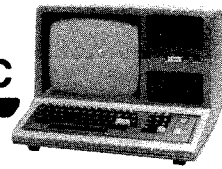


## Section 2: BASIC Language

```
960 FOR K = 1 TO N3
970 FOR J = 1 TO N2
980 FOR I = 1 TO N1
990 C(I,J,K) = A(I,J,K)
1000 NEXT I
1010 NEXT J
1020 NEXT K
1030 END
```

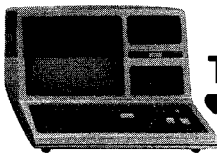


# 1 / BASIC Concepts

*This chapter gives an in-depth description of how to use the full power of Model III BASIC. Programmers require this information in order to build powerful and efficient programs. However, if you are still somewhat of a novice, you might want to skip this chapter for now, keeping in mind that the information is here when you need it.*

This chapter is divided into four sections:

- 1. Overview — Elements of a Program.** This section defines many of the terms we will be using in the chapter.
- 2. How BASIC Handles Data.** Here we discuss how BASIC classifies and stores data. This will show you how to get BASIC to store your data in its most efficient format.
- 3. How BASIC Manipulates Data.** This will give you an overview of all the different operators and functions you can use to manipulate and test your data.
- 4. How to Construct an Expression.** Understanding this topic will help you form powerful statements instead of using many short ones.



# Overview — Elements of a Program

This overview defines the elements of a program:

The **program** itself, which consists of . . .

Statements, which may consist of . . .

Expressions

We will refer to these terms during the rest of this chapter.

## Program

A program is made up of one or more numbered lines. Each line contains one or more BASIC statements. BASIC allows line numbers from 0 to 65529 inclusive. You may include up to 255\* characters per line, including the line number. You may also have two or more statements to a line, separated by colons.

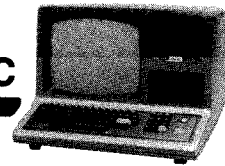
\*You can only type in 240 characters for new lines; using the Edit Mode, you can add the extra 15 characters.

Here is a sample program:

Line number    BASIC statement    Colon between statements    BASIC statement

```
100 CLS: PRINT "NORMAL MODE..."
110 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
120 FOR I = 1 TO 1000: NEXT I
130 CLS: PRINT CHR$(23); "DOUBLE-SIZE MODE..."
140 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
150 END
```

When BASIC executes a program, it handles the statements one at a time, starting at the first and proceeding to the last. Some statements, such as GOTO, ON . . . GOTO, GOSUB, change this sequence.



## Statements

A statement is a complete instruction to BASIC, telling the Computer to perform specific operations. For example:

```
GOTO 100
```

Tells the Computer to perform the operations of (1) locating line 100 and (2) executing the statement on that line.

```
END
```

Tells the Computer to perform the operation of ending execution of the program.

Many statements instruct the computer to perform operations with data. For example, in the statement:

```
PRINT "SEPTEMBER REPORT"
```

the data is SEPTEMBER REPORT. The statement instructs the Computer to print the data inside the quotes.

## Expressions

An expression is actually a general term for data. There are four types of expressions:

**1. Numeric expressions**, which are composed of numeric data. Examples:

```
(1 + 5.2) / 3
```

```
D
```

```
5 * B
```

```
3.7682
```

```
ABS(X) + RND(0)
```

```
SIN(3 + E)
```

**2. String expressions**, which are composed of character data. Examples:

```
A$
```

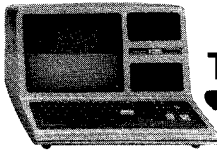
```
"STRING"
```

```
"STRING" + "DATA"
```

```
MO$ + "DATA"
```

```
MID$(A$,2,5) + MID$("MAN",1,2)
```

```
M$ + A$ + B$
```



### 3. Relational expressions, which test the relationship between two expressions.

Examples:

```
A = 1  
A$ > B$
```

### 4. Logical expressions, which test the logical relationship between two expressions. Examples:

```
A$ = "YES" AND B$ = "NO"  
C > 5 OR M < B OR O > 2  
578 AND 452
```

## Functions

Functions are automatic subroutines. Most BASIC functions perform computations on data. Some serve a special purpose such as controlling the video display or providing data on the status of the computer. You may use functions in the same manner that you use any data — as part of a statement.

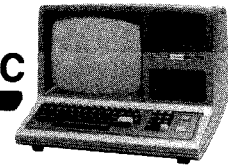
These are some of BASIC's functions:

```
INT  
ABS  
STRING$
```

## How Basic Handles Data

Model III BASIC offers several different methods of handling your data. Using these methods properly can greatly improve the efficiency of your program. In this section we will discuss:

1. Ways of Representing Data
  - a. Constants
  - b. Variables
2. How BASIC Stores Data
  - a. Numeric (integer, single precision, double precision)
  - b. String
3. How BASIC Classifies Constants
4. How BASIC Classifies Variables
5. How BASIC Converts Data



## Ways of Representing Data

BASIC recognizes data in two forms — either directly, as constants, or by reference to a memory location, as variables.

### Constants

All data is input into a program as “constants” — values which are not subject to change. For example, the statement:

```
PRINT "1 PLUS 1 EQUALS"; 2
```

contains one string constant,

```
1 PLUS 1 EQUALS
```

and one numeric constant

```
2
```

In these examples, the constants “input” to the PRINT statement. They tell PRINT what data to print on the Display.

These are more examples of constants:

3.14159	"L. O. SMITH"
1.775E + 3	"0123456789ABCDEF"
"NAME TITLE"	-123.45E-8
57	"AGE"

### Variables

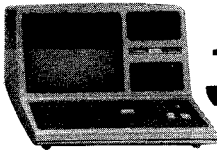
A variable is a place in memory — a sort of box or pigeonhole — where data is stored. Unlike a constant, a variable’s value can change. This allows you to write programs dealing with changing quantities. For example, in the statement:

```
A$ = "OCCUPATION"
```

The variable A\$ now contains the data OCCUPATION. However, if this statement appeared later in the program:

```
A$ = "FINANCE"
```

The variable A\$ would no longer contain OCCUPATION. It would now contain the data FINANCE.



## Variable Names

In BASIC, variables are represented by names. Variable names must begin with a letter, A through Z. This letter may be followed by one more character — either a digit or a letter.

For example

AM            A                    A1        B1            AB

are all valid and distinct variable names.

Variable names may be longer than two characters. However, only the first two characters are significant in BASIC.

For example:

SUM        SU        SUPERNUMERARY

are all treated as the same variable by BASIC.

## Reserved Words

Certain combinations of letters are reserved as BASIC keywords, and cannot be used in variable names. For example:

OR        LAND        LENGTH        MIFFED

cannot be used as variable names, because they contain the reserved of OR, AND, LEN, and IF, respectively.

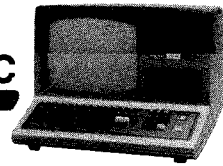
See the Appendix for a list of reserved words.

## Simple and Subscripted Variables

All of the variables mentioned above are simple variables. They can only refer to one data item.

Variables may also be subscripted so that an entire list of data can be stored under one variable name. This method of data storage is called an **array**. For example, an array named A may contain these elements (subscripted variables):

A(0)        A(1)        A(2)        A(3)        A(4)



You may use each of these elements to store a separate data item, such as:

```
A(0) = 5.3
A(1) = 7.2
A(2) = 8.3
A(3) = 6.8
A(4) = 3.7
```

In this example, array A is a one-dimensional array, since each element contains only one subscript. An array may also be two-dimensional, with each element containing two subscripts. For example, a two-dimensional array named X could contain these elements:

```
X(0,0) = 8.6          X(0,1) = 3.5
X(1,0) = 7.3          X(1,1) = 32.6
```

With BASIC, you may have as many dimensions in your array as you would like. Here is an example of a three-dimensional array named L which contains these 8 elements:

```
L(0,0,0) = 35233      L(0,1,0) = 96522
L(0,0,1) = 52000      L(0,1,1) = 10255
L(1,0,0) = 33333      L(1,1,0) = 96253
L(1,0,1) = 53853      L(1,1,1) = 79654
```

BASIC assumes that all arrays contain 11 elements in each dimension. If you want more elements you must use the DIM statement at the beginning of your program to dimension the array.

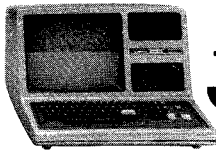
For example, to dimension array L, put this line at the beginning of the program:

```
DIM L(1, 1, 1)
```

to allow room for two elements in the first dimension; two in the second; and two in the third for a total of  $2 * 2 * 2 = 8$  elements.

See the Arrays chapter later on in this manual.





## How BASIC Stores Data

The way that BASIC stores data determines the amount of memory it will consume and the speed in which BASIC can process it.

### Numeric Data

You may get BASIC to store all numbers in your program as either integer, single precision, or double precision. In deciding how to get BASIC to store your numeric data, remember the tradeoffs. Integers are the most efficient and the least precise. Double precision is the most precise and least efficient.

#### Integers

**(Speed and Efficiency, Limited Range)**

To be stored as an integer, a number must be whole and in the range of  $-32768$  to  $32767$ . An integer value requires only two bytes of memory for storage. Arithmetic operations are faster when both operands are integers.

For example:

1          32000          -2          500          -12345

can all be stored as integers.

#### Single-Precision Type

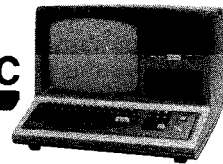
**(General Purpose, Full Numeric Range)**

Single-precision numbers can include up to 7 significant digits, and can represent normalized values\* with exponents up to  $\pm 38$ , i.e., numbers in the range:

$[-1 \times 10^{38}, -1 \times 10^{-38}] [1 \times 10^{-38}, 1 \times 10^{38}]$

A single-precision value requires 4 bytes of memory for storage. BASIC assumes a number is single-precision if you do not specify the level of precision.

\*In this reference manual, normalized value is one in which exactly one digit appears to the left of the decimal point. For example, 12.3 expressed in normalized form is  $1.23 \times 10$ .



For example:

10.001      -200034      1.774E6      6.024E-23      123.4567

can all be stored as single-precision values.

**Note:** When used in a decimal number, the symbol E stands for “single-precision times 10 to the power of...” Therefore 6.024E-23 represents the single-precision value:

$$6.024 \times 10^{-23}$$

### **Double-Precision Type (Maximum Precision, Slowest in Computations)**

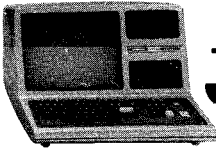
Double-precision numbers can include up to 17 significant digits, and can represent values in the same range as that for single-precision numbers. A double-precision value requires 8 bytes of memory for storage. Arithmetic operations involving at least one double-precision number are slower than the same operations when all operands are single-precision or integer.

For example:

1010234578  
-8.7777651010  
3.1415926535897932  
8.00100708D12

can all be stored as double-precision values.

**Note:** When used in a decimal number, the symbol D stands for “double-precision times 10 to the power of...” Therefore 8.00100708 D12 represents the value  $8.00100708 \times 10^{12}$



## String Data

Strings (sequences of characters) are useful for storing non-numeric information such as names, addresses, text, etc. You may store any ASCII characters as a string. (A list of ASCII characters is in the Appendix).

For example, the data constant:

Jack Brown, Age 38

can be stored as a string of 18 characters. Each character (and blank) in the string is stored as an ASCII code, requiring one byte of storage. BASIC would store the above string constant internally as:

Hex Code	4A	61	63	6B	20	42	72	6F	77	6E	2C	20	41	67	65	20	33	38
ASCII Character	J	a	c	k		B	r	o	w	n	,		A	g	e		3	8

A string can be up to 255 characters long. Strings with length zero are called “null” or “empty”.

## How BASIC Classifies Constants

When BASIC encounters a data constant in a statement, it must determine the type of the constant: string, integer, single precision, or double precision. First, we will list the rules BASIC uses to classify the constant. Then we will show you how you can override these rules, if you want a constant stored differently:

### Rule 1

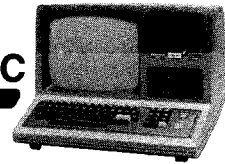
If the value is enclosed in double-quotes, it is a string. For example:

“YES”  
“3331 Waverly Way”  
“1234567890”

the values in quotes are automatically classified as strings.

### Rule 2

If the value is not in quotes, it is a number. (An exception to this rule is during data input by an operator, and in DATA lists. See INPUT, INKEY\$, and DATA)



For example:

123001  
1  
-7.3214E+6

are all numeric data.

### **Rule 3**

Whole numbers in the range of  $-32768$  to  $32767$  are integers. For example:

12350  
-12  
10012

are integer constants.

### **Rule 4**

If the number is not an integer and contains seven or fewer digits, it is single-precision. For example:

1234567  
-1.23  
1.3321

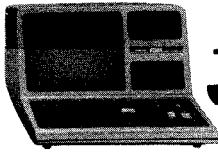
are all single-precision.

### **Rule 5**

If the number contains more than seven digits, it is double precision. For example, these numbers:

1234567890123456  
-100000000000.1  
2.777000321

are all double precision.



## Type Declaration Tags

You can override BASIC's normal typing criteria by adding the following "tags" to the end of the numeric constant:

**!** Makes the number single-precision. For example, in the statement:

```
A = 12.345678901234!
```

the constant is classified as single-precision, and shortened to seven digits:  
12.34567

**E** Single-precision exponential format. The E indicates the constant is to be multiplied by a specified power of 10. For example:

```
A = 1.2E5
```

stores the single-precision number 120000 in A.

**#** Makes the number double-precision. For example, in statement:

```
PRINT 3#/7
```

the first constant is classified as double-precision before the division takes place.

**D** Double-precision exponential format. The D indicates the constant is to be multiplied by a specified power of 10. For example:

```
A = 1.23456789D - 1
```

The double-precision constant has the value 0.123456789.

## How BASIC Classifies Variables

When BASIC encounters a variable name in the program, it classifies it as either a string, integer, single- or double-precision number.

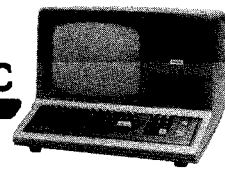
BASIC classifies all variable names as single-precision initially. For example:

```
AB      AMOUNT      XY      L
```

are all single-precision initially. If this is the first line of your program:

```
LP = 1.2
```

BASIC will classify LP as a single-precision variable.



However, you may assign different attributes to variables by using definition statements at the beginning of your program:

- DEFINT — Defines variables as integer
- DEFDBL — Defines variables as double-precision
- DEFSTR — Defines variables as string
- DEFSNG — Defines variables as single-precision. (Since BASIC classifies all variables as single-precision initially anyway, you would only need to use DEFSNG if one of the other DEF statements were used.)

For example:

```
DEFSTR L
```

makes BASIC classify all variables which start with L as string variables. After this statement, the variables:

```
L      LP      LAST
```

can all hold string values only.

### Type Declaration Tags

As with constants, you can always override the type of a variable name by adding a type declaration tag at the end. There are four type declaration tags for variables:

%	Integer
!	Single-precision
#	Double-precision n
\$	String

For example:

```
I%      FT%      NUM%      COUNTER%
```

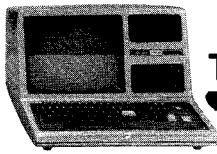
are all integer variables, **regardless** of what attributes have been assigned to the letters I, F, N and C.

```
T!      RY!      QUAN!      PERCENT!
```

are all single-precision variables, **regardless** of what attributes have been assigned to the letters T, R, Q and P.

```
X#      RR#      PREV#      LSTNUM#
```

are all double-precision variables, **regardless** of what attributes have been assigned to the letters X, R, P and L.



## TRS-80 MODEL III

---

Q\$      CA\$      WRD\$      ENTRY\$

are all string variables, **regardless** of what attributes have been assigned to the letters Q, C, W and E.

Note that any given variable name can represent four different variables. For example:

A5#      A5!      A5%      A5\$

are all valid and **distinct** variable names.

**One further implication of type declaration:** Any variable name used without a tag is equivalent to the same variable name used with one of the four tags. For example, after the statement:

```
DEFSTR C
```

the variable referenced by the name C1 is identical to the variable referenced by the name C1\$.

## How BASIC Converts Numeric Data

Often your program might ask BASIC to assign one type of constant to a different type of variable. For example:

```
A% = 2.34
```

In this example, BASIC must first convert the single precision constant 2.34 to an integer in order to assign it to the integer variable A%.

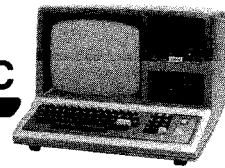
You might also want to convert one type of variable to a different type, such as:

```
A# = A%
```

```
A! = A#
```

```
A! = A%
```

The conversion procedures are listed on the following pages.



## Single- or double-precision to integer type

BASIC returns the largest integer that is not greater than the original value.

**Note:** The original value must be greater than or equal to -32768, and less than 32768.

### Examples

$A\% = -10.5$

Assigns A% the value -11.

$A\% = 32767.9$

Assigns A% the value 32767.

$A\% = 2.5D3$

Assigns A% the value 2500.

$A\% = -123.45678901234578$

Assigns A% the value -124.

$A\% = -32768.1$

Produces an Overflow Error (out of integer range).

## Integer to single- or double-precision

No error is introduced. The converted value looks like the original value with zeros to the right of the decimal place.

### Examples

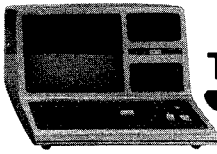
$A\# = 32767$

Stores 32767.000000000000 in A#.

$A\# = -1234$

Stores -1234.000 in A#.





## Double- to single-precision

This involves converting a number with up to 17 significant digits into a number with no more than seven. BASIC chops off (truncates) the least significant digits to produce a seven-digit number. Before Printing such a number, BASIC rounds it off (4/5 rounding) to six digits.

### Examples

```
A! = 1.234567890124567
```

Stores 1.234567 in A! However, the statement:

```
PRINT A!
```

will display the value 1.23457, because only six digits are displayed. The full seven digits are stored in memory.

```
A! = 1.3333333333333333
```

Stores 1.333333 in A!.

## Single- to double-precision

To make this conversion, BASIC simply adds trailing zeros to the single-precision number. If the original value has an exact binary representation in single-precision format, no error will be introduced. For example:

```
A# = 1.5
```

Stores 1.5000000000000 in A#, since 1.5 *does* have an exact binary representation.

However, for numbers which have no exact binary representation, an error is introduced when zeros are added. For example:

```
A# = 1.3
```

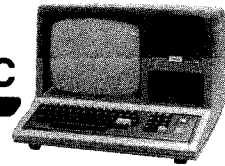
Stores 1.299999952316284 in A#.

Because most fractional numbers do not have an exact binary representation, you should keep such conversions out of your programs. For example, whenever you assign a constant value to a double-precision variable, you can force the constant to be double-precision:

```
A# = 1.3#      A# = 1.3D
```

Both store 1.3 in A#.

**Here is a special technique** for converting single-precision to double-precision, without introducing an error into the double-precision value. It is useful when the single-precision value is stored in a variable.



Take the single-precision variable, convert it to a string with `STR$`, then convert the resultant string back into a number with `VAL`. That is, use:

```
VAL (STR$ (single-precision variable))
```

For example, the following program:

```
10 A! = 1.3
20 A# = A!
30 PRINT A#
```

prints a value of:

```
1.299999952316284
```

Compare with this program:

```
10 A! = 1.3
20 A# = VAL (STR$(A!))
30 PRINT A#
```

which prints a value of:

```
1.3
```

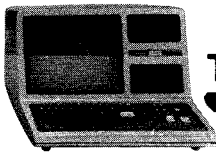
The conversion in line 20 causes the value in `A!` to be stored accurately in double-precision variable `A#`.

## Illegal Conversions

BASIC cannot automatically convert numeric values to string, or vice versa. For example, the statements:

```
A$ = 1234
A% = "1234"
```

are illegal. (Use `STR$` and `VAL` to accomplish such conversions.)



## How BASIC Manipulates Data

You have many fast methods you may use to get BASIC to count, sort, test and rearrange your data. These methods fall into two categories:

1. Operators
  - a. numeric
  - b. string
  - c. relational
  - d. logical
  
2. Functions

### Operators

An operator is the single symbol or word which signifies some action to be taken on either one or two specified values referred to as operands.

In general, an operator is used like this:

*operand-1 operator operand-2*

*operand-1* and *-2* can be expressions. A few operations take only one operand, and are used like this:

*operator operand*

Examples:

$6 + 2$

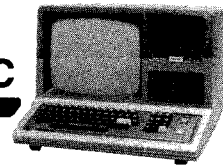
The addition operator  $+$  connects or relates its two operands, 6 and 2, to produce the result 8.

$-5$

The negation operator  $-$  acts on a single operand 5 to produce the result negative 5.

Neither  $6 + 2$  or  $-5$  can stand alone; they must be used in statements to be meaningful to BASIC. For example:

```
A = 6 + 2  
PRINT -5
```



Operators fall into four categories:

- Numeric
- String
- Relational
- Logical

based on the kinds of operands they require and the results they produce.

## Numeric Operators

Numeric Operators are used in numeric expressions. Their operands must always be numeric, and the result they produce is one numeric data item.

In the descriptions below, we use the terms integer, single-precision, and double-precision operations. Integer operations involve two-byte operands, single-precision operations involve four-byte operands, and double-precision operations involve eight-byte operands. The more bytes involved, the slower the operation.

There are five different numeric operators. Two of them, sign + and sign -, are unary, that is, they have only one operand. A sign operator has no effect on the precision of its operand.

For example, in the statement:

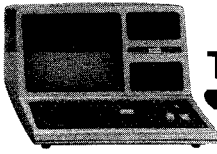
```
PRINT - 77, + 77
```

the sign operators - and + produce the values negative 77 and positive 77, respectively.

**Note:** When no sign operator appears in front of a numeric term, + is assumed.

The other numeric operators are all binary, that is, they all take two operands. These operators are:

- |                 |  |
|-----------------|--|
| +               | Addition   |
| -               | Subtraction  |
| *               | Multiplication   |
| /               | Division   |
| [ or $\uparrow$ | Exponentiation. Press the $\uparrow$ key to type in this operator. |



## Addition

The + operator is the symbol for addition. The addition is done with the precision of the more precise operand (the less precise operand is converted).

For example, when one operand is integer type and the other is single-precision, the integer is converted to single-precision and four-byte addition is done. When one operand is single-precision and the other is double-precision, the single-precision number is converted to double-precision and eight-byte addition is done.

Examples:

```
PRINT 2 + 3
```

Integer addition.

```
PRINT 3.1 + 3
```

Single-precision addition.

```
PRINT 1.2345678901234567 + 1
```

Double-precision addition.

## Subtraction

The - operator is the symbol for subtraction. As with addition, the operation is done with the precision of the more precise operand (the less precise operand is converted).

Examples:

```
PRINT 33 - 11
```

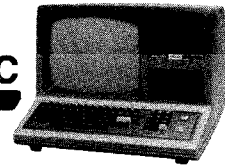
Integer subtraction.

```
PRINT 33 - 11.1
```

Single-precision subtraction.

```
PRINT 12.345678901234567 - 11
```

Double-precision subtraction.



## Multiplication

The \* operator is the symbol for multiplication. Once again, the operation is done with the precision of the more precise operand (the less precise operand is converted).

Examples:

```
PRINT 33 * 11
```

Integer multiplication.

```
PRINT 33 * 11.1
```

Single-precision multiplication.

```
PRINT 12.345678901234567 * 11
```

Double-precision multiplication.

## Division

The / symbol is used to indicate ordinary division. Both operands are converted to single or double-precision, depending on their original precision:

- If either operand is double-precision, then both are converted to double-precision and eight-byte division is performed.
- If neither operand is double-precision, then both are converted to single-precision and four-byte division is performed.

Examples:

```
PRINT 3/4
```

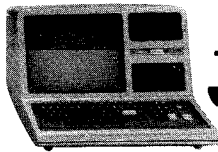
Single-precision division.

```
PRINT 3.8/4
```

Single-Precision division.

```
PRINT 3/1.2345678901234567
```

Double-precision division.



## Exponentiation

The symbol `[` denotes exponentiation. It converts both its operands to single-precision, and returns a single-precision result.

**Note:** To enter the `[` operator, press  $\text{\textcircled{4}}$ .

For example:

```
PRINT 6 [.3
```

prints 6 to the .3 power.

## String Operator

BASIC has a string operator ( `+` ) which allows you to concatenate (link) two strings into one. This operator should be used as part of a string expression. The operands are both strings and the resulting value is one piece of string data.

The `+` operator links the string on the right of the sign to the string on the left. For example:

```
PRINT "CATS" + "LOVE" + "MICE"
```

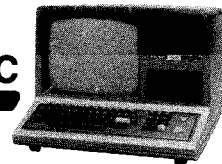
prints:

```
CATSLOVEMICE
```

Since BASIC does not allow one string to be longer than 255 characters, you will get an error if your resulting string is too long.

## Relational Operators

Relational operators compare two numerical or two string expressions to form a relational expression. This expression reports whether the comparison you set up in your program is true or false. It will return a `-1` if the relation is true; a `0` if it is false.



## Numeric Relations

This is the meaning of the operators when you use them to compare numeric expressions:

<	Less than
>	Greater than
=	Equal to
<> or ><	Not equal to
=< or <=	Less than or equal to
=> or >=	Greater than or equal to

Examples of true relational expressions:

```
1 < 2
2 <> 5
2 <= 5
2 <= 2
5 > 2
7 = 7
```

## String Relations

The relational operators for string expressions are the same as above, although their meanings are slightly different. Instead of comparing numerical magnitudes, the operators compare their ASCII sequence. This allows you to sort string data:

<	Precedes
>	Follows
=	Has the same precedence
>< or <>	Does not have the same precedence
<=	Precedes or has the same precedence
>=	Follows or has the same precedence

BASIC compares the string expressions on a character-by-character basis. When it finds a non-matching character, it checks to see which character has the lower ASCII code. The character with the lower ASCII code is the smaller (precedent) of the two strings.

**Note:** The appendix contains a listing of ASCII codes for each character.

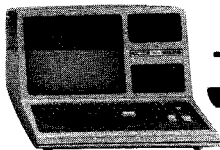
Examples of true relational expressions:

```
"A" < "B"
```

The ASCII code for A is decimal 65; for B it's 66.

```
"CODE" < "COOL"
```





## TRS-80 MODEL III

---

The ASCII code for O is 79; for D it's 68.

If while making the comparison, BASIC reaches the end of one string before finding non-matching characters, the shorter string is the precedent. For example:

```
"TRAIL" < "TRAILER"
```

Leading and trailing blanks are significant. For example:

```
" A" < "A"
```

ASCII for the space character is 32; for A, it's 65.

```
"Z-80" < "Z-80A"
```

The string on the left is four characters long; the string on the right is five.

### How to Use Relational Expressions

Normally, relational expressions are used as the test in an IF/THEN statement. For example:

```
IF A = 1 THEN PRINT "CORRECT"
```

BASIC tests to see if A is equal to 1. If it is, BASIC prints the message.

```
IF A$ < B$ THEN 50
```

If string A\$ alphabetically precedes string B\$, then the program branches to line 50.

```
IF R$ = "YES" THEN PRINT A$
```

If R\$ equals YES then the message stored as A\$ is printed.

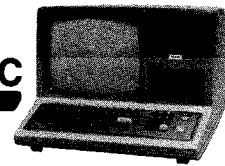
However, you may also use relational expressions simply to return the true or false results of a test. For example:

```
PRINT 7 = 7
```

Prints - 1 since the relation tested is true.

```
PRINT "A" > "B"
```

Prints 0 because the relation tested is false.



## Logical Operators

Logical operators make logical comparisons. Normally, they are used in IF/THEN statements to make a logical test between two or more relations. For example:

```
IF A = 1    OR  C = 2    THEN PRINT X
```

The logical operator, OR, compares the two relations  $A = 1$  and  $C = 2$ .

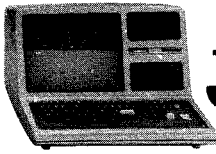
Logical operators may also be used to make bit-comparisons of two numeric expressions.

For this application, BASIC does a bit-by-bit comparison of the two operands, according to predefined rules for the specific operator.

**Note:** The operands are converted to integer type, stored internally as 16-bit, two's complement numbers. To understand the results of bit-by-bit comparisons, you need to keep this in mind.

The following table summarizes the action of Boolean operators in bit manipulation.

Operator	Meaning of Operation	First Operand	Second Operand	Result
AND	When both bits are 1, the result will be 1. Otherwise, the result will be 0.	1	1	1
		1	0	0
		0	1	0
		0	0	0
OR	Result will be 1 unless both bits are 0.	1	1	1
		1	0	1
		0	1	1
		0	0	0
NOT	Result is opposite of bit.	1		0
		0		1



## Hierarchy of Operators

When your expressions have multiple operators, BASIC performs the operations according to a well-defined hierarchy, so that results are always predictable.

### Parentheses

When a complex expression includes parentheses, BASIC always evaluates the expressions inside the parentheses before evaluating the rest of the expression. For example, the expression:

$$8 - (3 - 2)$$

is evaluated like this:

$$\begin{aligned} 3 - 2 &= 1 \\ 8 - 1 &= 7 \end{aligned}$$

With nested parentheses, BASIC starts evaluating the innermost level first and works outward. For example:

$$4 * (2 - (3 - 4))$$

is evaluated like this:

$$\begin{aligned} 3 - 4 &= -1 \\ 2 - (-1) &= 3 \\ 4 * 3 &= 12 \end{aligned}$$

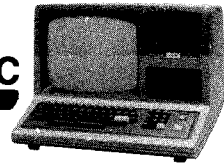
### Order of Operations

When evaluating a sequence of operations on the same level of parenthesis, BASIC uses a hierarchy to determine what operation to do first.

The two listings below show the hierarchy BASIC uses. Operators are shown in decreasing order of precedence. Operators listed in the same entry in the table have the same precedence and are executed as encountered **from left to right**:

#### Numerical operations:

[ or (Exponentiation)
+, - (Unary sign operands [ <b>not</b> addition and subtraction])
*, /
+, - (Addition and subtraction)
<, >, =, <=, >=, <>
NOT
AND
OR

**String operations:**

+ <, >, =, <=, >=, <>
--------------------------

For example, in the line:

```
X*X + 5[2.8
```

BASIC will find the value of 5 to the 2.8 power. Next, it will multiply  $X * X$ , and finally add this value to the value of 5 to the 2.8. If you want BASIC to perform the indicated operations in a different order, you must add parentheses. For example:

```
X*(X + 5[2.8)
```

or

```
X*(X + 5)[2.8
```

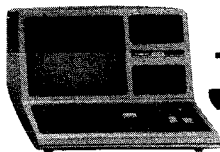
Here's another example:

```
IF X = 0 OR Y > 0 AND Z = 1 THEN 255
```

The relational operators  $=$  and  $>$  have the highest precedence, so BASIC performs them first, one after the next, from left to right. Then the logical operations are performed. AND has a higher precedence than OR, so BASIC performs the AND operation before OR.

If the above line looks confusing because you can't remember which operator is precedent over which, then you can use parentheses to make the sequence obvious:

```
IF X = 0 OR ((Y > 0) AND (Z = 1)) THEN 255
```



## Functions

A function is a built-in sequence of operations which BASIC will perform on data. A function is actually a subroutine which usually returns a data item. BASIC functions save you from having to write a BASIC routine, and they operate faster than a BASIC routine would.

A function consists of a keyword which is usually followed by the data that you specify. This data is always enclosed in parentheses; if more than one data item is required, the items are separated by commas.

If the data required is termed "number" you may insert any numerical expression. If it is termed "string" you may insert a string expression.

### Examples:

```
SQR(A + 6)
```

Tells BASIC to compute the square root of (A + 6).

```
MID$(A$, 3, 2)
```

Tells BASIC to return a substring of the string A\$, starting with the third character, with a length of 2.

Functions cannot stand alone in a BASIC program. Instead they are used in the same way you use expressions — as the data in a statement.

For example

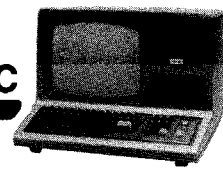
```
A = SQR(7)
```

Assigns A the data returned as the square root of 7.

```
PRINT MID$(A$, 3, 2)
```

Prints the substring of A\$ starting at the third character and two characters long.

If the function returns numeric data, it is a numeric function and may be used in a numeric expression. If it returns string data, it is a string function and may be used in a string expression.



# How to Construct an Expression

Understanding how to construct an expression will help you put together powerful statements – instead of using many short ones. In this section we will discuss the two kinds of expressions you may construct:

- Simple
- Complex

as well as how to construct a function.

As we have stated before, an expression is actually data. This is because once BASIC performs all the operations, it returns one data item. An expression may be string or numeric. It may be composed of:

- Constants
- Variables
- Operators
- Functions

Expressions may be either simple or complex:

A **simple expression** consists of a single term: a constant, variable or function. If it is a numeric term, it may be preceded by an optional + or – sign.

For example:

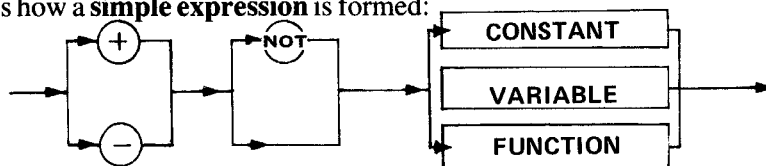
+A    3.3    -5    SQR(8)

are all simple numeric expressions, since they only consist of one numeric term.

A\$    STRING\$(20, A\$)    "WORD"    "M"

are all simple string expressions since they only consist of one string term.

Here's how a **simple expression** is formed:



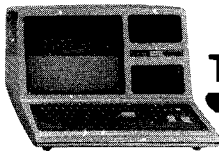
A **complex expression** consists of two or more terms (simple expressions) combined by operators. For example:

A-1    X+3.2-Y    1=1    A AND B    ABS(B)+LOG(2)

are all examples of complex numeric expressions. (Notice that you can use the relational expression (1 = 1) and the logical expression (5 AND 3) as a complex numeric expression since both actually return numeric data.)

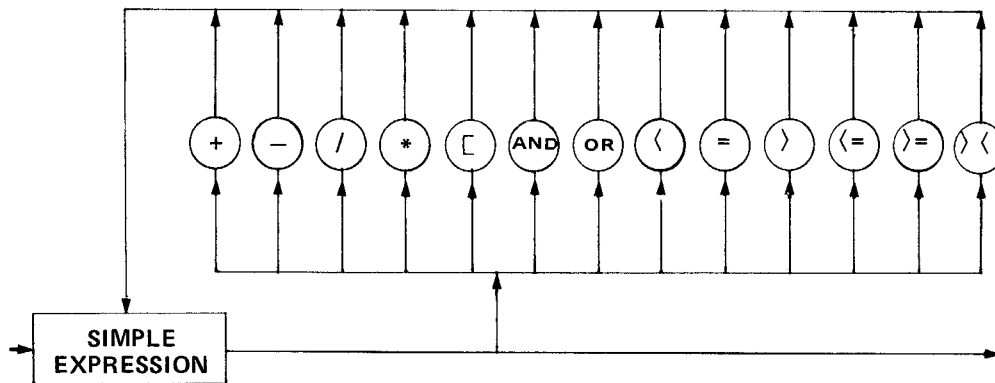
A\$+B\$    "Z"+Z\$    STRING\$(10, "A")+ "M"

are all examples of complex string expressions.

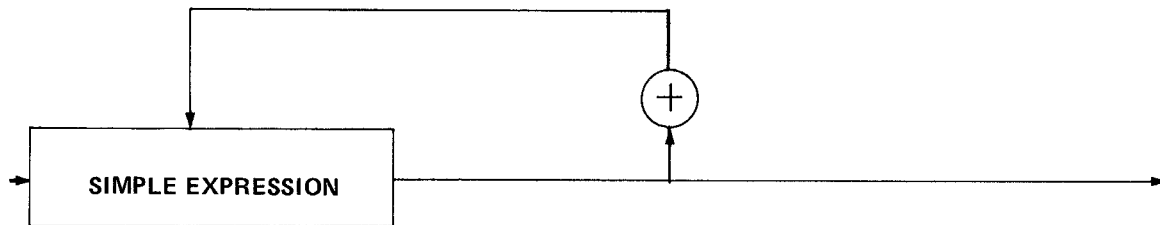


## TRS-80 MODEL III

This is how a **complex numeric expression** is formed:



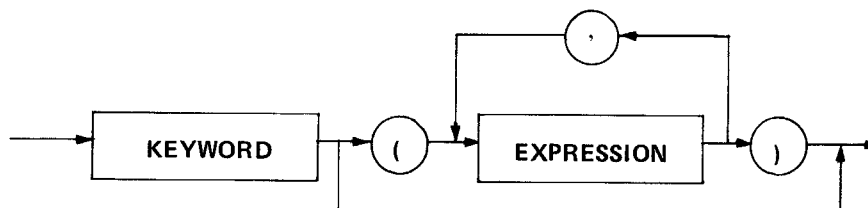
This is how a **complex string expression** is formed:



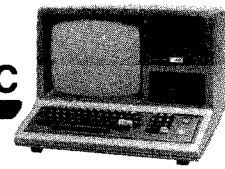
Most functions, except functions returning system information, require that you input either or both of the following kinds of data:

- One or more numeric expressions
- One or more string expressions.

This is how a **function** is formed:



If the data returned is a number, the function may be used as a term in a numeric expression. If the data is a string, the function may be used as a term in a string expression.



## 2/Commands

Whenever a prompt `>` is displayed, your Computer is in the "Immediate" or "Command" Mode. You can type in a command, **(ENTER)** it, and the Computer will respond immediately. This chapter describes the commands you'll use to control the Computer — to change modes, begin input and output procedures, alter program memory, etc. **All of these commands — except CONT — may also be used inside your program as statements.** In some cases this is useful; other times it is just for very specialized applications.

The commands described in this chapter are:

AUTO	CONT	EDIT	RUN
CLEAR	CSAVE	LIST	SYSTEM
CLOAD	DELETE	LLIST	TROFF
CLOAD?		NEW	TRON

### AUTO line number, increment

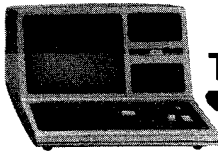
Turns on an automatic line numbering function for convenient entry of programs — all you have to do is enter the actual program statements. You can specify a beginning line number and an increment to be used between line numbers. Or you can simply type AUTO and press **(ENTER)**, in which case line numbering will begin at 10 and use increments of 10. Each time you press **(ENTER)**, the Computer will advance to the next line number.

#### Examples:

AUTO	to use line numbers	10, 20, 30, . . .
AUTO 5, 5		5, 10, 15, . . .
AUTO 100		100, 110, 120, . . .
AUTO 100, 25		100, 125, 150, . . .
AUTO ,10		0, 10, 20, . . .

To turn off the AUTO function, press the **(BREAK)** key. (Note: When AUTO brings up a line number which is already being used, an asterisk will appear beside the line number. If you do not wish to re-program the line, press the **(BREAK)** key to turn off AUTO function.)





### **CLEAR** *n*

When used without an argument (e.g., type CLEAR and press **ENTER**), this command resets all numeric variables to zero, and all string variables to null. When used with an argument (e.g., CLEAR 100), this command performs a second function in addition to the one just described: it makes the specified number of bytes available for string storage.

Example: CLEAR 100 makes 100 bytes available for strings. When you turn on the Computer a CLEAR 50 is executed automatically.

### **CLOAD** *“file name”*

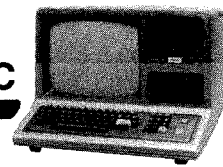
Lets you load a BASIC program stored on cassette. Place recorder/player in Play mode (be sure the proper connections are made and cassette tape has been re-wound to proper position). The file name may be any single character except the double-quote (").

**Note:** See “Using the Cassette Interface” in the Operation Section for instructions on which baud rate to use.

Entering CLOAD will turn on the cassette machine and load the first program encountered. BASIC also lets you specify a desired “file” in your CLOAD command. For example, CLOAD “A” will cause the Computer to ignore programs on the cassette until it comes to one labeled “A”. So no matter where file “A” is located on the tape, you can start at the beginning of the tape; file “A” will be picked out of all the files on the tape and loaded. As the Computer is searching for file “A”, the names of the files encountered will appear in the upper right corner of the Display, along with a blinking “\*”.

Only the first character of the file name is used by the Computer for CLOAD, CLOAD?, and CSAVE operations.

Loading a program from tape automatically clears out the previously stored program. See also CSAVE.



## CLOAD? *“file name”*

Lets you compare a program stored on cassette with one presently in the Computer. This is useful when you have saved a program onto tape (using CSAVE) and you wish to check that the transfer was successful. You may specify CLOAD? *“file-name”*. If you don't specify a file-name, the first program encountered will be tested. During CLOAD?, the program on tape and the program in memory are compared byte for byte. If there are any discrepancies (indicating a bad dump), the message “BAD” will be displayed. In this case, you should CSAVE the program again. (CLOAD?, unlike CLOAD, does not erase the program memory.)

Be sure to type the question mark or the Computer will interpret your command as CLOAD.

## CONT

When program execution has been stopped (by the **BREAK** key or by a STOP statement in the program), type CONT and **ENTER** to continue execution at the point where the stop or break occurred. During such a break or stop in execution, you may examine variable values (using PRINT) or change these values. Then type CONT and **ENTER** and execution will continue with the current variable values. CONT, when used with STOP and the **BREAK** key, is primarily a debugging tool.

**NOTE:** You cannot use CONT after EDITing your program lines or otherwise changing your program. CONT is also invalid after execution has ended normally. See also STOP.

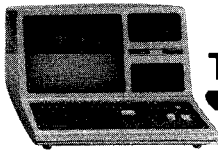
## CSAVE *“file name”*

Stores the resident program on cassette tape. (Cassette recorder must be properly connected, cassette loaded, and in the Record mode, before you enter the CSAVE command.) You must specify a file-name with this command. This file-name may be any alpha-numeric character other than double-quote ("). The program stored on tape will then bear the specified file-name, so that it can be located by a CLOAD command which asks for that particular file-name. You should always write the appropriate file-names on the cassette case for later reference.

### Examples:

CSAVE "1"            saves resident program and attaches label "1"  
CSAVE "A"            saves resident program and attaches label "A"

See also CLOAD. and “Using the Cassette Interface” in the Operation Section.



### **DELETE** *line number-line number*

Erases program lines from memory. You may specify an individual line or a sequence of lines, as follows:

<i>DELETE line number</i>	Erases one line as specified
<i>DELETE line number-line number</i>	Erases all program lines starting with first line number specified and ending with last number specified
<i>DELETE-line number</i>	Erases all program lines up to and including the specified number

The upper line number to be deleted must be a currently used number.

#### **Examples:**

<i>DELETE 5</i>	Erases line 5 from memory (error if line 5 not used)
<i>DELETE 11-18</i>	Erases lines 11, 18 and every line in between

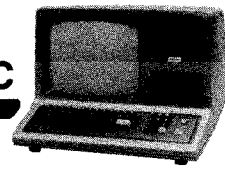
If you have just entered or edited a line, you may delete that line simply by entering *DELETE.* (use a period instead of the line number).

### **EDIT** *line number*

Puts the Computer in the Edit Mode so you can modify your resident program. The longer and more complex your programs are, the more important *EDIT* will be. The Edit Mode has its own selection of subcommands, and we have devoted Chapter 9 to the subject.

### **LIST** *line number-line number*

Instructs the Computer to display all programs lines presently stored in memory. If you enter *LIST* without an argument, the entire program will scroll continuously up the screen. To stop the automatic scrolling, press **(SHIFT)** and **@** simultaneously. This will freeze the display. Press any key to release the "pause" and continue the automatic scrolling.



To examine one line at a time, specify the desired line number as an argument in the LIST command. To examine a certain sequence of program lines, specify the first and last lines you wish to examine.

### Examples:

LIST 50	Displays line 50
LIST 50-150	Displays line 50, 150 and everything in between
LIST 50 -	Displays line 50 and all higher-numbered lines
LIST.	Displays current line (line just entered or edited)
LIST - 50	Displays all lines up to and including line 50

## LLIST

Works like LIST, but outputs to the Printer

LLIST	Lists current program to printer.
LLIST 100 -	Lists line 100 to the end of the program to the line printer.
LLIST 100-200	Lists line 100 through 200 to the line printer.
LLIST.	Lists current line to the line printer.
LLIST - 100	Lists all lines up to and including line 100 to the line printer.

See LIST.

## NEW

Erases all program lines, sets numeric variables to zero and string variables to null. It does not change the string space allocated by a previous CLEAR *number* statement.

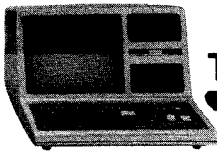
NEW is used in the following program to provide password protection.

```

10 INPUT A$: IF A$ <> "E" THEN 65520
20 REM
30 REM          REST OF PROGRAM HERE
40 REM
65519 END
65520 NEW

```

You can't run the rest of the program until you enter the correct password, in this case an E.



### **RUN** *line number*

Causes Computer to execute the program stored in memory. If no line number is specified, execution begins with lowest numbered program line. If a line number is specified, execution begins with the line number. (Error occurs if you specify an unused line number.) Whenever RUN is executed, Computer also executes a CLEAR.

#### **Examples:**

RUN	Execution begins at lowest-numbered line
RUN 100	Execution begins at line 100

RUN may be used inside a program as a statement; it is a convenient way of starting over with a clean slate for continuous-loop programs such as games.

To execute a program without CLEARing variables, use GOTO.

## **SYSTEM**

Puts the Computer in the System Mode, which allows you to load object files (machine-language routines or data). Radio Shack offers several machine-language software packages, such as the Editor-Assembler. You can also create your own object files using the TRS-80 Editor/Assembler.

To load an object file: Type **SYSTEM** and **(ENTER)**

\*?

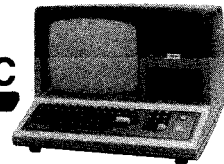
will be displayed. Now enter the file name (no quotes are necessary) and the tape will begin loading. During the tape load, the familiar asterisks will flash in the upper right-hand corner of the Video Display. When loading is complete, another

\*?

will be displayed. Type in a slash-symbol / followed by the address (in decimal form) at which you wish execution to begin. Or you may simply type in the slash-symbol and **(ENTER)** without any address. In this case execution will begin at the address specified by the object file.

**NOTE:** BASIC object files are stored as blocks. Further, each block has its own check sum. Should a check sum error occur while loading, the leftmost asterisk will change into the letter C. If this occurs you will have to reload the entire object file. (If the tape motion doesn't stop, hold down **(BREAK)** until READY returns.)

See "Using the Cassette Interface" in the Operation Section for information on which baud rate to use and the procedures for loading a system tape.



## TROFF

Turns off the Trace function. See TRON.

## TRON

Turns on a Trace function that lets you follow program-flow for debugging and execution analysis. Each time the program advances to a new program line, that line number will be displayed inside a pair of brackets.

For example, enter the following program:

```
10 PRINT "LINE 10"  
20 INPUT "PRESS <ENTER> TO BEGIN THE LOOP"; X  
30 PRINT "HERE WE GO..."  
40 GOTO 30
```

Now type in TRON **(ENTER)**, and RUN **(ENTER)**.

```
<10>LINE 10  
<20>PRESS <ENTER> TO BEGIN THE LOOP?  
<30>HERE WE GO...  
<40><30>HERE WE GO...  
<40><30>HERE WE GO...  
etc.
```

(Press **(SHIFT)** and **@** simultaneously to pause execution and freeze display. Press any key to continue with execution.)

As you can see from the display, the program is in an infinite loop.

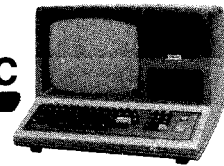
The numbers show you exactly what is going on. (To stop execution, press **(BREAK)**.)

To turn off the Trace function, enter TROFF. TRON and TROFF may be used inside programs to help you tell when a given line is executed.

For Example

```
50 TRON  
60 A = A + 1  
70 TROFF
```

might be helpful in pointing out every time line 60 is executed (assuming execution doesn't jump directly to 60 and bypass 50). Each time these three lines are executed, <60> <70> will be displayed. Without TRON, you wouldn't know whether the program was actually executing line 60. After a program is debugged, TRON and TROFF lines can be removed.



## 3/Input-Output

The statements described in this chapter let you send data from Keyboard to Computer, Computer to Display, and back and forth between Computer and the Cassette and the Line Printer (if you have one). These will primarily be used inside programs to input data and output results and messages.

Statements covered in this chapter:

PRINT

@ (PRINT modifier)  
 TAB ((PRINT modifier)  
 USING (PRINT formatter)

INPUT  
 DATA  
 READ  
 RESTORE  
 LPRINT  
 PRINT #-1 (Output to Cassette)  
 INPUT #-1 (Input to Cassette)

### PRINT item list

Prints an item or a list of items on the Display. The items may be either string constants (messages enclosed in quotes), string variables, numeric constants (numbers), variables, or expressions involving all of the preceding items. The items to be PRINTed may be separated by commas or semi-colons. If commas are used, the cursor automatically advances to the next print zone before printing the next item. If semi-colons are used, no space is inserted between the items printed on the Display. In cases where no ambiguity would result, all punctuation can be omitted.

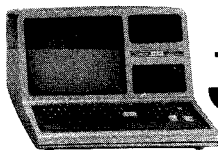
#### Examples:

```
30 X = 5
40 PRINT 25; "IS EQUAL TO"; X ↑ 2
50 END
```

```
80 A$ = "STRING"
90 PRINT A$; A$, A$; " "; A$
100 END
```

```
130 X = 25
140 PRINT 25 "IS EQUAL TO" X
150 END
```

```
180 A = 5: B = 10: C = 3
190 PRINT ABC
200 END
```



## TRS-80 MODEL III

---

**Postive numbers** are printed with a leading blank (instead of a plus sign); **all numbers** are printed with a trailing blank; and no blanks are inserted before or after **strings** (you can insert them with quotes as in line 90).

In line 140 no punctuation is needed; but in line 190 zero will print out because ABC is interpreted as a single variable which has not been assigned a value yet.

```
230 PRINT "ZONE 1", "ZONE 2", "ZONE 3", "ZONE 4", "ZONE 1 ETC"  
240 END
```

There are four 16-character print zones per line.

```
270 PRINT "ZONE 1", "ZONE 3"  
280 END
```

The cursor moves to the next print zone each time a comma is encountered.

```
300 PRINT "PRINT STATEMENT #10";  
310 PRINT "PRINT STATEMENT #20"  
320 END
```

A trailing semi-colon overrides the cursor-return so that the next PRINT begins where the last one left off (see line 300).

If no trailing punctuation is used with PRINT, the cursor drops down to the beginning of the next line.

### **PRINT** @ *position, item list*

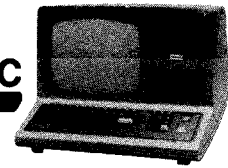
Specifies exactly where printing is to begin. The @ modifier must be a number from 0 to 1023. Refer to the Video Display worksheet, Appendix C, for the exact position of each location 0-1023:

```
100 PRINT @ 550, "LOCATION 550"
```

RUN this to find out where location 550 is.

```
100 PRINT @ 550, 550; @ 650, 650
```





Whenever you PRINT @ on the bottom line of the Display, there is an automatic line-feed, causing everything displayed to move up one line. To suppress this, use a trailing semi-colon at the end of the statement.

**Example:**

```
100 PRINT @ 1000, 1000;  
110 GOTO 110
```

Use a trailing semi-colon or comma any time you want to suppress the line feed.

**PRINT TAB (*expression*)**

Moves the cursor to the specified position on the current line (modulo \* 128 if you specify TAB positions greater than 127). TAB may be used several times in a PRINT list.

The value of *expression* must be between 0 and 255 inclusive.

**Example:**

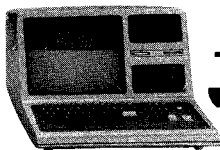
```
10 PRINT TAB (5) "TABBED 5"; TAB(25) "TABBED 25"
```

No punctuation is required after a TAB modifier.

```
340 ?FROM PRINT TAB(EXPRESSION)  
350 X = 3  
360 PRINT TAB(X) X; TAB(X ↑ 2) X ↑ 2; TAB(X ↑ 3) X ↑ 3  
370 END
```

Numerical expressions may be used to specify a TAB position. This makes TAB very useful for graphs of mathematical functions, tables, etc. TAB cannot be used to move the cursor to the left. If cursor is beyond the specified position, the TAB is ignored.

**\*Modulo** A cyclic counting system. Modulo 64 means the count goes from zero to 63 and then starts over at zero.



### PRINT USING *string; item list*

This statement allows you to specify a format for printing string and numeric values. It can be used in many applications such as printing report headings, accounting reports, checks, or wherever a specific print format is required.

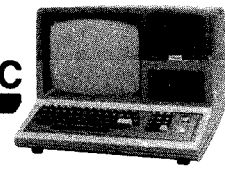
The PRINT USING statement uses the following format:

PRINT USING *string ; value*

*String* and *value* may be expressed as variables or constants. This statement will print the expression contained in the string, inserting the numeric value shown to the right of the semicolon as specified by the field specifiers.

The following field specifiers may be used in the string:

- # This sign specifies the position of each digit located in the numeric value. The number of # signs you use establishes the numeric field. If the numeric field is greater than the number of digits in the numeric value, then the unused field positions to the left of the number will be displayed as spaces and those to the right of the decimal point will be displayed as zeros.
- .
- ,
- \*\* Two asterisks placed at the beginning of the field will cause all unused positions to the left of the decimal to be filled with asterisks. The two asterisks will establish two more positions in the field.
- \$ A dollar-sign will be printed ahead of the number.
- \$\$ Two dollar signs placed at the beginning of the field will act as a floating dollar sign. That is, it will occupy the first position preceding the number.
- \*\*\$ If these three signs are used at the beginning of the field, then the vacant positions to the left of the number will be filled by the \* sign and the \$ sign will again position itself in the first position preceding the number.
- (↑) (↑) (↑) (↑) Causes the number to be printed in exponential (E or D) format.  
or [ [ [ [ This will be displayed as a "[[".



- +            When a + sign is placed at the beginning or end of the field, it will be printed as specified as a + for positive numbers or as a - for negative numbers.
- When a - sign is placed at the end of the field, it will cause a negative sign to appear after all negative numbers and will appear as a space for positive numbers.
- % spaces %**    To specify a string field of more than one character, *% spaces %* is used. The length of the string field will be 2 plus the number of spaces between the percent signs.
- !**            Causes the Computer to use the first string character of the current value.

Any other character that you include in the USING string will be displayed as a string literal.

The following program will help demonstrate these format specifiers:

```
10 INPUT "TYPE IN FORMAT, THEN DATA"; A$, A
20 PRINT USING A$; A
30 GOTO 10
```

RUN this program and try various specifiers and strings for A\$ and various values for A.

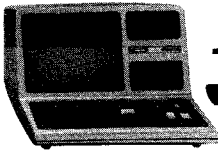
**For Example:**

```
>RUN
TYPE IN FORMAT, THEN DATA? ##.#, 12.12
12.1
TYPE IN FORMAT, THEN DATA? ##.#, 1.34
1.3
TYPE IN FORMAT, THEN DATA? ###.##, 1000.33
%1000.33
TYPE IN FORMAT, THEN DATA?
```

The % sign is automatically printed if the field is not large enough to contain the number of digits found in the numeric value. The entire number to the left of the decimal will be displayed preceded by this sign.

```
>RUN
TYPE IN FORMAT, THEN DATA? ##.##, 12.127
12.13
TYPE IN FORMAT, THEN DATA?
```

Note that the number was rounded to two decimal places.



## TRS-80 MODEL III

```
TYPE IN FORMAT, THEN DATA? +##.##, 12.12
+12.12
TYPE IN FORMAT, THEN DATA? "THE ANSWER IS +##.##", -12.12
THE ANSWER IS -12.12
TYPE IN FORMAT, THEN DATA? ##.##+, 12.12
12.12+
TYPE IN FORMAT, THEN DATA? ##.##+, -12.12
12.12-
TYPE IN FORMAT, THEN DATA? ##.##-, 12.12
12.12
TYPE IN FORMAT, THEN DATA? ##.##-, -12.12
12.12-

TYPE IN FORMAT, THEN DATA? "**** IN TOTAL.", 12.12
**12 IN TOTAL.
TYPE IN FORMAT, THEN DATA? $###.##, 12.12
$ 12.12
TYPE IN FORMAT, THEN DATA? $$$###.##, 12.12
$12.12
TYPE IN FORMAT, THEN DATA? **$###.##, 12.12
***$12.12
TYPE IN FORMAT, THEN DATA? "#,###,###", 1234567
1,234,570
TYPE IN FORMAT, THEN DATA?
```

Another way of using the PRINT USING statement is with the string field specifiers “!” and % spaces %.

### Examples:

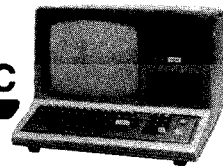
```
PRINT USING "!"; string
PRINT USING "% %"; string
```

The “!” sign will allow only the first letter of the string to be printed. The “% spaces %” allows spaces + 2 characters to be printed. Again, the *string* and specifier can be expressed as string variables. The following program will demonstrate this feature:

```
10 INPUT "TYPE IN THE FORMAT, THEN THE STRING DATA"; A$, B$
20 PRINT USING A$; B$
30 GOTO 10
```

and RUN it:

```
TYPE IN THE FORMAT, THEN THE STRING DATA? !, ABCDE
A
TYPE IN THE FORMAT, THEN THE STRING DATA? %%, ABCDE
AB
TYPE IN THE FORMAT, THEN THE STRING DATA? % %, ABCDE
ABCD
TYPE IN THE FORMAT, THEN THE STRING DATA?
```



Multiple strings or string variables can be joined together (concatenated) by these specifiers. The “!” sign will allow only the first letter of each string to be printed. For example:

```
10 INPUT "TYPE IN THREE NAMES"; A$, B$, C$
20 PRINT USING "!"; A$, B$, C$
30 GOTO 10
```

And RUN it. . .

```
>RUN
TYPE IN THREE NAMES? ABC, DEF, GHI
ADG
TYPE IN THREE NAMES?
```

By using more than one “!” sign, the first letter of each string will be printed with spaces inserted corresponding to the spaces inserted between the “!” signs. To illustrate this feature, make the following change to the last little program:

```
20 PRINT USING "! ! !"; A$, B$, C$
```

And RUN it. . .

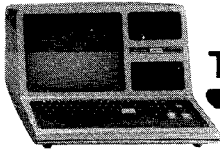
```
>RUN
TYPE IN THREE NAMES? ABC, DEF, GHI
A D G
TYPE IN THREE NAMES?
```

Spaces now appear between letters A, D and G to correspond with those placed between the three “!” signs.

Try changing “!!!” to “%%” in line 20 and run the program.

The following program demonstrates one possible use for the PRINT USING statement.

```
510 CLS
520 A$ = "***##,#####.## DOLLARS"
530 INPUT "WHAT IS YOUR FIRST NAME"; F$
540 INPUT "WHAT IS YOUR MIDDLE NAME"; M$
550 INPUT "WHAT IS YOUR LAST NAME"; L$
560 INPUT "ENTER THE AMOUNT PAYABLE"; P
570 PRINT: PRINT "PAY TO THE ORDER OF ";
580 PRINT USING "! . !. %" F$, M$, L$
600 PRINT: PRINT USING A$; P
620 END
```



## TRS-80 MODEL III

RUN the program. Remember, to save programming time, use the “?” sign for PRINT. Your display should look something like this:

```
WHAT IS YOUR FIRST NAME? ALBERT
WHAT IS YOUR MIDDLE NAME? BARCUSSI
WHAT IS YOUR LAST NAME? COOSEY
ENTER THE AMOUNT PAYABLE? 12385.34

PAY TO THE ORDER OF A. B. COOSEY

*****$12,385.30 DOLLARS
```

If you want to use a double-precision amount without rounding off or going into scientific notation, then simply add the double precision sign (#) after the variable P in Lines 560 and 600. You will then be able to use amounts up to 16 decimal places long.

### INPUT *item list*

Causes Computer to stop execution until you enter the specified number of values via the keyboard. The INPUT statement may specify a list of string or numeric variables to be input. The items in the list must be separated by commas.

```
100 INPUT X$, X1, Z$, Z1
```

This statement calls for you to input a string-literal, a number, another string literal, and another number, **in that order**. When the statement is encountered, the Computer will display a

?

You may then enter the values all at once or one at a time. To enter values all at once, separate them by commas. (If your string literal includes leading blanks, colons, or commas, you must enclose the string in quotes.)

For example, when line 100 (above) is RUN and the Computer is waiting for your input, you could type

```
JIM,50,JACK,40  (ENTER)
```

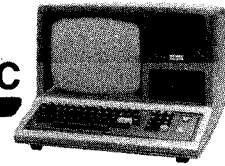
The Computer will assign values as follows:

```
X$ = "JIM"      X1 = 50      Z$ = "JACK"     Z1 = 40
```

If you (ENTER) the values one at a time, the Computer will display a

??

. . . indicating that more data is expected. Continue entering data until all the variables have been set, at which time the Computer will advance to the next statement in your program.



Be sure to enter the correct type of value according to what is called for by the INPUT statement. For example, you can't input a string-value into a numerical variable. If you try to, the Computer will display a

```
?REDO
?
```

and give you another chance to enter the correct type of data value, starting with the *first* value called for by the INPUT list. The Computer *will* accept numeric data for string input.

**NOTE:** You cannot input an expression into a numerical value — you must input a simple numerical constant.

**Example:**

```
10 INPUT X1, Y1$
20 PRINT X1, Y1$
30 END
>RUN
? 7 + 3
?REDO
? 10
?? "THIS IS A COMMA , "
10          THIS IS A COMMA ,
```

It was necessary to put quotes around "THIS IS A COMMA," because the string contained a comma.

If you type in more data elements than the INPUT statement specifies, the Computer will display the message

```
?EXTRA IGNORED
```

and continue with normal execution of your program.

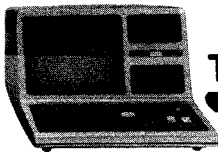
If you press **(ENTER)** without typing anything, the variables will have the values they were previously assigned.

You can also include a "prompting message" in your INPUT statement. This will make it easier to input the data correctly. The prompting message must immediately follow "INPUT", must be enclosed in quotes, and must be followed by a semi-colon.

**Example:**

```
10 INPUT "ENTER NAME, AGE"; N$, A
20 PRINT "HELLO, "; N$; ", YOU ARE AT LEAST"; A * 365; "DAYS OLD"

RUN
ENTER NAME, AGE? DO RAMEY, 31
HELLO, DO RAMEY, YOU ARE AT LEAST 11315 DAYS OLD
```



### **DATA** *item list*

Lets you store data inside your program to be accessed by READ statements. The data items will be read sequentially, starting with the first item in the first DATA statement, and ending with the last item in the last DATA statement. Items in a DATA list may be string or numeric constants — no expressions are allowed. If your string values include colons, commas or leading blanks, you must enclose these values in quotes.

It is important that the data types in a DATA statement match up with the variable types in the corresponding READ statement. DATA statements may appear anywhere it is convenient in a program. Generally, they are placed consecutively, but this is not required.

#### **Examples:**

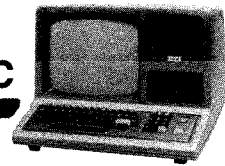
```
10 READ N1$, N2$, N3, N4
20 DATA THIS IS ITEM ONE, THIS IS ITEM TWO, 3, 4
30 PRINT N1$, N2$, N3, N4
```

See **READ**, **RESTORE**.

### **READ** *item list*

Instructs the Computer to read a value from a DATA statement and assign that value to the specified variable. The first time a READ is executed, the first value in the first DATA statement will be used; the second time, the second value in the DATA statement will be read. When all the items in the first DATA statement have been read, the next READ will use the first value in the second DATA statement; etc. (An Out-of-Data error occurs if there are more attempts to READ than there are DATA items.) The following program illustrates a common application for READ/DATA statements.





```
700 PRINT "NAME", "AGE"  
710 READ N$  
720 IF N$ = "END" THEN PRINT "END OF LIST": END  
730 READ AGE  
740 IF AGE < 18 PRINT N$, AGE  
750 GOTO 710  
760 DATA "SMITH, JOHN", 30, "ANDERSON, T.M.", 20  
770 DATA "JONES, BILL", 15, "DOE, SALLY", 21  
780 DATA "COLLINS, ANDY", 17, END
```

The program locates and prints all the minors' names from the data supplied. Note the use of an END string to allow reading lists of unknown length.

See **DATA, RESTORE**

## RESTORE

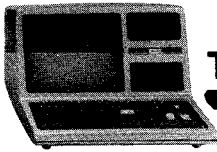
Causes the next READ statement executed to start over with the first item in the first DATA statement. This lets your program re-use the same DATA lines.

### Example:

```
810 READ X  
820 RESTORE  
830 READ Y  
840 PRINT X, Y  
850 DATA 50, 60  
860 END
```

Because of the RESTORE statement, the second READ statement starts over with the first DATA item.

See **READ, DATA**



### LPRINT

This command or statement allows you to output information to the Line Printer. For example, LPRINT A will list the value of A to the line printer. LPRINT can also be used with all the options available with PRINT **except** PRINT @.

#### Examples:

LPRINT *variable or expression* lists the variable or expression to the line printer.

LPRINT USING prints the information to the line printer using the format specified.

LPRINT TAB will move the line printer carriage position to the right as indicated by the TAB expression.

#### Example:

```
10 LPRINT TAB (5) "NAME" TAB (30) "ADDRESS" STRING$(63,32) "BALANCE"  
will print NAME at column 5, ADDRESS at column 30, and BALANCE at column 100.  
See PRINT.
```

### PRINT #-1, *item list*

Prints the values of the specified variables onto cassette tape. (Recorder must be properly connected and set in Record mode when this statement is executed.)

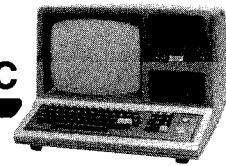
#### Example:

```
890 A1 = -30.334: B$ = "STRING-VALUE"  
900 PRINT #-1, A1, B$, "THAT'S ALL"  
910 END
```

This stores the current values of A1 and B\$, and also the string-literal "THAT'S ALL". The values may be input from tape later using the INPUT #-1 statement. The INPUT #-1 statement must be identical to the PRINT #-1 statement in terms of **number** and **type of items** in the PRINT #-1/INPUT lists. See INPUT #-1.

#### Special Note:

The values represented in *item list* must not exceed 248 characters total; otherwise all characters after the first 248 will be truncated. For example, PRINT #-1, A#, B#, C#, D#, E#, F#, G#, H#, I#, J#, A\$ will probably exceed the maximum record length if A\$ is longer than about 75 characters. If you have a lengthy list, you should break it up into two or more PRINT# statements.



## INPUT #-1, *item list*

Inputs the specified number of values stored on cassette and assigns them to the specified variable names.

### Example:

```
50 INPUT #-1,X,P$,T$
```

When this statement is executed, the Computer will turn on the tape machine, input values in the order specified, then turn off the tape machine and advance to the next statement. If a string is encountered when the INPUT list calls for a number, a bad file data error will occur. If there are not enough data items on the tape to "fill" the INPUT statement, an Out of Data error will occur.

**The Input list must be identical to the Print list that created the taped data-block (same number and type of variables in the same sequence.)**

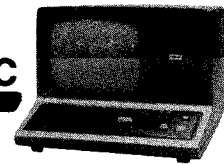
## Sample Program

Use the two-line program supplied in the PRINT# description to create a short data file. Then rewind the tape to the beginning of the data file, make all necessary connections, and put cassette machine in Play mode. Now run the following program.

```
10 INPUT #-1, A1, B$, L$
20 PRINT A1, B$, L$
30 IF L$ = "THAT'S ALL" THEN END
40 REM PROGRAM COULD GO BACK TO LINE 10 FOR MORE DATA
```

This program doesn't care how long or short the data file is, so long as:

- 1) the file was created by successive PRINT# statements **identical in form** to line 10
- 2) the last item in the last data triplet is "THAT'S ALL".



# 4/Program Statements

*MODEL III BASIC makes several assumptions about how to run your program. For example:*

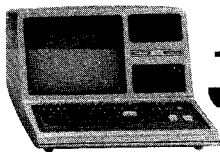
- \* *Variables are assumed to be single-precision (unless you use type declaration characters — see Chapter 1, “Variable Types”).*
- \* *A certain amount of memory is automatically set aside for strings and arrays — whether you use all of it or not.*
- \* *Execution is sequential, starting with the first statement in your program and ending with the last.*

*The statements described in this chapter let you override these assumptions, to give your programs much more versatility and power.*

**NOTE:** *All BASIC statements except INPUT and INPUT#-1 can be used in the Immediate Mode as well as in the Execute Mode.*

*Statements described in this chapter:*

<b>Type Definition</b>	<b>Assignment &amp; Allocation</b>	<b>Sequence of Execution</b>	<b>Tests (Conditional Statements)</b>
DEFINT	CLEAR <i>n</i>	END	IF
DEFSNG	DIM	STOP	THEN
DEFDBL	LET	GOTO	ELSE
DEFSTR		GOSUB	
		RETURN	
		ON . . . GOTO	
		ON . . . GOSUB	
		FOR-NEXT-STEP	
		ERROR	
		ON ERROR GOTO	
		RESUME	
		REM	



### **DEFINT** *letter range*

Variables beginning with any letter in the specified range will be stored and treated as integers, unless a type declaration character is added to the variable name. This lets you conserve memory, since integer values take up less memory than other numeric types. And integer arithmetic is faster than single or double precision arithmetic. However, a variable defined as integer can only take on values between  $-32768$  and  $+32767$  inclusive.

#### **Examples:**

```
10 DEFINT A, I, N
```

After line 10, all variables beginning with A, I or N will be treated as integers. For example, A1, AA, I3 and NN will be integer variables. However, A1#, AA#, I3# would still be double precision variables, because of the type declaration characters, which always over-ride DEF statements.

```
10 DEFINT I-N
```

Causes variables beginning with I, J, K, L, M or N to be treated as integer variables.

DEFINT may be placed anywhere in a program, but it may change the meaning of variable references without type declaration characters. Therefore it is normally placed at the beginning of a program.

See DEFSNG, DEFDBL, and Chapter 1.

### **DEFSNG** *letter range*

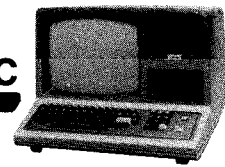
Causes any variable beginning with a letter in the specified range to be stored and treated as single precision, unless a type declaration character is added. Single precision variables and constants are stored with 7 digits of precision and printed out with 6 digits of precision. Since all numeric variables are assumed to be single precision unless DEFINed otherwise, the DEFSNG statement is primarily used to re-define variables which have previously been defined as double precision or integer.

#### **Example:**

```
100 DEFSNG I, W-Z
```

Causes variables beginning with the letter I or any letter W through Z to be treated as single precision. However, I% would still be an integer variable, and I# a double precision variable, due to the use of type declaration characters.

See DEFINT, DEFDBL, and Chapter 1.



### **DEFDBL** *letter range*

Causes variables beginning with any letter in the specified range to be stored and treated as double-precision, unless a type declaration character is added. Double precision allows 17 digits of precision; 16 digits are displayed when a double precision variable is PRINTed.

#### **Example:**

```
10 DEFDBL S-Z, A-E
```

Causes variables beginning with one of the letters S through Z or A through E to be double precision.

DEFDBL is normally used at the beginning of a program, because it may change the meaning of variable references without type declaration characters.

See **DEFINT**, **DEFSNG**, and Chapter 1.

### **DEFSTR** *letter range*

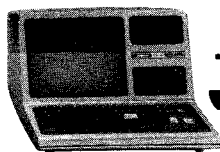
Causes variables beginning with one of the letters in the specified range to be stored and treated as strings, unless a type declaration character is added. If you have **CLEAR**ed enough string storage space, each string can store up to 255 characters.

#### **Example:**

```
10 DEFSTR L-Z
```

Causes variables beginning with any letter L through Z to be string variables, unless a type declaration character is added. After line 10 is executed, the assignment `L1 = "WASHINGTON"` will be valid.

See **CLEAR** *n*, Chapter 1, and Chapter 5.



## **CLEAR** *n*

When used with an argument *n* (*n* can be a constant or an expression), this statement causes the Computer to set aside *n* bytes for string storage. In addition all variables are set to zero. When the TRS-80 is turned on, 50 bytes are automatically set aside for strings.

The amount of string storage CLEARED must equal or exceed the greatest number of characters stored in string variables during execution; otherwise an Out of String Space error will occur.

### **Example:**

```
10 CLEAR 1000
```

Makes 1000 bytes available for string storage.

By setting string storage to the exact amount needed, your program can make more efficient use of memory. A program which uses no string variables could include a CLEAR 0 statement, for example. The CLEAR argument must be non-negative, or an error will result.

## **DIM** *name (dim1, dim2, . . . , dimK)*

Lets you set the “depth” (number of elements allowed per dimension) of an array or list of arrays. If no DIM statement is used, a depth of 11 (subscripts 0-10) is allowed for each dimension of each array used. To create an array with more than three dimensions, you must use DIM.

### **Example:**

```
10 DIM A(5), B(2,3), C$(20)
```

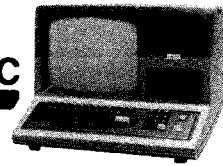
Sets up a one-dimension array A with subscripted elements 0-5; a two-dimension array B with subscripted elements 0,0 to 2,3; and a one-dimension string array C\$ with subscripted elements 0-20. Unless previously defined otherwise, arrays A and B will contain single-precision values.

DIM statements may be placed anywhere in your program, and the depth specifier may be a number or a numerical expression.

### **Example:**

```
40 INPUT "NUMBER OF NAMES"; N  
50 DIM NA(N,2)
```

To re-dimension an array, you must first use a CLEAR statement, either with or without an argument. Otherwise an error will result.

**Example Program:**

```
10 AA(4) = 11.5
20 DIM AA(7)
READY
>RUN
?DD ERROR IN 20
READY
```

See Chapter 6, ARRAYS.

**LET** *variable = expression*

May be used when assigning values to variables. Radio Shack Model III BASIC does not require LET with assignment statements, but you might want to use it to ensure compatibility with those versions of BASIC that do require it.

**Examples:**

```
100 LET A$ = "A ROSE IS A ROSE"
110 LET B1 = 1.23
120 LET X = X - Z1
```

In each case, the variable on the left side of the equals sign is assigned the value of the constant or expression on the right side.

**END**

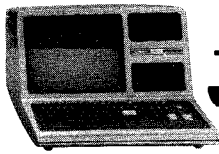
Terminates execution normally (without a BREAK message). Some versions of BASIC require END as the last statement in a program; with Model III BASIC it is optional. END is primarily used to force execution to terminate at some point other than the physical end of the program.

**Example:**

```
10 INPUT S1, S2
20 GOSUB 100
30 REM          MORE PROGRAM LINES HERE...
99 END          : REM          PROTECTIVE END-BLOCK
100 H = SQR(S1*S1 + S2*S2)
110 RETURN
```

The END statement in line 99 prevents program control from “crashing” into the subroutine. Now line 100 can only be accessed by a branching statement such as 20 GOSUB 100.





## STOP

Interrupts execution and prints a `BREAK IN line number` message. `STOP` is primarily a debugging aid. During the break in execution, you can examine or change variable values. The command `CONT` can then be used to re-start execution at the point where it left off. (If the program itself is altered during a break, `CONT` cannot be used.)

### Example:

```
10 X = RND(10)
20 STOP
30 GOSUB 1000
99 END
1000 REM
1010 RETURN
```

Suppose we want to examine what value for `X` is being passed to the subroutine beginning at line 1000. During the break, we can examine `X` with `PRINT X`.

## GOTO *line number*

Transfers program control to the specified line number. Used alone, `GOTO line number` results in an unconditional (or automatic) branch; however, test statements may precede the `GOTO` to effect a conditional branch.

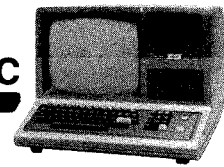
### Example:

```
200 GOTO 10
```

When 200 is executed, control will automatically jump back to line 10.

You can use `GOTO` in the Immediate Mode as an alternative to `RUN`. `GOTO line number` causes execution to begin at the specified line number, **without an automatic CLEAR**. This lets you pass values assigned in the Immediate Mode to variables in the Execute Mode.

See `IF, THEN, ELSE, ON... GOTO`.



## GOSUB *line number*

Transfers program control to the subroutine beginning at the specified line number and stores an address to RETURN to after the subroutine is complete. When the Computer encounters a RETURN statement in the subroutine, it will then return control to the statement which follows GOSUB.

If you don't RETURN, the previously stored address will not be deleted from the area of memory used for saving information, called the stack. The stack might eventually overflow, but, even more importantly, this address might be read incorrectly during another operation, causing a hard-to-find program error. So, . . . always RETURN from your subroutines. GOSUB, like GOTO may be preceded by a test statement. See IF,THEN,ELSE,ON...GOSUB.

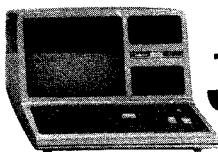
### Example Program:

```
100 GOSUB 200
110 PRINT "BACK FROM THE SUBROUTINE": END
200 PRINT "EXECUTING THE SUBROUTINE"
210 RETURN
READY
>RUN
EXECUTING THE SUBROUTINE
BACK FROM THE SUBROUTINE
```

Control branches from line 100 to the subroutine beginning at line 200. Line 210 instructs Computer to return to the statement immediately following GOSUB, that is, line 110.

## RETURN

Ends a subroutine and returns control to statement immediately following the most recently executed GOSUB. If RETURN is encountered without execution of a matching GOSUB, an error will occur. See GOSUB.



## **ON *n* GOTO** *line number, ..., line number*

This is a multi-way branching statement that is controlled by a test variable or expression. The general format for ON *n* GOTO is:

*ON expression GOTO 1st line number, 2nd line number, . . . , Kth line number*

*expression* must be between 0 and 255 inclusive.

When ON. . . GOTO is executed, first the expression is evaluated and the integer portion. . . INT(expression). . . is obtained. We'll refer to this integer portion as *J*. The Computer counts over to the *J*th element in the line-number list, and then branches to the line number specified by that element. If there is no *J*th element (that is, if  $J > K$  or  $J = 0$  in the general format above), then control passes to the next statement in the program.

If the test expression or number is less than zero, or greater than 255, an error will occur. The line-number list may contain any number of items.

For example:

```
100 ON MI GOTO 150, 160, 170, 150, 180
```

says "Evaluate MI. If integer portion of MI equals 1 then go to

line 150;

If it equals 2, then go to 160;

If it equals 3, then go to 170;

If it equals 4, then go to 150;

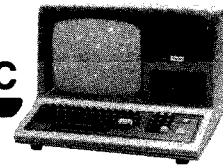
If it equals 5, then go to 180;

If the integer portion of MI doesn't equal any of the numbers 1 through 5, advance to the next statement in the program."

### **Sample Program**

```
100 INPUT "ENTER A NUMBER": X
110 ON SGN(X) + 2 GOTO 200, 210, 220
200 PRINT "NEGATIVE": END
210 PRINT "ZERO": END
220 PRINT "POSITIVE": END
```

SGN(X) returns -1 for X less than zero; 0 for X equal to zero; and +1 for X greater than 0. By adding 2, the expression takes on the values 1, 2, and 3, depending on whether X is negative, zero, or positive. Control then branches to the appropriate line number.



### ON *n* GOSUB *line number*, ..., *line number*

Works like ON *n* GOTO, except control branches to one of the subroutines specified by the line numbers in the line-number list.

#### Example:

```
100 INPUT "CHOOSE 1, 2, OR 3"; I
110 ON I GOSUB 200, 300, 400
120 END
200 PRINT "SUBROUTINE #1": RETURN
300 PRINT "SUBROUTINE #2": RETURN
400 PRINT "SUBROUTINE #3": RETURN
```

The test object *n* may be a numerical constant, variable or expression. It must have a non-negative value or an error will occur.

See ON *n* GOTO.

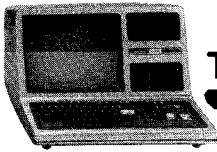
### FOR *counter* = *exp* TO *exp* STEP *exp* NEXT *counter*

Opens an iterative (repetitive) loop so that a sequence of program statements may be executed over and over a specified number of times. The general form is (brackets indicate optional material):

```
line # FOR counter-variable = initial value TO final value [STEP increment]
.
.
.
line # NEXT [counter-variable]
```

In the FOR statement, *initial value*, *final value* and *increment* can be constants, variables or expressions. The first time the FOR statement is executed, these three are evaluated and the values are saved; if the variables are changed by the loop, it will have no effect on the loop's operation. However, **the counter variable must not be changed** or the loop will not operate normally.

The FOR-NEXT-STEP loop works as follows: the first time the FOR statement is executed, the counter is set to the "initial value." Execution proceeds until a NEXT statement is encountered. At this point, the counter is incremented by the amount specified in the STEP *increment*. (If the *increment* has a negative value, then the counter is actually decremented.) If STEP *increment* is not used, an increment of 1 is assumed.



## TRS-80 MODEL III

Then the counter is compared with the *final value* specified in the FOR statement. If the counter is greater than the *final value*, the loop is completed and execution continues with the statement following the NEXT statement. (If *increment* was a negative number, loop ends when counter is **less** than *final value*.) If the counter has not yet exceeded the *final value*, control passes to the first statement after the FOR statement.

### Example Programs:

```
10 FOR I = 10 TO 1 STEP -1
20 PRINT I;
30 NEXT
READY
>RUN
 10 9 8 7 6 5 4 3 2 1
READY
>
```

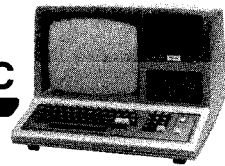
```
10 FOR K = 0 TO 1 STEP .3
20 PRINT K;
30 NEXT
READY
>RUN
 0 .3 .6 .9
READY
>
```

After  $K = .9$  is incremented by  $.3$ ,  $K = 1.2$ . This is greater than the *final value* 1, therefore loop ends without ever printing *final value*.

```
10 FOR K = 4 TO 0
20 PRINT K;
30 NEXT
READY
>RUN
 4
READY
>
```

No STEP is specified, so STEP 1 is assumed. After K is incremented the first time, its value is 5. Since 5 is greater than the *final value* 0, the loop ends.

```
10 J = 3: K = 8: L = 2
20 FOR I = J TO K + 1 STEP L
30 J = 0: K = 0: L = 0
40 PRINT I;
50 NEXT
READY
>RUN
 3 5 7 9
READY
>
```



The variables and expressions in line 20 are evaluated once and these values become constants for the FOR-NEXT-STEP loop. Changing the variable values later has no effect on the loop.

FOR-NEXT loops may be “nested”:

```
10 FOR I = 1 TO 3
20   PRINT "OUTER LOOP"
30     FOR J = 1 TO 2
40       PRINT "  INNER LOOP"
50     NEXT J
60 NEXT I
```

```
RUN
OUTER LOOP
  INNER LOOP
  INNER LOOP
OUTER LOOP
  INNER LOOP
  INNER LOOP
OUTER LOOP
  INNER LOOP
  INNER LOOP
```

```
READY
>
```

Note that each NEXT statement specifies the appropriate counter variable; however, this is just a programmer's convenience to help keep track of the nesting order. The counter variable may be omitted from the NEXT statements. But if you **do** use the counter variables, you **must** use them in the right order; i.e., the counter variable for the innermost loop must come first.

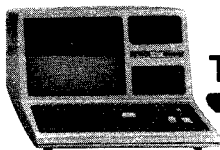
It is also advisable to specify the counter variable with NEXT statements when your program allows branching to program lines outside the FOR-NEXT loop.

Another option with nested NEXT statements is to use a counter variable list.

Delete line 50 from the above program and change line 60:

```
60 NEXT J,I
```

Loops may be nested 3-deep, 4-deep, etc. The only limit is the amount of memory available.



### **ERROR** *code*

Lets you “simulate” a specified error during program execution. The major use of this statement is for testing an ON ERROR GOTO routine. When the *ERROR code* statement is encountered, the Computer will proceed exactly as if that kind of error had occurred. Refer to Appendix B for a listing of error codes and their meanings.

#### **Example Program:**

```
100 ERROR 1
READY
>RUN
?NF Error in 100
READY
>
```

1 is the error code for “attempt to execute NEXT statement without a matching FOR statement”.

See ON ERROR GOTO, RESUME.

### **ON ERROR GOTO** *line number*

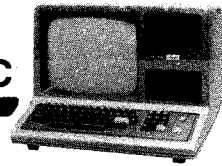
When the Computer encounters any kind of error in your program, it normally breaks out of execution and prints an error message. With ON ERROR GOTO, you can set up an error-trapping routine which will allow your program to “recover” from an error and continue, without any break in execution. Normally you have a particular type of error in mind when you use the ON ERROR GOTO statement. For example, suppose your program performs some division operations and you have not ruled out the possibility of division by zero. You might want to write a routine to handle a division-by-zero error, and then use ON ERROR GOTO to branch to that routine when such an error occurs.

#### **Example:**

```
10 ON ERROR GOTO 100
20 A = 1 / 0
90 END
100 PRINT"ERROR # "; ERR/2 + 1
110 RESUME 90
```

In this “loaded” example, when the Computer attempts to execute line 20, a divide-by-zero error will occur. But because of line 10, the Computer will simply ignore line 20 and branch to the error-handling routine beginning at line 100.

**NOTE:** The ON ERROR GOTO must be executed **before** the error occurs or it will have no effect.



The ON ERROR GOTO statement can be disabled by executing an ON ERROR GOTO 0. If you use this inside an error-trapping routine, BASIC will handle the current error normally.

The error handling routine must be terminated by a RESUME statement. See **RESUME**.

### **RESUME** *line number*

Terminates an error handling routine by specifying where normal execution is to resume.

RESUME without a line number and RESUME 0 cause the Computer to return to the statement in which the error occurred.

RESUME followed by a line number causes the Computer to branch to the specified line number.

RESUME NEXT causes the Computer to branch to the statement **following** the point at which the error occurred.

### **Sample Program with an Error Handling Routine**

```
605 ON ERROR GOTO 640
610 INPUT "SEEKING SQUARE ROOT OF"; X
620 PRINT SQR(X)
630 GOTO 610
640 PRINT "IMAGINARY ROOT:"; SQR(-X); " * I"
650 RESUME 610
660 END
```

RUN the program and try inputting a negative value.

You must place a RESUME statement at the end of your error trapping routine, so that later errors may also be trapped.





## REM

Instructs the Computer to ignore the rest of the program line. This allows you to insert comments (REMARKS) into your program for documentation. Then, when you (or someone else) look at a listing of your program, it'll be a lot easier to figure out. If REM is used in a multi-statement program line, it must be the last statement.

### Example Program:

```
710 REM ** THIS REMARK INTRODUCES THE PROGRAM **
720 REM ** AND POSSIBLY THE PROGRAMMER, TOO.   **
730 REM **                                     **
740 REM ** THIS REMARK EXPLAINS WHAT THE     **
750 REM ** VARIOUS VARIABLES REPRESENT:      **
760 REM ** C = CIRCUMFERENCE  R = RADIUS     **
770 REM ** D = DIAMETER                       **
780 REM
```

Any alphanumeric character may be included in a REM statement, and the maximum length is the same as that of other statements: 255 characters total.

In Model III BASIC, an apostrophe ' (**SHIFT** **7**) may be used as an abbreviation for :REM.

```
100 A=1 ' THIS, TOO IS A REMARK
```

## IF *true/false expression* THEN *action-clause*

Instructs the Computer to test the following logical or relational expression. If the expression is True, control will proceed to the "action" clause immediately following the expression. If the expression is False, control will jump to the matching ELSE statement (if there is one) or down to the next program line.

In numerical terms, if the expression has a non-zero value, it is always equivalent to a logical True.

### Examples:

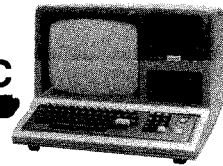
```
100 IF X > 127 THEN PRINT "OUT OF RANGE": END
```

If X is greater than 127, control will pass to the PRINT statement and then to the END statement. But if X is **not** greater than 127, control will jump down to the next line in the program, skipping the PRINT and END statements.

```
IF 0 <= X AND X <= Y THEN Y = X + 180
```

If both expressions are True then Y will be assigned the value X + 180. Otherwise control will pass directly to the next program line, skipping the THEN clause.

See THEN, ELSE.



### **THEN** *statement or line number*

Initiates the "action clause" of an IF-THEN type statement. THEN is optional except when it is required to eliminate an ambiguity, as in IF A < 0 100. THEN should be used in IF-THEN-ELSE statements.

### **ELSE** *statement or line number*

Used after IF to specify an alternative action in case the IF test fails. (When no ELSE statement is used, control falls through to the next program line after a test fails.)

#### **Examples:**

```
100 INPUT A$: IF A$ = "YES" THEN 300 ELSE END
```

In line 100, if A\$ equals "YES" then the program branches to line 300. But if A\$ does not equal "YES", program skips over to the ELSE statement which then instructs the Computer to end execution.

```
200 IF A < B THEN PRINT "A<B" ELSE PRINT "B<=A"
```

If A is less than B, the Computer prints that fact, and then proceeds down to the next program line, **skipping** the ELSE statement. If A is not less than B, Computer jumps directly to the ELSE statement and prints the specified message. **Then** control passes to the next statement in the program.

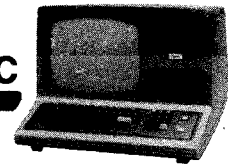
```
200 IF A>.001 THEN B = 1/A: A = A/5: ELSE 260
```

If A>.001 is True, then the next two statements will be executed, assigning new values to B and A. Then the program will drop down to the next line, skipping the ELSE statement. But if A>.001 is False, the program jumps directly over to the ELSE statement, which then instructs it to branch to line 260. Note that GOTO is not required after ELSE.

IF-THEN-ELSE statements may be nested, but you have to take care to match up the IFs and ELSEs.

```
810 INPUT "ENTER TWO NUMBERS"; A, B
820 IF A <= B THEN IF A < B PRINT A;: ELSE PRINT "NEITHER
";: ELSE PRINT B;
830 PRINT "IS SMALLER"
840 END
```

RUN the program, inputting various pairs of numbers. The program picks out and prints the smaller of any two numbers you enter.



## 5/Strings

*“Without string-handling capabilities, a computer is just a super-powered calculator.” There’s an element of truth in that exaggeration; the more you use the string capabilities of Model III BASIC, the truer the statement will seem.*

*In Model III BASIC any valid variable name can be used to contain string values, by the DEFSTR statement or by adding a type declaration character to the name. And each string can contain up to 255 characters.*

*Moreover, you can compare strings to alphabetize them, for example. You can take strings apart and string them together (concatenate them). For background material to this chapter, see Chapter 1, “Variable Types” and “Glossary”, and Chapter 4, DEFSTR.*

*Functions covered in this chapter:*

FRE (string)	LEFT\$	STRING\$
INKEY\$	MID\$	TIMES\$
LEN	RIGHT\$	VAL
ASC	STR\$	
CHR\$		

**NOTE:** Whenever string is given as a function argument, you can use a string expression or constant.

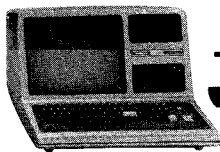
### String Space

Fifty bytes of memory are set aside automatically to store strings. If you run out of string space, you will get an OS error and you should use the CLEAR *n* command to save more space.

**Note:** CLEAR also sets variables to zero or null strings.

To calculate the space you’ll need, multiply the amount of space each variable takes (See VARPTR) by the number of string variables you are using, including temporary variables.

Temporary variables are created during the calculation of string functions. Therefore even if you have only a few short string variables assigned in your program, you may run out of string space if you concatenate them several times.



### **ASC** (*string*)

Returns the ASCII code for the first character of the specified string. The string-argument must be enclosed in parentheses. A null-string argument will cause an error to occur.

```
100 PRINT ASC("A")
110 T$ = "AB": PRINT ASC (T$)
```

Lines 100 and 110 will print the same number.

The argument may be an expression involving string operators and functions:

```
200 PRINT ASC(RIGHT$(T$, 1))
```

Refer to the ASCII Code Table, Appendix C.

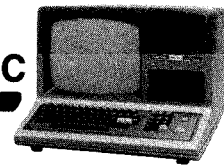
### **CHR\$** (*expression*)

Performs the inverse of the ASC function: returns a one-character string whose character has the specified ASCII, control or graphics code. The argument may be any number from 0 to 255, or any variable expression with a value in that range. Argument must be enclosed in parentheses.

```
100 PRINT CHR$(35)           Prints a number-sign #
```

Using CHR\$, you can even assign quote-marks (normally used as string-delimiters) to strings. The ASCII code for quotes " is 34. So A\$ = CHR\$(34) assigns the value "" to A\$.

```
410 A$ = CHR$(34)
420 PRINT "HE SAID, " ; A$ ; "HELLO." ; A$
```



CHR\$ may also be used to display any of the graphics or special characters. (See Appendix C, Character Codes.)

```
460 CLS
470 FOR I = 129 TO 191
480 PRINT I; CHR$(I);
490 NEXT
500 GOTO 500
```

(RUN the program to see the various graphics characters.)

Codes 0-31 are display control codes. Instead of returning an actual display character, they return a control character. When the control character is PRINTed, the function is performed. For example, 23 is the code for 32 character-per-line format; so the command, PRINT CHR\$(23) converts the display format to 32 characters per line. (Hit CLEAR, execute CLS, or execute PRINT CHR\$(28) to return to 64 character-per-line format.)

### **FRE** (*string*)

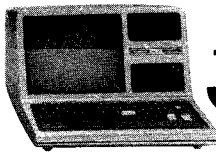
When used with a string variable or string constant as an argument, returns the amount of string storage space currently available. Argument must be enclosed in parentheses. FRE causes BASIC to start searching through memory for unused string space. If your program has done a lot of string processing, it may take several minutes to recover all the "scratch pad" type memory.

```
500 PRINT FRE(A$), FRE(L$), FRE("Z")
```

All return the same value.

The string used has no significance; it is a dummy variable. See Chapter 4, CLEAR *n*.

FRE(*number*) returns the amount of available memory (same as MEM).



### INKEY\$

Returns a one-character string determined by a keyboard check. The last key pressed before the check is returned. If no key has been pressed, a null string (length zero) is returned. This is a very powerful function because it lets you input values while the Computer is executing — without using the **(ENTER)** key. The popular video games which let you fire at will, guide a moving dot through a maze, play tennis, etc., may all be simulated using the INKEY\$ function (plus a lot of other program logic, of course).

Characters typed to an INKEY\$ are **not** automatically displayed on the screen.

INKEY\$ is often placed inside some sort of loop, so that the keyboard is scanned repeatedly.

#### Example Program:

```
540 CLS
550 PRINT @ 540; INKEY$: GOTO 550
```

RUN the program; notice that the screen remains blank until the first time you hit a key. The last key hit remains on the screen until you hit another one. (The last key hit is always saved. The INKEY\$ function uses it until it is replaced by a new value.)

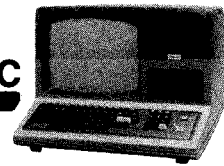
INKEY\$ may be used in sequences of loops to allow the user to build up a longer string.

#### Example:

```
590 PRINT "ENTER THREE CHARACTERS"
600 A$ = INKEY$: IF A$ = "" THEN 600 ELSE PRINT A$;
610 B$ = INKEY$: IF B$ = "" THEN 610 ELSE PRINT B$;
620 C$ = INKEY$: IF C$ = "" THEN 620 ELSE PRINT C$;
630 D$ = A$ + B$ + C$
```

A three-character string D\$ can now be entered via the keyboard without using the **(ENTER)** key.

**NOTE:** The statement IF A\$ = "" compares A\$ to the null string. There are **no** spaces between the double-quotes.

**LEFT\$** (*string*, *n*)

Returns the first *n* characters of *string*. The arguments must be enclosed in parentheses. *string* may be a string constant or expression, and *n* may be a numeric expression.

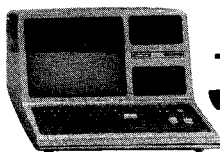
**Example Program:**

```
670 A$ = "TIMOTHY"  
680 B$ = LEFT$ (A$, 3)  
690 PRINT B$; " -THAT'S SHORT FOR "; A$
```

**LEN** (*string*)

Returns the character length of the specified string. The string variable, expression, or constant must be enclosed in parentheses.

```
730 A$ = ""  
740 B$ = "TOM"  
750 PRINT A$, B$, B$ + B$  
760 PRINT LEN(A$), LEN(B$), LEN(B$+B$)
```



## MID\$(string,p,n)

Returns a substring of *string* with length *n* and starting at position *p*. The string name, length and starting position must be enclosed in parentheses. *string* may be a string constant or expression, and *n* and *p* may be numeric expressions or constants. For example, MID\$(L\$,3,1) refers to a one-character string beginning with the third character of L\$.

If no argument is specified for the length *n*, the entire string beginning at position *p* is returned.

### Example Program:

The first three digits of a local phone number are sometimes called the "exchange" of the number. This program looks at a complete phone number (area code, exchange, last four digits) and picks out the exchange of that number.

```
800 INPUT "AREA CODE AND NUMBERS (NO HYPHENS, PLEASE)"; P$
810 EX$ = MID$(P$, 4,3)
820 PRINT "NUMBER IS IN THE "; EX$; " EXCHANGE."
```

## RIGHT\$(string, n)

Returns the last *n* characters of *string*. *string* and *n* must be enclosed in parentheses. *string* may be a string constant or variable, and *n* may be a numerical constant or variable. If LEN(*string*) is less than or equal to *n*, the entire *string* is returned.

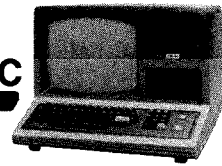
```
10 INPUT "ENTER A WORD"; M$
20 IF LEN(M$) = 0 THEN 10
30 PRINT "THE LAST LETTER WAS: "; RIGHT$(M$,1)
40 GOTO 10
```

## STR\$(expression)

Converts a numeric expression or constant to a string. The numeric expression or constant must be enclosed in parentheses. STR\$(A), for example, returns a string equal to the character representation of the value of A. For example, if A = 58.5, then STR\$(A) equals the string " 58.5". (Note the leading blank in " 58.5"). While arithmetic operations may be performed on A, only string operations and functions may be performed on the string " 58.5".

PRINT STR\$(X) prints X without a trailing blank; PRINT X prints X with a trailing blank.



**Example Program:**

```
860 A = 58.5: B = -58.5
870 PRINT STR$(A)
880 PRINT STR$(B)
890 PRINT STR$(A+B)
900 PRINT STR$(A) + STR$(B)
```

**STRING\$(*n*, "character" or number)**

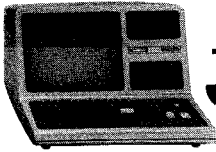
Returns a string composed of *n* character-symbols. For example, STRING\$(30, "\*") returns "\*\*\*\*\*". STRING\$ is useful in creating graphs, tables, etc.

The argument *n* is any numerical expression with a value of from zero to 255.

*character* can also be a number from 0-255; in this case, it will be treated as an ASCII, control, or graphics code.

**Example:**

```
10 CLEAR 200
20 FOR I=128 TO 191
30 A$ = STRING$(64,I)
40 PRINT A$:
50 NEXT I
```



## TRS-80 MODEL III

### TIMES

Returns today's date and time. Your Model III contains a built-in clock. To use this clock, you will want to first set it to the correct date and time. To do this, you may type and run this little program:

```
10 DEFINT A-Z
20 DIM TM(5)
30 CL = 16924
40 PRINT "INPUT 6 VALUES: MO, DA, YR, HR, MN, SS"
50 INPUT TM(0), TM(1), TM(2), TM(3), TM(4), TM(5)
60 FOR I = 0 TO 5
70     POKE CL - I, TM(I)
80 NEXT I
90 PRINT "CLOCK IS SET"
100 END
```

Once you have set the date and time with this program, you may request it any time you want. For example, this program line:

```
10 PRINT TIMES
```

causes the Computer to print today's date and time.

If you do not set the date and time, the Computer will keep time anyway. However, the date and time will be set at zero when you first turn on the Computer or reset it.

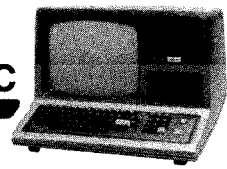
**NOTE:** The clock is turned off during cassette operations and at certain other times. Therefore it will need to be corrected periodically.

### VAL(*string*)

Performs the inverse of the STR\$ function: returns the number represented by the characters in a string argument. The numerical type of the result can be integer, single precision, or double precision, as determined by the rules for the typing of constants (See page 1/10 in this section). For example, if A\$ = "12" and B\$ = "34" then VAL(A\$ + "." + B\$) returns the value 12.34. VAL(A\$ + "E" + B\$) returns the value 12E34, that is  $12 \times 10^{34}$ .

VAL operates a little differently on mixed strings — strings whose values consist of a number followed by non-numeric characters. In such cases, only the leading number is used in determining VAL; the non-numeric remainder is ignored.

For example: VAL("100DOLLARS") returns 100.



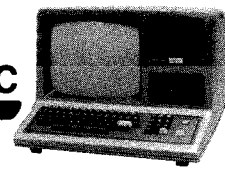
This can be a handy short-cut in examining addresses, for example.

**Example Program:**

```
940 REM "WHAT SIDE OF STREET?"
950 REM EVEN = NORTH. ODD = SOUTH
960 INPUT "ADDRESS: NUMBER AND STREET"; AD$
970 C = INT(VAL(AD$)/2) * 2
980 IF C = VAL(AD$) THEN PRINT "NORTH SIDE": GOTO 960
990 PRINT "SOUTH SIDE": GOTO 960
```

RUN the program, entering street addresses like "1015 SEVENTH AVE".

If the string is non-numeric or null, VAL returns a zero.



# 6/Arrays

An array is simply an ordered list of values. In Model III BASIC these values may be either numbers or strings, depending on how the array is defined or typed. Arrays provide a fast and organized way of handling large amounts of data. To illustrate the power of arrays, this chapter traces the development of an array to store checkbook data: check numbers, dates written, and amounts for each check.

In addition, several matrix manipulation subroutines are listed at the end of this chapter. These sequences will let you add, multiply, transpose, and perform other operations on arrays.

**Note:** Throughout this chapter, zero-subscripted elements are generally ignored for the sake of simplicity. But you should remember they are available and should be used for the most efficient use of memory. For example, after DIM A(4), array A contains 5 elements: A(0), A(1), A(2), A(3), A(4).

For background information on arrays, see Chapter 4, DIM, and Chapter 1, "Arrays".

## A Check-Book Array

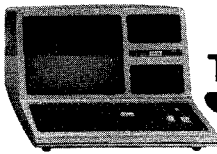
Consider the following table of checkbook information:

Check #	Date Written	Amount
025	1-1-78	10.00
026	1-5-78	39.95
027	1-7-78	23.50
028	1-7-78	149.50
029	1-10-78	4.90
030	1-15-78	12.49

Note that every item in the table may be specified simply by reference to two numbers: the row number and the column number. For example, (row 3, column 3) refers to the amount 23.50. Thus the number pair (3,3) may be called the "subscript address" of the value 23.50.

Let's set up an array, CK, to correspond to the checkbook information table. Since the table contains 6 rows and 3 columns, array CK will need two dimensions: one for row numbers, and one for column numbers. We can picture the array like this:

A(1,1) = 025	A(1,2) = 1.0178	A(1,3) = 10.00
.	.	.
.	.	.
.	.	.
.	.	.
A(6,1) = 030	A(6,2) = 1.1578	A(6,3) = 12.49



## TRS-80 MODEL III

Notice that the date information is recorded in the form *mm.dyy*, where *mm* = month number, *dd* = day of month, and *yy* = last two digits of year. **Since CK is a numeric array, we can't store the data with alpha-numeric characters such as dashes.**

Suppose we assign the appropriate values to the array elements. Unless we have used a DIM statement, the Computer will assume that our array requires a depth of 10 for each dimension. That is, the Computer will set aside memory locations to hold CK(7,1), CK(7,2), . . . , CK(10,1), CK(10,2) and CK(10,3). In this case, we don't want to set aside this much space, so we use the DIM statement at the beginning of our program:

```
40 DIM CK(6,3)
```

Now let's add program steps to read the values into the array CK:

```
50 FOR ROW = 1 TO 6
60 FOR COL = 1 TO 3
70 READ CK(ROW, COL)
80 NEXT COL, ROW
90 DATA 025, 1.0178, 10.00
100 DATA 026, 1.0578, 39.95
110 DATA 027, 1.0778, 23.50
120 DATA 028, 1.0778, 149.50
130 DATA 029, 1.1078, 4.90
140 DATA 030, 1.1578, 12.49
```

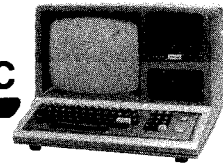
Now that our array is set up, we can begin taking advantage of its built-in structure. For example, suppose we want to add up all the checks written. Add the following lines to the program:

```
150 FOR ROW = 1 TO 6
160 SUM = SUM + CK(ROW,3)
170 NEXT
180 PRINT "TOTAL OF CHECKS WRITTEN":
190 PRINT USING "$####.##"; SUM
```

Now let's add program steps to print out all checks that were written on a given day.

```
200 PRINT "SEEKING CHECKS WRITTEN ON WHAT DATE (MM.DD YY)":
210 INPUT DT
220 PRINT: PRINT "ANY CHECKS WRITTEN ARE LISTED BELOW:"
230 PRINT "CHECK #", "AMOUNT": PRINT
240 FOR ROW = 1 TO 6
250 IF CK(ROW,2) = DT THEN PRINT CK(ROW,1), CK(ROW,3)
260 NEXT
```

It's easy to generalize our program to handle checkbook information for all 12 months and for years other than 1978.



All we do is increase the size (or “depth”) of each dimension as needed. Let’s assume our checkbook includes check numbers 001 through 300, and we want to store the entire checkbook record. Just make these changes:

```
40 DIM CK(300,3)      'SET UP A 300 BY 3 ARRAY
50 FOR ROW = 1 TO 300
```

and add DATA lines for check numbers 001 through 300. You’d probably want to pack more data onto each DATA line than we did in the above DATA lines.

And you’d change all the ROW counter final values:

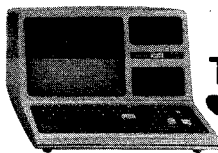
```
150 FOR ROW = 1 TO 300
240 FOR ROW = 1 TO 300
```

## Other Types of Arrays

Remember, in Model III BASIC the number of dimensions an array can have (and the size or depth of the array), is limited only by the amount of memory available. Also remember that **string** arrays can be used. For example, C\$(X) would automatically be interpreted as a string array. And if you use DEFSTR A at the beginning of your program, any array whose name begins with A would also be a string array. One obvious application for a string array would be to store text material for access by a string manipulation program.

```
10 CLEAR 1200
20 DIM TXT$(10)
```

would set up a string array capable of storing 10 lines of text. 1200 bytes were CLEARED to allow for 10 sixty-character lines, plus 600 extra bytes for string manipulation with other string variables.



## Array/Matrix Manipulation Subroutines

To use this subroutine, your main program must supply values for two variables N1 (number of rows) and N2 (number of columns). Within the subroutine, you can assign values to the elements in the array row by row by answering the INPUT statement.

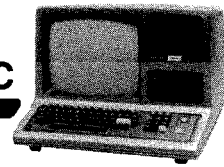
```
10 FOR ROW = 1 TO N1
20 FOR COL = 1 TO N2
30 PRINT "ENTER DATA FOR "; ROW; ":"; COL
40 INPUT A(ROW, COL)
50 NEXT COL
60 NEXT ROW
70 RETURN
```

To use this subroutine, your main program must supply values for three variables N1 (size of dim #1), N2 (size of dim #2) and N3 (size of dim #3). Within the subroutine, you can assign values to each element of the array using READ and DATA statements. You must supply I x J x K elements in the following order: row by row for K = 1, row by row for K = 2, row by row for K = 3, and so on for each value of N3.

```
400 REM REQUIRES DATA STMTS.
410 FOR K = 1 TO N3
420 FOR I = 1 TO N1
430 FOR J = 1 TO N2
440 READ A(I, J, K)
450 NEXT J, I, K
460 RETURN
```

Main program supplies values for variables N1, N2, N3. The subroutine prints the array.

```
560 FOR K = 1 TO N3
570 FOR I = 1 TO N1
580 FOR J = 1 TO N2
590 PRINT A(I, J, K),
600 NEXT J: PRINT
610 NEXT I: PRINT
620 NEXT K: PRINT
630 RETURN
```



Main program supplies values for variables N1, N2, N3. Within the subroutine, you can assign values to each element of the array using the INPUT statement.

```
660 FOR K = 1 TO N3
670 PRINT "PAGE"; K
680 FOR I = 1 TO N1
690 PRINT "INPUT ROW"; I
700 FOR J = 1 TO N2
710 INPUT A(I,J,K)
720 NEXT J
730 NEXT I
740 PRINT: NEXT K
750 RETURN
```

Multiplication by a Single Variable: Scalar Multiplication (3 Dimensional)

```
780 FOR K = 1 TO N3
790 FOR J = 1 TO N2
800 FOR I = 1 TO N1
810 B(I,J,K) = A(I,J,K) * X
820 NEXT I
830 NEXT J
840 NEXT K
850 RETURN
```

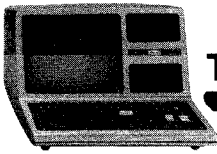
Multiplies each element in MATRIX A by X and constructs matrix B

Transposition of a Matrix (2 Dimensional)

```
880 FOR I = 1 TO N1
890 FOR J = 1 TO N2
900 B(J,I) = A(I,J)
910 NEXT J
920 NEXT I
930 RETURN
```

Transposes matrix A into matrix B





## TRS-80 MODEL III

---

### Matrix Addition (3 Dimensional)

```
960 FOR K = 1 TO N3
970 FOR J = 1 TO N2
980 FOR I = 1 TO N1
990 C(I,J,K) = A(I,J,K) + B(I, J, K)
1000 NEXT I
1010 NEXT J
1020 NEXT K
1030 RETURN
```

### Array Element-wise Multiplication (3 Dimensional)

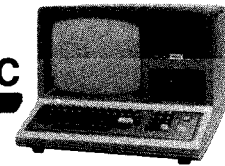
```
1060 FOR K = 1 TO N3
1070 FOR J = 1 TO N2
1080 FOR I = 1 TO N1
1090 C(I,J,K) = A(I,J,K) * B(I,J,K)
1100 NEXT I
1110 NEXT J
1120 NEXT K
1130 RETURN
```

Multiplies each element in A times its corresponding element in B.

### Matrix Multiplication (2 Dimensional)

```
1160 FOR I = 1 TO N1
1170 FOR J = 1 TO N2
1180 C(I,J) = 0
1190 FOR K = 1 TO N3
1200 C(I,J) = C(I,J) + A(I,K) * B(K,J)
1210 NEXT K
1220 NEXT J
1230 NEXT I
1240 RETURN
```

A must be an N1 by N3 matrix; B must be an N3 by N2 matrix. The resultant matrix C will be an N1 and N2 matrix. A, B, and C must be dimensioned accordingly.



## 7/Arithmetic Functions

*Model III BASIC offers a wide variety of intrinsic ('built-in') functions for performing arithmetic and special operations. The special-operation functions are described in the next chapter.*

*All the common math functions described in this chapter return single-precision values **accurate to six decimal places**. ABS, FIX and INT return values whose precision depends on the precision of the argument.*

*The conversion functions (CINT, CDBL, etc.) return values whose precision depends on the particular function. Trig functions use or return radians, not degrees. A radian-degree conversion is given for each of the functions.*

*For all the functions, the argument must be enclosed in parentheses. The argument may be either a numeric variable, expression or constant.*

*Functions described in this chapter:*

ABS	COS	INT	SGN
ATN	CSNG	LOG	SIN
CDBL	EXP	RANDOM	SQR
CINT	FIX	RND	TAN

### ABS(x)

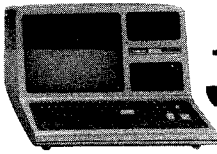
Returns the absolute value of the argument.  $ABS(X) = X$  for  $X$  greater than or equal to zero, and  $ABS(X) = -X$  for  $X$  less than zero.

```
100 IF ABS(X)<1E-6 PRINT "TOO SMALL"
```

### ATN(x)

Returns the arctangent (in radians) of the argument; that is,  $ATN(X)$  returns 'the angle whose tangent is  $X$ '. To get arctangent in degrees, multiply  $ATN(X)$  by 57.29578.

```
100 Y = ATN(B/C)
```



## **CDBL** (*x*)

Returns a double-precision representation of the argument. The value returned will contain 17 digits, but only the digits contained in the argument will be significant.

CDBL may be useful when you want to force an operation to be done in double-precision, even though the operands are single precision or even integers. For example CDBL (I%)/J% will return a fraction with 17 digits of precision.

```
100 FOR I% = 1 TO 25 : PRINT 1/CDBL(I%), : NEXT
```

## **CINT** (*x*)

Returns the largest integer not greater than the argument. For example, CINT (1.5) returns 1; CINT( - 1.5) returns - 2. For the CINT function, the argument must be in the range - 32768 to + 32767. The result is stored internally as a two-byte integer.

CINT might be used to speed up an operation involving single or double-precision operands without losing the precision of the operands (assuming you're only interested in an integer result).

```
100 K% = CINT(X#) + CINT(Y#)
```

## **COS** (*x*)

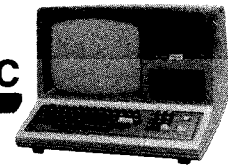
Returns the cosine of the argument (argument must be in radians). To obtain the cosine of X when X is in degrees, use COS(X\*.01745329).

```
100 Y = COS(X + 3.3)
```

## **CSNG** (*x*)

Returns a single-precision representation of the argument. When the argument is a double-precision value, it is returned as six significant digits with "4/5 rounding" in the least significant digit. So CSNG(.6666666666666667) is returned as .666667; CSNG(.3333333333333333) is returned as .333333.

```
100 PRINT CSNG (A# + B#)
```



### EXP(x)

Returns the “natural exponential” of  $X$ , that is  $e^X$ . This is the inverse of the LOG function, so  $X = \text{EXP}(\text{LOG}(X))$ .

```
100 PRINT EXP(-X)
```

### FIX(x)

Returns a truncated representation of the argument. All digits to the right of the decimal point are simply chopped off, so the resultant value is an integer. For non-negative  $X$ ,  $\text{FIX}(X) = \text{INT}(X)$ . For negative values of  $X$ ,  $\text{FIX}(X) = \text{INT}(X) + 1$ . For example,  $\text{FIX}(2.2)$  returns 2, and  $\text{FIX}(-2.2)$  returns -2.

```
100 Y = ABS(A - FIX(A))
```

This statement gives  $Y$  the value of the fractional portion of  $A$ .

### INT(x)

Returns an integer representation of the argument, using the largest whole number that is not greater than the argument. Argument is **not** limited to the range  $-32768$  to  $+32767$ . The result is stored internally as a single-precision whole number.  $\text{INT}(2.5)$  returns 2;  $\text{INT}(-2.5)$  returns -3; and  $\text{INT}(1000101.23)$  returns 1000101.

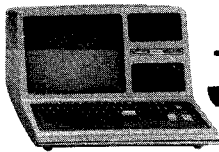
```
100 Z = INT(A*100 + .5)/100
```

Gives  $Z$  the value of  $A$  rounded to two decimal places (for non-negative  $A$ ).

### LOG(x)

Returns the natural logarithm of the argument, that is,  $\log_e(\text{argument})$ . This is the inverse of the EXP function, so  $X = \text{LOG}(\text{EXP}(X))$ . To find the logarithm of a number to another base  $b$ , use the formula  $\text{LOG}_b(X) = \text{LOG}_e(X)/\text{LOG}_e(b)$ . For example,  $\text{LOG}(32767)/\text{LOG}(2)$  returns the logarithm to base 2 of 32767.

```
100 PRINT LOG(3.3*X)
```



## RANDOM

RANDOM is actually a complete statement rather than a function. It reseeds the random number generator. If a program uses the RND function, you may want to put RANDOM at the beginning of the program. This will ensure that you get an unpredictable sequence of pseudo-random numbers each time you turn on the Computer, load the program, and run it.

```
10 RANDOM
20 PRINT RND(0);
30 GOTO 20           ' DO LINE 10 JUST ONCE
```

## RND(*x*)

Generates a pseudo-random number using the current pseudo-random "seed number" (generated internally and not accessible to user). RND may be used to produce random numbers between 0 and 1, or random integers greater than 0, depending on the argument.

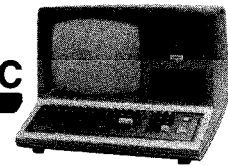
RND(0) returns a single-precision value between 0 and 1. RND(*integer*) returns an integer between 1 and *integer* inclusive (*integer* must be positive and less than 32768). For example, RND(55) returns a pseudo-random integer greater than zero and less than 56. RND(55.5) returns a number in the same range, because RND uses the INTEger value of the argument.

```
100 X = RND(2) : ON X GOTO 200,300
```

## SGN(*x*)

The "sign" function : returns - 1 for X negative, 0 for X zero, and + 1 for X positive.

```
100 ON SGN(X) + 2 GOTO 200,300,400
```

**SIN(*x*)**

Returns the sine of the argument (argument must be in radians). To obtain the sine of *X* when *X* is in degrees, use SIN(*X*\*.01745329).

```
100 PRINT SIN(A*B - B)
```

**SQR(*x*)**

Returns the square root of the argument. SQR(*X*) is the same as  $X^{(1/2)}$ , only faster.

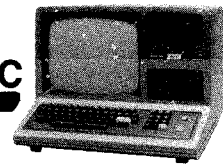
```
100 Y = SQR(X[ 2 - H[ 2)
```

**TAN(*x*)**

Returns the tangent of the argument (argument must be in radians). To obtain the tangent of *X* when *X* is in degrees, use TAN(*X*\*.01745329).

```
100 Z = TAN(2*A)
```

**NOTE:** A great many other functions may be created using the above functions. See Appendix E, "Derived Functions".



## 8/Special Features

*Model III BASIC offers some unusual functions and operations that deserve special highlighting. Some may seem highly specialized; as you learn more about programming and begin to experiment with machine-language routines, they will take on more significance. Other functions in the chapter are of obvious benefit and will be used often (for example, the graphics functions).*

*Functions, statements and operators described in this chapter:*

<b>Graphics:</b>	<b>Error-Routine Functions:</b>	<b>Other Functions and Statements:</b>
SET	ERL	INP
RESET	ERR	MEM
CLS		OUT
POINT		PEEK
		POKE
		POS
		USR
		VARPTR

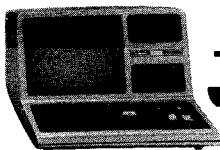
### **SET**(*x,y*)

Turns on the graphics block at the location specified by the coordinates *x* and *y*. For graphics purposes, the Display is divided up into a 128 (horizontal) by 48 (vertical) grid. The *x*-coordinates are numbered from left to right, 0 to 127. The *y*-coordinates are numbered from top to bottom, 0 to 47. Therefore the point at (0,0) is in the extreme upper left of the Display, while the point at (127,47) is in the extreme lower right corner. See the Video Display Worksheet in the Appendix.

The arguments *x* and *y* may be numeric constants, variables or expressions. They need not be integer values, because SET(*x,y*) uses the INTEger portion of *x* and *y*. SET(*x,y*) is valid for:

$$0 \leq x < 128$$

$$0 \leq y < 48$$



## TRS-80 MODEL III

---

### Examples:

```
100 SET (RND(128) - 1, RND(48) - 1)
```

Lights up a random point on the Display.

```
100 INPUT X,Y: SET (X,Y)
```

RUN to see where the blocks are.

### RESET (x,y)

Turns off a graphics block at the location specified by the coordinates  $x$  and  $y$ . This function has the same limits and parameters as SET( $x,y$ ).

```
200 RESET (X,3)
```

### CLS

“Clear-Screen” — turns off all the graphics blocks on the Display and moves the cursor to the upper left corner. This wipes out alphanumeric characters as well as graphics blocks. CLS is very useful whenever you want to present an attractive Display output.

```
10 CLS  
20 SET (RND(128) - 1, RND(48) - 1)  
30 GOTO 20
```

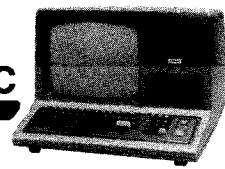
### POINT(x,y)

Tests whether the specified graphics block is “on” or “off”. If the block is “on” (that is, if it has been SET), then POINT returns a binary True (-1 in Model III BASIC). If the block is “off”, POINT returns a binary False (0 in Model III BASIC). Typically, the POINT test is put inside an IF-THEN statement.

```
100 SET (50,28) : IF POINT (50,28) THEN PRINT "ON" ELSE PRINT "OFF"
```

This line will always print the message, “ON”, because POINT(50,28) will return a binary True, so that execution proceeds to the THEN clause. If the test failed, POINT would return a binary False, causing execution to jump to the ELSE statement.





## ERL

Returns the line number in which an error has occurred. This function is primarily used inside an error-handling routine accessed by an ON ERROR GOTO statement. If no error has occurred when ERL is called, line number 0 is returned. However, if an error has occurred since power-up, ERL returns the line number in which the error occurred. If error occurred in direct mode, 65535 is returned (largest number representable in two bytes).

### Example Program using ERL

```
10 CLEAR 10
20 ON ERROR GOTO 1000
30 INPUT "ENTER YOUR MESSAGE": M$
40 INPUT "NOW ENTER A NUMBER": N
50 Z = 1/N
60 PRINT "INPUT VALUES OKAY--TRY AGAIN TO CAUSE AN ERROR"
70 GOTO 30
1000 IF ERL=30 AND (ERR/2 + 1 = 14) THEN 1040
1010 IF ERL=40 AND (ERR/2 + 1 = 6) THEN 1050
1020 IF ERL=50 AND (ERR/2 + 1 = 11) THEN 1060
1030 ON ERROR GOTO 0: RESUME
1040 PRINT "MESSAGE TOO LONG--10 LETTERS MAXIMUM": RESUME
1050 PRINT "NUMBER TOO LARGE": RESUME
1060 PRINT "DIVISION BY ZERO IN LINE 50--ENTER NON-ZERO NUMBER"
1070 RESUME 40
```

RUN the program. Try entering a long message; try entering zero when the program asks for a number. Note that ERL is used in line 1000 to determine where the error occurred so that appropriate action may be taken.

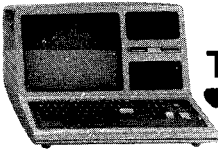
## ERR/2 + 1

Similar to ERL, except ERR returns a value **related** to the **code** of the error rather than the line in which the error occurred. It is commonly used inside an error handling routine accessed by an ON ERROR GOTO statement. See Appendix B, "Error Codes."

$ERR/2 + 1 = \text{true error code}$   
 $(\text{true error code} - 1) * 2 = ERR$

### Sample Program

See ERL.



## TRS-80 MODEL III

---

### **INP** (*port*)

Returns a byte-value from the specified port. There are 256 ports, numbered 0-255.  
For example

```
100 PRINT INP(50)
```

inputs a byte from port 50 and prints the decimal value of the byte.

You do **not** need to access the Z-80 ports to make full use of the TRS-80.

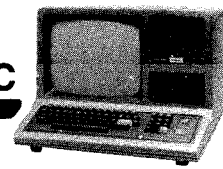
### **MEM**

Returns the number of unused and unprotected bytes in memory. This function may be used in the Immediate Mode to see how much space a resident program takes up; or it may be used inside the program to avert OM (Out of Memory) errors by allocating less string space, DIMensioning smaller array sizes, etc. MEM requires no argument.

#### **Example:**

```
100 IF MEM < 80 THEN 900
```

Enter the command PRINT MEM (in the Immediate Mode) to find out the amount of memory not being used to store programs, variables, strings, stack, or reserved for object-files.



## **OUT** *port, value*

Outputs a byte value to the specified port. OUT is not a function but a statement complete in itself. It requires two arguments separated by a comma (no parenthesis): the port destination and the byte value to be sent. *port* and *value* are in the range 0 to 255.

## **PEEK**(*address*)

Returns the value stored at the specified byte address (in decimal form). To use this function, you'll need to refer to two sections of the Appendix: the Memory Map (so you'll know where to PEEK) and the Table of Function ASCII and Graphics Codes (so you'll know what the values represent).

If you're using PEEK to examine object files, you'll also need a microprocessor instruction set manual (one is included with the TRS-80 Editor/Assembler Instruction Manual).

PEEK is valuable for linking machine language routines with Model III BASIC programs. The machine language routine can store information in a certain memory location, and PEEK may be used inside your BASIC program to retrieve the information. For example,

```
A = PEEK (17999)
```

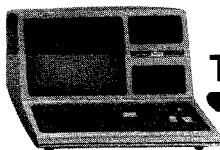
returns the value stored at location 17999 and assigns that value to the variable A.

Peek may also be used to retrieve information stored with a POKE statement. Using PEEK and POKE allows you to set up very compact, byte-oriented storage systems. Refer to the Memory Map in the Appendix to determine the appropriate locations for this type of storage. See **POKE, USR**.

## **POKE** *address, value*

Loads a value into a specified memory location. POKE is not a function but a statement complete in itself. It requires two arguments: a byte address (in decimal form) and a value. The value must be between 0 and 255 inclusive. Refer to the Memory Map in the Appendix to see which addresses you'd like to POKE.

To POKE (or PEEK) an address **above** 32767, use the following formula:  $-1 * (65536 - \text{desired address}) = \text{POKE OR PEEK address}$ . For example, to POKE into address 32769, use POKE  $-32767$ , *value*.



## TRS-80 MODEL III

---

Since the Video Display is memory-mapped, you can output to the Display directly by POKEing ASCII data into Video RAM. Video RAM is from 15360 to 16383.

**Example:**

```
10 CLS
20 FOR M = 15360 TO 16383
30 POKE M,191
40 NEXT M
50 GOTO 50
```

RUN the program to see how fast the screen is "painted" white.

Since POKE can be used to store information anywhere in memory, it is very important when we do our graphics to stay in the range for display locations. If we POKE outside this range, we may store the byte in a critical place. We could be POKEing into our program, or even in worse places like the stack. Indiscriminate POKEing can be disastrous. You might have to reset or power off and start over again. Unless you know where you are POKEing — don't.

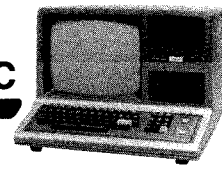
See PEEK, USR, SET, and CHR\$ for background material. Also see the Owners Section for examples on special uses of POKE.

### POS(x)

Returns a number from 0 to 63 indicating the current cursor position on the Display. Requires a "dummy argument" (any numeric expression).

```
100 PRINT TAB(40);POS(0)
```

prints 40 at position 40. (Note that a blank is inserted before the "4" to accommodate the sign; therefore the "4" is actually at position 41.) The "0" in "POS(0)" is the dummy argument.



## USR (x)

This function lets you call a machine-language subroutine and then continue execution of your BASIC program.

“Machine language” is the low-level language used internally by your Computer. It consists of Z-80 microprocessor instructions. Machine-language subroutines are useful for special applications (things you can’t do in BASIC) and simply because they can do things very fast (like white-out the Display).

Writing such routines requires familiarity with assembly-language programming and with the Z-80 instruction set. For more information on this subject, see the Radio Shack book, *TRS-80 Assembly-Language Programming*, by William Barden, Jr., and the instruction manual for Radio Shack’s EDITOR-ASSEMBLER (26-2002).

### Getting the USR routine into memory

1. You should first reserve the area in high memory where the routine will be located. This is done immediately after power-up by answering the MEMORY SIZE? question with the address **preceding the start address of your USR routine**. For example, if your routine starts at 32700, then type 32699 in response to MEMORY SIZE?.
2. Then load the routine into memory.
  - A. If it is stored on tape in the SYSTEM format (created with EDITOR-ASSEMBLER), you must load it via the SYSTEM command, as described in Chapter 2. After the tape has loaded press **(BREAK)** to return to the BASIC immediate mode.
  - B. If it is a short routine, you may simply want to POKE it into high memory.

### Telling BASIC where the USR routine starts

Before you can make the USR call, you have to tell BASIC the entry address to the routine. Simply POKE the two-byte address into memory locations 16526-16527: least significant byte (LSB) into 16526, most significant byte (MSB) into 16527.

For example, if the entry point is at 32700:

32700 decimal = 7FBC hexadecimal

LSB = BC hexadecimal = 188 decimal

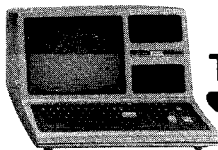
MSB = 7F hexadecimal = 127 decimal

So use the statements:

```
POKE 16526, 188
```

```
POKE 16527, 127
```

to tell BASIC that the USR routine entry is at 32700.



### Making the USR call

At the point in your BASIC program where you want to call the subroutine, insert a statement like

```
X = USR(N)
```

where N can be an expression and must have a value between - 32768 and + 32767 inclusive. This argument, N, can be used to pass a value to your routine (see below) or you can simply consider it a dummy argument and not use it at all.

When BASIC encounters your X = USR(N) statement, it will branch to the address stored at 16526-16527. At the point in your USR routine where you want to **return** to the BASIC program, insert a simple RET instruction — unless you want to return a value to BASIC, in which case, see below.

### Passing an argument to the USR routine

If you want to pass the USR(N) argument to your routine, then include the following CALL instruction at the **beginning** of your USR routine.:

```
CALL 0A7FH
```

This loads the argument N into the HL register pair as a two-byte signed integer.

### Returning an argument from the USR routine

To return an integer value to the USR(N) function, load the value (a two-byte signed integer) into HL and place the following jump instruction at the end of your routine:

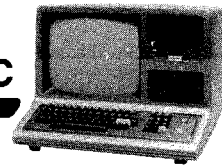
```
JP 0A9AH
```

Control will pass back to your program, and the integer in HL will replace USR(N). For example, if the call was

```
X = USR(N)
```

Then X will be given the value in HL.

USR routines are automatically allocated up to 8 stack levels or 16 bytes (a high and low memory byte for each stack level). If you need more stack space, you can save the BASIC stack pointer and set up your own stack. See **SYSTEM**, **PEEK**, and **POKE**. Also see the Technical Information Chapter in the Owners Section.



## VARPTR (*variable name*)

Returns an address-value which will help you locate where the variable name and its value are stored in memory. If the variable you specify has not been assigned a value, an FC error will occur when this function is called.

If VARPTR(*integer variable*) returns address K:

Address K contains the least significant byte (LSB) of 2-byte *integer*.

Address K + 1 contains the most significant byte (MSB) of *integer*.

You can display these bytes (two's complement decimal representation) by executing a PRINT PEEK (K) and a PRINT PEEK (K + 1).

If VARPTR(*single precision variable*) returns address K:

(K)\* = LSB of value

(K + 1) = Next most significant byte (Next MSB)

(K + 2) = MSB with hidden (implied) leading one. Most significant bit is the sign of the number

(K + 3) = exponent of value excess 128 (128 is added to the exponent).

If VARPTR(*double precision variable*) returns K:

(K) = LSB of value

(K + 1) = Next MSB

(K + ...) = Next MSB

(K + 6) = MSB with hidden (implied) leading one. Most significant bit is the sign of the number.

(K + 7) = exponent of value excess 128 (128 is added to the exponent).

For single and double precision values, the number is stored in normalized exponential form, so that a decimal is assumed before the MSB. 128 is added to the exponent. Furthermore, the high bit of MSB is used as a sign bit. It is set to 0 if the number is positive or to 1 if the number is negative. See examples below.

You can display these bytes by executing the appropriate PRINT PEEK(*x*) where *x* = the address you want displayed. Remember, the result will be the decimal representation of byte, with bit 7 (MSB) used as a sign bit. The number will be in normalized exponential form with the decimal assumed before the MSB. 128 is added to the exponent,

If VARPTR(*string variable*) returns K:

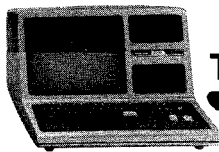
K = length of string

(K + 1) = LSB of string value starting address

(K + 2) = MSB of string value starting address

\* (K) signifies "contents of address K"

The address will probably be in high RAM where string storage space has been set aside. But, if your string variable is a constant (a string literal), then it will point to the area of memory where the program line with the constant is stored, in the program buffer area. Thus, program statements like A\$ = "HELLO" do not use string storage space.



## TRS-80 MODEL III

For all of the above variables, addresses  $(K - 1)$  and  $(K - 2)$  will store the TRS-80 Character Code for the variable name. Address  $(K - 3)$  will contain a descriptor code that tells the Computer what the variable type is. Integer is 02; single precision is 04; double precision is 08; and string is 03.

$\text{VARPTR}(\text{array variable})$  will return the address for the first byte of that element in the array. The element will consist of 2 bytes if it is an integer array; 3 bytes if it is a string array; 4 bytes if it is a single precision array; and 8 bytes if it is a double precision array.

The first element in the array is preceded by:

1. A sequence of two bytes per dimension, each two-byte pair indicating the "depth" of each respective dimension.
2. A single byte indicating the total number of dimensions in the array.
3. A two-byte pair indicating the total number of elements in the array.
4. A two-byte pair containing the ASCII-coded array name.
5. A one-byte type-descriptor (02 = Integer, 03 = String, 04 = Single-Precision, 08 = Double-Precision).

Item (1) immediately precedes the first element, Item (2) precedes Item (1), and so on.

The elements of the array are stored sequentially with the first dimension-subscripts varying "fastest", then the second, etc.

### Examples:

$A! = 2$  will be stored as follows

$2 = 10$  Binary, represented as  $.1E2 = .1 \times 2^2$

So exponent of A is  $128 + 2 = 130$  (called excess 128)

MSB of A is 10000000;

however, the high bit is changed to zero since the value is positive (called hidden or implied leading one).

So  $A!$  is stored as

Exponent $(K + 3)$	MSB $(K + 2)$	Next MSB $(K + 1)$	LSB $(K)$
130	0	0	0

$A! = -.5$  will be stored as

Exponent $(K + 3)$	MSB $(K + 2)$	Next MSB $(K + 1)$	LSB $(K)$
128	128	0	0

$A! = 7$  will be stored as

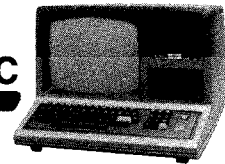
Exponent $(K + 3)$	MSB $(K + 2)$	Next MSB $(K + 1)$	LSB $(K)$
131	96	0	0

$A! = -7$ :

Exponent $(K + 3)$	MSB $(K + 2)$	Next MSB $(K + 1)$	LSB $(K)$
131	224	0	0

Zero is simply stored as a zero-exponent. The other bytes are insignificant.





## 9/Editing

*You have probably found it is very time consuming to retype long program lines, simply because of a typo, or maybe just to make a small change.*

*Model III editing features eliminate much of this extra work. In fact, it's so easy to alter program lines, you'll probably want to experiment with multi-statement lines, complex expressions, etc.*

*Commands, subcommands, and special function keys described in this chapter:*

EDIT	(L)	n(D)
(ENTER)	(X)	n(C)
n(SPACEBAR)	(I)	n(S)c
n ←	(A)	n(K)c
(SHIFT) ↑	(E)	
	(Q)	
	(H)	

### EDIT *line number*

This command puts you in the Edit Mode. You must specify which line you wish to edit, in one of two ways:

EDIT <i>line-number</i>	Lets you edit the specified line. If line number is not in use, an FC error occurs
or	
EDIT.	Lets you edit the current program line — last line entered or altered or in which an error has occurred.

For example, type in and (ENTER) the following line:

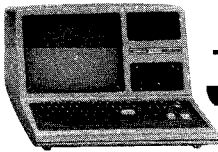
```
100 FOR I = 1 TO 10 STEP .5 : PRINT I, I [ 2, I [ 3 : NEXT
```

This line will be used in exercising all the Edit subcommands described below.

Now type EDIT 100 and hit (ENTER). The Computer will display:

```
100■
```

You are now in the Edit Mode and may begin editing line 100.



## TRS-80 MODEL III

---

**NOTE:** EDITing a program line automatically clears all variable values and eliminates pending FOR/NEXT and GOSUB operations. If BASIC encounters a syntax error during program execution, it will automatically put you in the EDIT mode. Before EDITing the line, you may want to examine current variable values. In this case, you must type Q as your first EDIT command. This will return you to the command mode, where you may examine variable values. Any other EDIT command (typing E, pressing ENTER, etc.) will clear out all variables.

### **ENTER** key

Hitting **ENTER** while in the Edit Mode causes the Computer to record all the changes you've made (if any) in the current line, and returns you to the Command Mode.

### **n** **SPACEBAR**

In the Edit Mode, hitting the Space-bar moves the cursor over one space to the right and displays any character stored in the preceding position. For example, using line 100 entered above, put the Computer in the Edit Mode so the Display shows:

```
100 ■
```

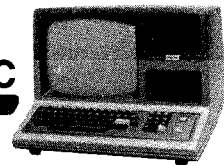
Now hit the Space-Bar. The cursor will move over one space, and the first character of the program line will be displayed. If this character was a blank, then a blank will be displayed. Hit the Space-Bar until you reach the first non-blank character:


```
100 F ■
```

is displayed. To move over more than one space at a time, hit the desired number of spaces first, and then hit the space-bar. For example, type 5 and hit Space-bar, and the display will show something like this (may vary depending on how many blanks you inserted in the line):

```
100 FOR I = ■
```


Now type 8 and hit the Space-bar. The cursor will move over 8 spaces to the right, and 8 more characters will be displayed.



**n** 


Moves the cursor to the left by  $n$  spaces. If no number  $n$  is specified, the cursor moves back one space. When the cursor moves to the left, all characters in its "path" are erased from the display, but **they are not deleted from the program line**. Using this in conjunction with D or K or C can give misleading Video Displays of your program lines. So, be careful using it! For example, assuming you've used  $n$ Space-Bar so that the Display shows:

```
100 FOR I = 1 TO 10 ■
```

type 8 and hit the  key. The display will show something like this:

```
100 FOR I = ■          (will vary depending on number of blanks in
                        your line 100)
```

**SHIFT** 

Hitting SHIFT and  keys together effects an escape from any of the Insert subcommands: X, I and H. After escaping from an Insert subcommand, you'll still be in the Edit Mode, and the cursor will remain in its current position. (Hitting **ENTER** is another way to exit these Insert subcommands).

## L (List Line)

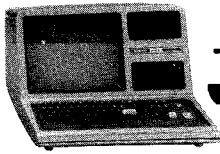
When the Computer is in the Edit Mode, and is not currently executing one of the subcommands below, hitting L causes the remainder of the program line to be displayed. The cursor drops down to the next line of the Display, reprints the current line number, and moves to the first position of the line. For example, when the Display shows

```
100 ■
```

hit L (without hitting **ENTER** key) and line 100 will be displayed:

```
100 FOR I = 1 TO 10 STEP .5 : PRINT I, I [ 2, I [ 3 : NEXT
100 ■
```

This lets you look at the line in its current form while you're doing the editing.



## X (Extend Line)

Causes the rest of the current line to be displayed, moves cursor to end of line, and puts Computer in the Insert subcommand mode so you can add material to the end of the line. For example, using line 100, when the Display shows

```
100 ■
```

hit X (without hitting **ENTER**) and the entire line will be displayed; notice that the cursor now follows the last character on the line:

```
100 FOR I = 1 TO 10 STEP .5 : PRINT I, I [ 2, I [ 3 : NEXT ■
```

We can now add another statement to the line, or delete material from the line by using the **X** key. For example, type :PRINT "DONE" at the end of the line. Now hit **ENTER**. If you now type LIST 100, the Display should show something like this:

```
100 FOR I = 1 TO 10 STEP .5 : PRINT I, I [ 2, I [ 3 : NEXT : PRINT "DONE"
```

## I (Insert)

Allows you to insert material beginning at the current cursor position on the line. (Hitting **I** will actually delete material from the line in this mode.) For example, type and **ENTER** the EDIT 100 command, then use the Space Bar to move over to the decimal point in line 100. The Display will show:

```
100 FOR I = 1 TO 10 STEP . ■
```

Suppose you want to change the increment from .5 to .25. Hit the I key (don't hit **ENTER**) and the Computer will now let you insert material at the current position. Now hit 2 so the Display shows:

```
100 FOR I = 1 TO 10 STEP .2 ■
```

You've made the necessary change, so hit **SHIFT** **↑** to escape from the Insert Subcommand. Now hit L key to display remainder of line and move cursor back to the beginning of the line:

```
100 FOR I = 1 TO 10 STEP .25 : PRINT I, I [ 2, I [ 3 : NEXT : PRINT "DONE"  
100 ■
```

You can also exit the Insert subcommand and save all changes by hitting **ENTER**. This will return you to Command mode.



## A (Cancel and Start Again)

Moves the cursor back to the beginning of the program line and cancels editing changes already made. For example, if you have added, deleted, or changed something in a line, and you wish to go back to the beginning of the line and cancel the changes already made: first hit SHIFT **⏏** (to escape from any subcommand you may be executing); then hit A. (The cursor will drop down to the next line, display the line number and move to the first program character.)

## E (Exit)

Causes Computer to end editing and save all changes made. You must be in Edit Mode, not executing any subcommand, when you hit E to end editing.

## Q (Quit)

Tells Computer to end editing and cancel all changes made in the current editing session. If you've decided not to change the line, type Q to cancel changes and leave Edit Mode.

## H (Hack)

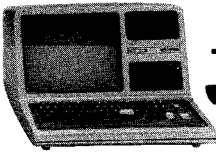
Tells Computer to delete remainder of line and lets you insert material at the current cursor position. Hitting **H** will actually delete a character from the line in this mode. For example, using line 100 listed above, enter the Edit Mode and space over to the last statement, PRINT "DONE". Suppose you wish to delete this statement and insert an END statement. Display will show:

```
100 FOR I = 1 TO 10 STEP .25 : PRINT I, I [ 2, I [ 3 : NEXT : ■
```

Now type H and then type END. Hit **ENTER** key. List the line:

```
100 FOR I = 1 TO 10 STEP .25 : PRINT I, I [ 2, I [ 3 : NEXT : END
```

should be displayed.



### ***nD* (Delete)**

Tells Computer to delete the specified number *n* characters to the right of the cursor. The deleted characters will be enclosed in exclamation marks to show you which characters were affected. For example, using line 100, space over to the PRINT command statement:

```
100 FOR I = 1 TO 10 STEP .25 :■
```

Now type 19D. This tells the Computer to delete 19 characters to the right of the cursor. The display should show something like this:

```
100 FOR I = 1 TO 10 STEP .25 : !PRINT I, I | 2, I | 3 : !■
```

When you list the complete line, you'll see that the PRINT statement has been deleted.

### ***nC* (Change)**

Tells the Computer to let you change the specified number of characters beginning at the current cursor position. If you type C without a preceding number, the Computer assumes you want to change one character. When you have entered *n* number of characters, the Computer returns you to the Edit Mode (so you're not in the *nC* Subcommand). For example, using line 100, suppose you want to change the final value of the FOR-NEXT loop, from "10" to "15". In the Edit Mode, space over to just before the "0" in "10".

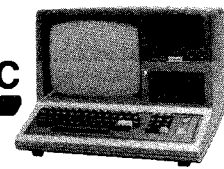
```
100 FOR I = 1 TO 1■
```

Now type C. Computer will assume you want to change just one character. Type 5, then hit L. When you list the line, you'll see that the change has been made.

```
100 FOR I = 1 TO 15 STEP .25 : NEXT : END
```

would be the current line if you've followed the editing sequence in this chapter.

The  $\leftarrow$  does not work as a backspace under the C command in Edit mode. Instead, it replaces the character you want to change with a backspace. So it should not be used. If you make a mistake while typing in a change, Edit the line again to correct it, instead of using  $\leftarrow$ .



### ***nSc* (Search)**

Tells the Computer to search for the *n*th occurrence of the character *c*, and move the cursor to that position. If you don't specify a value for *n*, the Computer will search for the first occurrence of the specified character. If character *c* is not found, cursor goes to the end of the line. Note: The Computer only searches through characters to the right of the cursor.

For example, using the current form of line 100, type EDIT 100 ( **ENTER** ) and then hit 2S: . This tells the Computer to search for the second occurrence of the colon character. Display should show:

```
100 FOR I = 1 TO 15 STEP .25 : NEXT ■
```

You may now execute one of the subcommands beginning at the current cursor position. For example, suppose you want to add the counter variable after the NEXT statement. Type I to enter the Insert subcommand, then type the variable name, I. That's all you want to insert, so hit SHIFT (↑) to escape from the Insert subcommand. The next time you list the line, it should appear as:

```
100 FOR I = 1 TO 15 STEP .25 : NEXT I : END
```

### ***nKc* (Kill)**

Tells the Computer to delete all characters up to the *n*th occurrence of character *c*, and move the cursor to that position. For example, using the current version of line 100, suppose we want to delete the entire line up to the END statement. Type EDIT 100 ( **ENTER** ), and then type 2K:. This tells the Computer to delete all characters up to the 2nd occurrence of the colon. Display should show:

```
100 !FOR I = 1 TO 15 STEP .25 : NEXT !! ■
```

The second colon still needs to be deleted, so type D. The Display will now show:

```
100 !FOR I = 1 TO 15 STEP .25 : NEXT !!!:! ■
```

Now hit **ENTER** and type LIST 100 ( **ENTER** ).

Line 100 should look something like this:

```
100 END
```