In explaining the interaction of the numeric field characters, the term "subfield" will be used. A subfield is

a. that part of a field which is used to print or accept a number without its exponent, *or*

b. that part of a field which is used to print or accept the exponent of a number.

Thus, in the FORM

"6D 3D.DESDD 5$"

- 6D, 3D.D and 5$ are subfields of type a. 6D and 5$ also constitute fields in themselves; 3D.D does not.
- SDD is a subfield of type b. It is not a field in this example; 3D.DESDD is a field.

It can be seen from this explanation that a field will contain more than one subfield only when there is an E or a K in the field. The first subfield will be for the mantissa; the second will be the exponent subfield.

D And Y In Numeric Fields

Output

The numeric field characters D and Y can be used to print digits. D will print leading, embedded, and trailing zeroes, while Y will replace each leading, embedded, or trailing zero with a blank. For example,

Field	Prints	As
4D 4Y	2307	2307 23 7
3D.3D YDD.DDY	15	015.000 15.00
D.DESDD Y.YESYD	600	6.0E+02 6. E+ 2

Since neither D nor Y will print the sign of a number, the values in the middle column above can be multiplied by −1 and the output (column 3) will be the same as shown. *NOTE: S, +, and − are the only precise FORM characters that will print the sign of a number.*

Input

D and Y are equivalent when used in input fields. Both will accept only digits or blanks; blanks are converted to zeroes. Any attempt to accept characters other than digits and blanks will cause an error message to be printed. For example, either 3D or 3Y will read the value 307 from either 307 or 3 7. Notice that D and Y will accept neither a decimal point nor the sign of a number.

As another example, consider the input

20660217

or

2 66 217

If A, B, C, and D are read from either of these input lines with the FORM

"4(DD)/" *or* **"4(YY)/"**

the values will be read as follows: A=20, B=66, C=2, and D=17.

Floating Versus Static: *, $, S, +, -

The occurrence of more than one successive *, $, S, +, or – indicates that a character will "float" to the position immediately to the left of the number printed. The character that will float and an example for each are listed here.

More Than One	Floats	Example		
		Field	Prints	As
*	* (fills any preceding positions with *'s)	3*	6	**6
$	$ (like S, +, and –, leaves spaces in preceding positions)	3$	6	$6
S	sign	3S	6 –6	+6 –6
+	+ if positive number; space if negative	3+	6 –6	+6 6
–	– if negative number; space if positive	3–	–6 6	–6 6

On the other hand, a single occurrence of any of the above field characters will not cause a character to float to a certain position. Instead, whatever would otherwise float is simply printed in the position specified. That position can precede or follow the number printed.

Thus, more than one successive *, $, S, +, or - implies that these are floating field characters, while a single occurrence of these characters is "static" and not floating.

Static	Prints	Example		
		Field	Prints	As
*	*	*DD	6	*06
$	$	$QQ	6	$6
S	sign	ZZS	–6	6–
+	+ if positive number; space if negative	+DDD	–102	102
–	- if negative number; space if positive	–YYY	–102	–1 2

More than one static occurrence of *, $, S, +, or – in a single subfield is permitted. For example,

```
>PRINT IN FORM "*$QQQQS*":-15↵
* $15-*
>PRINT IN FORM "SDD.D3B+B-/":16,13.2,-23↵
+ 16.0   +
+ 13.2   +
- 23.0   -

>
```

Note the following additional information on *, $, S, +, and –:

- **Floating * and $.** When these characters are used, at least one * or $ must be printed. If the number of characters used is too few for an * or $ to print, an error message will be given, indicating that the field is too short. For example,

  **** cannot print 1234, but can print –123 (sign is not printed)

  $$.$$ cannot print 16.32

- **The sign characters.** S, +, and – are the only precise FORM characters that will print the sign of a number. The sign of an exponent subfield can be printed only in front of the exponent. After the exponent, a static occurrence of S, +, or – will print the sign of the mantissa. For example,

```
>PRINT IN FORM "SDDESDD3B+B-/": ↵
  1600,-1600↵
+ 16E+02   +
- 16E+02   -
>     refer to mantissa
```

Conditional Field Characters

The conditional field characters are Q, Z, and the floating characters (more than one successive *, $, S, +, or –). Certain rules must be followed when concatenating these with other characters. The rules are:

Rule 1.

Two different conditional field characters cannot appear in the same subfield. For example,

```
$$$ZZZZ
****$$$$   —— not permitted
+++++.QQ
```

The field

QQQQESSS

is permitted because the Q's constitute one subfield and the S's constitute another.

A single *, $, S, +, or – can appear in the same subfield as a conditional field character. Rule 1 does not prohibit this, since the conditional characters include only the floating and not the static use of these symbols. Thus, the following fields are permitted:

Field	FIELD CHARACTERS USED	
	Static	Conditional
*QQQ	*	Q
$– – – –	$	floating –
5$.2$BB+B–	+ and –	floating $
SZZZE–QQ	mantissa:S exponent:–	mantissa:Z exponent:Q

Rule 2.

The conditional field characters can be used before but not after D or Y in a subfield. For example,

$$$DDDD
+++++YD ——— *permitted*
SSS.DD

but

DDDD$$$
YD+++++ ——— *not permitted*
DD.SSS

A conditional field character (floating –) appears after D in the field

DD.DK– – –

but since the D's are in one subfield and the –'s are in another, this field is permitted.

Rule 2 does not prohibit such fields as

DDDS
6Y–

since here the S and – are static, not floating, and therefore are not conditional field characters.

Rule 3.

If conditional field characters are used to print the decimal part of a number, the result will be the same as if D's had been used instead. For example, the following pairs of fields are equivalent:

Field	Same As
$$$$.$$	$$$$.DD
SSSVSSS	SSSVDDD
$– – –.– –	$– – –.DD
ZZ.ZESQQ	ZZ.DESQQ

Rule 4.

When the comma is used with conditional field characters, a comma will print only if a digit of the number being printed appears to the left of the comma. When the comma is used with D or Y, a comma prints unconditionally. For example,

Field	Prints	As
,.DD	1203.6 32.61	** 1,203.60 *****32.61
$$,$$$–	2341.7 –28	$2,342 $28–
SQ,3Q,3Q.3D	24369.6 –315.01	+24,369.600 –315.010
Z,ZZZ	1200 39	1, 200 39
but . . .		
–DD,DDD.DD	–1337.4 16.25	–01 ,337.40 00 ,016.25
Y,YYY	3297 8	3,297 , 8

H, O, and W

H, O, and W fields can be used to specify hexadecimal, octal, and binary conversion, respectively. For example,

```
>10 C="6H 3B 8O/ 24W"↵
>20 N=7293891↵
>30 PRINT IN FORM C:N,N,N↵
>RUN↵
6F4BC3   33645703
011011110100101111000011
>
```

Although H, O, and W cannot be concatenated with any field characters, they can be used with the utility characters of FORM. In the following example, blanks are embedded in a field of W's:

```
>PRINT IN FORM "8[3WB]":652190↵
000 010 011 111 001 110 011 110
>
```

Hexadecimal, octal, or binary input can be converted to decimal using H, O, or W respectively, but only if the input value is read into a variable that has been declared integer. If the variable is undeclared or declared string, the input value will be read as a string. No other type of declaration is permitted. For example, if the file HEX contains the data

```
0003EC
0003EC
```

then . . .

```
>10 INTEGER A
>20 OPEN "HEX",INPUT,3
>30 INPUT FROM 3 IN FORM "6H/":A,B
>40 PRINT A:"   (READ INTO A)"
>50 PRINT B:"   (READ INTO B)"
>RUN
  1004   (READ INTO A)
0003EC   (READ INTO B)

>
```

Since A is declared integer in this example, the value read into A is the number 1004, which is the decimal equivalent of hexadecimal 0003EC. Since B is undeclared, the value read into B is a string comprised of the input characters 0003EC.

String Field Characters

The string field characters are X, A, D, and Y. X and A operate on strings only. X will print or accept any string character, while A will print or accept only the letters A through Z and blanks. D and Y act similarly in string fields as they do in numeric fields. Both operate on the characters 0 through 9 and blanks only. On output, D will print the string character 0 in place of a blank, and Y will print a blank in place of the string character 0. On input, D and Y are equivalent; blanks are converted to zeroes. *NOTE: Concatenation of X and A with any numeric field characters other than D and Y is not permitted.*

If D or Y is concatenated with any numeric field characters, the field is numeric. Similarly, D or Y concatenated with A or X results in a string field. If D, Y, or both are the only characters in a field, the field type will be taken from the variable type. This means that such a field can print a numeric or a string value, but can accept a string value only if the variable is declared string. For example,

```
-COPY DATA1 TO T
6  3
86 1

-SBASIC

>10 STRING A,B
>20 OPEN "DATA1",INPUT,1
>30 INPUT FROM 1 IN FORM "4D/4Y/":A,B
>40 PRINT
>50 PRINT A,B
>RUN

6003          8601

>
```

Since A and B are declared string, D and Y are interpreted as string field characters (line 30). Notice that the blanks typed as input to A and B are converted to zeroes.

A string field will print or accept no more (and no fewer) than the exact number of characters specified in the field. On output, if the field width is longer than the string to be printed, trailing spaces will be appended to the string (trailing 0's if D is used). If the field width is shorter than the string to be printed, only the specified number of characters will print. For example,

```
>10 T="8X 6X 3X", A="SUNDAY"
>20 PRINT IN FORM T:A,A,A
>RUN
SUNDAY   SUNDAYSUN
```

printed by 8X — *printed by 6X* — *printed by 3X*

To illustrate string input fields, suppose that Q, R, and S are read from

```
SUNDAY     SUNDAYSUN
```

with the FORM "8X 6X 3X". The result would be Q="SUNDAY ", R="SUNDAY" and S="SUN".

In the following example, only one character of the input is read.

```
-COPY DATA2 TO T

YES
NNO
D1652840
```

```
– SBASIC ↵

> 10 OPEN "DATA2",INPUT,6 ↵
> 20 ON ENDFILE(6) GO TO 70 ↵
> 30 INPUT FROM 6 IN FORM "X/":A ↵
> 40 PRINT "VALUE READ: ":A ↵
> 50 PRINT ↵
> 60 GO TO 30 ↵
> 70 CLOSE 6 ↵
> RUN ↵
VALUE READ: Y

VALUE READ: N

VALUE READ: D

>
```

The X in the format reads one character; the / causes all characters up to the next Carriage Return to be skipped.

The most valuable use of A and D is to check the composition of string input that must be of a certain type (alphabetic, numeric, or a particular alphabetic-numeric combination). For example, if the input is supposed to contain nothing but the characters A through Z and blanks, and the field character A is used, then any illegal input will cause an error message to print.

The following example illustrates the use of X and Y in printing the string returned by the DATE function. DATE usually prints a string that looks like

```
01/12  09:36
  |    |  |
date two time
   blanks
```

Consider the effect of printing this string with the field Y2XY3XY4X, which is equivalent to YXXYXXXYXXXX. The Y's are situated so that a leading 0 in either part of the date or in the hour designation of the time will be replaced by a blank when this field is used. For example,

```
> 10 PRINT DATE ↵
> 20 PRINT IN FORM "/Y2XY3XY4X/":DATE ↵
> 30 PRINT IN FORM "/Y2XY3X4BY4X/":DATE ↵
> 40 S="'DATE:  'Y2XYX\'TIME:'2XY4X/" ↵
> 50 PRINT IN FORM S:DATE ↵
> RUN ↵
03/20  09:24          Usual printout.

 3/20  9:24           No leading 0's.

 3/20     9:24        Four extra blanks embedded.

DATE:  3/20           Text and Line Feed embedded.
TIME:  9:24

>
```

The X format is useful in entering strings which contain quotes and commas. These characters normally terminate the input, but are accepted as text with the X format. For example,

```
> 10 STRING T ↵
> 20 INPUT IN FORM "33X/":T ↵
> 30 PRINT IN FORM "33X/":T ↵
> RUN ↵
THIS IS A GOOD EXAMPLE, ISN'T IT? ↵
THIS IS A GOOD EXAMPLE, ISN'T IT?

>
```

Carriage Return And Line Feed: / and \

The / symbol generates a Carriage Return on output. On input, / causes all characters up to the next Carriage Return to be skipped. The / is a field terminator. Thus, the FORM "3X/3X/" contains two fields, as illustrated below.

```
> 10 A="FIRST", B="SECOND" ↵
> 20 S="3X/3X/"          !OR "2(3X/)" ↵
> 30 PRINT IN FORM S:A,B ↵
> RUN ↵
FIR
SEC

>
```

The first field, 3X, prints three characters of the string A. A / separates the field for A from the field for B and causes a Carriage Return to be printed. The second field, also 3X, prints three characters of B and the subsequent / prints another Carriage Return.

The \ symbol (shift L on some terminals) generates a Line Feed on output. On input, \ causes the next input character to be skipped. The \ is not a field terminator. Thus, the FORM "3X\3X\" contains only one field.

Example

```
>10 A="FIRST", B="SECOND"↵
>20 S="3X\3X\"          !OR "2[3X\]"↵
>30 PRINT IN FORM S:A,B ↵
>RUN ↵
FIR
ST
SEC
OND

>
```

Here the field 3X\3X\ is used twice: first to print the value of A and then to print B. This field specifies that after every 3 characters of each value, a Line Feed will be printed. Since A is only 5 characters long and 6 field characters are specified in the output field, a trailing blank would normally be printed after the value of A. However, since only a Line Feed follows, there is no need to print the trailing blank, so SUPER BASIC does not do so. *NOTE: The suppression of blanks before a Line Feed or a Carriage Return applies to terminal output only.*

Field And Character Replication: ( ) and [ ]

Parentheses may be used to specify multiple use of a field or fields. For example,

Equivalent:		
3(DD)	*and*	DD DD DD
3(3Z B)	*and*	3Z B 3Z B 3Z B
2(AX 4X)	*and*	AX 4X AX 4X

The number preceding the (may be any integer. If negative or zero, this number will cause whatever is enclosed in the parentheses to be ignored. For example,

```
>05 PRINT↵
>10 PRINT "REPLICATION FACTOR IS -1"↵
>20 PRINT IN FORM "D 2B D -1(2B D)/": ↵
N FOR N=1 TO 4↵
>30 PRINT "REPLICATION FACTOR IS 0"↵
>40 PRINT IN FORM "D 2B D 0(2B D)/": ↵
N FOR N=1 TO 4 ↵
>50 PRINT "REPLICATION FACTOR IS 1"↵
>60 PRINT IN FORM "D 2B D 1(2B D)/": ↵
N FOR N=1 TO 4↵
>RUN ↵

REPLICATION FACTOR IS -1
1  2
3  4
REPLICATION FACTOR IS 0
1  2
3  4
```

The effective FORM when the factor is -1 or 0 is: "D 2B D/".

```
REPLICATION FACTOR IS 1
1  2  3
4
>
```

The effective FORM when the factor is 1 is: "D 2B D 2B D/".

Four levels of nesting are allowed. An example of two levels of nesting is the format

"20(–4Z.DD B 4(–3Z B)/)"

which can be used to print twenty lines, each consisting of a decimal number and four integers.

Character replication is denoted either by an integer before the character or, if more than one character is to be replicated, by an integer before the first square bracket enclosing the characters. *NOTE: E, K, ., and V cannot be replicated.* For example,

Equivalent:		
4Z	*and*	ZZZZ
Y3D	*and*	YDDD
6[HB]	*and*	HBHBHBHBHBHB
$Z2[,3Z]	*and*	$Z,ZZZ,ZZZ

Nesting the brackets (such as 2[2[YD]B]) is not permitted.

If the number preceding either the character to be replicated or the [is zero, the characters which otherwise would be replicated are ignored. For example,

"0[DB] 3D/" is the same as "3D/"
"1[DB] 3D/" is the same as "DB 3D/"
"2[DB] 3D/" is the same as "DBDB 3D/"

A minus sign followed by a number, if placed before a character or a [, is interpreted as follows:

- The minus sign is interpreted as a static –. A – will print if the output value is negative, and a space will print if the value is positive.

- The number following the – is assumed to be the number of times that the particular character or characters are to be replicated.

For example,

"–2[DB] 3D/" is interpreted as "–DBDB 3D/".

The STR function may be used to create a variable replication factor. For example, STR(I) may be concatenated to "[DB] 3D/".

If I=-2, STR(I)+"[DB] 3D/"
is equivalent to
"-2[DB] 3D/".

If I=0, STR(I)+"[DB] 3D/" is equivalent to "3D/".

Notice that [ ] differ from ( ) in that using brackets results in a single field while parentheses act as field terminators and thus result in more than one field. Compare 2(XA) with 2[XA]. The former is two fields, XA XA, while the latter is one field, XAXA.

Examples Using Precise Form Characters

Example 1

This simple example lists the numbers 0 through 7 and their binary equivalents. A space is printed before each binary digit. Notice that the FORM in line 10 prints Carriage Returns, text, and blanks, but no variable values. There are only utility characters (not field characters) in the FORM.

```
>LIST↵
10 PRINT IN FORM "/' DECIMAL' 4B 'BINARY'/"
20 S="3B D 7B 3[BW]/"
30 PRINT IN FORM S:J,J FOR J=0 TO 7
>RUN↵

 DECIMAL     BINARY
    0        0 0 0
    1        0 0 1
    2        0 1 0
    3        0 1 1
    4        1 0 0
    5        1 0 1
    6        1 1 0
    7        1 1 1

>
```

Example 2

In this example, numbers are accepted from the terminal, added, and printed (together with the sum) in four different ways. The variable F, which is part of the formats used (lines 80 and 90), is changed three times.

```
>LIST↵
10 VAR = ZERO
20 DIM A(0:100)
30 PRINT "ENTER THE AMOUNTS, TERMINATE WITH 'END'"
40 INPUT A(I) FOR I=1 UNTIL STR(A(I-1)) = "END"
50 F = "$8Z.ZZ-/"
60 S = S+A(J) FOR J=1 TO I-2
70 PRINT
80 PRINT IN FORM "7B"+F: A(J) FOR J=1 TO I-2
90 PRINT IN FORM "21'-'/'TOTAL: '"+F:S
```

```
>RUN↵
ENTER THE AMOUNTS, TERMINATE WITH 'END'
? 1204.61,-36209.33,1209642.11,43260.40,-5163↵
? 307689.29,END↵

        $     1204.61
        $    36209.33-
        $ 1209642.11
        $    43260.40
        $     5163.00-
        $  307689.29
---------------------
TOTAL: $ 1520424.08

>!CHANGE TO FLOATING $↵
>F="9$.$$-/"↵

>GO TO 70↵

           $1204.61
          $36209.33-
        $1209642.11
          $43260.40
           $5163.00-
         $307689.29
---------------------
TOTAL:  $1520424.08

>!CHANGE TO INCLUDE COMMAS↵
>F="2[3$,]3$.$$-/"↵

>GO TO 70↵

            $1,204.61
           $36,209.33-
       $1,209,642.11
           $43,260.40
            $5,163.00-
          $307,689.29
---------------------
TOTAL:  $1,520,424.08

>!CHANGE TO INCLUDE SPACES AND ALL SIGNS↵
>F="2[3$,]3$.$$BBS/"↵

>GO TO 70↵

            $1,204.61  +
           $36,209.33  -
       $1,209,642.11  +
           $43,260.40  +
            $5,163.00  -
          $307,689.29  +
---------------------
TOTAL:  $1,520,424.08  +

>
```

Example 3

In this example, data is read from a file and printed on the terminal using a field of D's. The first two characters of each line of input are skipped, as specified by BB. Then 5D reads five characters (which must be numeric), and a / causes all characters up to the next Carriage Return to be skipped. The input is then printed using the same FORM; BB prints two blanks, 5D prints five digits, and / prints a Carriage Return.

```
-COPY ADZ TO T↵

CA95112SAN JOSE
NY11423HOLLIS
NJ07047NOBERGEN
CA94040MTNVIEW
NJ07110NUTLEY
NM87544LOSALAMOS
CA90241DOWNEY
CA94025MENLOPARK
NM87100ALBUQUERQUE
  00000



-SBASIC↵

>LOAD SAMPLE↵
>LIST↵
10 OPEN "ADZ",INPUT,3
20 S = "BB5D/"
30 FOR I = 1 TO 1000
40 INPUT FROM 3 IN FORM S:N
50 IF N = 0 THEN GO TO 80
60 PRINT IN FORM S:N
70 NEXT I
80 CLOSE 3
  >RUN↵
  95112
  11423
  07047
  94040
  07110
  87544
  90241
  94025
  87100

>
```

Example 4

This program will accept any string and print the string 10 characters per line. A Line Feed is printed after every 10 characters, and a Carriage Return is printed at the end of the string. The number to precede [10X\], which prints 10 characters and a Line Feed, varies according to the length of the string and is computed in the FORM definition (line 40). Notice from the ↴ and ↵ designations below that trailing blanks (those followed by only a Line Feed or a Carriage Return) are not printed. This is a feature of all SUPER BASIC terminal output.

```
>LOAD DEMO↵
>LIST↵
10 STRING S
20 PRINT "TYPE A STRING OF ANY LENGTH:"
30 INPUT S
40 A = STR(INT((LENGTH(S)-1)/10))+"[10X\]10X/"
50 PRINT IN FORM A:S
60 PRINT "ANOTHER:"
70 GO TO 30
  >RUN↵
TYPE A STRING OF ANY LENGTH:
? STRING↵
STRING
ANOTHER:
? THIS IS A STRING↵
THIS IS A
STRING
ANOTHER:
? THIS IS THE THIRD STRING TYPED!!↵
THIS IS TH
E THIRD ST
RING TYPED
!!
ANOTHER:
? ABCDEFGHIJKLMNOPQRSTUVWXYZ↵
ABCDEFGHIJ
KLMNOPQRST
UVWXYZ
ANOTHER:
?⊕
INTERRUPTED IN 30
>
```

SECTION 7
FILES

Files are a convenient method of supplying a program with large amounts of data, saving the results of the execution of a program, or giving commands directly to SUPER BASIC. As many as four files can be used concurrently for input to or output from a program. The data stored on these files can be in either symbolic or binary form, explained below. In addition, both sequential and random access files are available. This section describes the commands for using both sequential and random access files.

SUPER BASIC may be directed to take commands from a command file rather than from the terminal. Command files are discussed in this section on page 89.

No matter what kind of file is being used, it must be opened before use with an OPEN statement and closed after use with a CLOSE statement.

SEQUENTIAL DATA FILE INPUT AND OUTPUT

A sequential file is a file in which reading and writing must take place in the same sequence that data is stored on that file. In other words, once the file is opened, the first data item in the file must be read or written, then the second, and so on. Sequential file input/output will thus prove much slower than random file input/output in many cases. However, sequential files have the advantage of requiring less program overhead than random files.

OPENING A FILE

Before a data file can be read or written, it must be opened (and at the same time given a number) with the command:

OPEN file name FOR { **SYMBOLIC** / *or* / **BINARY** } { **INPUT** / *or* / **OUTPUT** } **AS FILE n**

or the short form:

OPEN file name, { **SYMBOLIC** / *or* / **BINARY** } { **INPUT** / *or* / **OUTPUT** } **, n**

The file name must be a string of characters[1]. If a literal file name is specified in the OPEN statement, it must be enclosed in single or double quote marks. If the file name is a string variable or expression, it must not be enclosed in quote marks. See page 39 for a discussion of string expressions in the OPEN statement.

The file number n, which can be zero or any positive numeric expression, is necessary in every OPEN statement to specify which file the user is working with, since he may have up to four files open at one time. A file number that is not an integer will be truncated. The OPEN statement must also specify whether the file is to be used as an input file or an output file.

Input or output files may be symbolic or binary. Since data written on a binary file is not in the usual character representation, but in internal machine code, the file cannot be printed on the terminal (and be meaningful). Binary form, however, is especially useful if a program creates a large number of results that are to be used as input to another program. Using binary data files can significantly reduce both the disk storage required for data and input/output computing

1 - See *Appendix C* for file naming rules.

time. Matrix input and output from binary data files using MAT commands is especially fast and should be used in preference to other methods where practical[1]. *NOTE: If the MAT PRINT or MAT WRITE command is used to write on a binary file, the MAT INPUT command must be used to read the same data.*

Single variables and arrays must be declared before reading their values from or writing their values on a binary file, using any of the declaration statements INTEGER, REAL, DOUBLE, COMPLEX, LOGICAL, STRING, or TEXT. When reading values from a binary file, the variables and arrays should be declared to be of the same type as they were when written. Since the amount of storage used for a value stored on a binary file depends on its value type, SUPER BASIC cannot know how many words to read unless this is specified in a declaration statement[2].

When writing on a file, variables should be used instead of expressions. For example,

```
70 REAL X
80 X=I+L*B↑2-SQRT(B)
90 PRINT ON 3:X
```

is better than

```
70 PRINT ON 3:I+L*B↑2-SQRT(B)
```

NOTE: Formatted input and output (discussed under **FORMATTING WITH IMAGE**, *page 51, and* **FORMATTING WITH FORM**, *page 56) can be used with symbolic files only; it can never be used with binary files.*

When a file is opened for input, it need not be specified in the OPEN statement as symbolic or binary. If the word SYMBOLIC or BINARY is omitted, SUPER BASIC will check to see what type of file it is and will read it as such. If the file type is specified but does not match the file, an error message will be printed.

When a file is opened for output, the user must specify if the file is to be binary; otherwise, a symbolic output file will be written. Thus,

```
OPEN "BDATA",BINARY OUTPUT,M*N
```

will open for binary output the file BDATA, the file number of which equals the value of M*N. The following

```
OPEN "SDATA",OUTPUT,4
```

will open for symbolic output the file SDATA.

A file need not already exist to be opened for output; the OPEN command will automatically create a file of the specified name and type in the user's directory.

NOTE: Opening a file initializes input or output at the beginning of the file. Thus, with sequential files, data must be read or written beginning with the first location in the file. To read or write data beginning at other than the first data item in a file, use the random file commands discussed in **RANDOM ACCESS DATA FILES**, *page 77.*

INPUT FROM A FILE

The command used to read data from a file takes the form:

INPUT FROM n:variable list

where n is the input file number. For example,

```
10 OPEN 'AFILE',INPUT,2
20 INPUT FROM 2:X,Y,Z
```

reads three values from AFILE and assigns them to the variables X, Y, and Z, respectively.

The entries in a data file may be separated by commas or spaces, with a Carriage Return at the end of each line of data. The entries can be numbers but **not** expressions.

OUTPUT TO A FILE

To write on a file, use either of the equivalent forms:

WRITE ON n: *or* **PRINT ON n:**

followed by a list of numbers, variables, or expressions whose values are to be written on the file, where n is the output file number. For example,

```
80 OPEN "DATA1",OUTPUT,3
85 OPEN "DATA2",BINARY OUTPUT,N
90 WRITE ON 3:P,Q,R,W
95 WRITE ON N:A,B,C
```

Line 90 writes the values of the variables P, Q, R, and W on the symbolic file DATA1 (file 3). Line 95 writes the values of the variables A, B, and C on the binary file DATA2 (file number equal to the value of N).

CLOSING A FILE

After the last input or output operation is performed on a data file, the CLOSE command should be used to close the file. *NOTE: An input or output file is closed automatically after a RUN, a DELETE ALL, or a return to the EXECUTIVE.*

1 - MAT commands are discussed under *MATRIX ARITHMETIC*, page 20.

2 - For the amount of storage reserved by each declaration statement, see *Section 5, USING DECLARATION STATEMENTS.*

Files to be closed are specified by their file numbers in the CLOSE command. For example,

```
120 CLOSE 4,B-2          Closes files 4 and B-2.
200 CLOSE 3              Closes file 3.
```

Once a sequential file has been read or written, it can be read or rewritten only by closing the file and opening it again. This restriction is avoided with random files, discussed on page 77.

If four files are open concurrently, any of them may be closed with a CLOSE command so that other files can be opened. Once a file has been closed the number of that file may be used later to designate another file.

NOTE: Files created during execution of a program remain after the program has terminated.

Example: Data File Commands

Twelve numbers are read from a file named XDATA. The positive numbers are written on POSX, the negative are written on NEGX, and zeroes are ignored.

```
– COPY XDATA TO T↵
1,16,-4,6,-11,-2,30,-4,6,8,0,-7

– SBASIC↵

>10 OPEN "XDATA",INPUT,4↵
>20 OPEN "POSX",OUTPUT,2↵
>30 OPEN "NEGX",OUTPUT,3↵
>40 FOR I = 1 TO 12↵
>50 INPUT FROM 4:X↵
>60 IF X>0 THEN WRITE ON 2:X;↵
    ELSE IF X<0 THEN WRITE ON 3:X;↵
>70 NEXT I↵
>80 CLOSE 4,2,3↵
>RUN↵

>QUIT↵

– COPY POSX TO T↵

  1     16     6     30     6     8
– COPY NEGX TO T↵

 -4    -11    -2    -4    -7
```

DELETING A FILE

Files may be deleted within a SUPER BASIC program by using a form of the CLOSE command as follows:

CLOSE "filename"

where filename is the name of the file to be deleted.

Both sequential and random files may be deleted in this manner.

Before a file can be deleted, it must be closed with the

CLOSE number

command, where number is the number given to the file when it was opened.

Deletion of the file does not need to follow immediately after the **CLOSE number** command. In fact, a file may be deleted in a program without ever having been opened in that program.

Example

The file INMEAN is opened for symbolic input as a sequential file. Calculations are performed using the data, the file is closed, then deleted.

```
– COPY INMEAN TO T↵

6.7,5.9,8.76
50.8,24.7,33
422.1,516.49,731.07
29.07,31.41,42.10

– SBA↵

>LOAD AVG↵
>LIST↵
10 OPEN "INMEAN",INPUT,3
20 FOR I=1 TO 4
30 INPUT FROM 3:A,B,C
40 PRINT (A+B+C)/3
50 NEXT I
60 CLOSE 3
70 CLOSE "INMEAN"
  >RUN↵
 7.12
 36.166667
 556.55333
 34.193333

>RUN↵

ERROR IN STEP 10:

FILE NAME NOT IN FILE DIRECTORY

>QUIT↵
```

Note that the program cannot be executed a second time because the file INMEAN has been deleted from the user's directory.

Lines 60 and 70 above could have been combined as

```
60 CLOSE 3,"INMEAN"
```

TESTING FOR END OF FILE

At any time after a file is opened, a command can be given that will cause SUPER BASIC to transfer to a specified statement when the end of the input file is encountered. The form of the command is

ON ENDFILE(*n*) GO TO line number

where *n* is the input file number. The ON ENDFILE statement must be executed after the input file is opened and before the INPUT statement on which the test is to be made. For example, the program shown above that reads twelve numbers from XDATA can be generalized so that the exact number of data values on the input file need not be known.

```
10 OPEN "XDATA",INPUT,4
20 OPEN "POSX",OUTPUT,2
30 OPEN "NEGX",OUTPUT,3
35 ON ENDFILE(4) GO TO 80
40 FOR I = 1 TO 5000
50 INPUT FROM 4:X
60 IF X>0 THEN WRITE ON 2:X;
   ELSE IF X<0 THEN WRITE ON 3:X;
70 NEXT I
80 CLOSE 4,2,3
```

When line 50 encounters the end of the input file XDATA, a transfer to line 80 will be made, as specified in the ON ENDFILE statement. This program is designed so that:

- If XDATA contains 5000 or fewer data values, all values will be read.
- If XDATA contains more than 5000 data values, only the first 5000 values will be read.

Note that the value of I upon exit from the FOR loop is one greater than the number of data values read from XDATA. Consider, as another example, the following data file and program:

```
-COPY D TO T↵
1 2 3 4
-SBASIC↵
>10 OPEN "D",INPUT,6↵
>20 ON ENDFILE(6) GO TO 60↵
>30 FOR I = 1 TO 100↵
>40 INPUT FROM 6:Y↵
>50 NEXT I↵
>60 PRINT "I=":I↵
>70 PRINT "BUT":I-1:" Y VALUES WERE
  READ"↵
>80 CLOSE 6↵
>RUN↵
I= 5
BUT 4 Y VALUES WERE READ

>
```

The ON ENDFILE statement is no longer effective once the file being checked is closed. Thus, if another file is opened for input as file 6 after the above statements are executed, encountering the end of that file will cause an error message unless an ON ENDFILE statement is executed for the new file 6.

If more than one ON ENDFILE statement is executed for the same data file, the last transfer specified will be made when the end of that file is encountered.

NOTE: The ON ENDFILE statement can be used with random as well as sequential file input.

THE TERMINAL AS A FILE

The terminal behaves exactly like a sequential file. The rules for file input and output are applicable to the INPUT, PRINT, and WRITE statements used for terminal input and output.

In addition, the OPEN, CLOSE, INPUT, and WRITE statements may be used to refer to the terminal as a numbered file. For example,

```
OPEN "TEL", INPUT,3
INPUT FROM 3 IN IMAGE "#":A,B,C
CLOSE 3
```

If TEL is used in the OPEN statement, the terminal is treated as an open file and is counted as one of the four files which may be open concurrently. If the terminal is used for input and output without being opened in an OPEN statement, it is not considered one of the four files which may be open concurrently.

The input question mark is not suppressed upon execution of the command

INPUT FROM n:variable list

where TEL has been opened for input as file n. To suppress the terminal input question mark and still retain the advantages of free form input, use the single # field in either IMAGE or FORM.

RANDOM ACCESS DATA FILES

SUPER BASIC allows random access disk files of either symbolic or binary type in addition to the usual sequential files. The user may read information beginning at any location in a file, write information on any part of a file without destroying the rest of the file, and erase selected parts of a file. He may specify variable record length for a random file, or a fixed record length of any desired length. These direct access features allow implementation of many applications that are either impossible or take too long to execute with sequential files. In particular, applications requiring information retrieval and updating of a small part of the total data file should use significantly less processing time when implemented with random files.

BASIC CONCEPTS AND DEFINITIONS

Elements

The elements of a random file are simply the "units" stored in the file:

- If a file is symbolic, an element is a character in the file.
- If a file is binary, an element is a word (24 bits).

Records

Random files are considered to consist of records. A record is a group of related elements that can be retrieved as a single unit. For example, each record in a payroll data file might consist of an employee name, his hourly pay rate, and weekly deductions. SUPER BASIC random files may either have records of a fixed length specified by the user, or have records of arbitrary, variable length.

Record Length

The record length of a random file is the number of elements (characters or words) in a record. Any record length may be specified by the user when he opens a random file.[1] If he does not specify a record length in the OPEN command, his file is a **variable record length** file; that is, it may contain records of different lengths. If a record length is specified in the OPEN command, the file is a **fixed record length file** and may contain only records of the specified length.

Location

Each element in a variable record length file, and each record in a fixed record length file, is assigned a positive integer called the **location** of the element or record. If a random file has a fixed record length, the first record is at location 1, the second at location 2, and so on. In other words, a location is simply a record number. If a random file has variable record length, a location is an element number.

Example

Suppose we have the following records stored on a symbolic variable record length file, and we are considering a record to be any sequence of characters terminated by a Carriage Return. Then

This record	Begins at Location
THIS IS A STRING ↵	1
456789 ↵	18
1 ↵	25
END ↵	27

Note that Carriage Returns and spaces are counted as elements.

There are several ways to specify the location for the next input or output operation to be performed on a random file. These will be discussed later in this section.

Multiple Blanks In Sequential And Random Files

SUPER BASIC writes sequential files with multiple blanks compressed, using a special multiple blank character to abbreviate any sequence of multiple blanks. Random files, on the other hand, are written with multiple blanks not compressed. *Do not attempt to use a sequential file as a random file; doing so will cause locations to be defined unpredictably due to the multiple blank compression.* If you have a symbolic sequential file that you wish to use as a random file, you may rewrite it in EDITOR with multiple blanks not compressed by using the WRITE↵ command[2]. The file may then be used as a random file in SUPER BASIC.

1 - See *Opening A Random File*, page 78.

2 - See the *Tymshare EDITOR Reference Manual* for details.

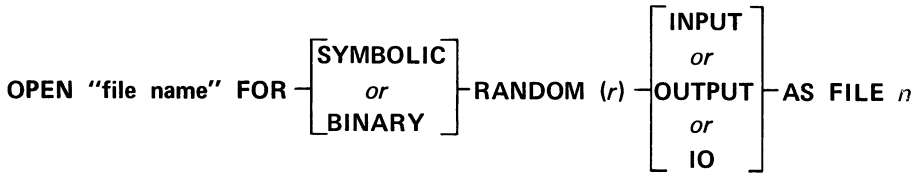
OPENING A RANDOM FILE

Like sequential files, random files must be opened (and assigned a number) before they can be read or written. As many as four files may be opened simultaneously. Random files must be opened in one of three different modes:

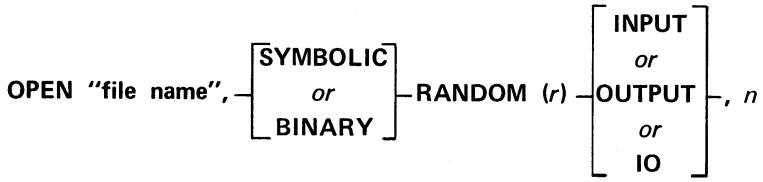
1. For input only, indicated by the word INPUT.
2. For output only, indicated by the word OUTPUT.
3. For both input and output, indicated by IO.

Opening a previously created random file for output does not erase the contents of the file as does opening a previously created sequential file for output; neither does opening a random file for input and output.

To open a random file, use the following form of the OPEN command, which may be direct or indirect:



Or use the equivalent short form:



NOTE: The file name must be enclosed in double or single quote marks unless it is a string variable or expression. See page 39 for a discussion of string expressions in the OPEN statement.

Here, *n* is the file number and *r* is the record length, which **must** be enclosed in parentheses. Both *n* and *r* may be constants, variables, or numeric expressions; *n* must be zero or positive and *r* must be positive. If either value is not an integer, it will be truncated.

If the record length *r* is omitted, the file is a variable record length file.

Example 1

```
>OPEN "RDATA" FOR SYMBOLIC RANDOM
 IO AS FILE M*N
```

opens the symbolic file named RDATA for random IO as a variable record length file, and assigns it file number M*N.

Example 2

The command

```
10 OPEN 'BR',BINARY RANDOM (50) INPUT,3
```

opens the binary file named BR for random input as file 3, and specifies a fixed record length of 50 words.

If the word RANDOM is omitted from the OPEN command, the file is opened as a sequential file (no record length may be specified). *NOTE: A sequential file may not be opened for IO.*

When a file is opened for random input, it need not be specified as symbolic or binary in the OPEN statement. If the word SYMBOLIC or BINARY is omitted, SUPER BASIC checks to see what type the file is and will read it as such.

When a file is opened for random output or for IO, it will be opened as a symbolic file unless the user specifies that the file is to be binary; attempting to open a binary file for output or IO without specifying the file type yields an error message. Thus, assuming there is no binary file in the user's directory called SFILE,

```
30 OPEN "SFILE",RANDOM (I+1) OUTPUT,5
```

opens the file named SFILE for symbolic random output as file number 5. A fixed record length of I+1 characters is specified.

If an old file is opened for random output or IO, SUPER BASIC will check the file type against the type specified in the OPEN statement. If they do not agree, an error message will be given.

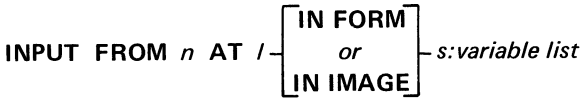
RANDOM FILE INPUT AND OUTPUT

Special forms of the INPUT FROM and PRINT ON or WRITE ON commands are available for random file input and output. The commands to use are the same for both variable and fixed record length files; however, the rules for formatting input and output differ for each type of file. In this section we first present general information about the input and output commands, and then discuss separately the different rules for symbolic variable record length and fixed record length input and output formats.

NOTE: Binary file input and output can never be formatted, neither for variable or fixed record length random files, nor for sequential files.

Input From A Random File

Data may be read from a random file that has been opened for INPUT or IO with the following form of the INPUT FROM command, which may be direct or indirect:



Here, *n* is the file number specified in the OPEN command, */* is the location at which input is to begin, and *s* is a string constant, variable, or expression specifying the input format. Both *n* and */* may be numeric constants, variables, or expressions. The location */* must be positive; if its value is not an integer, it will be truncated.

Example

Suppose the file FILE1 contains the strings

```
STRING 1
STRING 2
STR,ING 3
```

then

```
10 OPEN "FILE1",RANDOM INPUT,4
20 INPUT FROM 4 AT 10 IN FORM 'R':S,T
```

reads the strings STRING 2 and STR,ING 3 from FILE1 and assigns them to the variables S and T, respectively.

The location */* need not always be specified in the INPUT FROM command. SUPER BASIC keeps track of the location at which input or output is taking place and defines the next location to be affected by an input or output operation, called the **current location**, as follows:

The current location is always the location following the location most recently read or written, unless

- It is otherwise specified in an input or output command or with the LOCATE command (discussed on page 84);.
- The file has been opened but nothing has yet been read or written. In this case, the current location is location 1.

If the location has been omitted from an INPUT FROM command, input begins at the current location. Thus, the statements

```
>OPEN 'THIS',RANDOM IO,3↵
>INPUT FROM 3 IN FORM 'D':N↵
```

read the first digit in the file named THIS, since the current location is location 1 just after a file has been opened. If we then execute the statement

```
>INPUT FROM 3 IN FORM 'D':M↵
```

M is assigned a value equal to the second digit in the file, since the current location is location 2 when this statement is encountered.

Matrix data may be read from a random file using MAT INPUT FROM in the same form as INPUT FROM given above.

Example

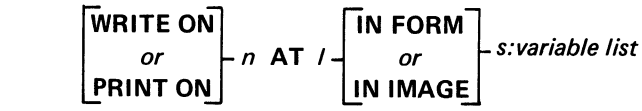
```
10 TEXT A(100):15
20 OPEN "MATDATA",RANDOM (15) IO,3
30 MAT INPUT FROM 3 AT 5 IN FORM
   "15D/":A
```

This program dimensions the one dimensional string array A and opens the file MATDATA for RANDOM IO, specifying a fixed record length of 15 elements. Then it reads the 100 array elements

of A beginning at location 5 in the file. Since the format 15D/ is used, and the specified record length is 15, one record is assigned to each array element: Record 5 is assigned to A(1), record 6 to A(2), etc. The / is needed to indicate the end of each record, see *Fixed Record Length File Input and Output Formats,* page 81.

Output To A Random File

To write data on a random file opened for OUTPUT or IO, use the following form of the WRITE ON or PRINT ON command, which may be direct or indirect:



In the above form, *n* is the file number, *l* is the location at which output is to begin, and *s* is a string constant, variable, or expression specifying the output format. As with INPUT FROM, *n* and *l* may be numeric constants, variables, or expressions. This command may also be preceded by the word MAT to write matrix data on a random file.

Example

```
-COPY PAY TO TEL↵
```

Location of beginning of line[1].

```
JONES, 2.50                    1
BROWNE, 3.00                   13
SWAN, 1.50                     26
JOHANSEN, 4.75                 37
THOMAS, 3.50                   52

-SBASIC↵

>OPEN 'PAY',RANDOM IO,3↵

>PRINT ON 3 AT 32 IN FORM 'D.DD/':2↵

>PRINT ON 3 AT 65 IN FORM↵
 '8X BD.DD/':'STEVENS,',2.25↵

>QUIT↵

-COPY PAY TO TEL↵

JONES, 2.50
BROWNE, 3.00
SWAN, 2.00                     This line was updated.
JOHANSEN, 4.75
THOMAS, 3.50
STEVENS, 2.25                  This line was added to
                               the file.
```

Notice that writing data at a location replaces whatever was previously at that location, element for element. Thus, 2.00 replaced 1.50 beginning at location 32. This fact should be kept in mind especially when writing on variable record length files, since writing a record longer than an existing record will replace elements in the following record. For example, if we now execute

```
>PRINT ON 3 AT 32 IN FORM 'DDD.DD/':2↵
```

the resulting file will look like

```
JONES, 2.50
BROWNE, 3.00
SWAN, 002.00
HANSEN, 4.75        . . . . The first two characters in
THOMAS, 3.50               this line (JO) were lost.
STEVENS, 2.25              They were replaced by the
                           last two characters added to
                           the preceding line (0 and a
                           Carriage Return).
```

The location *l* need not be specified in the PRINT ON or WRITE ON command. If *l* is omitted, output begins at the current location. Thus, using the file PAY from the previous example the following may occur:

```
>LIST↵
10 OPEN 'PAY',RANDOM IO,3
20 INPUT FROM 3 AT 26:S
30 IF S='SWAN' THEN PRINT ON 3 IN
   FORM 'BD.DD':2.5
>RUN↵

>QUIT↵

-COPY PAY TO TEL↵

JONES, 2.50
BROWNE, 3.00
SWAN, 2.50
JOHANSEN, 4.75
THOMAS, 3.50
STEVENS, 2.25
-
```

In line 20 the string SWAN was read from the file, using unformatted input so that the terminating

1 - The Carriage Return at the end of the line is counted as one character in determining this location.

comma was read but not assigned to the variable S. Thus, when the PRINT ON command in line 30 is encountered, the current location is 31 (the location following the comma). A blank and the number 2.50 are printed after the comma.

Variable Record Length File Input And Output Formats

Both unformatted and formatted input and output are allowed with symbolic variable record length random files; they work exactly as in the sequential case. Thus, the following rules apply:

1. On input, the usual terminators (comma, space, Carriage Return) may serve to separate the entries on the data file.
2. When a data item is read using unformatted input, all characters up to and including the next terminating character are read, even though the terminator is not included in the value assigned to the variable being read. Thus, the current location after an unformatted input statement will be the location following the last terminating character encountered. For example, if the data

   13.5,19.21

   is stored beginning at location 38 on file 4, the statement

   >INPUT FROM 4 AT 38:A↵

   assigns A the numeric value 13.5 and sets the current location to 43 (the location following the comma).
3. On output, the comma, semicolon, colon, and Carriage Return in the output statement have their usual meanings.

   Examples

   >PRINT ON 4:A↵ *Prints the value of A followed by a Carriage Return on file 4.*

   >PRINT ON 3:13.2,25↵ *Prints 13.2 followed by 11 spaces followed by 25 and a Carriage Return on file 3.*

   >PRINT ON 2 AT 38:A,↵ *Prints the value of A followed by as many spaces as are needed to fill a 15 space print zone, at location 38 on file 2.*
4. All IMAGE and FORM formats allowed in symbolic sequential file input and output may be used with symbolic variable record length random files as well. These include free form and fixed length formats in IMAGE and FORM, and the single R field in FORM. All formats work just as they do for sequential files.

Example

```
-COPY VRLDATA TO TEL↵

JONES, 55 S. STREET
DAVIS, 3490 OCEAN DRIVE
THOMPSON, 90125 WASHINGTON BLVD.
ADAMS, 49 WEST WAY

-SBASIC↵

>5 STRING S1,S2↵
>10 OPEN "VRLDATA",RANDOM IO,5↵
>20 INPUT FROM 5 IN FORM 'R':S1,S2↵
>30 PRINT S1↵
>40 PRINT S2↵
>!FIRST WE USE THE SINGLE R IN ↴
 FORM (LINE 20)↵
>RUN↵
JONES, 55 S. STREET
DAVIS, 3490 OCEAN DRIVE

>!NOW, CHANGE TO A FIXED LENGTH ↴
 IMAGE↵
>20 I="%%%%%"↵
>25 INPUT FROM 5 AT 1 IN IMAGE I:S1,S2↵
>GO TO 20↵
JONES
DAVIS

>!NOW, A FIXED LENGTH FORM↵
>20 F="5X/"↵
>25 INPUT FROM 5 IN FORM F:S1,S2↵
>GO TO 20↵
THOMP
ADAMS

>
```

Fixed Record Length File Input And Output Formats

The structure of fixed record length files is quite different from that of variable record length files.

Fixed record length files are designed to store and retrieve records of a fixed, specified length with no special terminating characters either within or at the end of each record. *Thus, all characters on a symbolic fixed record length file are treated identically by SUPER BASIC; commas, spaces, and Carriage Returns are handled just like any other characters and should be included on a symbolic fixed record length file only if they are part of the data to be processed.* Most fixed record length files will consist of alphanumeric data to be looked up and updated from time to time as in an inventory control system. They probably will not be listable in a readable form from the EXECUTIVE, since if Carriage Returns are absent, characters will overprint at the end of a physical line on the terminal.

To make efficient handling of symbolic fixed length records easy, SUPER BASIC provides special rules for formatting input and output on fixed record length files. These include:

1. Rules to insure that reading and writing data units occur only within record boundaries, discussed in *Record Protection Features* below.
2. Special rules for IMAGE and FORM formats to be used to access all or part of any desired record, including a format which can read or write exactly one record. These are discussed in *Special Formatting Rules,* below.

Record Protection Features

Since there are no terminators on fixed record length files to delimit input, and since data written on a record cannot be longer than the record length, fixed length formats must be used to read or write data on a fixed record length symbolic file. Thus,

- no unformatted input or output, and
- no free form (the single #) input or output

are allowed on fixed record length symbolic files. Attempts to use unformatted or free form input or output result in an error message.

If a fixed length format specifies a field length longer than the space available in the current record, SUPER BASIC will print an error message. Input or output will not be allowed to continue into the next record.

Example 1

The statements

```
10 OPEN "DATA6",RANDOM (10) IO,6
20 INPUT FROM 6 IN FORM '15D':X
```

yield an error message when executed, since the FORM '15D' specifies that more characters be read (15) than there are in a record on the file (10).

Example 2

```
10 OPEN "DATA6",RANDOM (10) IO,4
20 INPUT FROM 4 IN FORM '5D':X
30 INPUT FROM 4 IN FORM '6D':Y
```

These statements also yield an error message when line 30 is executed. Since five digits have already been read from the first record in the file, SUPER BASIC is positioned at the sixth character in record 1 when line 30 is encountered; thus, there are only five characters left in the record to be read. The form '6D' tells SUPER BASIC to read six digits, so an error message is printed; SUPER BASIC will not cross the record boundary to read the sixth digit.

On output, when the format specifies a length longer than the space left in the record, characters will be written up to the end of the current record even though an error message is printed. Thus, the statements

```
10 OPEN "DATA6",RANDOM (10) IO,3
20 PRINT ON 3 AT 3 IN FORM
   '15D':'012345678901234'
```

yield an error message when executed, but will print 0123456789 on record 3. Output is stopped at the end of the current record, so that nothing is printed on record 4.

Special Formatting Rules

Of the formats that may be used for fixed record length file input and output, the FORM characters R and / have special functions; IMAGE formats also differ when used with fixed record length files. All other FORM characters have the same functions as they do with sequential and variable record length files and will not be discussed here.

The Single R Field

The single R in FORM has a special meaning with symbolic fixed record length files: It tells SUPER BASIC to read or write a string up to the end of the current record. If the user is positioned at the beginning of a record, the R field specifies that an entire record be read or written. If, on the other hand, part of the current record has already been read, the R field specifies that the rest of the record be read or written.

On input, the single R field in FORM simply reads the rest of the current record as a string. Carriage Returns are treated as any other character. Thus, if the file FRLTEST contains the data

```
STRING 1STRING↵
2,123456↵
```

the R field reads records of length 8 as follows:

```
>5 STRING S(3)↵
>10 OPEN "FRLTEST",RANDOM(8) INPUT,3↵
>20 INPUT FROM 3 IN FORM 'R':S(I)↵
  FOR I=1 TO 3↵
>30 PRINT "THIS IS RECORD":I:":";S(I)↵
  FOR I=1 TO 3↵
>RUN↵
THIS IS RECORD 1:     STRING 1
THIS IS RECORD 2:     STRING ←┐
   A Carriage Return is the last character in record 2.
2
THIS IS RECORD 3:     ,123456 ←┐
   A Carriage Return is the last character in record 3.

>
```

On output, the R field writes the specified string as the rest of the current record. If the string to be written is longer than the space available in the current record, characters are omitted from the right of the string. If the string to be written does not fill the current record, trailing blanks are printed to fill the record.

Example

The statements

```
10 OPEN "RTEST",RANDOM(10) IO,4
20 S='01234567899999',T='STRING'
30 PRINT ON 4 AT 13 IN FORM 'R':S,T
```

print 0123456789 as record number 13, and STRING followed by four blanks as record number 14.

The / In A Form Format

On both input and output, the / in a FORM corresponds to the end of the record with fixed record length files.

To illustrate the rules for using the /, we assume that the symbolic file MAT has been opened for random IO with the statement

```
OPEN "MAT" FOR RANDOM(30) IO AS FILE 2
```

We suppose that this file contains a 100 by 3 matrix A consisting of 300 strings of length 10 each with three strings per record (record length equals 30).

On input from a fixed record length file, the / in a FORM format causes SUPER BASIC to seek the end of the record. Thus, for the file assumed above, the FORM '10B10X/' may be used to read the middle string of any record; the matrix element A(67,3) could be read with the command

```
INPUT FROM 2 AT 67 IN FORM
'10B10X/':A(67,3)
```

assuming the records are stored beginning at location 1 in the file. The 10B causes the first ten characters in the record to be skipped, the 10X causes the next ten characters to be read and assigned to A(67,3) and the / causes SUPER BASIC to seek the end of the record, thus skipping the last ten characters. The current location is set to 68 after execution of the statement.

On output to a fixed record length file, the / causes an entire record to be read or written and the current location to be set to the next record number. If the record is not filled when the / is encountered, trailing blanks will be printed to fill the record. For example, the statements

```
S='P648910500',T='P769185439'
PRINT ON 2 AT 5 IN FORM '10X 10X/':S,T
```

print S and T followed by ten blanks as record number 5. If the data to be printed is longer than the record length, it will be cut off on the right. Thus,

```
S='ABCDEFGHIJKLMNOPQRSTUVWXYZ123456
PRINT ON 2 IN FORM '30X/':S
```

prints ABCDEFGHIJKLMNOPQRSTUVWXYZ1234 at the current location on file 2.

The / must be used in FORM to read or write across a record boundary in a fixed record length file; failure to use it results in an error message. Thus, the statement

```
40 MAT PRINT ON 2 IN FORM '3(10X)/':A
```

is legal, but using '3(10X)' is not, since, in writing the entire matrix A, this statement then attempts to write more than one record without indicating the end of the record. Similarly,

```
30 FOR I=1 TO 100
.
.
.
55 INPUT FROM 2 IN FORM '3(10X)/':A(I,J)
 FOR J=1 TO 3
```

```
.
.
.
80 NEXT I
```

is legal, but using the FORM '3(10X)' in statement 55 would result in an error message.

This rule does not apply if the current location is specified for each input or output statement, since specifying the location tells SUPER BASIC to begin a new record. (The location may be specified either in each input or output statement or with the LOCATE command discussed below.) Thus, the loop

```
30 FOR I=1 TO 100
.
.
.
55 INPUT FROM 2 AT I IN FORM
 '3(10X)':A(I,J) FOR J=1 TO 3
.
.
.
80 NEXT I
```

is legal.

Image Formats

When used for fixed record length file input and output, an IMAGE format acts as a record image rather than a line image. Thus, SUPER BASIC will automatically seek the end of the record being read when an input IMAGE is exhausted, just as it automatically seeks a Carriage Return with input from sequential and variable record length files. Any characters remaining in the record are skipped.

Example

Suppose the file ITEST contains the data

```
RECORD ONERECORD TWO
```

then,

```
>10 OPEN 'ITEST',RANDOM (10) IO,2↵
>20 STRING X,Y↵
>30 INPUT FROM 2 IN IMAGE↓
  '%%%%%%':X,Y     !OR FORM '6%/'↵
>40 PRINT X,Y↵
>RUN↵
RECORD          RECORD

>
```

The last four characters in each record are skipped.

On output, SUPER BASIC always assumes that the end of the IMAGE specifies the end of the record and will set the current location to the next record number when the IMAGE is exhausted. If the number of characters specified by the IMAGE format does not fill the record, trailing spaces are printed. Thus,

```
>PRINT ON 2 AT 3 IN IMAGE↓
  '%%%%%%':'RECORD'↵
```

where file 2 is the same as in the previous example, prints RECORD followed by four blanks as record number 3.

If the IMAGE specifies a length longer than the number of positions remaining in the current record, an error message is printed; the data written is cut off at the end of the record.

SETTING THE CURRENT LOCATION: THE LOCATE COMMAND

Instead of specifying the current location in the input or output command, the user may specify the current location by using the LOCATE command. This command, which may be direct or indirect, has the following form:

LOCATE / ON *n*

It tells SUPER BASIC to perform the next input or output done on file number *n* at location number /. The location number / is, as usual, an element number for a variable record length file and a record number for a fixed record length file; its value must be a positive integer. Both *n* and / may be numeric constants, variables, or expressions; non-integer values will be truncated.

Examples

```
>LOCATE 6 ON 3↵
```

sets the current location to location 6 on file number 3.

```
45 LOCATE (J+K)/2 ON M*N
```

sets the current location to the (truncated) value of (J+K)/2 on file M*N.

NOTE: The LOCATE command always sets the position function POS(n) to 1.

THE POS(n) FUNCTION WITH RANDOM FILES

In the case of random files, POS(*n*) specifies the position within the current record at which the next input or output operation on file *n* will be performed, instead of the number of print positions from the beginning of the line.

Example

For a fixed record length file with record length 3 opened as file 2:

```
10 LOCATE 12 ON 2      Sets the current location
                       to record 12 and POS(2)
                       to 1.
20 INPUT FROM 2 IN FORM 'X':S
                       Reads one element from
                       record 12 and sets POS(2)
                       =2, assuming this statement
                       is executed just after state-
                       ment 10 above.
```

Because of the flexible record structure allowed with variable record length files, SUPER BASIC must make certain assumptions in setting the position function with these files. These are as follows:

- Whenever a Carriage Return is encountered in reading or writing on file *n*, POS(*n*) is reset to 1.

- The command

  LOCATE / ON *n*

  assumes location / contains the first character of some record and sets POS(*n*) to 1.

Example

```
– COPY VRLDATA TO TEL↵

JONES, 55 S. STREET
DAVIS, 3490 OCEAN DRIVE
THOMPSON, 90125 WASHINGTON BLVD.
ADAMS, 49 WEST WAY

– SBASIC ↵

> OPEN "VRLDATA",RANDOM IO,3↵

> LOCATE 21 ON 3↵

> PRINT 'POSITION IS':POS(3)↵
POSITION IS 1

> INPUT FROM 3 IN FORM '5X':S↵

> PRINT 'AFTER READING ':S:' POS ↓
  IS':POS(3)↵
AFTER READING DAVIS POS IS 6

> ! CURRENT LOCATION IS NOW 26↵
> LOCATE 26 ON 3↵

> PRINT 'CURR LOC. STILL 26, NOW POS ↓
  IS':POS(3)↵
CURR LOC. STILL 26, NOW POS IS 1

> INPUT FROM 3 IN FORM 'R':S↵

> PRINT ' CARRIAGE RETURN IS SEEN, SO ↓
  POS IS':POS(3)↵
CARRIAGE RETURN IS SEEN, SO POS IS 1

>
```

THE LOCATION FUNCTION: LOC(n)

The user may determine the current location of an opened random file by executing the LOC function, of the form

LOC (*n*)

The single argument *n* is the file number of the file whose location is desired. The value returned by the LOC function is the current location on file *n*; that is, the next location to be affected by an input or output operation on file *n*.

Example

```
> 10 OPEN "FRL", RANDOM(50) IO,5↵
> 20 LOCATE 7 ON 5↵
> 30 PRINT "CURRENT LOCATION IS":LOC(5)↵
> 40 INPUT FROM 5 IN FORM 'R':X↵
> 50 PRINT "NOW, CURRENT LOCATION ↓
  IS":LOC(5)↵
> RUN↵
CURRENT LOCATION IS 7
NOW, CURRENT LOCATION IS 8

>
```

In this example, the file is a fixed record length file; thus, location numbers are record numbers. After record 7 is read in line 40, the current location increases by 1.

NOTE: When reading from or writing on a fixed record length file, the current location does not change until the entire record is read or written. Thus, if a LOC function is executed while the program is in mid-record, the value returned is the location of the record being processed. For example,

```
>10 OPEN "FRL", RANDOM (10) INPUT,3
>20 INPUT FROM 3 IN FORM '5D':N
>30 PRINT "NOW WE ARE READING
 RECORD 1"
>40 PRINT "SO THE CURRENT LOCATION
 IS":LOC(3)
>50 INPUT FROM 3 IN FORM '5D':M
>60 PRINT
>70 PRINT "WE HAVE FINISHED READING
 RECORD 1"
>80 PRINT "SO THE CURRENT LOCATION
 IS":LOC(3)
>RUN
NOW WE ARE READING RECORD 1
SO THE CURRENT LOCATION IS 1

WE HAVE FINISHED READING RECORD 1
SO THE CURRENT LOCATION IS 2

>
```

THE SIZE FUNCTION: SIZE(n)

The size of an opened random file may be obtained by using the function

SIZE(*n*)

where the argument *n* is the number of the file whose size is to be computed. The function returns the size of file *n*, defined as follows:

- If the file is of variable record length, the size is the location of the last character (or word if the file is binary) written on the file.
- If the file is a fixed record length file, the size is the location, or record number, of the last record written on the file.

Example

Suppose the file RDATA contains the following data:

```
012479
653721
598743
```

Then the SIZE function yields the following:

```
>10 OPEN "RDATA",RANDOM(7) INPUT,2
>20 Y=SIZE(2)
>30 PRINT "FIXED RECORD LENGTH:
 SIZE =":Y
>40 CLOSE 2
>50 OPEN "RDATA", RANDOM INPUT,3
>60 Y=SIZE(3)
>70 PRINT "VARIABLE RECORD LENGTH:
 SIZE =":Y
>RUN
FIXED RECORD LENGTH: SIZE = 3
VARIABLE RECORD LENGTH: SIZE = 21
```

ERASING PART OF A FILE: THE ERASE COMMAND

To erase an area of a random file, use the ERASE command, directly or indirectly. It has the general form

ERASE *n* FROM l_1 TO l_2

where *n* is the file number, l_1 is the first location to be erased, and l_2 is the last location to be erased. The command erases all data from l_1 to l_2, inclusive.

NOTE: Data may be erased from random files opened for OUTPUT or IO, but not from random files opened for INPUT.

Example

```
30 ERASE 3 FROM 1000 TO 1560
```

erases the contents of file 3 from location 1000 to location 1560, inclusive.

The ERASE command writes blanks in all locations erased. Thus, erasing a portion of the file does not change the file size, with the following exception. If locations are erased from the end of the file, the size is changed to the last location not erased. For example, if SIZE(2)=3469 and the command

ERASE 2 FROM 3000 TO 3469

is executed, SIZE(2) will then become 2999.

EXAMPLE: A DICTIONARY FOR A VARIABLE RECORD LENGTH FILE

The user of random files will often want a dictionary for a particular file, containing pointers to the records in the file. In the case of variable record length files, it is almost necessary to have a dictionary containing at least the location of each record in the file, so that the user will know where to begin accessing a record. The first program presented here illustrates creating such a dictionary. Using as input a file containing only the data records (names and addresses in this case), it creates a new file containing a dictionary in the first 200 locations of the file and the data records in the rest of the file. Note that much more space is used for the dictionary than is actually needed, to permit adding to the data records and dictionary without requiring total restructuring of the file. SUPER BASIC prints blanks between the last location in the dictionary and the first data record at location 201. If the dictionary eventually becomes full, the data must be moved to allow more space for the dictionary and the entries in the dictionary must be recomputed.

Note the use of the functions LOC and SIZE, and the command LOCATE, as well as of the commands for random file input and output.

After creating the dictionary, we illustrate using the dictionary to access records.

```
-COPY DIR TO T↵

MR. JOHN B. CAREY,285 COTTLE AVENUE,CAMPBELL
MRS. LESLIE FISHER,1964 HAMPTON DRIVE,DANVILLE
MR. CARL LARSON,985 SOUTH 9TH STREET,SAN JOSE
MR. DALE MOSS,1650 SARATOGA AVENUE,SARATOGA
MR. JOHN REY,106 FORMAN STREET,CAMPBELL
MR. DANIEL TORRES,24 SCHARF AVENUE,LOS GATOS
MISS DONNA WILKES,315 SOUTH 3RD STREET,SAN JOSE
MR. MICHAEL YOUNG,60 WILSON ROAD,CHESTER
MR. HENRY C. ZIMMER,15 JACKSON STREET,PALO ALTO

-SBASIC↵

>LOAD DCT↵
>LIST↵
10 OPEN "DIR",RANDOM IO,2  ! INPUT FILE
20 OPEN "RDIR",RANDOM IO,3  ! OUTPUT FILE WITH DICTIONARY
30 INTEGER A(200)
40 STRING S
50 LOCATE 201 ON 3
60 FOR I=1 UNTIL LOC(2)=SIZE(2)+1  ! LOOP TERMINATES ON END OF FILE
70 INPUT FROM 2 IN FORM 'R':S
80 A(I)=LOC(3)  ! ARRAY A CONTAINS DICTIONARY ENTRIES
90 PRINT ON 3 IN FORM 'R':S
100 NEXT I
110 LOCATE 1 ON 3
120 PRINT ON 3 IN FORM 'DDDD/':A(J) FOR J= 1 TO I-1
130 CLOSE 2,3
 >RUN↵

>QUIT↵
```

```
-COPY RDIR TO T↵

0201
0246
0293
0339
0383
0423
0468
0516
0557

MRS. LESLIE FISHER,1964 HAMPTON DRIVE,DANVILLE
MR. CARL LARSON,985 SOUTH 9TH STREET,SAN JOSE
MR. DALE MOSS,1650 SARATOGA AVENUE,SARATOGA
MR. JOHN REY,106 FORMAN STREET,CAMPBELL
MR. DANIEL TORRES,24 SCHARF AVENUE,LOS GATOS
MISS DONNA WILKES,315 SOUTH 3RD STREET,SAN JOSE
MR. MICHAEL YOUNG,60 WILSON ROAD,CHESTER
MR. HENRY C. ZIMMER,15 JACKSON STREET,PALO ALTO

-
```

This is location 201. The first data item in the file overprints since SUPER BASIC prints 155 blanks between the last entry in the dictionary and the record at location 201

Now that the dictionary has been created, it may be used to determine the location of any record in the file. For example, if RDIR has been opened for INPUT or IO as file 2, the statements

```
30 INPUT FROM 2 AT 5*N-4 IN FORM 'DDDD/':L
40 INPUT FROM 2 AT L IN FORM 'R':S
```

will read the Nth record in the file, for any integer N. This is illustrated in the following:

```
>LIST↵
10 OPEN "RDIR",RANDOM IO,2
20 FOR N=1 UNTIL LOC(2) = SIZE(2)+1
30 INPUT FROM 2 AT 5*N-4 IN FORM 'DDDD/':L
40 INPUT FROM 2 AT L IN FORM 'R':S
50 PRINT 'RECORD':N:' AT':L:' IS: ':S
60 NEXT N
>RUN↵
RECORD 1 AT 201 IS: MR. JOHN B. CAREY,285 COTTLE AVENUE,CAMPBELL
RECORD 2 AT 246 IS: MRS. LESLIE FISHER,1964 HAMPTON DRIVE,DANVILLE
RECORD 3 AT 293 IS: MR. CARL LARSON,985 SOUTH 9TH STREET,SAN JOSE
RECORD 4 AT 339 IS: MR. DALE MOSS,1650 SARATOGA AVENUE,SARATOGA
RECORD 5 AT 383 IS: MR. JOHN REY,106 FORMAN STREET,CAMPBELL
RECORD 6 AT 423 IS: MR. DANIEL TORRES,24 SCHARF AVENUE,LOS GATOS
RECORD 7 AT 468 IS: MISS DONNA WILKES,315 SOUTH 3RD STREET,SAN JOSE
RECORD 8 AT 516 IS: MR. MICHAEL YOUNG,60 WILSON ROAD,CHESTER
RECORD 9 AT 557 IS: MR. HENRY C. ZIMMER,15 JACKSON STREET,PALO ALTO

>
```

COMMAND FILES

It is possible to instruct SUPER BASIC to take its commands from a file rather than from the terminal. Simply create a file containing all the commands (direct or indirect) which are to be executed. Type the commands into the file exactly as they would normally be given from the terminal.

Using Command Files

In SUPER BASIC, specify that a file is a command file by using the OPEN statement with the * symbol in place of a file number. For example,

OPEN "file name" FOR INPUT AS FILE *

or the short form,

OPEN "file name",INPUT,*

This statement can be executed directly or indirectly. When executed directly, it causes commands to be taken from the file immediately. *However, when executed indirectly, this statement does not interrupt program execution. Instead, it specifies that at the next program pause or stop, commands will be taken from the command file.*

The file name must be enclosed in double or single quote marks unless it is a string variable or expression. See page 39 for a discussion of string expressions in the OPEN statement.

As an example of using this statement directly, consider the file C2 which contains the commands

```
LIST
RUN
PRINT "NOW, QUIT AND LOGOUT"
QUIT
LOGOUT
```

and is used as follows:

```
- SBASIC ↵
> LOAD COMPLEX ↵
> OPEN "C2",INPUT, * ↵
                          Command taken from C2:
10 COMPLEX X,Y,Z,P        LIST
20 READ X,Y,Z
30 P=(X+Y)*Z
40 PRINT "PRODUCT IS":P
50 DATA 5,1,4,3,2,-6
PRODUCT IS 42, -46        RUN
NOW, QUIT AND LOGOUT      PRINT "NOW,
                          QUIT AND
                          LOGOUT"
CPU TIME: 1 SECS          QUIT
TERMINAL TIME: 0:2:57     LOGOUT

PLEASE LOG IN:
```

Notice that the command file in the above example includes an EXECUTIVE command (LOGOUT) as well as SUPER BASIC commands. Any commands that can be typed at the terminal can be included in a command file that is opened in SUPER BASIC.

The system will take its commands from the command file until

- The end of the command file is reached, which causes the system to return to taking commands from the terminal, or
- Another OPEN file name, INPUT, * statement is executed to specify a new command file (the old command file is closed automatically). If the file name specified is "TELETYPE" (or "T" or "TEL"), the system will return to taking commands from the terminal. The statement CLOSE *, which closes the old command file, assumes that the terminal is the new command file, and is therefore equivalent to OPEN "TEL", INPUT, *. *NOTE: When the terminal is opened as the command file, it is not considered as one of the four files that can be open simultaneously in SUPER BASIC.*

When a command file is opened indirectly, the next program pause or stop causes commands to be taken from the file. If it is a PAUSE statement that causes this transfer of control, a subsequent GO in the command file will result in a return to the statement following the PAUSE. For example, suppose the file TEST contains the commands

```
PRINT "WE ARE IN TEST"
PRINT "NOW, THE GO COMMAND"
GO
PRINT "BACK IN TEST!"
PRINT "NOW, CLOSE *"
CLOSE *
```

This command file is opened with an indirect statement:

```
- SBASIC ↵
> 10 OPEN "TEST",INPUT,* ↵
> 20 PRINT "MAIN PROGRAM" ↵
> 30 PRINT "NOW, A PAUSE" ↵
```

```
>40 PAUSE ↵
>50 PRINT "BACK IN MAIN PROGRAM!" ↵
>60 PRINT "NOW, THE PROGRAM END" ↵
>RUN ↵
MAIN PROGRAM
NOW, A PAUSE

PAUSE AT 40
WE ARE IN TEST

NOW, THE GO COMMAND


BACK IN MAIN PROGRAM!
NOW, THE PROGRAM END

BACK IN TEST!

NOW, CLOSE *

>
```

First, a PAUSE in the main program causes commands to be taken from TEST. A GO in the command file results in a transfer to line 50 (the statement following PAUSE). Then the program end is encountered and commands are again taken from TEST. The CLOSE * statement returns control to the terminal (as would an OPEN "TEL", INPUT, * statement) and SUPER BASIC prints a >.

To prevent the PAUSE AT 40 message from printing in this example, line 40 can be replaced by

40 STOP (or 40 END)

and the GO in the command file replaced by GO TO 50. The transfer of control will be the same but no messages will print except those that are specified in PRINT statements.

Input data will be read from the command file if the INPUT statement is of the form

INPUT FROM *: variable list

The statement

INPUT variable list

always accepts input from the terminal only. For example, consider the command file

```
LOAD D
LIST
RUN
7
CLOSE *
```

This command file, named COM, is used as follows:

```
- SBASIC ↵
>OPEN "COM",INPUT,* ↵        Command taken
                              from COM:
                            LOAD D
5 PRINT                     LIST
10 PRINT "A=":                This is the pro-
20 INPUT A                    gram stored in
30 PRINT                      D.
40 INPUT FROM *:B
50 PRINT "THIS IS B:":B
                            RUN
A=? 15 ↵                      Here, input is
                              taken from the
                              terminal.
THIS IS B: 7                  The value of B
                              was read from
                              COM.
>                           CLOSE *
```

The file name is enclosed in quotes in the OPEN statement. Input is taken from the command file only when the INPUT FROM * statement is executed.

The TCP Function

The TCP function is used to indicate whether a command file is open. It returns zero if no command file is open and a non-zero integer if a command file is open. The TCP function requires no argument.

Example

```
>LIST ↵
100 PRINT "NO COMMAND FILE OPEN."
110 PRINT "TCP =":TCP
120 OPEN "XYZ",INPUT,*
130 PRINT "COMMAND FILE IS OPEN."
140 PRINT "TCP =":TCP
150 CLOSE *
160 PRINT "COMMAND FILE CLOSED."
170 PRINT "TCP =":TCP
>RUN ↵
NO COMMAND FILE OPEN.
TCP = 0
COMMAND FILE IS OPEN.
TCP = 3
COMMAND FILE CLOSED.
TCP = 0

>
```

SECTION 8
SUBROUTINES, PROGRAMMER DEFINED FUNCTIONS, FUNCTION SUBROUTINES

SUBROUTINES

When a part of a program is repeated several times in different places, it can be programmed more efficiently as a subroutine. Subroutine statements are written only once but can be used many times from any place in the main program.

GOSUB AND RETURN

The command used to transfer to a subroutine may be executed directly or indirectly. Its form is GOSUB followed by the line number of the first statement of the subroutine. The GOSUB command is similar to GO TO followed by a line number in that it transfers unconditionally to another part of the program. GOSUB differs in that it will not go beyond the end of the subroutine, which must be indicated by a RETURN command. If the GOSUB command was executed indirectly, the return will be to the statement following the one in which the GOSUB command was given. If GOSUB was executed directly, SUPER BASIC will simply stop when it reaches the end of the subroutine.

The following example of a small subroutine shows two sections of the main program in which the GOSUB command is used.

```
10 S = 3
20 GOSUB 400
30 PRINT H,P,X
.
.
.
100 S = 7
110 GOSUB 400
120 Z = 3*H+P/X
.
.
.
400 H = S*SQR(2),P = 2*S+H
410 IF P<= 10 THEN X = 1 ELSE X = 2
420 RETURN
```

When this program is run, line 20 instructs SUPER BASIC to transfer to the subroutine beginning at line 400. When the RETURN command at the end of the subroutine is reached, a return is made to line 30 (the line following the GOSUB command). Similarly, when the subroutine is called later from line 110, the return will be to line 120.

As an example of the GOSUB command used directly, suppose that the above program has been loaded into SUPER BASIC. A direct GOSUB can be used to execute only the subroutine for a particular value of S as shown below.

```
> S=4 ↵
> GOSUB 400 ↵
```

A subroutine can contain a GOSUB statement which calls either another subroutine or itself. *NOTE: A GO TO statement should not be used within a subroutine to transfer out of the subroutine before the RETURN command is reached.*

Example 1

```
.
.
.
40 X = SIN(Y+Z)
50 GOSUB 200
60 PRINT X
.
.
.
200 Q = X+R/S
210 IF Q<.5 THEN RETURN
220 PRINT "Q =";Q
230 GOSUB 500
240 RETURN
.
.
.
```

```
500 V = Q+R/S
510 PRINT "V =";V
520 RETURN
.
.
.
```

The subroutine beginning at line 200 contains a GOSUB command which calls another subroutine. The program continues in order until it reaches the GOSUB 500 command. A transfer is then made to the subroutine beginning at line 500. Note the effect of the RETURN commands in this program: Line 520 causes a return to line 240, which in turn causes a return to line 60 (the statement following the GOSUB 200 command).

Example 2

```
10 INPUT A
20 IF A#0 THEN GOSUB 1000
30 B = 1/COS(A)
.
.
.
1000 A = 1/SIN(A/3)
1010 IF A>0 THEN RETURN
1020 GOSUB 1000
1030 RETURN
.
.
.
```

Line 20 instructs SUPER BASIC to execute the subroutine beginning at line 1000 if A is not zero. The specified subroutine assigns a new value to A (on line 1000), and a return is made to line 30 if A is positive. If A is not positive, the GOSUB 1000 command in line 1020 is executed. The subroutine will continue to call itself in this way until A is positive. Then a return will be made to line 1030, which in turn causes a return to line 30.

Note that a subroutine which calls itself must contain at least one condition on which a transfer out of the subroutine can be made (such as line 1010 above); otherwise, an infinite loop will result.

ISOLATING SUBROUTINES

Subroutines must be isolated from the main program; this is not done automatically by SUPER BASIC. The sequence of steps in the program should be designed so that the statements of the subroutine are executed **only** after a GOSUB command.

Either of the indirect commands STOP or END may be used to isolate subroutines. These commands cause execution of the program to terminate. All subroutines can be placed at the end of the main program and separated from the main program by a STOP or END statement as illustrated below:

```
10 ! MAIN PROGRAM BEGINS
.
.
.
100 GOSUB 700
.
.
.
690 STOP ! MAIN PROGRAM ENDS
700 ! SUBROUTINE BEGINS
.
.
.
790 RETURN ! SUBROUTINE ENDS
```

NOTE: A STOP or END statement may be used anywhere in a program to terminate execution. Remember that no such command is required at the end of an entire program, since SUPER BASIC stops automatically as soon as there are no more statements to be executed.

COMPUTED GOSUB STATEMENT

A computed GOSUB statement, which may be executed directly or indirectly, causes transfer to one of several different parts of a program depending on the value of a specified expression.

The form of the computed GOSUB statement is

ON expression GOSUB $line_1$, $line_2$,...

where $line_1$, $line_2$,... is a sequence of line numbers to which the program will transfer depending on the value of the expression. If the value of the expression is 1, the program will transfer to the subroutine starting on $line_1$; if the value is 2, the transfer will be to the subroutine starting on $line_2$, and so on. After the subroutine is executed, the program returns to the next statement in order after the computed GOSUB statement.

Example

```
>10 FOR A = 1,2,3↵
>20 ON A GOSUB 100,200,300↵
>30 PRINT "NEXT"↵
>40 NEXT A↵
>50 STOP↵
>100 PRINT "SUBROUTINE AT 100,A =":A↵
>110 RETURN↵
>200 PRINT "SUBROUTINE AT 200,A =":A↵
>210 RETURN↵
>300 PRINT "SUBROUTINE AT 300, A =":A↵
>310 RETURN↵
>RUN↵
SUBROUTINE AT 100, A = 1
NEXT
SUBROUTINE AT 200, A = 2
NEXT
SUBROUTINE AT 300, A = 3
NEXT

>
```

PROGRAMMER DEFINED FUNCTIONS

The defined function as explained in this section can consist of only a single line. Multiple line functions (function subroutines) are described below under *FUNCTION SUBROUTINES*.

In addition to the standard SUPER BASIC functions, the user may define any other function which he expects to use a number of times in a program. The indirect command DEF is used for this purpose. The names of programmer defined functions must contain three letters, the first two of which must be FN. The form of the DEF statement is shown below; the programmer defines a function which will calculate the sine of an angle in degrees.

```
10 DEF FNS(X) = SIN(X*PI/180)
```

NOTE: If more than one DEF statement is used with the same function name, the current definition is the last one executed.

An argument used in defining a function (X in the above example) is called a parameter. A programmer defined function can have either no parameters or any number of parameters (separated by commas and enclosed in parentheses). Parameters are "dummy" arguments; that is, when a defined function is used, certain specified values will temporarily replace the parameters where they appear in the function definition. For example,

```
>10 DEF FND(A,B) = 4*A*B+A↑2↵
>20 Y = FND(2,1)↵
>30 PRINT Y↵
>RUN↵
  12
>
```

When the defined function was used in line 20, 2 and 1 replaced A and B respectively in the function definition in line 10. Thus, Y was set to $(4X2X1)+2^2$, or 12.

A parameter can have any variable name, including the name of a variable used in the same program, without affecting the value of the program variable having the same name. In other words, the parameters are **local** to the function definition. Continuing from the above example, if the lines

```
5 A = 6,B = 4
35 PRINT A,B
```

are written into the program, the A and B parameters will still be replaced by 2 and 1 (as specified in line 20). Once the function has been evaluated however, A and B are restored to their former values as assigned in line 5. Therefore, line 35 will print 6 and 4 as the values of A and B.

Any variables in a function definition which are not parameters of that function simply take the values assigned to them in some previous part of the program; that is, these variables are **global**. For example, consider the following defined function:

```
35 DEF FNK = 6.21083*R↑2+W
```

When the function FNK is used, the variables R and W must have been assigned values previously; these values will be used in evaluating FNK.

When a defined function with parameters is used in a program, any argument (number, variable, or expression) can replace the parameters in the definition. For example, the following is permitted:

```
60 DEF FNP (X,Y,Z) = X/2-4*Y*Z+Z↑2
65 B = FNP(3,Q,R↑3)
```

When line 65 is executed, the parameters X, Y, and Z are set temporarily to the values of 3, Q, and R^3.

NOTE: When any statement of a program is deleted or modified in any way, all programmer-defined functions become undefined.

The defining expression in a DEF statement may include other programmer defined functions as well as parameters, program variables, and standard functions. For example,

```
40 DEF FNR(A) = TAN(B)+A↑2/W
50 DEF FNF(X,Y,Z,K) = 2*Y*Z+LOG(X)-FNR(K)
60 G = FNF(M,N,P,Q)
```

In this example, the DEF statement on line 50 calls for another function previously defined by the programmer, namely, FNR on line 40. When line 60 is executed, the current values of M,N,P, and Q will be transferred directly to the defining expression of line 50. The value of G will be set to

```
2*N*P+LOG(M)-TAN(B)-Q↑2/W
```

Note that when a DEF statement uses one or more previously defined functions, it is possible that parameters will be listed which do not appear directly in the defining expression. For example,

```
100 DEF FNY(Q) = A+6*EXP(Q)
105 DEF FNZ(A) = FNY(2)↑B
110 M = FNZ(5)
```

The parameter A of the function FNZ does not appear in the defining expression, but it specifies that when FNY(2) is evaluated as a part of that function, A will be replaced temporarily by the argument of FNZ (5 in line 110). Thus A is local to FNZ even though it is global (assigned the value of the program variable A) in the function FNY.

NOTE: Functions may not be recursive; that is, a function may not be defined using itself. For example, the following is an illegal, recursive function.

```
DEF FNA(X) = FNA(X-1)*X
```

FUNCTION SUBROUTINES

The use of DEF is not limited to those cases in which the value of the function can be computed within a single statement. SUPER BASIC allows more complicated programmer defined functions, consisting of several lines (as a subroutine) and returning a single value (as a function). The combined use of DEF and GOSUB makes this possible, as illustrated in the following example.

```
10 DEF FNT(X,Y): GOSUB 400
.
.
.
55 PRINT FNT(5,.6)
.
.
.
75 G1=FNT(1,.73)
.
.
.
100 STOP !END OF MAIN PROGRAM
400 IF X<4 THEN 420
410 X=ABS(X)+Y
420 A=X-Y↑3+2
430 RETURN A
```

The function FNT with parameters X and Y is defined as the subroutine beginning at line 400. The subroutine ends with the statement RETURN A, which indicates that the value of A as calculated in the subroutine will be returned when FNT is used.

The usual rules regarding programmer defined functions apply here:

- The function name must contain three letters, the first two of which must be FN.
- The function may have either no parameters or any number of parameters.
- Parameters are dummy arguments; when the function is used, certain specified values replace the parameters where they appear in the func-

tion definition. But since the parameters are local to the function, program variables may have the same names as parameters and will not be affected when the function is used.

- Variables in the function definition which are not parameters simply take the values previously assigned to them in the program.

The last two statements of the function subroutine in our example program are

```
420 A=X-Y↑3+2
430 RETURN A
```

The RETURN at the end of any function subprogram can be followed by a variable or expression. Thus statements 420 and 430 above can be combined as follows:

```
420 RETURN X-Y↑3+2
```

The only difference between the two methods is that the first creates an additional program variable, A.

Remember that only the parameters listed in the DEF statement are **local** to the function. Any other variables appearing in the function subroutine are **global;** that is, they are equivalent to variables with the same names that appear outside the subroutine. For example, if the parameters X and Y were referred to in line 76 of the example program, the error message VARIABLE HAS NO VALUE would be printed unless X and Y previously had been given values **in the main program.** But if A were referred to in line 76, the value of A would be that value assigned it in the function subroutine when FNT was used in line 75. In other words, A would equal $1-.73^3+2$.

RETURN may also be used alone at the end of a function subroutine; it need not be followed by a variable or expression. This is done when a function is needed simply to execute a group of statements without returning a value, although in fact, a value of 0 is returned. For example,

```
30 DEF FNC(C1,C2,S,T):GOSUB 700
.
.
.
50 Z=FNC(2,3,1,X)
.
.
.
75 Z=FNC(3,4,2,Y)
.
.
.
690 STOP
700 FOR I=1 TO 20
710 IF A(C1,I)>A(C2,I) THEN 730
720 D(S)=A(C2,I),E=T+1
730 NEXT I
740 RETURN
```

When the function FNC is used in line 50 above, the subroutine starting at line 700 is executed. This subroutine assigns values to an element of the array D and to the variable E. (Note that the value of the program variable X replaces the parameter T in the subroutine.) The value returned, and therefore stored in Z in line 50, is 0 because RETURN was not followed by a variable or expression. Z is a "dummy" variable in the sense that it is used in the program only as a means of executing the subroutine defining FNC. The fact that 0 is returned becomes significant if the user does not want to use a "dummy" variable of this kind. For example,

```
50 Z=FNC(2,3,1,X)
55 Q=SQR(D(S))
```

can be replaced by

```
50 Q=FNC(2,3,1,X)+SQR(D(S))
```

When the function name is encountered, the defining subroutine is executed. Since 0 is returned, the value of SQR(D(S)) alone is assigned to Q.

The number of parameters in the function definition and the number of arguments in the function call should be equal. An error message is printed if more arguments are included than required. For example,

```
>10 DEF FNC(A,B)=A+B ↵
>20 PRINT FNC(2,5,1) ↵
>RUN ↵

ERROR IN STEP 20:
WRONG NUMBER OF ARGUMENTS IN
                        FUNCTION CALL
>
```

However, if fewer arguments are included in the call, the unlisted arguments cause an error message unless they have been assigned a previous value in the program. For example,

```
>10 DEF FNA(A,B)=A*B ↵
>20 PRINT FNA(4) ↵
>RUN ↵
```

```
ERROR IN STEP 20:
VARIABLE HAS NO VALUE
>
```

If B has been assigned a value in the main program, that value will be used to compute the function value.

```
> LIST ↵
10 B=6
20 DEF FNA(A,B)=A*B
30 PRINT FNA(4)
  > RUN ↵
 24

>
```

Variables whose values are strings of more than eleven characters must not be used as function arguments.

All functions become undefined if the program is modified in any way. Thus, if a modification is made in a program containing a single or multiple line function, the program should be reexecuted with the RUN command. For example,

```
> 10 DEF FNC(A,B)=A+B ↵
> 20 PRINT FNC(4) ↵
> RUN ↵

ERROR IN STEP 20:
VARIABLE HAS NO VALUE
> 20 PRINT FNC(4,5) ↵
> RUN ↵
  9

>
```

Arrays may not be used as function arguments, but subscripted variables may be used.

SECTION 9
PROGRAM CONTROL AND EDITING

PROGRAM CONTROL FEATURES

This section describes the SUPER BASIC features allowing control of running programs, the interactive functions TEL and WAIT, program loading and linking, binary program files, and the SUPER BASIC Optimizer.

CONTROL OF RUNNING PROGRAMS

SUPER BASIC gives the user complete control of his running program. An indirect PAUSE, STOP, or END statement causes program execution to be interrupted, as does pressing the ALT MODE/ESCAPE key. The user then can enter direct statements which will, for example, assign or change variable values, print values, or list parts of the program. He then may resume execution at the point of interruption or anywhere else in his program.

PROGRAM CONTROLS		
Command	**Effect**	**To Continue**
100 PAUSE	Interrupts the program at statement 100. The message PAUSE IN 100 is printed. Direct statements can be entered.	a) Type GO to continue execution at the point of interruption. All information in the program before interruption is retained. *NOTE: After interruption, if the user types an indirect statement or deletes a statement, GO will not continue execution. Any information about FOR loops or GOSUB commands is lost.* b) Type GO TO *line number* to continue execution from anywhere in the program, or START (same as GO TO the first statement). All information in the program is retained. c) Type RUN to reinitialize execution from the beginning of the program. No information is retained; that is, all values are reinitialized and all files are closed.[1]

1 - Except the command file, explained on page 89.

PROGRAM CONTROLS (Continued)		
Command	**Effect**	**To Continue**
ALT MODE/ ESCAPE	Finishes execution of the statement that was being executed when ALT MODE was pressed and prints the message INTERRUPTED BEFORE followed by the line number of the next statement. (Note exception below.)	Same as PAUSE.
ALT MODE/ ESCAPE (Twice)	To interrupt execution of an INPUT statement or a statement with an infinite loop (such as PRINT A WHILE A>1), press ALT MODE twice. The message printed is INTERRUPTED IN followed by the line number of the statement interrupted.	Same as PAUSE except that GO will not resume execution reliably.
Normal end of program	Terminates program execution.	a) Type START (same as GO TO the first statement), or GO TO *line number* to continue execution anywhere in the program except inside a FOR loop or a subroutine. All information in the program is retained. b) Type RUN to reinitialize execution from the beginning of the program. No information is retained; that is, all values are reinitialized and all files are closed.
35 STOP *or* 35 END	Terminates program execution at statement 35.	
Program execution error	Terminates program execution and prints the message ERROR IN followed by the line number and an error diagnostic.	
>QUIT *or* >Q *or* 100 QUIT	Returns to the EXECUTIVE. Closes all files.[1]	a) Type REENTER to return to SUPER BASIC and continue. b) Type SBASIC to reinitialize SUPER BASIC.

NOTE: Although RUN normally retains no information, a VAR=ZERO or VAR=UNDEF command will be retained when the RUN command is given.

SUPER BASIC ERROR MESSAGES

When an error occurs during execution of a SUPER BASIC program, an error message is printed, and execution is terminated.

Division by zero is indicated by an error message.

For example,

```
>10 A=5,B=0 ↵
>20 PRINT A/B ↵
>RUN ↵
```

1 - Except the command file, explained on page 89.

```
ERROR IN STEP 20:
ATTEMPT TO DIVIDE BY ZERO
>
```

Certain other arithmetic operations cause errors if a number is too large to be converted to the type of variable in which it is being stored.

TEL AND WAIT

TEL

TEL is a logical function with no argument. It returns 1 (true) if terminal input is waiting to be processed, and 0 (false) if no terminal input is waiting.

For example, suppose a program contains a FOR... UNTIL loop that the user knows may take a long time to terminate. If this should happen, he wants to be able to check the value of the indexing variable at any time during the execution of the loop to see how far it has progressed. If the program is set up in part as

```
10 STRING A
.
.
.
90 PRINT "NOW WE CALCULATE X"
95 FOR I=1 UNTIL ABS(X-Y)<1E-8
.
.
.
115 IF TEL THEN INPUT A ELSE 125
120 PRINT "I=":I
125 NEXT I
130 PRINT "X IS":X
.
.
.
```

then any string typed during the execution of the loop will cause the current value of I to be printed. Although the first character of the string is enough to cause TEL in line 115 to be true, the INPUT statement waits, as usual, for the terminating Carriage Return. Note that the string typed is stored in A, not because it is going to be used, but to clear the input so that TEL will again be false until another string is typed.

The use of this program and the resulting printout for lines 90 to 130 are shown below. Notice that the question mark usually supplied by INPUT is suppressed.

```
NOW WE CALCULATE X
S↵
I= 57
NOW↵
I= 102
X IS 3.1349
```

Adding the statement

```
121 PAUSE
```

would enable the user to continue in the loop by typing GO after the value of I is printed, or to change or print the values of some variables and then type GO.

WAIT

A statement of the form

WAIT(*n*)

causes SUPER BASIC to wait n seconds before proceeding with the program. The number of seconds, n, may be any numeric variable or expression. For example,

```
>10 STRING A2↵
>20 PRINT "IN WHAT YEAR DID BALBOA ↓
  DISCOVER THE PACIFIC?"↵
>30 WAIT(10)↵
>40 IF TEL THEN 80 ELSE PRINT "DO ↓
  YOU GIVE UP ":↵
>50 INPUT A2↵
>60 IF A2="YES" THEN PRINT "1513" ↓
  ELSE 100↵
>70 STOP↵
>80 INPUT A1↵
>90 GO TO 120 IF A1=1513↵
>100 PRINT "TRY AGAIN..."↵
>110 GO TO 30↵
>120 PRINT "VERY GOOD!!!"↵
>RUN↵
IN WHAT YEAR DID BALBOA
DISCOVER THE PACIFIC?
ten seconds pass
DO YOU GIVE UP ? NO↵
TRY AGAIN...
1532↵
TRY AGAIN...
1513↵
VERY GOOD!!!

>
```

Notice that when TEL was true the question mark usually supplied by INPUT (line 80) was suppressed, but when TEL was false, the question mark was printed (line 50).

LOAD AND LINK

The LOAD command can be used to enter statements from a file into SUPER BASIC. Any variable values stored before a LOAD are retained when the LOAD command is given. All declarations are preserved, and files remain open. New statements loaded are merged with statements entered before the LOAD. Each previously entered statement is retained unless a statement with the same line number is loaded, in which case the old statement with that line number will be replaced.

The LINK command also enters statements from a file into SUPER BASIC, preserves variable values and declarations, and leaves files open. However, LINK differs from LOAD in two ways:

1. All statements entered before a LINK are deleted when the LINK command is given.
2. LINK causes immediate execution of the program entered.

NOTE: When any statement of a program is deleted or modified in any way, all programmer-defined functions become undefined.

When executed directly, the LOAD and LINK commands take the form

LOAD file name

LINK file name

For example,

```
>LOAD FILE2↵
>LINK THISFILE↵
>LINK (HRC)@COPY↵
```

However, when LOAD and LINK are executed indirectly, certain rules for specifying file names must be followed so that SUPER BASIC can provide maximum flexibility. The rules which also apply to file names used in the OPEN command are:

- If the file name is a literal name, enclose the name in single or double quote marks.

```
30 LOAD "DATA2"
85 LINK 'MATR'
157 LINK "(SCM)PROG'
```

but...

```
75 LOAD "'MPY'"          Loads 'MPY'
110 LINK "'ASDF'"        Links 'ASDF'
```

Note that in the last two examples above, the file names begin and end with a single quote, therefore double quotes must be used to enclose each name.

- The file name can be any string variable or expression.

```
10 LOAD A        Where A has the string value
                 PROB, loads PROB from the
                 user's own directory.
95 LINK '(JOE)'+F
                 Where F has the string value
                 @FILE, links @FILE from the
                 directory JOE.
30 LOAD "/"+STR(I)+"/"
                 Where I=2 loads / 2/ from
                 the user's own directory.
```

After a direct or indirect LOAD, control returns to the terminal or to the command file from which commands are being taken.

Direct statements may be placed on a program file in EDITOR. If the program file is used as a LOAD or LINK file, the direct statements are executed when encountered on the file.

BINARY PROGRAM FILES

This feature allows the user to save a compiled SUPER BASIC program on a binary program (or GO) file, thus allowing extremely rapid loading and execution of production programs. To create a binary program file, first enter the program into SUPER BASIC and then use the direct command of the form

SAVE BINARY file name

Example

```
>LOAD ROOT↵
>SAVE BINARY BROOT↵
  NEW FILE↵
>
```

NOTE: No line ranges may be specified with the SAVE BINARY command; only the entire program may be saved on a binary program file.

A binary program file may be executed either:

- In SUPER BASIC with an **indirect** LINK command.
- In the EXECUTIVE with the command

  GO file name

When an indirect LINK is used to execute a binary program file, all current program statements are deleted, the binary program file is loaded into SUPER BASIC, and execution of the program is started automatically. As usual, LINK preserves values of variables and declarations and leaves files open. After execution of the program, the user is automatically returned to the EXECUTIVE.

NOTE: Binary program files should be used for debugged, production programs only since no debugging is allowed during execution of a binary program file. All program interruptions (error messages; the commands STOP, END, and PAUSE; and ALT MODE/ESCAPE) cause a return to the EXECUTIVE.

Example

```
>10 LINK "BROOT"
>RUN
TYPE THE NUMBER ? 7777
CUBE ROOT: 19.812413

TYPE THE NUMBER ? -45.9
CUBE ROOT:-3.5804496

TYPE THE NUMBER ? DONE

-
```

The EXECUTIVE command GO causes execution of the program directly from the EXECUTIVE. Thus,

```
-GO BROOT
TYPE THE NUMBER ? 7777
CUBE ROOT: 19.812413

TYPE THE NUMBER ? -45.9
CUBE ROOT:-3.5804496

TYPE THE NUMBER ? DONE

-
```

LINK may be used to link two or more binary program files. For example, suppose the program set up in part as

```
10 !SORT ROUTINE
.
.
.
200 IF K>1 THEN LINK "BMERGE" ELSE STOP
```

is stored on the binary program file BSORT. When it is run using LINK or GO, execution of line 200 causes execution of the binary program file BMERGE if K is greater than 1; the values of K and all other variables are retained for use in executing BMERGE. If K is not greater than 1 at line 200, BSORT is stopped and the user is returned to the EXECUTIVE.

NOTE: After a LINK statement has been used to execute a binary program file, linking to a symbolic program is not permitted.

Program files created with the SAVE BINARY command can never be listed in SUPER BASIC since any interruptions of the running program cause an immediate return to the EXECUTIVE, and since loading the program causes immediate execution. Thus, the user can insure complete protection for a program he wishes to share by storing it on a binary program file. If the program is then declared PUBLIC PROPRIETARY with no WRITE ACCESS using the EXECUTIVE command DECLARE, other users may run the program but may never copy it, write on it, or list it. This may be done as follows:

```
-DECLARE
FILE(S): BROOT
PRIVATE:
 WRITE ACCESS? Y
 READ ACCESS? Y
 PUBLIC? Y
PUBLIC:
 PROPRIETARY? Y
 WRITE ACCESS? N

-
```

NOTE: Copying a binary program file always causes the copied file to be nonexecutable.

The execution speed of a binary program file can be increased using the SUPER BASIC Optimizer discussed below.

SUPER BASIC OPTIMIZER

The SUPER BASIC Optimizer is a Tymshare feature which enables SUPER BASIC programs to be run at the fastest possible speeds. To use the Optimizer feature, the SUPER BASIC program should be completely debugged and contain no reference to linked files.

Procedure

To use the Optimizer, load the program to be run into SUPER BASIC. Then type the SUPER BASIC command

SAVE BINARY file name

followed by a Carriage Return. The computer will respond with the OLD FILE/NEW FILE message. Type a Carriage Return and the file will be saved in binary code.[1]

When the SUPER BASIC mark appears, type QUIT↵ to return to the EXECUTIVE. When the EXECUTIVE dash appears, type FSB↵ to call the Optimizer. The Optimizer responds with the version number and the message

INPUT FILE:

Type the name of the binary program file on which the program currently exists. The Optimizer writes

OUTPUT FILE:

Reply with a file name to indicate the file on which the optimized program is to be written.

When the system responds with the OLD FILE/ NEW FILE message, type the appropriate response. Next, the program indicates the change in size accomplished by the optimization.

For example:

ORIGINAL PROGRAM SIZE=3023 WORDS
FINAL PROGRAM SIZE=2090 WORDS

When the EXECUTIVE dash appears, type GO file name↵. The file name must be the same output file assigned in answer to OUTPUT FILE: above. The program then runs. A typical run is shown below.

The program is loaded into SUPER BASIC.

```
>SAVE BINARY SLOWFILE↵
  NEW FILE↵
>QUIT↵
-FSB↵
FSB A2.03   This number is the latest update of FSB.
INPUT FILE: SLOWFILE↵
OUTPUT FILE: FASTFILE↵
  NEW FILE↵
ORIGINAL PROGRAM SIZE=3224 WORDS
FINAL PROGRAM SIZE=3015 WORDS
-GO FASTFILE↵
```

The program runs.

Notes

1. Any file names may be used for the input and output files in FSB, as long as the two file names are different.

2. It is unnecessary to return to the EXECUTIVE and run FSB immediately after saving the SUPER BASIC program as a binary file. FSB may be called at any time from the EXECUTIVE.

3. Similarly, the optimized program need not be run immediately after FSB has completed the optimization. The instruction GO FASTFILE can be executed at any time. In fact, once FSB has optimized a SUPER BASIC program, it may be run at any time by invoking the EXECUTIVE GO command.

4. The optimized program is accessible only from the EXECUTIVE.

Suggestions

To obtain the best optimized program, three areas should be given special attention.

1. Declarations

Real and integer variables should be declared as such. This applies to simple variables as well as to arrays, and should be made for **all** real and integer variables which retain their original mode throughout the program. Thus, the single most important rule for maximum performance is:

DECLARE EVERYTHING

The variable declarations must be executed before the variable is used. For example, the following sequence of statements will cause errors because the declaration statement is not encountered until after the variable has been used.

```
1 GO TO 500
2 INTEGER N,I,J
.
.
.
500 INPUT N
501 REAL X(N)
502 GO TO 2
```

2. Mixed Mode Expressions

A mixed mode expression (one which contains both real and integer variables) generates slightly more binary code than one which is either all real or all integer. On the other hand, if there are several integer variables which can be reduced to one computed value before being converted to floating point, a mixed mode expression of this type will run slightly

1 - Binary program files are discussed on page 100.

faster than an all real expression of the same size. The method of handling mixed mode expressions is left to the judgment of the programmer.

The sequence of operations is not changed by the Optimizer. Thus, if I and J are declared integer and X is declared real,

I+J+X or X+(I+J) is faster than **I+X+J or X+I+J.**

3. IF...THEN...ELSE...Statements

The use of IF. . .THEN . . . ELSE statements is encouraged, even when they become very long. Using several statements to perform the same computation is less efficient. For example:

```
100 IF A(I)<B(J) THEN B(J)=A(I) ELSE
                      B(J)=B(J)-A(I)
```

will generate less code and run faster than

```
100 IF A(I)<B(J) THEN 130
110 B(J)=B(J)-A(I)
120 GO TO 140
130 B(J)=A(I)
140 . . .
```

Optimized Operations

The Optimizer is effective in optimizing certain operations on real and integer variables. CPU time of programs which contain a large amount of arithmetic computation will be reduced significantly in proportion to the number of arithmetic operations involved.

The following operations are optimized:

1. Simple Variable Expressions

Expressions involving simple variables of a single type—real or integer—are optimized if the variables have been declared and retain their original mode throughout the program.

2. Array Access

Access to arrays of one and two dimensions is optimized. Not only is the address computation for an array element optimized, but also the address is remembered and used again where possible.

3. Arithmetic Operations

The arithmetic operations of subtraction, negation (unary minus), and multiplication are optimized. Exponentiation to a constant integer power is performed as the appropriate sequence of multiplications. The division operator, /, is optimized for real numbers and variables. For integers, the MOD and DIV operators are optimized. References to the standard mathematical functions (SQRT, SIN, etc.) invoke the same routines as before, but do not prevent the optimization of the rest of the expression.

4. Relational Operators

All relational operators (<, =, >=, etc.) are optimized.

5. Logical Operators

The three logical operators AND, OR, and NOT are optimized.

6. Binary Operators

The binary operators BAN, BEX, and BOR are optimized.

Restrictions

There are two restrictions to remember when using optimized SUPER BASIC programs.

1. Optimized GO files may not be linked.

2. Optimized variables will be set to zero when declared. Thus, a declared variable is never undefined.

EDITING FEATURES

The editing commands and characters described beginning on page 14 are only a small part of the extensive editing features available in SUPER BASIC. Instead of retyping an entire line that needs changing, the user may let certain control characters do the editing for him. These control characters, which are the same as those available in the Tymshare EDITOR language, are summarized in the table on page 106.

The first set of characters listed in the table can be used at any time – while typing direct and indirect statements, file names, and even data input from the keyboard. The second set of characters is used to edit lines already typed, even if a syntax error was made in the line. The EDIT and MODIFY commands allow editing of any existing line in a program. Further explanation and examples of these editing features are given below.

EDITING THE LINE BEING TYPED

In the following example, Control Q (Q^C) is used to delete the line being typed. While retyping the line, two incorrect characters are deleted with A^C's.

```
>40 FOR I = 1 TO Q^C↑
40 PRINT I↑3 FOR I = 1 TO 25A^C←A^C←50 ↵
>LIST 40 ↵
40 PRINT I↑3 FOR I = 1 TO 50
>
```

The TABS Command

The tab stops which determine to which character position I^C will type are initialized at 7, 15, and at every fifth position from 15 on. The direct command TABS allows the user to set any other tabs that he wishes. For example,

```
>TABS 10,20,30 ↵
```

sets the tab stops at the specified positions. A Control I subsequently typed at the beginning of a line will space to position 10. *NOTE: A maximum of ten tabs may be set with the TABS command.*

File Name Editing

File names typed in SUPER BASIC commands can be edited also. For example,

```
>SAVE XYA^C←Z ↵
```

will save the program on a file named XZ.

To include a control character in a file name, precede the character by V^C so that no editing will occur. For example,

```
>LOAD /PV^CW^CR/ ↵
```

must be typed to load from a file named /PW^{C}R/.

Data Input Editing

The control characters A^C, W^C, and Q^C have special properties when used to edit data typed in response to the INPUT command.

Control A will delete the preceding character unless that character is a comma (or space) used to separate data items. Once such a character is typed, the preceding value is stored in a variable and is not available for editing.

If a string data item is being entered enclosed in quote marks, Control A will delete any character except the initial and terminal quote marks. If the first character in the string is deleted, a question mark is printed and the data item must be reentered, including the quote marks. For example,

```
>10 INPUT A,B,C ↵
>20 PRINT A;B;C ↵
>RUN ↵
?  123,56A^C←5,"ERA^C←A^C←? "STRING" ↵
   123          55    STRING

>
```

Once the comma was typed after 123, no editing could be done to that value. The first Control A deleted 6. The second and third A^C's deleted ER; the question mark was printed and the user retyped the value of the variable C.

Control W, which deletes the preceding data item, also has no effect on the characters which A^C cannot delete. For example,

```
>INPUT X,Y,Z ↵
? "SMYTW^C \? "SMITH",64,92W^C \? 93.8 ↵

>PRINT X:Y:Z ↵
SMITH  64 93.8

>
```

The first W^C deleted SMYT and the leading quote marks. The second W^C deleted 92; another Control W typed there would have done nothing, since 64 was already stored in the variable Y.

Control Q restarts the entire statement containing the INPUT command, causing SUPER BASIC to print another question mark. Since direct statements are not saved and therefore cannot be restarted, Q^C applies only when the INPUT command was executed indirectly. For example,

```
>10 INPUT A(I) FOR I=1 TO 8 ↵
>RUN ↵
? 11.17,33.9,46.1,39,21.8,5.62 ↵
? 13.7Q^C↑? 11.17,85,33.9,46.1,39,21.86 ↵
? 13.7,10.8 ↵

>
```

Note that the values for A(1) to A(6) were actually stored before the Q^C was typed. The user then entered new input values. Thus, if the INPUT command were in a statement such as

```
>55 IF A=0 THEN INPUT A,B ↵
                ELSE PRINT "NO" ↵
```

the following might occur

```
? 5,7.5Q^C↑NO
```

(execution continues from the statement after 55)

Statement 55 was restarted, but since A was actually assigned the value of 5 before Q^C was typed, A was no longer equal to zero and INPUT A,B was not executed.

EDITING A LINE ALREADY TYPED

EDIT And MODIFY

The direct commands EDIT and MODIFY allow the user to edit any statement in his program by using an extensive set of control characters. EDIT followed by a line number causes SUPER BASIC to print the specified line and wait for the user to edit. MODIFY (or MOD) is the same as EDIT except that the specified line is not printed.

Example 1

```
>EDIT 20 ↵
20 A = SQR(PI*M↑2)        This is line 20.
Z^C*20 A = SQR(PI*ND^C↑2)
                          Z^C * copies up to and including the *. The user typed N to replace the incorrect M, and D^C to copy the rest of the line.
>LIST 20 ↵
20 A = SQR(PI*N↑2)        This is the new line 20.
```

Example 2

```
>10 INPUT A(I) FOR I = 1 TO 10 ↵
>20 GOSUB 100 ↵
>MODIFY 10 ↵              Line 10 does not print.
30^CA10 INPUT BF^C        3 replaces 1 so that the edited line will be line 30. O^CA copies up to but not including A. The user types B to replace the A, and F^C which copies but does not print the rest of the line.
>LIST ↵
10 INPUT A(I) FOR I = 1 TO 10
20 GOSUB 100
30 INPUT B(I) FOR I = 1 TO 10
>
```

Editing The Previous Line

After the user types any indirect statement, that statement is immediately available for edit as though the EDIT or MODIFY command had been given. For example,

```
>45 IF Y = 20 THEN NEXT I ↵
>Z^C245 IF Y = 25D^C THEN NEXT I
                          Z^C and D^C are used to edit the line just typed. The 20 is changed to 25.
>LIST 45 ↵
45 IF Y = 25 THEN NEXT I
>
```

This can be done even if a syntax error is made in the statement just typed.

Direct statements can be edited after they are typed **only** if a syntax error is made. Once the statement begins to execute, it is no longer available for edit. For example,

```
>PRINT "AREA IS:A ↵
MISSING "                 This is a syntax error.
>Z^CSPRINT "AREA ISE^C<"E^C>D^C:A
                          The statement is edited. D^C copies the rest of the line and causes the statement to be executed.
AREA IS 35

>PRINT "VOLUME IS":X ↵
                          This statement contained no syntax errors, so SUPER BASIC began to execute it.
VOLUME IS
VARIABLE HAS NO VALUE     The variable X was not defined (a program error).

>                         Control characters will have no effect here.
```

Correcting Syntax Errors

If a syntax error is made while typing a direct or indirect statement, Control B can be used to copy the statement up to the syntax error. The user then corrects the error and completes the statement, using other control characters if he wishes. For example,

```
>10 C23=B↑3+Y ↵
SYNTAX ERROR
>B^C10 CE^C<(E^C>23E^C<)E^C>D^C=B↑3+Y
```

The user forgot to put parentheses around the subscript of C. He typed Control B which copied up to the syntax error. He used Control E to insert the parentheses and Control D to copy the rest of the statement and to end the edit.

Examples

```
>194 OPEN "DATA",INPUT 3↵
SYNTAX ERROR
>B^C194 OPEN "DATA",INPUT,3↵

>A=LEFT(V,INDEX(V," ")↵
MISSING )
>B^CA=LEFT(V,INDEX(V," "))↵
```

EDITING CONTROL CHARACTERS

The following control characters may be used at any time while typing direct and indirect statements, file names and data input from the terminal.

Control Character	Symbol Printed	Function
A^C *or* ←	←	Deletes the preceding character typed.
W^C	\	Deletes the preceding word typed.
Q^C	↑	Deletes the entire line being typed.
V^C and a character		Indicates that the control character that follows is to be accepted as any other character (it will not perform its editing function).

The following control characters can be used only during EDIT, MODIFY, and edit of a previous line for deleting, copying, and inserting.

Control Character	Symbol Printed	Function
For Deleting		
S^C	%	Deletes the next character in the line being edited (the "old line").
K^C		Deletes the next character in the old line; prints the character it deletes.
P^C and a character	%	Deletes up to but not including the character typed after it.
X^C and a character	%	Deletes up to and including the character typed after it.
Carriage Return		Deletes the rest of the old line and ends the edit.
For Copying		
B^C		Copies up to syntax error. *Used only to correct a line containing a syntax error.*
C^C		Copies the next character in the old line.
O^C and a character		Copies up to but not including the character typed after it.
Z^C and a character		Copies up to and including the character typed after it.
D^C[1]		Copies and prints the rest of the old line and ends the edit.
F^C[1]		Copies but does not print the rest of the old line and ends the edit.
H^C		Copies and prints the rest of the old line and continues the edit at the end of the line.
Y^C		Copies but does not print the rest of the old line and continues the edit at the beginning of the new line (same as F^C followed by MODIFY of the line as edited).
R^C		Copies and prints the rest of the old line plus the new line; continues the edit from where R^C was typed.
T^C		Same as R^C except that it aligns the rest of the old line and the new line.
U^C		Copies from the old line up to the next tab stop in the new line.
For Inserting		
E^C *text* E^C	< >	Inserts text into the old line; first E^C prints <, second E^C prints >.
Other		
N^C		Backspaces in the old and in the new line.
I^C		Types spaces up to the next tab stop.

1 - If the user has typed past the end of the old line, he should use a Carriage Return to end the line, not D^C or F^C.

THE RENUMBER COMMAND

Renumbering To The End Of The Program

All or some of the statements in a program may be renumbered with a direct command which takes the form:

RENUMBER N1,N2,N3 *or*
REN N1,N2,N3

where N1 will be the first new line number, N2 is the number of the line in the program where renumbering will begin, and N3 is the increment to be used in assigning the new line numbers.

Example

```
> 1 !THIS IS A TEST PROGRAM↵
> 10 INPUT P,I,N↵
> 11 M = P*(I+1)↑N↵
> 15 PRINT M↵
> 20 GO TO 10↵
> RENUMBER 20,10,2↵
> LIST↵
1 !THIS IS A TEST PROGRAM
20 INPUT P,I,N
22 M = P*(I+1)↑N
24 PRINT M
26 GO TO 20
>
```

In this example, the program is renumbered from line 10 to the end of the program, in steps of 2, with 20 as the first new line number. Line 1 remains unchanged. Notice that the line number referred to in the GO TO statement also has been changed correctly.

NOTE: If this program had been run before the renumbering, the variable values would have been lost as a result of the RENUMBER command.

Certain words may be included in the RENUMBER command to help the user remember the order and meaning of the three arguments. For example,

RENUMBER 20,10,2

can be typed as

RENUMBER AS 20 FROM 10 BY 2 *or*
RENUMBER AS 20 FROM 10 INC 2

Any of these prompting words may be used or not as desired. AS is optional, and either FROM, BY, or INC may be replaced by a comma.

Renumbering A Range Of Lines

A range of lines may be specified for renumbering. For example,

RENUMBER 200,90-205,10

will renumber lines 90 to 205 as 200, 210, 220 and so on.

An additional prompting word may be included in this form of the RENUMBER command; namely, the dash used in indicating the line range may be replaced by the word TO.

When the RENUMBER command is given, SUPER BASIC first checks to see that after the requested renumbering is done, the renumbered line range will still have line numbers that are different from the rest of the program. If this is not the case, an error message will be printed, since it is impossible for two program lines to begin with the same number.

Omitting Parts Of The RENUMBER Command

One or more parts of the RENUMBER command may be omitted, with the following results:

Omitted	Result
N1	First new line number is assumed to be 100.
N2	Program is renumbered from the beginning (the lowest numbered statement).
N3	Increment is assumed to be 10.

Examples

RENUMBER	All three parts are omitted. RENUMBER 100, 0, 10 is assumed. (The 0 will cause renumbering to begin from the lowest numbered statement.)
RENUMBER , , 5 *or* RENUMBER BY 5 *or* RENUMBER INC 5	First two parts are omitted. RENUMBER 100, 0, 5 is assumed.
RENUMBER 10 *or* RENUMBER AS 10	Last two parts are omitted. RENUMBER 10, 0, 10 is assumed.
RENUMBER 10,,5 *or* RENUMBER AS 10 BY 5	Second part is omitted. RENUMBER 10, 0, 5 is assumed.
RENUMBER 150, 115-210 *or* RENUMBER AS 150 FROM 115 TO 210	Third part is omitted. RENUMBER 105, 115-210, 10 is assumed.

RENUMBER With ADD

There is another form of the RENUMBER command in which the numbers of the specified lines are increased by a certain amount. For example,

RENUMBER 150 ADD 10 *or*
RENUMBER FROM 150 ADD 10

will renumber from line 150 to the end of the program by adding 10 to every line number.

A range of lines may be specified, such as

RENUMBER 210-340 ADD 20 *or*
RENUMBER FROM 210 TO 340 ADD 20

which will add 20 to the line numbers 210-340 inclusive.

A negative number may be typed after ADD to decrease the specified lines by a certain amount. For example,

RENUMBER 500-545 ADD -100

will subtract 100 from the line numbers 500-545 inclusive.

SECTION 10

DEBUGGING AIDS

The Tymshare system provides two useful aids to assist the SUPER BASIC programmer in debugging his program. The SUPER BASIC index generator is a Tymshare Library program which lists all variables, arrays, programmer-defined functions, subroutines, and statement transfers in a specified program. The SUPER BASIC MAP statement prints a table of storage allocation for a program.

SUPER BASIC INDEX GENERATOR

The SUPER BASIC index generator is designed to aid the SUPER BASIC programmer in developing, debugging, modifying, and documenting his programs. The index generator accepts as input a symbolic SUPER BASIC program and lists all references to variables, arrays, programmer-defined functions, and line numbers, including transfers to subroutines.

The information supplied by the index generator is extremely useful if modifications are to be made to an unfamiliar program. The index generator is also useful in program development since it provides a list of all variables used and the statements in which they appear. This provides assistance in assigning new variables and in making corrections to program logic.

The SUPER BASIC index generator is called by typing SBIG and a Carriage Return in the EXECUTIVE. The program replies with

TYPE FILE NAME OF SBASIC PROGRAM:

Enter the name of the symbolic file containing the program to be indexed. Follow the file name with a Carriage Return. SBIG then types

OUTPUT REPORT TO:

Enter the name of the file on which the reference information is to be written. Follow the file name with a Carriage Return. If the report is to be listed on the terminal and not written on a file, enter T followed by a Carriage Return.

After the reference information has been generated, SBIG asks

DO YOU HAVE ANOTHER PROGRAM TO PROCESS?

Type NO ↵ to terminate SBIG. Typing YES ↵ causes SBIG to request another file name.

NOTE: Each GOSUB reference listed is preceded by an S.

Example

```
– COPY TESTPROG TO T↵

5   DIM A(3)
10  INPUT X,Y,Z
12  MAT INPUT A
15  DEF FNF(I,J,K)=SQRT(I↑2+J↑2+K↑2)
20  GOSUB 500
25  PRINT X;Y;Z,FNF(X,Y,Z)
30  GOSUB 500
35  GO TO 900
500 PRINT
501 PRINT "THIS IS A TEST"
502 PRINT
503 RETURN
900 MAT PRINT A
910 IF A(3)<Z THEN 10 ELSE PRINT "END"
```

```
-SBIG↵
TYPE FILE NAME OF SBASIC PROGRAM:TESTPROG↵
OUTPUT REPORT TO:INDXTESTPROG↵


DO YOU HAVE ANOTHER SBASIC PROGRAM TO PROCESS? NO↵

-COPY INDXTESTPROG TO T↵



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

PROGRAM NAME IS: TESTPROG



DICTIONARY OF VARIABLES WITH LINE REFERENCES

VAR.--REFERENCE LINE #
NAME
I    15
J    15
K    15
X    10 25
Y    10 25
Z    10 25 910


DICTIONARY OF ARRAYS WITH LINE REFERENCES

ARRAY--REFERENCE LINE #
A    5 12 900 910


DICTIONARY OF PROGRAM DEFINED FUNCTIONS

NAME--REFERENCE LINE #
FNF 15 25


CROSS REFERENCE OF LINE NUMBERS

LINE #   REFERENCE LINE #
10       910
500      S20 S30          The S's indicate that 500 is referenced by GOSUB in
900      35               these lines.



**************************************************
```

THE MAP STATEMENT

The computer has a specified amount of memory available for each SUPER BASIC program. The total number of statements that can be used can be greatly increased by linking programs together, that is, running one program, loading another program and running it, and so forth.

Approximately 8000 words of memory are available for each program. The actual program size and storage use may be determined by using the MAP statement.

The MAP statement may be used as a direct or indirect statement. This statement prints a three-column table. The first column contains the name of the item described in each row. The second column contains the core location, in octal notation, of the last space used by that item; the third column contains the number of words, in decimal notation, used by that item.

The abbreviations used in the first column and the blocks to which they refer are:

REF	Lines referenced by GO TO or GOSUB statements
PRG	Program storage
STR	Strings of more than six characters
EXP	Expression calculations
UNU	Unused storage
FOR	FOR loops
GSB	GOSUB statements
ARY	Arrays

If a program exceeds the maximum size allowed, it must be reduced in size before it will run. The program size can be reduced by deleting statements from the program, reducing the size of dimensioned variables, or modifying program logic.

Example

```
> MAP ↵
        LOC   SIZE WORDS
        OCT   DEC
REF     3521    79
PRG     14617   4495
STR     14616   0
EXP     14620   2
UNU     20043   1683
FOR     20043   0
GSB     20043   0
ARY     24000   2013

>
```

SECTION 11

SAMPLE SUPER BASIC PROGRAMS

This section contains programs written in SUPER BASIC and executed on the Tymshare system. These programs demonstrate many of the features of the SUPER BASIC language.

SOLVING QUADRATIC EQUATIONS

The program below computes roots for quadratic equations of the form

$$AX^2 + BX + C = 0$$

The roots are given by the formulas

$$R1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \qquad R2 = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$$

If B^2-4AC is less than zero, there are no real roots. In this case the program prints the message NO REAL SOLUTION. Each time roots are calculated, the next values for A, B, and C are requested automatically. A zero value for A terminates the program.

```
-SBASIC↵

>LOAD QUADRATIC↵
>LIST↵
100 PRINT ! THIS COMMAND CAUSES A BLANK LINE TO BE PRINTED.
110 PRINT "WHAT ARE A,B, AND C":
120 INPUT A,B,C
130 IF A=0 THEN STOP
140 D=B↑2-4*A*C
150 IF D<0 THEN GO TO 200
160 X1=(-B+SQRT(D))/(2*A)
170 X2=(-B-SQRT(D))/(2*A)
180 PRINT "ROOT 1 IS ";X1,"ROOT 2 IS ";X2
190 GO TO 100
200 PRINT "NO REAL SOLUTION"
210 GO TO 100
 >RUN↵

WHAT ARE A,B, AND C? 3.4,5.6,.08↵
ROOT 1 IS   -1.4411818E-02    ROOT 2 IS   -1.632647

WHAT ARE A,B, AND C? 2.1,1.1,5.3↵
NO REAL SOLUTION

WHAT ARE A,B, AND C? 0,0,0↵

>
```

LISTING STOCKS

This program reads up to 100 items of string and numeric data. Note the use of the ON ENDFILE statement to determine the end of data. The data is printed on the terminal with a picture format, followed by the sum of the numeric information. The left justification of strings and the right justification of numbers is an extremely useful feature of picture formatting.

```
-COPY BSTOCKS TO T↵

ACME LABS,100,AMERICAN INSTRUMENT,100,CONTINENTAL BAKING,100
DIVERSIFIED CONTROL,100,EASTERN METALS,125,GOODSON & CO,140
"HOWARD, J.H.",100,INT'L INDUSTRIES,12,LINCOLN PUBLISHING,50
NATIONAL PHARMACEUTICALS,64,ROYAL CHEMICAL,66,UNITED FURNITURE,135

-SBASIC↵

>LOAD STOCKS↵
>LIST↵
100 S=0
110 STRING C(100)
120 INTEGER N(100)
130 OPEN "BSTOCKS",INPUT,2
140 ON ENDFILE(2) GO TO 190
150 FOR I=1 TO 100
160 INPUT FROM 2:C(I),N(I)
170 S=S+N(I)
180 NEXT I
190 PRINT
200 PRINT IN FORM "27% 3%/":C(J),N(J) FOR J=1 TO I-1
210 PRINT
220 PRINT "TOTAL NUMBER OF SHARES IS":S
230 CLOSE 2
  >RUN↵

ACME LABS                    100
AMERICAN INSTRUMENT          100
CONTINENTAL BAKING           100
DIVERSIFIED CONTROL          100
EASTERN METALS               125
GOODSON & CO                 140
HOWARD, J.H.                 100
INT'L INDUSTRIES              12
LINCOLN PUBLISHING            50
NATIONAL PHARMACEUTICALS      64
ROYAL CHEMICAL                66
UNITED FURNITURE             135

TOTAL NUMBER OF SHARES IS 1092

>
```

PERCENTAGE BAR CHART

The DATA statement in the following program lists the frequency counts for 10 class intervals, denoted by the numbers 1 through 10. The program calculates the percentage frequency of each class interval expressed as a percent of total. Each percentage frequency is rounded to the nearest integer and plotted on a bar chart.

This program demonstrates the usefulness of the FOR statement modifier, the TAB and ROUN functions, and the MOD operator.

```
>LIST
100 READ Y(I) FOR I=1 TO 10
110 N = Y(1)
120 N = N+Y(I) FOR I=2 TO 10
130 N = 100/N,  K = 0
140 S(I) = ROUN(N*Y(I)),  K = MAX(K,S(I))  FOR I=1 TO 10
150 PRINT
160 PRINT TAB(11):"PERCENTAGE BAR CHART"
170 PRINT
180 FOR Y=K TO 1 STEP -1
190 PRINT Y:IF Y MOD 5 =0
200 PRINT TAB(3*I+3):"XX":IF S(I)>=Y FOR I= 1 TO 10
210 PRINT
220 NEXT Y
230 PRINT
240 PRINT IN FORM "3B 9(3%) 4%/":I FOR I=1 TO 10
250 DATA 1,5,16,19,30,40,25,21,7,2
 >RUN
```

```
           PERCENTAGE BAR CHART

                        XX
                        XX
                        XX
                        XX
 20                     XX
                        XX
                     XX XX
                     XX XX
                     XX XX
 15                  XX XX XX
                     XX XX XX
                     XX XX XX XX
                     XX XX XX XX
                  XX XX XX XX XX
 10            XX XX XX XX XX XX
               XX XX XX XX XX XX
               XX XX XX XX XX XX
               XX XX XX XX XX XX
               XX XX XX XX XX XX
 5             XX XX XX XX XX XX
               XX XX XX XX XX XX XX
            XX XX XX XX XX XX XX XX
            XX XX XX XX XX XX XX XX
         XX XX XX XX XX XX XX XX XX XX

         1  2  3  4  5  6  7  8  9  10

>
```

DIRECTORY OF ADDRESSES

The file DIR is a variable length random file containing the names and addresses of a number of California residents. The program asks the user whose address he wants. The user may enter any part of the person's name, for example, DALE, MOSS, or DALE MOSS, and that person's full name and address is printed. If the name is not found, an appropriate message is printed and the LOCATE statement is used to reposition the file pointer at the beginning.

Note the value of using the string function INDEX in this example. Since INDEX searches each name in the directory for whatever is typed by the user, any part of a name is acceptable for input. INDEX returns 0 if it does not find the string for which it has searched.

```
-COPY DIR TO T↵

MR. JOHN B. CAREY,285 COTTLE AVENUE,CAMPBELL
MRS. LESLIE FISHER,1964 HAMPTON DRIVE,DANVILLE
MR. CARL LARSON,985 SOUTH 9TH STREET,SAN JOSE
MR. DALE MOSS,1650 SARATOGA AVENUE,SARATOGA
MR. JOHN REY,106 FORMAN STREET,CAMPBELL
MR. DANIEL TORRES,24 SCHARF AVENUE,LOS GATOS
MISS DONNA WILKES,315 SOUTH 3RD STREET,SAN JOSE
MR. MICHAEL YOUNG,60 WILSON ROAD,CHESTER
MR. HENRY C. ZIMMER,15 JACKSON STREET,PALO ALTO

-SBASIC↵

>LOAD ADDRESS↵
>LIST↵
100 STRING N1,N,A,C
110 OPEN "DIR", RANDOM INPUT,2
120 ON ENDFILE(2) GO TO 230
130 PRINT
140 PRINT "ADDRESS OF":
150 INPUT N1
160 IF N1="NONE" THEN 260
170 PRINT
180 INPUT FROM 2:N,A,C
190 IF INDEX(N,N1)=0 THEN 180 ELSE PRINT N
200 PRINT A
210 PRINT C:", CALIFORNIA"
220 GO TO 240
230 PRINT "THE ADDRESS IS NOT LISTED HERE."
240 LOCATE 1 ON 2
250 GO TO 130
260 CLOSE 2
>RUN↵
```

```
ADDRESS OF? JOHN REY↵

MR. JOHN REY
106 FORMAN STREET
CAMPBELL, CALIFORNIA

ADDRESS OF? ZIMMER↵

MR. HENRY C. ZIMMER
15 JACKSON STREET
PALO ALTO, CALIFORNIA

ADDRESS OF? MORRIS↵

THE ADDRESS IS NOT LISTED HERE.

ADDRESS OF? DONNA↵

MISS DONNA WILKES
315 SOUTH 3RD STREET
SAN JOSE, CALIFORNIA

ADDRESS OF? NONE↵

>
```

FUNDAMENTAL FREQUENCY

This program uses the formula

$$F = \frac{.467T}{R^2}\sqrt{\frac{Y}{D(1-P^2)}}$$

to find F, the fundamental frequency of a circular clamped plate. D, Y, and P (the density, Young's modulus, and Poisson's ratio) are read from a DATA statement, and the value of T (the thickness) is requested. Using this data, the program calculates F for a range of radii (R) from .1 to 1 in steps of .1, from 1 to 10 in steps of 1, and from 10 to 100 in steps of 10. Picture formatting is used to print the results on a file.

Line 180 in this example illustrates the use of many instructions in one statement. A number and the value of a programmer-defined function are printed with picture formatting on a file for three distinct ranges of values.

```
-SBASIC ↵

>LOAD FREQ ↵
>LIST ↵
100 READ D,P,Y
110 PRINT "WHAT IS THE THICKNESS OF THE DRUM MATERIAL":
120 INPUT T
130 DEF FNF(X)=(.467*T/X↑2)*SQR(Y/(D*(1-P↑2)))
140 OPEN "X",OUTPUT,2
150 PRINT ON 2:"RADIUS","FUND. FREQ."
160 PRINT ON 2
170 A="%%%.%          %%%%%%%%.%%%"
180 PRINT ON 2 IN IMAGE A:R,FNF(R)
   FOR R=.1 TO .9 BY .1, 1 TO 9 BY 1, 10 TO 100 BY 10
190 CLOSE 2
200 DATA 7.8,.3,20E11
>RUN ↵
WHAT IS THE THICKNESS OF THE DRUM MATERIAL? .672 ↵

>QUIT ↵

-COPY X TO T ↵

RADIUS          FUND. FREQ.

   .1          16658394.930
   .2           4164598.733
   .3           1850932.770
   .4           1041149.683
   .5            666335.797
   .6            462733.193
   .7            339967.243
   .8            260287.421
   .9            205659.197
  1.0            166583.949
  2.0             41645.987
  3.0             18509.328
  4.0             10411.497
  5.0              6663.358
  6.0              4627.332
  7.0              3399.672
  8.0              2602.874
  9.0              2056.592
 10.0              1665.839
 20.0               416.460
 30.0               185.093
 40.0               104.115
 50.0                66.634
 60.0                46.273
 70.0                33.997
 80.0                26.029
 90.0                20.566
100.0                16.658

-
```

CUBE ROOT

This program uses the approximation method to compute the cube root of any number typed by the user. The first approximation is A=N/3, which is compared to the next approximation, $A1=(2A^3+N)/3A^2$. Each time through the loop, the last value of A1 is stored in A, and a new approximation is calculated. As soon as A1 is equal to A when rounded to ten decimal places, that is ABS(A1-A)<EPS, the program prints the cube root, A1, and the number of passes through the iteration loop, I-1. The program terminates if the number 0 is entered.

Two important characteristics of FOR when used with UNTIL or WHILE are illustrated here:

1. A and A1 must be initialized because the terminating condition is checked before the loop is entered. Thus, if A had not been initialized, SUPER BASIC would not have been able to define ABS(A1-A) upon first encountering the loop.
2. The value of I upon exit from the loop is that value which caused the exit to occur, that is, 1 more than the value of I the last time through the loop. For this reason, the number of iterations is I-1, not I.

Note that a binary file is created in SUPER.BASIC and executed in the EXECUTIVE with the GO command. When the program terminates, the user is returned to the EXECUTIVE.

```
-SBASIC↵

>LOAD ROOT↵
>LIST↵
10 PRINT "ENTER THE NUMBER: ";
20 INPUT IN FORM "#":N
30 IF N=0 THEN STOP
40 A=0,A1=N/3
50 A=A1,A1=(2*A↑3+N)/(3*A↑2) FOR I=1 UNTIL ABS(A1-A)<EPS
60 PRINT "CUBE ROOT:";A1
70 PRINT "NUMBER OF ITERATIONS:";I-1
80 PRINT
90 GO TO 10
>SAVE BINARY CUBE↵
 NEW FILE↵

>QUIT↵

-GO CUBE↵
ENTER THE NUMBER: 7777↵
CUBE ROOT: 19.812413
NUMBER OF ITERATIONS: 18

ENTER THE NUMBER: -45.9↵
CUBE ROOT:-3.5804496
NUMBER OF ITERATIONS: 9

ENTER THE NUMBER: 8↵
CUBE ROOT: 2
NUMBER OF ITERATIONS: 6

ENTER THE NUMBER: 0↵

-
```

CHECKING ACCOUNT SERVICE CHARGES

In this problem, we wish to compute the monthly service charge for a regular checking account. The amount of the service charge is based on the average monthly balance and the number of checks written. The charge may be computed from the following table:

AVERAGE MONTHLY BALANCE

NUMBER OF CHECKS	UNDER $200	$200 to $299	$300 to $399	$400 to $499	$500 to $599	$600 to $699	$700 to $799	$800 to $899	$900 to $999	$1000 to $1099	$1100 to $1199	$1200 to $1299	$1300 to $1399	$1400 to $1499	$1500 to $1599
0	$ .75	$ .47	$ .33	$	$	$	$	$	$	$	$	$	$	$	$
1	.82	.54	.40												
2	.89	.61	.47	.33											
3	.96	.68	.54	.40											
4	1.03	.75	.61	.47											
5	1.10	.82	.68	.54											
6	1.17	.89	.75	.61											
7	1.24	.96	.82	.68	.54										
8	1.31	1.03	.89	.75	.61										
9	1.38	1.10	.96	.82	.68	.54									
10	1.45	1.17	1.03	.89	.75	.61									
11	1.52	1.24	1.10	.96	.82	.68	.54								
12	1.59	1.31	1.17	1.03	.89	.75	.61								
13	1.66	1.38	1.24	1.10	.96	.82	.68	.54							
14	1.73	1.45	1.31	1.17	1.03	.89	.75	.61							
15	1.80	1.52	1.38	1.24	1.10	.96	.82	.68	.54						
16	1.87	1.59	1.45	1.31	1.17	1.03	.89	.75	.61						
17	1.94	1.66	1.52	1.38	1.24	1.10	.96	.82	.68	.54					
18	2.01	1.73	1.59	1.45	1.31	1.17	1.03	.89	.75	.61					
19	2.08	1.80	1.66	1.52	1.38	1.24	1.10	.96	.82	.68	.54				
20	2.15	1.87	1.73	1.59	1.45	1.31	1.17	1.03	.89	.75	.61				
21	2.22	1.94	1.80	1.66	1.52	1.38	1.24	1.10	.96	.82	.68	.54			
22	2.29	2.01	1.87	1.73	1.59	1.45	1.31	1.17	1.03	.89	.75	.61			
23	2.36	2.08	1.94	1.80	1.66	1.52	1.38	1.24	1.10	.96	.82	.68	.54		
24	2.43	2.15	2.01	1.87	1.73	1.59	1.45	1.31	1.17	1.03	.89	.75	.61		
25	2.50	2.22	2.08	1.94	1.80	1.66	1.52	1.38	1.24	1.10	.96	.82	.68	.54	.00

The program accepts as input C, the current balance; A, the average monthly balance; and N, the number of checks written. It then computes and prints the monthly service charge and new balance.

The program illustrates the use of logical operators and the DIV operator.

```
>LIST↵
100 ! CHECKING ACCOUNT PROGRAM
110 PRINT "THIS IS A PROGRAM TO COMPUTE THE MONTHLY SERVICE CHARGE"
120 PRINT "FOR A REGULAR CHECKING ACCOUNT AT A COMMERCIAL BANK."
130 PRINT FOR I= 1 TO 2
140 REAL M(15,0:26)
150 M(1,26)=2.50
160 M(15,26)=0
170 M(I,26)=2.50-(I*.14) FOR I= 2 TO 14
180 FOR I= 1 TO 15
190 FOR J=0 TO 25
200 M(I,J)=M(I,26)-((25-J)*.07)
210 NEXT J,I
220 PRINT "CURRENT BALANCE ":
230 INPUT C
240 PRINT "AVERAGE BALANCE ":
250 INPUT A
260 IF (C=0) AND (A=0) THEN STOP
270 X = A DIV 100
280 IF X<1 THEN X=1 ELSE IF X>15 THEN X=15
290 PRINT "NUMBER OF CHECKS THIS MONTH ":
300 INPUT Y
310 IF M(X,Y)>.54 THEN 360
320 M(X,Y)=0
330 IF (X=2 AND Y=0) OR (X=3 AND Y=2) OR (X=4 AND Y=4) THEN M(X,Y)=.47
340 IF (X=3 AND Y=1) OR (X=4 AND Y=3) THEN M(X,Y)=.40
350 IF (X=3 AND Y=0) OR (X=4 AND Y=2) THEN M(X,Y)=.33
360 PRINT
370 PRINT "THIS MONTH'S SERVICE CHARGE =":M(X,Y)
380 C=C-M(X,Y)
390 PRINT "THE NEW CURRENT BALANCE =":C
400 PRINT
410 GO TO 220
>RUN↵
THIS IS A PROGRAM TO COMPUTE THE MONTHLY SERVICE CHARGE
FOR A REGULAR CHECKING ACCOUNT AT A COMMERCIAL BANK.


CURRENT BALANCE ? 667.86↵
AVERAGE BALANCE ? 844.11↵
NUMBER OF CHECKS THIS MONTH ? 17↵

THIS MONTH'S SERVICE CHARGE = .82
THE NEW CURRENT BALANCE = 667.04

CURRENT BALANCE ? 0↵
AVERAGE BALANCE ? 0↵

>
```

PLOTTING

SUPER BASIC can be used to create plots of mathematical functions. The TAB function spaces to the appropriate position, and the plot symbol is placed there.

The program below plots the sine and cosine functions on the interval $[0,2\pi]$. If the values of the two functions are within .03 of each other, only the sine is plotted.

Note the use of the GOSUB statement in the IF...THEN...ELSE statement. In addition, each subroutine contains a GOSUB statement. These nested subroutines allow both function values to be plotted on the same line with the smaller value plotted first.

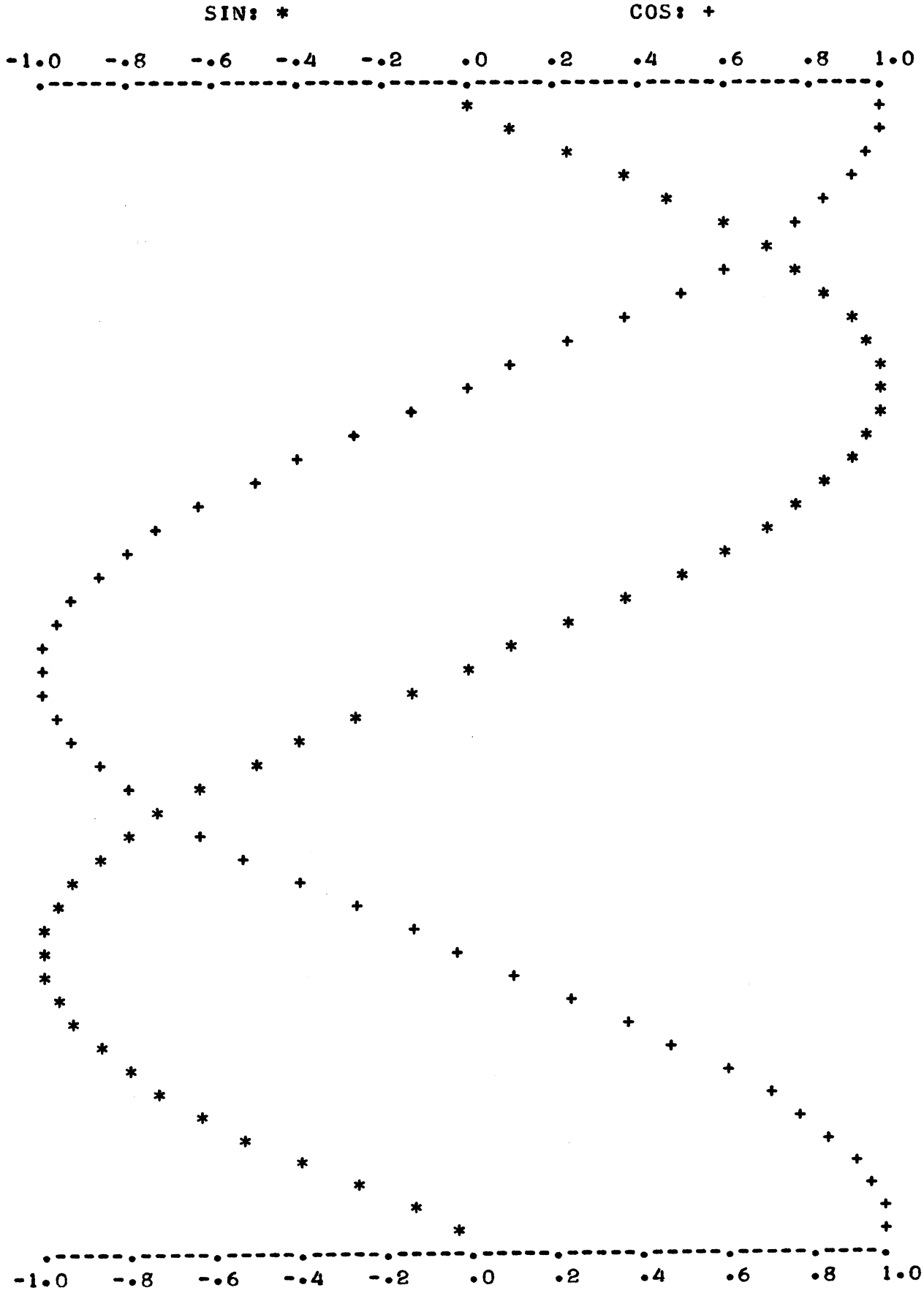
```
-COPY PLOT TO T↵

100 PRINT
110 PRINT TAB(23);"SIN; *";TAB(53);"COS; +"
120 PRINT
130 PRINT IN FORM "8B 11(%%.% 2B)/";I FOR I=-1 TO -.2 BY .2,0 TO 1 BY .2
140 PRINT "RADIANS    ";
150 PRINT ".-----"; FOR I= 1 TO 9
160 PRINT ".-----."
170 K,L=0
180 FOR I=0 TO 2*PI BY PI/24
190 PRINT IN FORM "%.%%%%";I
200 A=ABS(SIN(I)-COS(I))
210 IF A>.03 THEN GO TO 240
220 PRINT TAB(41+30*SIN(I));"*"
230 GO TO 260
240 IF SIN(I)<COS(I) THEN GOSUB 320 ELSE GOSUB 380
250 PRINT
260 NEXT I
270 PRINT TAB(11);
280 PRINT ".-----"; FOR I= 1 TO 9
290 PRINT ".-----."
300 PRINT IN FORM "8B 11(%%.% 2B)/";I FOR I=-1 TO -.2 BY .2,0 TO 1 BY .2
310 STOP
320 PRINT TAB(41+30*SIN(I));"*";
330 IF L=1 THEN 360
340 K=1
350 GOSUB 380
360 L=0
370 RETURN
380 PRINT TAB(41+30*COS(I));"+";
390 IF K=1 THEN 420
400 L=1
410 GOSUB 320
420 K=0
430 RETURN

-
```

```
-SBASIC ↵

>LOAD PLOT ↵
>RUN ↵
```




```
>
```

LEAST SQUARE LINE

This program fits a least square line of the form X=A+BX to a set of data where X is the independent and Y the dependent variable. The program accepts the number of data points, the matrix X of independent variable values, and the matrix Y of dependent variable values.

The regression coefficients A and B are calculated from the equations.

$$A = \frac{(\Sigma Y)(\Sigma X^2) - (\Sigma X)(\Sigma XY)}{N\Sigma X^2 - (\Sigma X)^2}$$

$$B = \frac{N\Sigma XY - (\Sigma X)(\Sigma Y)}{N\Sigma X^2 - (\Sigma X)^2}$$

Note the use of the MAT INPUT statements to read the X and Y values.

```
-SBASIC↵

>LOAD LSQ↵
>LIST↵
100 ! THIS PROGRAM FITS A LEAST SQUARE LINE OF THE FORM: Y=A+BX
110 ! TO A SET OF DATA (X,Y) WHERE X IS THE INDEPENDENT VARIABLE.
120 T1,T2,T3,T4=0
130 PRINT
140 PRINT "NUMBER OF DATA POINTS = ";
150 INPUT IN FORM "#":N
160 DIM X(N),Y(N),P(N),S(N)
170 PRINT "THE X VALUES ARE ";
180 MAT INPUT X
190 PRINT "THE Y VALUES ARE ";
200 MAT INPUT Y
210 ! LOOP TO CALCULATE TOTALS
220 FOR I=1 TO N
230 T1=T1+X(I)
240 T2=T2+Y(I)
250 T3=T3+X(I)*Y(I)
260 T4=T4+X(I)↑2
270 NEXT I
280 ! CALCULATION OF COEFFICIENTS A,B
290 D=N*T4-T1*T1
300 A=(T2*T4-T1*T3)/D
310 B=(N*T3-T1*T2)/D
320 PRINT
330 F="'THE LEAST SQUARE LINE IS: Y=' %%.%%% ' +' %%.%%% 'X'/"
340 PRINT IN FORM F:A,B
350 PRINT
360 PRINT "ANOTHER SET OF DATA ";
370 INPUT R
380 IF LEFT(R,1)="Y" THEN GO TO 120
 >RUN↵
```

```
NUMBER OF DATA POINTS = 8↵
THE X VALUES ARE ? 1,3,4,6,8,9,11,14↵
THE Y VALUES ARE ? 1,2,4,4,5,7,8,9↵

THE LEAST SQUARE LINE IS: Y=  .545 +  .636X

ANOTHER SET OF DATA ? YES↵

NUMBER OF DATA POINTS = 5↵
THE X VALUES ARE ? 1,2,0,7,-4↵
THE Y VALUES ARE ? -1,1,-3,11,-11↵

THE LEAST SQUARE LINE IS: Y=-3.000 + 2.000X

ANOTHER SET OF DATA ? NO↵

>
```

COPYING A FILE

This program copies a file and substitutes a Carriage Return for each semicolon in the file. This technique can be used within a program, thus eliminating the need for a command file.

Note the use of the string functions LEFT, LENGTH, INDEX, and SUBSTR. Line 210 sets C equal to the left n characters of TELETYPE, where n is the length of the name of the output file. This step allows the program to determine if the output file is the terminal, in which case certain steps are eliminated.

Lines 230–240 determine whether a file by the specified name is already in the user's directory by opening the file as a random file and calculating the size. If the size is greater than zero, the program prints OLD FILE and waits ten seconds. If the user types a character, the program requests a new output file name.

```
-COPY FILECOPY TO T↵

100 INTEGER K,I,J,N
110 STRING L,F2,F3,C,G
120 K=0
130 N=20
140 TEXT S(N):80
150 PRINT "COPY FROM FILE: ":
160 INPUT IN FORM 'R':F2
170 PRINT "TO FILE: ":
180 OPEN F2,INPUT,2
190 ON ENDFILE(2) GO TO 510
200 INPUT IN FORM 'R':F3
210 C=LEFT('TELETYPE',LENGTH(F3))
220 IF C=F3 THEN 330
230 OPEN F3,RANDOM OUTPUT,3
240 IF SIZE(3)=0 THEN 340
250 PRINT "OLD FILE":
260 WAIT(10)
270 IF NOT TEL THEN 320
280 INPUT G
290 CLOSE 3
300 PRINT "TO FILE: ":
310 GO TO 200
320 CLOSE 3
330 OPEN F3, OUTPUT,3
340 PRINT
350 PRINT DATE
360 INPUT FROM 2 IN FORM 'R':S(I) FOR I= 1 TO N
370 FOR I=1 TO N
380 J=INDEX(S(I),';'), L=LEFT(S(I),J-1), S(I)=SUBSTR(S(I),J+1)
390 IF J THEN PRINT ON 3:L ELSE 410
400 GO TO 380
410 PRINT ON 3:S(I)
420 NEXT I
430 PRINT "*": UNLESS C=F3
440 PRINT
450 GO TO 360 UNLESS K
460 CLOSE 2,3
470 PRINT "FILE COPIED"
480 PRINT "ANOTHER FILE ":
490 INPUT G
500 IF LEFT(G,1)="Y" THEN GO TO 130 ELSE STOP
510 K=1,N=I-1
520 GO TO 370

-
```

```
-COPY SEMIFILE TO T↵

FIRST;SECOND;THIRD;FOURTH
FIFTH;SIXTH
SEVENTH
EIGHTH;NINTH;TENTH

-SBASIC↵

>LOAD FILECOPY↵
>RUN↵
COPY FROM FILE: SEMIFILE↵
TO FILE: REWRITE↵
OLD FILEN↵
TO FILE: SEMIDEL↵

03/01  11:20
*
FILE COPIED
ANOTHER FILE ? YES↵
COPY FROM FILE: SEMIFILE↵
TO FILE: TE↵

03/01  11:21
FIRST
SECOND
THIRD
FOURTH
FIFTH
SIXTH
SEVENTH
EIGHTH
NINTH
TENTH

FILE COPIED
ANOTHER FILE ? NO↵

>
```

APPENDIX A
SUMMARY OF SUPER BASIC

All commands can be executed both directly and indirectly unless otherwise specified.

VARIABLES AND ARRAYS

VARIABLE NAMES

single letter
single letter followed by single digit
single letter followed by $
special variable, EPS, preset by SUPER BASIC to 10^{-10}, can be reset by user

SUBSCRIPTED VARIABLE (ARRAY) NAMES

single letter
single letter followed by $

VALUE TYPES

Type	May Be Declared As
Real Number Integer; e.g.,15,7 Decimal; e.g.,13.6,-.03 E Notation; e.g.,6E2 (E2 means times 10^2)	REAL INTEGER
Logical Value All variables with a numeric value have a logical value as well. TRUE if the numeric value $\neq 0$. FALSE if the numeric value = 0.	LOGICAL If declared, value returned is: 1 for TRUE 0 for FALSE
Complex Number *NOTE: The logical value of a complex variable is set to the logical value of its real part.*	COMPLEX Declaration is required.
Double Precision Number (up to 17 significant digits) D notation; e.g., 1.234567890123D-3	DOUBLE Declaration is required.
String Value Any combination of characters.	STRING TEXT (arrays only)

Octal constants, for example O05 and O42, are not acceptable in DATA statements or in reply to INPUT.

VARIABLE INITIALIZATION

Variables ordinarily are not initialized.

VAR = ZERO assumes undefined arithmetic variables to be zero, logical variables to be FALSE, and string variables to be the null string.

VAR = UNDEF nullifies VAR = ZERO.

See page 47 for array initialization.

DIM AND DECLARATION STATEMENTS

DIM reserves space for array elements which may be integer, real, or string. Defines no elements. Arrays with a subscript greater than 10 or with more than two dimensions (subscripts) require DIM.

DIM A(20),B(50,2) *Subscript base 1 is implied.*

DIM A(0:20),B(-6:6) *Base other than 1 is specified.*

BASE n specifies subscript base n; for example,

BASE 0

DIM C(2,4)

reserves space for a 3 by 5 matrix C.

Declaration statements declare simple variables and arrays. Arrays may be dimensioned in the declaration statement using the same form as DIM; for example,

INTEGER A(20),B(-6:6)

ASSIGNMENT STATEMENT

Assigns values to variables. LET is optional.

Examples

Real: **LET A = 6**
X,Y,Z = 0
B(1,5) = 3,K = C*N↑2

Complex: **A = CMPLX(6,X)** *A must be declared.*

String: **X = "DOUBLE"**
Y = 'SINGLE'

OPERATORS

Type	Operators		Operate On
Arithmetic	↑	exponentiation	Numeric variables and expressions. *NOTE: The exponentiation operator can be used to calculate fractional roots. For example, the fifth root of 29.3 is 29.3↑(1/5).*
	-	unary minus	
	MOD	modulo	
	*	multiplication	
	/	division	
	DIV	division (integer result)	
	+	addition	
	-	subtraction	
Relational	<	less than	String or numeric variables and expressions.
	<=	less than or equal to	
	=	equal to	
	>=	greater than or equal to	
	>	greater than	
	# *or* <>	not equal to	
	<<	very much less than	
	>>	very much greater than	
	=#	approximately equal to	
Logical	NOT		Relational expressions and logical values of numeric variables and expressions.
	AND		
	OR		
	XOR	exclusive OR	
	IMP	implication	
	EQV	equivalence	
String	+	concatenation	String variables and expressions.
Binary	BAN	binary AND	Integer variables or expressions.
	BOR	binary OR	
	BEX	binary exclusive OR	

PRECEDENCE OF OPERATORS

Operator	Precedence
Expressions in parentheses	1
Evaluation of functions	2
Exponentiation (↑)	3
Unary minus (-)	4
MOD, BAN, BOR, BEX	5
Multiplication and division (*, /, and DIV)	6
Addition and subtraction (+ and -)	7
Relational operators	8
NOT	9
AND	10
OR, XOR	11
IMP	12
EQV	13

FUNCTIONS

STANDARD FUNCTIONS	
Function	**Brief Description**
Mathematical Functions[1]	
ABS(X)	Absolute value of X.
ACOS(X)	Angle in radians whose cosine is X.
ASIN(X)	Angle in radians whose sine is X.
ATN(X) *or* ATAN(X)	Arctangent (in radians, over the range $-\pi/2$ to $+\pi/2$) of X.
ATN(Y,X) *or* ATAN(Y,X)	Arctangent (in radians, over the range $-\pi$ to $+\pi$) of Y/X.
COMP(X,Y)	Compares values of X and Y. (-1 if X<Y, 0 if X=Y, 1 if X>Y)
COS(X)	Cosine of X (X in radians).
COSH(X)	Hyperbolic cosine of X.
DET	Determinant of last matrix inverted.
EXP(X)	Natural exponential of X, e^X.
EXP2(X)	2^X
FIX(X)	X truncated: equal to SGN(X) * INT(ABS(X))
FP(X)	Fractional part of X: equal to X-IP(X).
INT(X) *or* IP(X)	Greatest integer less than or equal to X.
LOG(X)	Natural logarithm of X.
LOG2(X)	Base 2 logarithm of X.
LOG10(X) *or* LGT(X)	Base 10 logarithm of X.
MAX(X_1,X_2,...,X_n)	Returns value of largest argument.
MIN(X_1,X_2,...,X_n)	Returns value of smallest argument.
PDIF(X,Y)	Positive difference of X and Y, if X-Y>0; otherwise 0.
PI	Mathematical constant π.

STANDARD FUNCTIONS (Continued)	
Function	**Brief Description**
RND(X)	Random number generator. *NOTE: RND(0) may be typed as RND.*
ROUN(X)	Value of X rounded to nearest integer: equal to IP(X+.5).
SGN(X)	Sign function (1 for positive X, 0 for X = 0 and -1 for negative X).
SIN(X)	Sine of X (X in radians).
SINH(X)	Hyperbolic sine of X.
SQR(X) *or* SQRT(X)	Positive square root of X.
TAN(X)	Tangent of X (X in radians).
TANH(X)	Hyperbolic tangent of X.
String Functions	
S denotes string argument; N denotes numeric argument.	
ASC(S)	Returns ASCII code of as many as first 3 characters of S; for example, ASC("A") = 33.
CHAR(N)	Returns string character whose ASCII code is equal to N; for example, CHAR(33) = A.
COMP(S_1,S_2)	Compares S_1 and S_2. -1 if $S_1 < S_2$ 0 if $S_1 = S_2$ 1 if $S_1 > S_2$
INDEX(S_1,S_2)	Position of first occurrence of S_2 within S_1; for example, INDEX ("ABC", "B") = 2.
INDEX(S_1,S_2,N)	Position of S_2 within S_1; search begins at position N in S_1.
LEFT(S,N)	Substring of S; N characters, starting from the left.
LENGTH(S)	Length of string S.

1 - Unless otherwise specified, functions return values of the same type as their argument, e.g., IP(3/2) is real because the quotient 3/2 is real. Some functions, however, convert a logical or integer argument to real before computing the function value.

STANDARD FUNCTIONS (Continued)	
Function	**Brief Description**
MAX(S_1,S_2,...,S_n)	Returns value of greatest argument.
MIN(S_1,S_2,...,S_n)	Returns value of smallest argument.
RIGHT(S,N)	Substring of S; N characters, starting from the right.
SPACE(N)	String of N spaces.
STR(N)	String of the characters comprising N; for example, STR(3) = " 3".
SUBSTR(S,N_1)	Substring of S, from N_1 th character to end of S.
SUBSTR(S,N_1,N_2)	Substring of S; N_2 characters, starting from N_1 th character.
VAL(S)	Numeric value of S, where S must be a numeric string; for example VAL("-6") = -6.
Complex Functions	
CMPLX(X,Y)	Complex number with real part X and imaginary part Y.
CONJ(C)	Conjugate of the complex argument C.
IMAG(C)	Imaginary part of the complex argument C.
PHASE(C)	Phase in radians of complex argument C.
POLAR(X,Y)	Complex number with magnitude X and phase Y, in radians.
REAL(C)	Real part of the complex argument C.
Double Precision Functions	
DBL(X)	Double precision value of X.
DPI	Double precision π.

STANDARD FUNCTIONS (Continued)	
Function	**Brief Description**
Binary Functions	
V denotes integer variable or expression; N denotes shift count.	
LSH(V,N)	Logical left shift of N digits of binary equivalent of V.
RSH(V,N)	Logical right shift of N digits of binary equivalent of V.
Random File Functions	
LOC(N)	Current location of file number N.
POS(N)	Position within current record at which the next input to or output from file number N will be performed.
SIZE(N)	Size of file number N.
Print Functions	
POS	Position of print head (terminal output).
POS(N)	One greater than the number of characters since the last Carriage Return, or beginning of record in the case of fixed length random files. For binary files, POS(N) is word position.
TAB(X)	Tab to print position X on terminal (used with PRINT).
TAB(X,N)	Tab to position X on file N.
Utility Functions	
DATE	Date and time of day.
TIME	Reads internal clock of computer and returns integer value increased by 1 every 1/60 second.

PROGRAMMER DEFINED FUNCTIONS

DEF (indirect only) defines a function with name FN followed by a single letter; for example,

```
80 DEF FNS(X,Y) = 2*SIN(X)-FNA(2)
100 DEF FNK = 2.165*R↑2
```

A function may be defined in a subroutine by using DEF and GOSUB. For example,

```
10 DEF FNT(X,Y): GOSUB 400

400 IF X < 4 THEN 420
410 X = ABS(X)+Y
420 A = X-Y↑3+2
430 RETURN A
```

INPUT/OUTPUT STATEMENTS

FUNDAMENTAL INPUT/OUTPUT STATEMENTS				
Method Of Input Or Output	**Example**			
	Real	**Complex** *Must be declared*	**Strings**	
			Undeclared	**Declared**
INPUT prints ? , accepts input typed in reply.	INPUT A,B ? 4.5,6	INPUT A ? 11,-4.1	INPUT A ? "STRING" or 'STRING'	INPUT A ? STRING (but quotes must enclose strings with commas or leading spaces).
READ reads data from **DATA** statements. (DATA is indirect only.) **RESTORE** allows rereading from beginning of DATA statements.	10 READ A,B 50 DATA 4.5,6	10 READ A 50 DATA 11,-4.1	10 READ A 50 DATA "STRING" or 50 DATA 'STRING'	10 READ A 50 DATA STRING (but quotes must enclose strings with commas or leading spaces).
PRINT prints numbers, text, values of variables and expressions. PRINT zones: , normal (15 spaces) ; packed : concatenated	A = 6 PRINT A;A/2 6 3	A = CMPLX(2,3) PRINT "A=":A A = 2, 3	X = "XX",Y = 'YY' PRINT X,Y XX YY	

DATA FILE STATEMENTS[1]

Statement Model	Type Of File	Remarks
OPEN "file name" FOR {[SYMBOLIC *or* BINARY]} [INPUT *or* OUTPUT] AS FILE n	SEQUENTIAL	Four files may be open concurrently[2]. File number n may be zero or any positive numeric expression. The file name may be a string variable or expression. If the file name is a literal file name, it must be surrounded by double or single quote marks. For fixed length random files, the record length r must be enclosed in parentheses and must be a positive numeric expression. If the record length r is omitted, the file is a variable record length file.
OPEN "file name" FOR {[SYMBOLIC *or* BINARY]} RANDOM {(r)} [INPUT *or* OUTPUT *or* IO] AS FILE n	RANDOM	
Short Forms		
OPEN "file name", {[SYMBOLIC *or* BINARY]} [INPUT *or* OUTPUT] , n	SEQUENTIAL	
OPEN "file name", {[SYMBOLIC *or* BINARY]} RANDOM {(r)} [INPUT *or* OUTPUT *or* IO] , n	RANDOM	
INPUT FROM n {[IN FORM *or* IN IMAGE] s} :variable list	SEQUENTIAL	n is file number of data input file, I is the location at which input is to begin, and s is string constant, variable, or expression specifying the input format.
INPUT FROM n {AT I} {[IN FORM *or* IN IMAGE] s} :variable list	RANDOM	

1 - Optional items are enclosed in braces, { }.

2 - Including "TELETYPE" (or "TEL" or "T") to denote the terminal.

DATA FILE STATEMENTS (Continued)		
Statement Model	**Type Of File**	**Remarks**
[PRINT / *or* / WRITE] ON n {[IN FORM / *or* / IN IMAGE] s} :list of variables or expressions [PRINT / *or* / WRITE] ON n {AT l} {[IN FORM / *or* / IN IMAGE] s} :list of variables or expressions	SEQUENTIAL RANDOM	n is file number of data output file, l is location at which output is to begin, and s specifies output format. If output format is not given, the usual PRINT functions and zones apply.
LOCATE l ON n	RANDOM	Indicates that next file operation on file n is to be at location l.
ERASE n FROM l_1 TO l_2	RANDOM	n is file number, l_1 is first location to be erased, and l_2 is last location to be erased. Data may not be erased from random files opened for INPUT.
CLOSE n	SEQUENTIAL *and* RANDOM	Closes data file n. (Automatic after RUN, DELETE ALL, and return to EXECUTIVE.) May be given as CLOSE n_1, n_2, ...
CLOSE "file name"	SEQUENTIAL *and* RANDOM	Deletes the specified file from the user's file directory.
ON ENDFILE(n) GO TO line number	SEQUENTIAL *and* RANDOM	n is input file number. Causes transfer to specified line number when the end of the input file is encountered.

PICTURE FORMATTING

INPUT IN IMAGE string:
INPUT IN FORM string:
PRINT IN IMAGE string:
PRINT IN FORM string:
} **list of variables or expressions**

Format string contains field specification symbols. All specifications apply to input and output.[1]

Type Of Field	Field Specification		Remarks
	IN IMAGE or IN FORM	IN FORM	
FREE FORMAT	"#"	"#"	Accepts or prints any string or number up to 11 significant digits.
INTEGER	One or more % signs "%%%"	"3%"	Leading space for positive numbers can be suppressed when printing.
DECIMAL	One or more % signs with embedded decimal point. "%%%.%%"	"3%.2%"	IN FORM requires at least one % before the decimal point. Leading space for positive numbers can be suppressed when printing.
E NOTATION	"#######" "#.#####"	"7#" "#.5#"	When printing positive numbers, leading space cannot be suppressed.
STRING	"%%%%%%" "#######"	"6%" "7#"	Strings are left justified. Fixed length # field must be 7 or more characters long.
	"R"	"R"	May be used with IN FORM only.
FLOATING $ *and* * FIELDS	"$$$.$$" "***.**"	"3$.2$" "3*.2*"	IN IMAGE: fewer than four $'s or *'s will not be interpreted as a field specification. IN FORM: fields designated by $ and * do not print minus sign of negative numbers.
DESCRIPTIVE TEXT *and* SPACES	IN IMAGE: Characters other than above symbols will be printed directly (including spaces); for example, "X IS %%". IN FORM: Spaces are used to separate field specifications. Text to be printed is enclosed in single quote marks. Spaces may be denoted by B's; for example, "'X IS' %% 2B %%".		Note IMAGE exception above. Fewer than four $'s or four *'s will not be interpreted as field specification. IN FORM descriptive text will be enclosed in double quotes if format string is in single quotes.

1 - See pages 60–62 for tables of precise FORM characters.

Type Of Field	Field Specification (continued)		Remarks
	IN IMAGE or IN FORM	IN FORM	
END OF FORMAT	INPUT IN IMAGE: end of input IMAGE causes all characters up to the next Carriage Return to be skipped. INPUT IN FORM: end of input FORM does not seek Carriage Return, but starts rescan of form. PRINT IN IMAGE: Carriage Return supplied at end of image. PRINT IN FORM: Carriage Return denoted by /; for example, "%/".		

Field replication may be done in FORM formats.
FORM "3(2% B)" is equivalent to
"%% B%% B%% B"

Examples of picture formatting with data files.

```
PRINT ON 3 IN IMAGE S: A,B,C
INPUT FROM 2 AT 17 IN FORM "#": D
```

MATRIX STATEMENTS[1]

Name	Example	Remarks
Input		
MAT READ	MAT READ A,B,C, MAT READ K(15),L(-1:1,3)	Matrices are read in row order (i.e., second subscript varies more rapidly).
MAT INPUT	MAT INPUT A,B,C MAT INPUT R(2,3),S(0:M) MAT INPUT FROM 1:A(N+1)	
Output		
MAT PRINT	MAT PRINT A,B;C MAT PRINT ON 2:R;S; MAT WRITE ON 2:R;S;	May be formatted. Matrices are printed in row order.
Mathematical Operations		
Addition	MAT C=A+B	
Subtraction	MAT C=A-B	
Multiplication	MAT C=A*B	MAT A = A*B is illegal.
Scalar Multiplication	MAT C=(X-5)*A MAT C=A	
Transpose	MAT C=TRN(A)	MAT A=TRN(A) is illegal.
Inverse	MAT C=INV(A)	Square matrices only. Uses Gauss-Jordan method.
Determinant	A = DET	No argument. Returns determinant of last matrix inverted.
Matrix Initialization		
ZER	MAT C=ZER MAT C=ZER(M) MAT C=ZER(15,N)	Sets all elements to zero.
CON	MAT C=CON MAT C=CON(M) MAT C=CON(15,N)	Sets all elements to one.
IDN	MAT C=IDN MAT C=IDN(M,M)	Square matrices only. Sets identity matrix.

1 - Except for multiplication, TRN, INV, and IDN, all statements apply to both matrices and vectors.

CONTROL STATEMENTS

FOR and NEXT described in separate table.

Statement Model	Remarks
END	Not needed at end of program.
GO	Direct only.
GO TO line number	When used directly, retains all previous information.[1]
GOSUB line number	Be sure to isolate subroutine from main program.
IF logical expression THEN statement IF logical expression THEN statement ELSE statement	The statement after the THEN or ELSE clause can be any indirect statement except DATA, REM, or !.
ON numeric expression GO TO $line_1$, $line_2$,. . . ON numeric expression GOSUB $line_1$, $line_2$,. . .	Value of numeric expression will be truncated if not an integer.
PAUSE	Indirect only.
QUIT (or Q ↵ when used directly)	Also can be used indirectly.
RETURN	
RETURN expression	
START	Equivalent to GO TO first statement in program.
STOP	Equivalent to END.

1 - Except that it would reinitialize the reading of any DATA statements.

FOR and NEXT

These commands are indirect only.

Example of FOR	Remarks
30 FOR X = 1 TO 10	Implied step of 1.
30 FOR X = 1 TO 10 STEP 2	Step specified as 2.
30 FOR X = 10 TO 1 STEP -2	Negative step specified.
30 FOR X = 10 TO 1 BY -2 30 FOR X = 10 STEP -2 TO 1 30 FOR X = 10 BY -2 TO 1	Alternate forms of above example.
30 FOR X = 1,2,7,8	Values listed.
30 FOR X = 1,2,6 TO 18 STEP 3,50	Values and range listed.
30 FOR X = N*Q TO Q/3 STEP N	Variables used in defining range.
30 FOR X = 1 WHILE X<=Y 30 FOR X = 1 UNTIL X>Y 30 FOR A = 10 STEP 2 WHILE A<Y	WHILE and UNTIL used to specify final value. Change in Y within loop will alter final value.
30 FOR X = 1 TO Y	Change in Y within loop will not alter final value.

FOR statement is accompanied by NEXT.

80 NEXT X
80 NEXT I,J *Equivalent to* { *80 NEXT I*
81 NEXT J

STATEMENT MODIFIERS

Most direct statements and all indirect statements except DATA can be modified.

Modifier	Example	Effect Of Modifier
IF	INPUT N IF M=SQR(7)	INPUT N executed only if M equals SQR(7).
UNLESS	INPUT N UNLESS M=SQR(7)	INPUT N executed only if M does not equal SQR(7).
FOR	PRINT X↑2 FOR X=1 TO 10	Equivalent to: 10 FOR X=1 TO 10 20 PRINT X↑2 30 NEXT X FOR modifier takes the same forms as the FOR statement.
WHILE	X=2*X WHILE X<Y	X = 2*X executed repeatedly as long as X is less than Y.
UNTIL	X=2*X UNTIL X<Y	X = 2*X executed repeatedly as long as X is greater than or equal to Y.

WHILE and UNTIL also may be used with FOR to specify the final value.

ENTERING, LOADING, AND SAVING THE PROGRAM

Command	Example	Purpose
ENTER	>ENTER 100 BY 5 If BY increment is not given, assumed to be 10.	To enter the program statements from the terminal with line numbers prompted. Direct only.
LINK	>LINK FILE2 525 LINK "NEXTFILE"	To load program statements saved on a file. Deletes previous statements and begins execution. Direct or indirect. If used as indirect statement, literal file name must be enclosed in double or single quote marks.
LOAD	>LOAD A 1550 LOAD "THISFILE"	To load program statements saved on a file. Direct or indirect. If used as indirect statement, literal file name must be enclosed in double or single quote marks.
SAVE	>SAVE PROGFILE1 >SAVE XY, 1-15,30,70-100	To save all or part of program. Direct only.
SAVE BINARY	>SAVE BINARY ROOT	To save compiled program on a GO file. Direct only.

EDITING AND UTILITY COMMANDS

All of these commands are direct only except MAP, REM, and !.

Command	Example	Remarks
DELETE *or* DEL	>DELETE 10 >DEL 8-10,70 >DELETE ALL	DELETE ALL has the same effect as returning to EXEC and recalling SBASIC.
EDIT	>EDIT 25 >25 EDIT	The line to be edited will be printed out.
LIST	>LIST 25 >LIST 10,65-90 >LIST	LIST alone lists the entire program.
LOL	>LOL 120	Sets length of line on output to determine when new line is to begin. Length of line assumed to be 72 if LOL is not used.
MAP	>MAP >512 MAP	Prints table of storage allocation for SUPER BASIC program.
MODIFY *or* MOD	>MOD 10 >10 MOD	The line to be edited will not be printed out.
REM and !	>10 REM PRINT A >10!SUBROUTINE >55 A = A+1 !ADD 1	Only ! can append comments to statements (see the last example).
RENUMBER *or* REN	>RENUMBER 20,10,5 >REN AS 20 FROM 10 BY 5 >REN 20,10-95,5 >REN AS 20 FROM 10 TO 95 INC 5 >RENUMBER BY 5 >REN >REN 30 ADD 10 >REN FROM 30 TO 65 ADD 10 >RENUMBER 200-310 ADD -100	When omitted, first new line number is assumed to be 100, first old to be 0 (program is renumbered from the beginning), and increment to be 10.
TABS	>TABS 5,10,15,20	Tabs are initialized at 7, 15, and in steps of 5 from 15 on. I^C spaces to next tab stop.

1 - A list of editing control characters appears on page 106.

APPENDIX B

ALPHABETIC LIST OF ALL SUPER BASIC STATEMENTS AND CHARACTERISTICS

The following is an alphabetic list of all SUPER BASIC statements.

D – A direct statement
I – An indirect statement
B – Either a direct or an indirect statement
Y – Statement may be modified by statement modifiers
N – Statement may not be modified by statement modifiers

Statement	Statement Type	Modification Possible
BASE	B	Y
CLOSE	B	Y
COMPLEX	B	Y
DATA	I	N
DEF	I	Y
DELETE	D	N
DIM	B	Y
DOUBLE	B	Y
EDIT	D	N
END or STOP	B	Y
ERASE	B	Y
FOR	I	Y[1]
GO	D	N
GOSUB	B	Y
GO TO	B	Y
IF	B	N[2]
INPUT	B	Y
INTEGER	B	Y
LET (Assignment)	B	Y
LINK	B	Y
LIST	D	N
LOAD	D	N
LOL	D	N
LOCATE	B	Y
LOGICAL	B	Y
MAP	B	Y
MAT	B	Y
MODIFY	D	N
NEXT	I	Y
ON	B	Y
ON ENDFILE	B	Y
OPEN	B	Y
PAUSE	I	Y
PRINT	B	Y
QUIT	B	Y
READ	B	Y
REAL	B	Y
REM or !	B	N
RENUMBER	D	N
RESTORE	B	Y
RETURN	B	Y
RUN	D	N
SAVE	D	N
SAVE BINARY	D	N
START	D	N
STRING	B	Y
TABS	D	N
TAPE	D	N
TEXT	B	Y
VAR=UNDEF	B	Y
VAR=ZERO	B	Y
WRITE	B	Y

1 - But not by FOR.
2 - IF statement cannot be modified, but THEN or ELSE clause can be modified if the statement comprising the clause is modifiable.

APPENDIX C
THE EXECUTIVE

ENTERING THE SYSTEM

To gain access to the Tymshare system, the user must log in.

As soon as the computer has answered, place the telephone handset into the MARK V data modem, press the ORIGINATE button on the MARK V, and turn on your terminal. If your terminal is a Teletype Model 33 or 35, the system will ask that the identifying character D be typed. *NOTE: For the identification character for other terminals, consult your local Tymshare representative.* Now, log into the Tymshare system. See the *Tymshare EXECUTIVE Reference Manual* for details.

After the user logs in, the system will respond with a message and a dash. The dash indicates that the user is in the EXECUTIVE and may give an EXECUTIVE command. Calling a language is an EXECUTIVE function.

CALLING SUPER BASIC

To call SUPER BASIC, type the EXECUTIVE command

– SBASIC ↵

SUPER BASIC will reply with a > when it is ready to accept a command.

RETURNING TO SUPER BASIC

If for some reason you return to and work in the EXECUTIVE and then wish to continue from where you left off in SUPER BASIC, you can use the REENTER command. The program and data that you worked with in SUPER BASIC were not destroyed by the return to the EXECUTIVE. However, any files which were open when you left SUPER BASIC are closed when you return to the EXECUTIVE and remain closed if you use REENTER to return to SUPER BASIC.

Example

```
–SBASIC ↵
>
.                          The user types part of a
.                          SUPER BASIC program.
.
>QUIT ↵
–
.                          He does some work in the
.                          EXECUTIVE.
.
.
–REENTER ↵
SBASIC
>                          He continues to type the
                           program.
```

If the user had typed SBASIC↵ or called any other language instead of giving the REENTER command, all of his previous work would have been destroyed.

RULES FOR NAMING FILES

Files are designated by the name assigned by the user when he creates them. A file name can contain any combination of digits, letters, and @. In addition, a file name can contain a series of characters including any characters except Line Feed or Carriage Return if the series is enclosed in single quote marks or slashes. A series of characters which obeys the following rules is an acceptable file name.

Rule 1 A file name may contain any combination of the characters

0 through 9
A through Z
@

Rule 2 A file name can contain protected strings, that is, a series of characters in slashes or single quotes. Protected strings can contain any characters except Line Feed (J^C), Carriage Return (M^C), and the delimiting character itself (/ or '). *NOTE: To include a Control A in a file name in the EXECUTIVE, the A^C must be preceded by V^C.*

Rule 3 Certain reserved file names cannot be used:

TELETYPE	NOTHING
TELETYP	NOTHIN
TELETY	NOTHI
TELET	NOTH
TELE	NOT
TEL	NO
TE	N
T	
These names always designate a terminal.	*These names always designate a null file.*

Rule 4 A file name may contain a maximum of 45 characters.

THE COPY COMMAND

In the EXECUTIVE system, the COPY command is used to

- Create files.
- Print files on the terminal.
- Copy the contents of one file to another.

The general form of this command is

COPY source file TO destination file

The letter T may be used in place of a file name in the COPY command to indicate that the source or destination file is the user's terminal.

Examples

– COPY TRI TO T ↵

– COPY A1 TO B ↵

– COPY T TO SBPROG ↵

THE DECLARE COMMAND

The DECLARE command is used to set file security controls. These controls determine who can access the files, who can read them, and who can write on them. These controls provide a maximum of security.

DECLARE asks two sets of questions: PRIVATE and PUBLIC. These questions and the effect of the user's response are listed in the table below. The private controls refer to what the user himself can or cannot do to his file. The other file access controls refer to sharing files with other users in the same account.

A file with @ or a control character in its name is always public. It can be accessed by any user on the system if he knows the full name of the file. Therefore, to share a file without risking its exposure to everyone in the account, a user may include some nonprinting control characters in the file name and inform only those other users who are allowed to access the file. Since the control characters do not print, there will never be any written record of the complete file name to jeopardize its security.

The general form of the DECLARE command is

DECLARE file names

The user specifies the file or group of files which he wishes to declare. After the last file name, he types a Line Feed or Carriage Return. The EXECUTIVE then asks questions which can be answered Y or N followed by a Line Feed. (The WRITE ACCESS question can also be answered with an A.)

If the Y or N is followed by a Carriage Return rather than a Line Feed, DECLARE will skip the remaining questions and make no further changes in the condition of the file or files.

Example

SQU is a GO file belonging to user Jones (user name JONES). He declares this file to be proprietary as follows:

```
– DECLARE ↵
FILE(S): SQU ↵
PRIVATE:
 WRITE ACCESS? Y ↵
 READ ACCESS? Y ↵
 PUBLIC? Y ↵
PUBLIC:
 PROPRIETARY? Y ↵
 WRITE ACCESS? N ↵

–
```

The file may still be read, written on, or deleted by Jones. In addition, all the other users in the account may use it by typing

– GO (JONES)SQU ↵

No user can copy the file to his own directory.

EFFECT OF RESPONSES			
Question	**Y (for YES)**	**N (for NO)**	**A (for APPEND)**
PRIVATE: WRITE ACCESS?	The user may write on the file or delete it.	The file can only be read or opened for input; the user may neither write on it nor delete it. The next question is bypassed.	The user may add to the file, but he cannot write over existing information. Append-only files can be deleted.
READ ACCESS?	The user may read the file.	The user may not read the file. The file may not be opened for input or loaded into a language.	
PUBLIC?	Other users in the same account may access the file subject to the public controls.	Other users may not copy or use the file unless the file name contains a control character or @. All subsequent questions are bypassed.	
PUBLIC? PROPRIETARY?	The file can be executed by other users but cannot be listed or copied. It can be accessed only by the GO command if it is a GO file, or the RUN command if it is a dump file. Memory is cleared whenever a return to the EXECUTIVE is made.	No further limitation is placed on copying or using.	
WRITE ACCESS?	In addition to being able to copy and use the file, other users in the same account may write on it.	Other users in the account can copy or load this file, but may not write on it or delete it.	

LISTING FILE NAMES

When the EXECUTIVE command

– FILES ↵

is given, a complete listing of all your files will be printed, and the type of file will be indicated (SYM for symbolic, BIN for binary, DUM for dump, and GO for GO file).

Example

– FILES ↵

SYM MORTGAGE
SYM JUNK
SYM DATA
BIN BDATA
SYM VEN
SYM ABC

DELETING FILES

If there is no further use for a particular file, delete it by typing:

DELETE file name

Example

– DELETE ABC ↵
–

A single DELETE command may be used to delete more than one file. The file names must be separated by commas as follows:

– DELETE PGM,JUNK,VEN ↵
–

LEAVING THE SYSTEM

To exit from the Tymshare system, you first must be in the EXECUTIVE. To return to the EXECUTIVE from SUPER BASIC, type:

> QUIT ↵

or

> Q ↵

After the EXECUTIVE dash appears, type:

– LOGOUT ↵

The system will then type

CPU TIME: n SECS.

Number of computing seconds used.

TERMINAL TIME: 0:00:00

Number of minutes connected.

When the computer types

PLEASE LOG IN:

you may disconnect the line or let another user log in.

INDEX

NOTE: Page numbers which appear in **bold** *face type refer to those pages where the listed item receives the most detailed discussion.*