SAMS

22246

TI-99/4A BASIC Reference Manual

Carol Ann Casciato
and Donald J. Horsfall

TI-99/4A BASIC
Reference Manual

Carol Ann Casciato and **Donald J. Horsfall** are the principals in International Technical Communications, Inc., a computer systems research and consulting firm in the Philadelphia area. For the last 12 years, they have done management and systems consulting, research, and writing for a variety of Fortune 500 clients.

Their first exposure to professional writing came when they produced more than 25 manuals for a large technical documentation project. Since that time, they have written in-depth computer industry research reports, detailed technical product analyses, and manuals for microcomputer manufacturers.

When not reading, writing, or consulting on computers, Carol Ann prepares elaborate chocolate desserts and cares for her large collection of exotic plants. Don's interests include science fiction, restoring his Victorian home, and collecting space art.

TI-99/4A BASIC
Reference Manual

Carol Ann Casciato
and
Donald J. Horsfall

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

FIRST EDITION
FIRST PRINTING—1984

International Standard Book Number: 0-672-22246-9
Library of Congress Catalog Card Number: 83-51181

Edited by: *Don Silengo*
Illustrated by: *D. B. Clemons*

Printed in the United States of America.

Preface

This is a TI-99/4A BASIC Language Reference Manual. We wrote this book to make the TI BASIC commands and statements more easily accessible to you (and to us). The organization is intended to make programming the TI-99/4A in BASIC a more pleasant, rewarding, and productive experience for you.

Chapter 1 is an introduction to BASIC, with a short review of the expanded capabilities offered by Extended BASIC.

In Chapters 2 and 3 you will find a complete investigation of TI BASIC, the rules for formulation of expressions, and information on file processing.

But, by far the largest and the most important part of this book is Chapter 4, the reference section. This chapter contains detailed descriptions of all the commands and statements in TI BASIC. They are arranged in alphabetical order with complete examples and error analysis.

There are more than 130 sample programs included to illustrate the use of the TI BASIC commands and statements. We encourage you to enter and run the example programs on any BASIC instructions that you may be having trouble understanding. Take the program as presented and make changes to it, some of which we suggest to you. You will know you understand what is going on when you can successfully alter a program to make it do what you want.

If you are interested in the internal workings of the TI BASIC interpreter, Chapter 5 provides some technical insight into what is going on when you run a BASIC program.

Finally, the Appendices provide a quick reference to detailed information required by many BASIC instructions. This includes color and sound tables, derived trigonometric functions, and, in the last Appendix, error analysis designed to help you recognize and correct the programming errors that inevitably appear in even the best programmer's code.

CAROL ANN CASCIATO
DONALD J. HORSFALL

We thank Anna Marie Casciato-Sekerke for preparing the figures and for testing some of the sample programs in this book.

Contents

APPENDIX B

APPENDIX C

APPENDIX D

APPENDIX E

APPENDIX F

APPENDIX G

APPENDIX H

APPENDIX I

APPENDIX J

CHAPTER 1

Introduction to TI BASIC

This chapter gives you a brief history of the BASIC language and tells why it's a good language for home computers and beginning programmers.

We tell you about the TI BASIC that is in the TI-99/4A ROM and how you can easily write programs that use the TI sound and graphics features. We point out where you reach the limits of TI BASIC and tell you (briefly) what you can do with Extended BASIC.

BASIC BACKGROUND

BASIC, Beginners All Purpose Symbolic Instruction Code, is the programming language most often used on home computers today. Like any programming language, BASIC is a set of instructions and rules that are used to tell your computer what to do.

Virtually all home computers come with BASIC as their standard programming language. BASIC is relatively easy to learn, yet powerful enough that you can write substantial programs in it. It is usually built into the machine in the form of a ROM-based (Read Only Memory) BASIC interpreter. This makes BASIC a permanent part of the computer and it means that you need no other parts (like a disk drive) to write, or to run programs written in BASIC.

Why BASIC? Why not *LOGO* or *C* or *PL/1* or *FORTRAN?* The answer is the same as you will find in many areas of computer standardization. BASIC was in the right place at the right time.

BASIC was developed at Dartmouth College for students learning to program on a time-sharing mainframe computer. Because it was designed to be mainly a teaching tool, it was very strong in error detection and diagnosis. Since it had to support many students simultaneously, it was kept simple and was implemented as an *interpreter* rather than as a *compiler.* An interpreter is a good deal easier to write and much easier to use than a compiler.

Along came microcomputers. They lacked the large quantities of memory, the sophisticated software development tools, and the installed base of software. What they needed was a language that was already in use, easy to implement, strong on error detection, and easy to use. My goodness! Sounds just like BASIC. And so it was.

BASIC became the standard language for microcomputers. Unfortunately, no fundamental standards for the language were agreed upon before a bewildering variety of dialects emerged, each claiming that its particular extensions or modifications made its version more powerful, useful, user-friendly, or whatever.

Only after most of the dust had settled did any officious body—in this case ANSI (American National Standards Institute)—bother adopting a standard for the BASIC language, *ANSI Minimal BASIC*. Texas Instruments, Inc., has adopted this standard in the TI BASIC that you have in your TI-99/4A Home Computer.

This is a pretty good standard. It has much to recommend it. It's consistent, allows the use of long variable names, it includes powerful program control statements, and it eliminates most of the PEEK and POKE nonsense that plagues nearly all other home computer implementations of BASIC.

For the beginning programmer, TI BASIC is superior to most home computer implementations of BASIC. In TI BASIC, you access facilities such as color graphics, sound, and voice synthesis through the CALL statement. For example, to change the color of the screen, you CALL SCREEN(color-code). On many home computers you would have to POKE(obscure-address,color-code), which makes no sense to anyone.

There is much debate over whether BASIC is the best language for use in the home computer. There are probably better languages that are easier to learn and more appropriate to the home computer. But, these languages generally require much more hardware to support them. Some authorities prefer languages such as LOGO, which are based on artificial intelligence concepts. LOGO is a fine language, especially for children and beginners. LOGO, though, requires extended memory and the LOGO command module. This is a minimum of several hundred dollars more than the cost of the TI-99/4A.

So BASIC wins for the same reason it was adopted in the first place—it's inexpensive and easy to implement, requires minimal machine resources, and has a large body of software already written for it.

TI BASIC FEATURES

The standard TI BASIC that comes in your TI-99/4A console conforms to the American National Standard Institute for Minimal BASIC. Plus, TI BASIC offers features beyond the minimal standard, such as:

- *Color graphics*—you can control up to 16 colors and can create user-definable characters.

- *Sound*—you can control the duration (.001 to 4.25 seconds), volume, and the frequency of three independent tones, plus eight *periodic* or *white* noises. The frequency varies from 110 to 44,733 hertz (Hz) for tones.
- *Joystick control*—you determine the position of the joystick levers and the condition of the joystick *fire buttons* (pressed or not pressed).
- *Special keyboard scanning routines*—you can trap control character codes and/or split the keyboard into two (right and left) sections for multiple control (as in a two player game).
- *Special screen control and graphics routines*—you can easily define new characters, set character colors, and write at specific screen positions.
- *Arrays*—you can allocate arrays with up to *three* dimensions.
- *Line Editor*—built-in and easy to use and includes automatic line numbering and resequencing

These are only some of the excellent features available in standard TI BASIC. TI BASIC represents a good BASIC language set. It is more than sufficient for solving beginning programming problems.

TI BASIC programs, however, cannot use more than the 16K of memory that comes in the TI-99/4A console. To write programs larger than 16K, you must use Extended BASIC.

EXTENDED BASIC

In this book, we are looking only at the TI BASIC that comes in the TI-99/4A console. Texas Instruments offers another version of BASIC in a command module called Extended BASIC.

While the TI BASIC that comes with the TI-99/4A is a very good version of BASIC, you may find that you need an even more powerful version. Texas Instruments developed their Extended BASIC language to take full advantage of many of the sophisticated features in the TI-99/4A Home Computer.

One important feature of Extended BASIC is that you can use the 32K Memory Expansion card. Programs that are stored and run in this memory can be larger, and often execute faster, than those run from the standard 16K RAM in the TI-99/4A console.

Extended BASIC can access up to 48K, the 16K that comes in the TI-99/4A and the 32K from the Memory Expansion card. After everything is taken out, you end up with about 36,000 bytes of program and data space under Extended BASIC. This doesn't include the approximately 8000 bytes available for Assembler programs that you can link to from Extended BASIC.

Extended BASIC comes in a cartridge and offers these enhancements to the standard TI BASIC:

- *Sprites*—you can define up to 28 independent graphic figures that can move around on the screen.

- *Speech*—your BASIC programs can speak through the Speech Synthesizer Peripheral.
- *Extended memory support*—you can use more than the 16K RAM that comes in your computer. You can access up to 48K with the 32K Memory Expansion card.
- *Multiple statements on one line*—makes it easier to enter programs, saves space since only the first statement on a line needs a line number, and results in faster execution of the program.
- *More functions*—such as MAX, MIN, and PI.
- *Arrays*—you can allocate arrays with up to seven dimensions (increased from three).
- *Error handling*—you control error or warning conditions and take appropriate action within your program.
- *IF . . . THEN . . . ELSE enhancements*—you can enter multiple statements after the THEN and ELSE keywords, instead of only line numbers.
- *Assembly language subroutine support*—you can load and link to TMS9900 Assembly language routines. You'll need the Editor/Assembler to enter Assembly language statements.
- *Named subroutines with local variables and passed parameters*—make it easier for you to write programs using the most modern structured programming methods.
- *Enhanced input/output statements*—including formatted printing and cursor positioning control.
- *Merging of programs*—allows you to store commonly used routines on disk and merge them automatically into new programs as you write them.

CHAPTER 2

Data in BASIC

In this chapter, we are going to explore data and data manipulation. We look at how to define data items, what they can contain, and how to relate one data item to another.

INTRODUCTION

Data is what a program is written to process. Data can represent something real, like your birthdate, or the value of Pi, or the current balance in your savings account. Data can also contain something of value, but only within the confines of your program. For example, which line number to go to in the next ON . . . GOTO statement, or the value of a loop variable, or the column to tab to in the report that you're writing.

DATA TYPES

Whatever use it is put to, and however it relates to the *real world,* data comes in two types:

- Numeric data
- Character (or string) data

Obviously, numeric data represents numbers—numbers of all sorts. And only numbers, in whatever format.

String data, on the other hand, can contain: numbers, upper-case letters, lower-case letters, special characters, and punctuation marks.

In addition to these simple data types, TI BASIC allows you to define a collection, or list of items of the same data type, in a group data element called an *array.* You can then access the individual data items in the array through the same array name.

Numeric Data

In TI BASIC, numeric data comes in only one type:

- Floating point

The floating point numeric format can store any kind of number, including whole (integer) numbers, numbers with fractional parts (real), and numbers expressed in scientific notation. The values can be positive (+) or negative (−), and in scientific notation the exponent can be either positive or negative.

The following are examples of valid numbers in TI BASIC:

3
5
−6.783452
1258948567.236
2.36958E15
8.254689E − 12

The last two elements in this list are expressed in *exponential* or *scientific notation*. This is a shorthand method used by engineers and scientists to express very large and very small numbers. In the above example,

2.36958E15
is equal to
2.36958 times 10^{15}
or
2,369,580,000,000,000

and

8.254689E − 12
is equal to
8.254689 times 10^{-12}
or
0.000000000008254689

Scientific notation has the general form:

{[+]| − }m.mmmmmmmmmmmmmE{ + | − }xx

where:

m.mmmmmmmmmmmmm is the mantissa,
xx is the exponent.

Thus the value is derived by:

$$\text{Value} = \text{mantissa times } 10^{xx}$$

A *positive* exponent indicates that the decimal point is moved xx digits to the right; a *negative* exponent value indicates that the decimal point should be moved xx positions to the left.

There are limits to the size of a number that you can store. This is the result of allocating a fixed amount of storage to each number.

The largest absolute numeric value (ignoring the sign) that you can use in TI BASIC is:

9.9999999999999E127
or
9.9999999999999 times 10^{127}

A value greater than 9.9999999999999E127 or less than −9.9999999999999E127 causes an *overflow* and results in the warning message:

```
NUMBER TOO BIG
```

When this occurs, the maximum value (or its negative if the number was negative) is substituted and your program continues to execute.

The smallest absolute numeric value (ignoring the sign) that you can use in TI BASIC is:

1E−128
or
1 times 10^{-128}

A value between −1E−128 and 1E−128 is automatically replaced by zero (0). No warning message is printed and the program continues to execute normally.

These maximum and minimum absolute values apply equally to intermediate results in numeric calculations. If you are not careful, you can end up with the wrong answer caused by a substitution during a calculation. For example, the expression:

ANS = 1E100 * 3E30 / 3E30

results in a NUMBER TOO BIG error message and the wrong answer because 1E100 times 3E30 produces an overflow. The maximum value 9.9999999999999E127 is substituted for the result of this multiplication, yielding the answer (in ANS)

3.3333333333333E97

rather than the correct answer

1E100

This can be corrected simply by changing the order of the computation:

ANS = 3E30 / 3E30 * 1E100

No overflow occurred, because no substitution was made, and the result is correct.

In some calculations, it is important to know the *precision* with which numbers are handled by the computer.

You have already seen that numeric data is stored as a mantissa times 10 raised to an exponent. Precision is expressed as the number of significant digits maintained in the mantissa. Obviously, the more digits you keep track of, the more accurate your calculations will be. Depending on the value being stored, TI BASIC maintains a precision of up to 14 decimal places. TI BASIC is very accurate, indeed.

Maintaining precision to 14 decimal places is quite high, comparing favorably with business-oriented BASICs. Most home computers maintain only six or seven decimal places.

This means that in certain types of calculations, those involving very large numbers or very small numbers or both, you can rely on the TI-99/4A to produce accurate results.

String Data

The other type of data that BASIC recognizes is *character string data*. String data can include: letters, numbers, punctuation marks, and even nonprintable characters such as the carriage return character and the FCTN X back space character.

> *A character string data item is a sequence, or string, of one-byte characters which together are treated as a single data item. Each character in a character string occupies one 8-bit byte of memory.*

When you code a character string into a program, you enclose it in *quotation marks* (").

A string data item is identified by its value, just as a numeric data item is, and by its length. The maximum length of a string data item is 255 characters.

Should you exceed the 255 character maximum string length, no error or warning message results. TI BASIC simply *truncates* the string at the 255th character, throwing away all characters beyond the 255th. The following are examples of valid string data values:

"Hi There!!"
"227.678"
"Albert Einstein"
"4 pair @ $4.00/pair"

TI BASIC provides a set of *functions* designed to make it easy for you to manipulate string data. You can, for example, use the CHR$ function to insert otherwise inaccessible characters into a character string. We will examine these more closely when we look at expressions.

NOTE

Although a character string can contain numbers, it cannot be used as though it were a numeric data item. It must be converted to numeric format (using the VAL function) before you can use it as a number in, for example, a calculation.

DATA IN PROGRAMS—CONSTANTS AND VARIABLES

Now that we have examined the types of data that you can use in TI BASIC, we will take a look at how that data can be included in your programs.

Numeric and string data appear in programs in two forms: *constants* and *variables*. We are going to consider the differences between these two forms and show you where to use each.

Constants

You use *constants* to express a *fixed value* in your program. Constants are just that—literal representations of value. You might include numeric constants in a calculation, such as:

KILOMETERS = MILES * 0.625
or
number of kilometers equals number of miles times 0.625

The value 0.625 is a numeric constant. It is a factor that is used to convert miles into kilometers and, since it does not change, we can code it into a program as an unchangeable numeric constant.

String data constants included in a program are enclosed in double quotes ("), which sets them apart from BASIC statements and variable names. String data constants appear in many places, such as:

```
PRINT "INVESTMENT REPORT"
```

The string data constant "INVESTMENT REPORT" is printed to the screen each time this statement is executed.

Although a string data item can be as long as 255 characters, the *length of a string constant* is determined by the maximum length of an input line, which is 112 characters.

If you want to embed a double quote within a string value, you must code two double quotes in succession. For example:

```
PRINT "He said, " "How are you?" " "
```

prints this on the screen:

```
He said, "How are you?"
```

Variables

A *variable* represents a named storage area. Because it does not represent a value, only a place to store a value, a variable assumes any value that you assign to it.

You reference a variable by its *variable name*. This is a tag that you select, which is associated with a particular area of storage, and hence the value stored there.

The name you choose for your variables determines whether they can store numeric or string data. A single variable cannot store both types. TI BASIC has some rules concerning variable names. A few of them are listed below:

1. A TI BASIC variable name can have a maximum of 15 characters.
2. The first character of the variable name must be one of the following:
    - Upper case letter (A–Z)
    - At sign (@)
    - Left or right square bracket ([ ])
    - Back slash (\)
    - Under bar (_)
3. Within the variable name, you may include:
    - Upper-case letters (A–Z)
    - At sign (@)
    - Under bar (_)

If you enter a variable name longer than 15 characters, one that begins with an illegal character, or one that includes an illegal character, you will see the error message:

```
BAD NAME
```

Variables are what give programs flexibility. You assign actual values to them as the program executes so that different data values can be handled by the same program. Variables obtain their values from the following:

- Assignment statements (X = 53)
- Input statements (INPUT "ENTER X.":X)
- DATA statements (READ X)
- FOR statements (FOR I = 1 TO 4)

Your variable names cannot be the same as BASIC keywords or function names. These names are *reserved words* and are not available for use as variable names. Table 2-1 lists the TI BASIC reserved words. You may

Table 2-1. TI BASIC Reserved Words

These words cannot be used as variable names.

ABS
APPEND
ASC
ATN
BASE
BREAK
BYE
CALL
CHR$
CLOSE
CON
CONTINUE
COS
DATA
DEF
DELETE
DIM
DISPLAY
EDIT
ELSE
END
EOF
EXP
FIXED
FOR
GO
GOSUB
GOTO
IF
INPUT
INT
INTERNAL
LEN
LET
LIST
LOG
NEW
NEXT
NUM
NUMBER
OLD
ON
OPEN
OPTION
OUTPUT
PERMANENT
POS
PRINT
RANDOMIZE
READ
REC
RELATIVE
REM
RES
RESEQUENCE
RESTORE
RETURN
RND
RUN
SAVE
SEG$
SEQUENTIAL
SGN
SIN
SQR
STEP
STOP
STR$
SUB
TAB
TAN
THEN
TO
TRACE
UNBREAK
UNTRACE
UPDATE
VAL
VARIABLE

embed these reserved words within your variable names. For example, the name

SAVE

is not a legal TI BASIC variable name. The name

SAVE_TOTAL

however, is a valid variable name.

Numeric Variables—To store numeric data you must follow these rules when you name your numeric variables.

1. Numeric variable names may vary from one to 15 characters in length.
2. A numeric variable name must not end in a dollar sign ($).

The following are valid numeric variable names:

A
MILES
ZZ
VALUE
TOTAL
NUM_OF_STUDENTS
PRESENT_VALUE

The values you store in a numeric variable must follow the rules that apply to numeric data as described earlier in this chapter.

String Variables—String variables contain string data . . . numbers, letters, special characters, and nonprintable characters. There are two rules that you must follow when you name your string variables:

1. String variable names must end with a dollar sign ($) character.
2. A string variable name may vary from two to 15 characters, including the final dollar sign ($).

The following are valid string variable names:

Y$
NAME$
LIST$
MIDDLE_INITIAL$
STATE$

A string variable can contain up to 255 characters. TI BASIC has provided a set of functions to help you manipulate string variables. The following functions are available to:

- Determine the current length of a string variable (LEN)
- Determine the location of one string within another (POS)
- Extract a smaller string (sub-string) from a larger string (SEG$)
- Convert a string containing only numeric characters to numeric format (VAL)
- Convert a numeric variable to string format (STR$)
- Convert a character to its numeric (ASCII) value (ASC)
- Convert a numeric (ASCII) value to character string format (CHR$).

In addition to these functions, you can *add* strings to each other in a process called *concatenation*. Concatenation is like addition in numeric data items and is indicated by the ampersand (&) operator. For example:

A$ = "First Part. "&"Second Part."

yields a single string value in A$

"First Part. Second Part."

You can also concatenate string variables, like this

A$ = "HELLO "
B$ = "THERE, "
C$ = A$&B$

which is the same as:

C$ = "HELLO THERE, "

Array Variables—An *array* is a tabular collection of data elements of the same data type all of which are accessible through a single variable name.

An *array variable name* follows the same rules that apply to simple variable names. The type of data stored in an array (numeric or character), is determined by the variable name just as it is for simple variables. If the array name ends in a dollar sign ($), it is a character string array; otherwise, it is a numeric variable array.

NOTE

A simple variable and an array, or two arrays with different number of subscripts, cannot share the same name.

Arrays are useful in a wide variety of programming situations. In a numeric array, for example, you can store the cost of gasoline for each day of a trip. In a character string array, you can store the destination you reached on each day of the trip.

Arrays are characterized by their *number of dimensions* and their *array bounds*.

In TI BASIC, you can assign a maximum of three dimensions to an array. You do this in a DIM statement whose general format is:

DIM array-variable(dim1[,dim2[,dim3]])

The values you supply for the dimensions *(dim1, dim2,* and *dim3)* determine the bounds of the array. That is the number of elements stored in the array. There are limits to the size of an array. The maximum number of elements that can be stored in an array is 32,767.

If you attempt to dimension an array with a bound greater than 32,767, you will see the error message:

```
BAD VALUE
```

A single dimensioned array can be thought of as a *list* of *dim1* values in memory as shown in Fig. 2-1. A two-dimensional array is more like a table of *dim1* rows and *dim2* columns as shown in Fig. 2-2.

A three-dimensional array can be thought of as a repeating arrangement of two-dimensional arrays as illustrated in Fig. 2-3.

Although TI BASIC allows array bounds of up to 32,767, the actual limit is determined by the amount of available storage for the array. For example, each element in a numeric array occupies eight bytes of storage. Thus, a 32,767 element numeric array would occupy 262,136 bytes of memory, which is much more than is available.

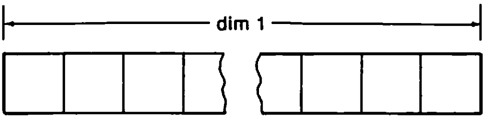


Fig. 2-1. Single dimensioned array.

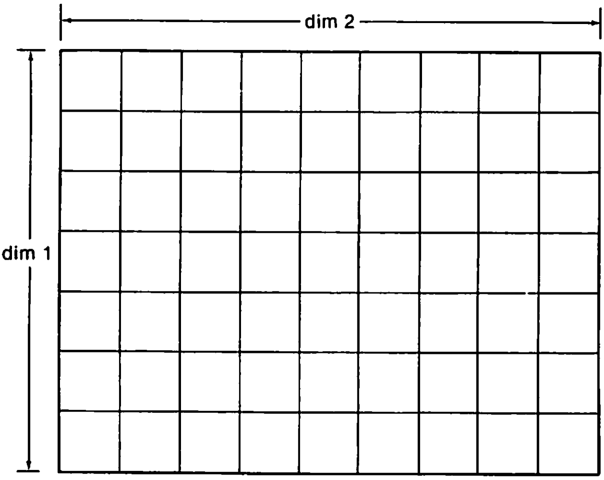


Fig. 2-2. Two dimensioned array.

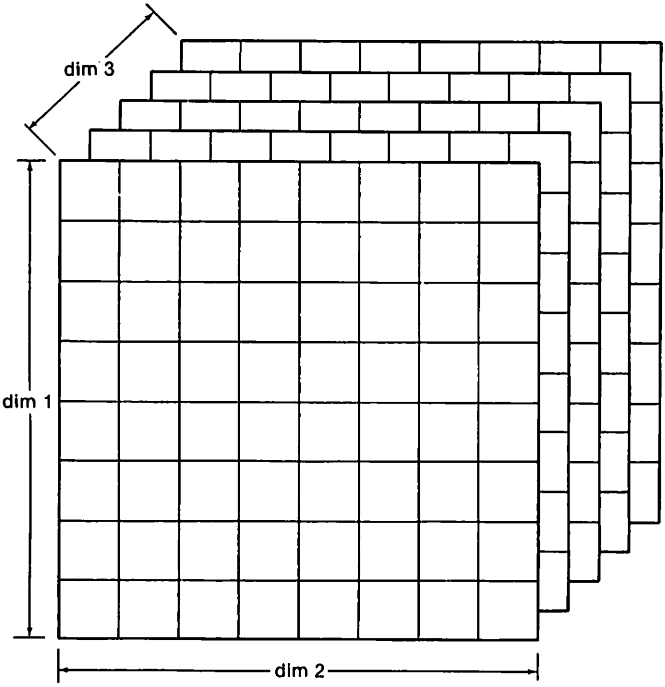


Fig. 2-3. Three dimensioned array.

The array bounds you can use in practice depends on the size of your program (which also occupies memory) and on the type of array it is.

Numeric arrays use eight bytes per element plus some overhead for symbol table entries and descriptors. The overhead is usually not a significant percentage of the array space.

String arrays are more complicated because the actual size of a string data item is not fixed. Nevertheless, a string array has some overhead that can be easily calculated. Two bytes are used for each element in a string array even if the element is not allocated (these two bytes store a pointer to the element once it is allocated). An allocated string element also has a one-byte length indicator attached to it. Plus there is the length of the string itself.

As you can see, once you enter data into the elements of a string array, you must plan for three bytes of memory in addition to the memory used by the string data itself, which is one byte per character.

Determining the actual number of elements in a TI BASIC array is a little confusing. While nearly everybody in the world starts counting at one, just about every BASIC starts counting array elements at zero (0). So an array declared as:

```
DIM GAS(60)
```

actually has 61 elements in it. It has the 60 you see, plus the zero element.

The *zero element* in an array can be useful. For example, if the GAS array is storing your cost of gasoline for each day of a trip, you can use the zero element to store the current total of all the other elements in the array. This way you can keep a running total of your gasoline costs without having to add up the individual elements each time.

The only time the zero element is really important (unless you intend to use it) is when you are approaching the limits of memory. In a multi-dimensioned array the zero element can add up to a significant amount of memory wasted. To avoid this problem, TI BASIC offers the

```
OPTION BASE 1
```

statement. This statement sets the lowest array element subscript to one (1) for all arrays in the program. If you put this in, you cannot use the zero element of any array.

You can calculate the total number of elements in an array by multiplying the number of elements in each dimension. For example, suppose you create the following array:

```
DIM A(10,5,7)
```

This array has 11 elements in the first dimension (10 plus the zero element); 6 in the second dimension; and 8 in the third dimension. Thus, the total number of elements allocated for this array is:

11 times 6 times 8 equals 528

If you include an *OPTION BASE 1* statement in your program, the number of elements allocated for this same array is given by

10 times 5 times 7 equals 350

or 178 fewer elements in the array. At eight bytes per numeric element, that translates into a savings of 1424 bytes.

Arrays, especially multidimensioned arrays, can use a great deal of memory very quickly. The actual bounds that you can declare for an array depend on how much memory you have to work in. When you have exceeded available memory, you will see the error message:

```
MEMORY FULL
```

Very often you can eliminate this error by reducing the size of your arrays.

It is not necessary to expressly dimension all arrays. You can implicitly dimension an array by coding a reference that includes a subscript. Such *default arrays* are limited to an upper bound of ten.

The lower bound of the dimension is either zero (0) or, in the presence of an *OPTION BASE 1* statement, a one (1). The following are examples of implicit array dimensions:

```
A(5)=3
B(3,4,5)=78
READ X(I)
```

Referencing Array Elements—You gain access to a particular element in an array through its unique *subscript(s).* Each element in an array has a unique subscript or, for arrays with more than one dimension, a set of subscripts.

An array reference must include a valid subscript value, enclosed in parentheses following the array variable name, for each dimension declared for the array. The general form of an array reference is:

array-variable(sub1[,sub2[,sub3]])

The subscripts *sub1*, *sub2*, and *sub3* must be numeric constants, variables, or expressions resulting in a value within the bounds of their respective dimension.

Subscripts must always be *integer* (whole) numbers. Noninteger subscripts are rounded to the nearest integer which is used as the subscript value.

If you supply a subscript outside the bounds declared for a dimension, you will see the error message:

```
BAD SUBSCRIPT
```

Some valid subscripted array references are:

```
A$(4)
X(1,5,7)
```

```
GAS(DAY)
GALAXY(QUAD1,QUAD2+2)
TOTAL(YEAR+1)
SALES(DAY,MONTH,YEAR)
```

Using Arrays—Arrays would be no easier to use than simple variables unless there were some way to streamline access to the individual elements.

We will consider a simple example to explore ways to use arrays. Suppose you have 50 stocks that you are keeping track of. You need to record the stock name and its latest selling price. You could use 50 different variables, but then you would have to write code to do processing separately on each of the 50 variables. A better solution is to dimension a few arrays:

```
DIMENSION STOCK_NAME(50),PRICE(50)
```

Now let's suppose you want to print the stock name and its latest selling price (we will ignore how the data got into the arrays for now). You could do that by coding:

```
100   PRINT STOCK_NAME(1),PRICE(1)
110   PRINT STOCK_NAME(2),PRICE(2)
120   PRINT STOCK_NAME(3),PRICE(3)
                  .
                  .
                  .
590   PRINT STOCK_NAME(50),PRICE(50)
```

But, that is no better than using individual variable names for each stock. A nicer solution is to use a FOR statement:

```
100   FOR I = 1 TO 50
110   PRINT STOCK_NAME(I),PRICE(I)
120   NEXT I
```

The FOR statement and its corresponding NEXT statement cause the variable I to vary from one to fifty, by one, each time through the loop (the loop is statements 100–120). This code accomplishes in three statements what straight coding took 50 statements to do. Hence the power of arrays.

Let's expand our example a little to include multidimensioned arrays. Suppose you want to keep track of the high, the low, and the closing prices for each stock. You can do that by dimensioning the PRICE array as follows:

```
DIMENSION PRICE(50,3)
```

The first dimension, 50, corresponds to the stock name, while the second dimension, 3, is used to keep track of the three prices for that stock. Think of this as a table of 50 rows, one for each stock, and three columns, one for the low, one for the high, and one for the closing price of the stock.

If you want to print the closing price of the tenth stock, you code:

```
PRINT PRICE(10,3)
```

To print the low price for the day of the 49th stock, code:

```
PRINT PRICE(49,1)
```

Functions

Functions, both those that are a part of TI BASIC and those that you create yourself, behave very much like variables except that they cannot be assigned a value (cannot appear on the left side of an equals sign).

The functions supplied as part of TI BASIC (built-in functions) provide fundamental support for many operations in geometry, math, and string handling.

Functions you write yourself can save a lot of coding and precious memory. Rather than repeating a lengthy calculation each time it is needed, you code it once in a function, then simply refer to the function name whenever you use the calculation.

A function is a variable-like reference, with an optional argument, that returns a single numeric or string value to your program. The general format of a function is:
function-name[(*arg*)]

Function names are constructed according to the same rules as variable names. Those that end in a dollar sign ($) return a string value, all others return a numeric value.

TI BASIC comes equipped with a large number of built-in functions. The names of these functions (shown in Table 2-2) are reserved words that you cannot use as variable names or user-defined function names.

User-defined functions are used in exactly the same way as built-in functions. You create a user-defined function with the DEF statement. See the discussion of the DEF statement in Chapter 4 for a full explanation of its use.

Consider, for example, a function which rounds a number to two places past the decimal point (n.nn):

```
DEF ROUND(X)=INT(X*100+.5)/100
```

Table 2-2. TI BASIC Functions

Function	Returns
ABS	The absolute value of the argument.
ASC	The ASCII value of the first character of the argument character string.
ATN	The arctangent of the argument angle.
CHR$	The character whose value is equal to the value of the numeric argument.
COS	The cosine of the argument angle.
EOF	True (– 1) if argument file is at end; otherwise false (0).
EXP	The value of "e" raised to the power of the argument.
INT	The integer part (largest whole number) of the numeric argument.
LEN	The length of the argument string.
LOG	The base 10 log of the argument value.
POS	The position in one string of another string.
RND	A random number between 0 and 1.
SEG$	A specified segment of the argument string.
SGN	– 1 if the argument is negative; 0 if the argument is zero; and + 1 if the argument is positive.
SIN	The sine of the argument angle.
SQR	The square root of the argument value.
STR$	The character string representation of the numeric argument.
TAN	The tangent of the argument angle.
VAL	The value, in numeric format, of the string argument.

In this function, *X* is a *parameter* that you replace with an actual variable name when you reference the function. The following are valid references to the *ROUND* function:

```
TOTAL = ROUND(TOTAL)
PRINT ROUND(TOTAL)
```

TI BASIC EXPRESSIONS

Expressions do the work in your programs. They appear in assignment statements, function references, screen or file output statements, IF and ON statements, and a host of other places.

> *An expression is a sequence of variables, constants, and/or function references, connected by operators, that resolves to a single value.*

Consider the following example:

```
COUNT + 1
```

This simple expression contains a variable (COUNT), and operator (+), and a constant (1). The result of this expression is a single numeric value.

Expressions can vary considerably in complexity as shown below:

```
COUNT=COUNT+1
HYPOT=SQR(SIDE1^2+SIDE2^2)
CIRCLE_AREA=3.14159*RADIUS*RADIUS
```

Operators

Operators determine the *action* that occurs between the constants, variables, and function references within an expression. TI BASIC defines the following three types of operators:

- Numeric operators
- String operators
- Relational operators

Numeric operators, as shown in Table 2-3, operate only on numeric data.

Table 2-3. Numeric Operators

Operator	Function
+	addition (X+3)
−	subtraction (X−3)
*	multiplication(X*3)
/	division (X/3)
^	raise to a power (X^3)
+	unary plus (+7)
−	unary minus (−X)

The *string operator*, shown in Table 2-4, works only on character string variables and constants. There is only one string operator because most string operations are performed using TI BASIC string functions (POS, SEG$, VAL, etc.).

Table 2-4. String Operator

Operator	Function
&	concatenation ("TIME "&"AND AGAIN")

The *relational operators*, shown in Table 2-5, are used mainly in IF statements. Relational operators always result in a TRUE (−1) or FALSE (0) numeric value.

Order of Evaluation—When dealing with numeric and relational expressions, it is vitally important to know the *order of evaluation* of the operations contained in the expression.

Consider, for example, the expression:

```
A = 6 + 5 * 4
```

Table 2-5. Relational Operators

Operator	Function
<	less than(A<B)
>	greater than (A>B)
=	equal to (A=B)
<>	not equal (A<>B)
<=	less than or equal to (A<=B)
>=	greater than or equal to (A>=B)

The result of this expression is different depending on which operation (* or +) is performed first:

- If the multiplication is performed first, the answer is 26 (5 * 4 = 20 + 6 = 26).
- If the addition is performed first, the answer is 44 (6 + 5 = 11 * 4 = 44).

The default *order of evaluation* of operations in an expression is given by the following hierarchy of operators:

1. ^
2. * and /
3. +, −, unary + and −
4. <, >, =, <>, <=, >=
5. &

Operators at the same level in the hierarchy (for example, * and /) are evaluated from left to right based on their placement in the expression.

Changing the Order of Evaluation—It is sometimes necessary to force a particular order to be followed in evaluating an expression. You do not always want expressions evaluated in the default order. You force an order on the evaluation by enclosing subexpressions within parentheses ( ).

A subexpression enclosed in parentheses is evaluated and its result used in the evaluation of any larger expression of which it is a part.

Subexpressions, enclosed in parentheses, may be *nested* with one subexpression embedded in another. The expression is evaluated beginning with the *inner-most* set of parentheses and proceeding outward.

Consider the simple expression:

$$A + B * C$$

According to the rules for order of evaluation:

1. B is first multiplied by C
2. The result of (B * C) is added to A

Suppose you want A added to B first, with the result multiplied by C. You achieve that by using parentheses as follows:

$$(A + B) * C$$

The subexpression enclosed in parentheses (A + B) is evaluated first and its result is used in evaluation of the remainder of the expression.

By nesting parentheses, you easily create highly complex expressions. For example, consider the expression:

((A + B) / (C + D))^3

The expression is evaluated as follows:

1. The expressions in the inner-most parentheses are evaluated first (A + B) and (C + D).
2. The result of the A + B is divided by the result of C + D.
3. That quantity is raised to the third power.

Numeric Expressions

In BASIC and in many other programming languages, numeric expressions, especially those contained in assignment statements, resemble algebraic expressions. They are not, however, algebraic expressions.

An *algebraic expression* has no firm values associated with it and therefore many possible answers are valid. A *numeric expression* in BASIC is a demand for immediate action, involving only the current values of the variables appearing in the expression.

Numeric expressions appear most frequently in *assignment statements*. For example, the assignment statement

AREA = LENGTH * WIDTH

calculates the area of a rectangle. Specifically, it demands evaluation of the expression (LENGTH * WIDTH), with the result stored in the variable named AREA.

Another frequently encountered location for numeric expressions is in IF statements:

IF (LENGTH * WIDTH) > 100 THEN 3000

In this case, the expression (LENGTH * WIDTH) is evaluated and the result is compared to 100. Note that the result is not saved, but retained only long enough for the comparison to be made.

String Expressions

There is only one string operator (&) which invokes the string addition operation called *concatenation*. Concatenation is the addition, in their order of appearance, of two or more string variables or constants, or both. Consider the program fragment:

```
100  MSG$="TIME "
110  CONCAT$=MSG$ & "AND AGAIN."
120  PRINT CONCAT$
```

When RUN, this program prints:

```
TIME AND AGAIN.
```

The length of the resulting string is equal to the sum of the lengths of the strings you are concatenating.

If the concatenation operation results in a string longer than the maximum allowed (255 characters), no warning message is given and all characters beyond the 255th are lost. (The string is truncated at the 255th character.)

Relational Expressions

Relational expressions compare two values. This is one of the most powerful things you can do in any programming language. Relational expressions provide you the opportunity to make decisions as to how your program should execute based solely on the current values of the variables in it.

A relational expression is the assertion of a particular relationship between two values. On evaluating the relational expression, TI BASIC returns:

- TRUE (−1 numeric value) if the asserted relationship is true
- FALSE (0 numeric value) if the asserted relationship is false

Consider the example:

```
DEDUCTIONS > GROSS_PAY
```

If the current value of the variable DEDUCTIONS is greater than the current value of GROSS_PAY, the evaluation of this expression yields a TRUE result and returns a −1 to the program. If GROSS_PAY is greater than or equal to DEDUCTIONS, then the relational expression is FALSE, and a 0 is returned to the program.

You use relational expressions most often in IF statements to control the order of statement execution through your program. When the evaluation of the relational expression yields a TRUE result, program execution is transferred to the statement whose line number appears in the THEN clause of the IF statement.

Otherwise, program execution passes to the statement indicated by the ELSE clause, if one is present, or to the statement immediately following the IF statement.

For example, consider:

```
800   IF CAPITAL_GAIN > 100000 THEN 5500 ELSE 1200
```

When the assertion, CAPITAL_GAIN is greater than 100000, is TRUE, statement 5500 is the next to be executed. When CAPITAL_GAIN is less than or equal to 100000, statement 1200 is the next statement executed.

When you compare character strings of different lengths, the comparison is carried out to the *length of the shorter string*. If they are equal at this length, the longer string is considered to be greater. The comparison is performed one character at a time from left to right according to the standard ASCII character collating sequence as shown in Table 2-6.

Table 2-6. ASCII Collating Sequence

ASCII Char	Decimal	ASCII Char	Decimal	ASCII Char	Decimal
(space)	32	@	64	`	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
–	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	|	124
=	61	]	93	}	125
>	62	^	94	~	126
?	63	_	95	(DEL)	127

Although FALSE is always signaled by a zero (0) value, TRUE is signaled to the IF statement by any nonzero value. For example, if the variable X is equal to 3, then the statement

```
IF X THEN 500 ELSE 1000
```

causes statement 500 to execute next. Only when X is zero (0), will statement 1000 be executed next.

This becomes important as you construct more complex relational expressions, connecting independent relational expressions with numeric operators to achieve the logical operations AND, OR, and XOR.

> *When you AND relational expressions together, the result is TRUE only when all of the constituent relational expressions are TRUE.*

In TI BASIC, the AND logical operation is performed by multiplying individual relational expressions together. The general form is:

(rel – exp1)*(rel – exp2)* . . . (rel – expn)

Where rel – exp1, rel – exp2, and rel – expn are valid relational expressions.

For example, consider:

```
IF (CHILDREN>3)*(WORK_STATUS=2) THEN 500
```

Statement 500 is executed only when *both* the variable CHILDREN is greater than three and WORK_STATUS is equal to two. If either of these relational expressions is FALSE, execution continues with the statement following the IF statement.

We can construct an AND logical operation this way because relational expressions return numeric values. Thus, when you multiply these numeric values, any FALSE relational expression causes a multiplication by zero, which always causes a zero (FALSE) final result.

> *When you OR relational expressions together, the result is TRUE when one or more of the constituent relational expressions are TRUE.*

In TI BASIC, the OR logical operation is performed by adding individual relational expressions together. The general form is:

(rel – exp1) + (rel – exp2) + . . . (rel – expn)

Where rel – exp1, rel – exp2, and rel – expn are valid relational expressions.

For example, consider:

```
IF (ANGLE=0)+(ANGLE=360) THEN 500
```

Statement 500 is executed next when the variable ANGLE is equal to 0 or to 360. Only when both relational expressions are FALSE (angle equal to other than 0 or 360) does execution continue with the statement following the IF statement.

In building the OR we rely on the fact that any nonzero result is interpreted as TRUE in the IF statement. When you add the results of evaluation of the relational expressions, one or more TRUE results will yield a nonzero answer. This satisfies the requirements for the OR.

When you XOR (eXclusive OR) relational expressions together, the result is TRUE when one and only one of the constituent relational expressions is TRUE.

In TI BASIC, the XOR logical operation is performed by adding individual relational expressions together and comparing the result to −1. The general form is:

((rel−exp1)+(rel−exp2)+ . . . (rel−expn))=−1

Where rel−exp1, rel−exp2, and rel−expn are valid relational expressions.

For example, consider:

IF ((CAT$="ASLEEP")+(DOG$="ASLEEP"))=−1 THEN 500

Statement 500 is executed next when either the CAT is asleep or the DOG is asleep, *BUTNOTBOTH*. If both the CAT and the DOG are asleep, execution continues with the statement following the IF statement.

The XOR works because relational expressions always yield a −1 or a zero. The sum of a series of relational expressions will equal −1 *if one and only one* of the constituent relational expressions is TRUE. In all other cases, the sum is zero (all FALSE) or less than −1 (more than one TRUE).

CHAPTER 3

Using BASIC

In this chapter, we tell you about TI BASIC's elements and how to tell your computer to do things immediately or as part of a program. We give you some hints on entering and editing TI BASIC programs.

TI BASIC operates in two modes, Direct Mode or Program Mode. These modes allow you to get your programs from a cassette tape and to make changes to a particular line in a program. They also allow you to RUN a program.

In *Direct Mode*, you type commands or statements, without line numbers, and they are executed as soon as you press the ENTER key.

In *Program Mode*, you enter and edit statements and commands that include line numbers, making them part of a program. The statements and commands are not executed until you RUN the program.

TI BASIC ELEMENTS

The *elements* of the TI BASIC language fall into three categories:

- Commands
- Statements
- Functions

Some commands work only in Direct Mode, some statements work only in Program Mode, but most operate in both modes. Nearly all functions operate in both modes.

Commands do something to your program or data files. Commands do not operate directly on your data, nor do they control the flow of execution through your program.

Commands are often executed in Direct Mode. When used in Direct Mode, commands are executed *immediately* after you press the ENTER key. Some commands can also be included in a TI BASIC program.

Table 3-1 lists the TI BASIC commands. Table 3-2 lists those commands that can also be included within a program.

Table 3-1. TI BASIC Commands

Command	Function
BREAK	Set a breakpoint.
BYE	End a TI BASIC session.
CONTINUE	Continue program execution.
DELETE	Delete a file.
EDIT	Edit a program.
LIST	List part or all of a program.
NEW	Clear memory.
NUMBER or **NUM**	Provide automatic line numbers.
OLD	Read a program into memory.
RESEQUENCE or **RES**	Resequence (renumber) a program.
RUN	Execute a program.
SAVE	Write a program to tape or disk.
TRACE	Write program line numbers before executing the statements.
UNBREAK	Remove all breakpoints.
UNTRACE	Reverse the TRACE action.

Table 3-2. Commands That Can Be Included in Programs

Command	Function
BREAK	Set a breakpoint.
UNBREAK	Remove all breakpoints.
DELETE	Delete a file.
TRACE	Write program line numbers before executing the statements.
UNTRACE	Reverse the TRACE action.

RESEQUENCE is an example of a command that *cannot* be used as a statement. RESEQUENCE is always used in Direct Mode, as in:

```
RESEQUENCE
or
RESEQUENCE 1000,50
```

DELETE is an example of a command that can be used as a statement. You use DELETE in Direct Mode, as in:

```
DELETE "DSK1.AFILE"
```

You use DELETE as a *statement* in Program Mode, as in:

```
500   DELETE "DSK1.AFILE"
or
900   DELETE "DSK2." & FILENAME$
```

Statements do the work in programs; they change the value of variables, alter the flow of execution of the program, and control input and output operations. Table 3-3 lists the TI BASIC statements.

Table 3-3. TI BASIC Statements

Statement	Result
CHAR	Define a character.
CLEAR	Clear the screen.
CLOSE #	Close a file.
COLOR	Change character foreground and background colors.
DATA	Store values in a program.
DEF	Define a user function.
DIM	Dimension one or more arrays.
DISPLAY	Write to the screen.
END	End a program.
FOR . . . TO . . . STEP	Repeatedly execute the statements between FOR and NEXT.
GCHAR	Read a character from a specific row and column on the screen.
GOSUB or **GO SUB**	Call a subprogram.
GOTO or **GO TO**	Branch to a statement.
HCHAR	Write a character at a specific row and column on the screen.
IF . . . THEN . . . ELSE	Evaluate a condition and branch to either of two statements.
INPUT	Get data from the keyboard.
INPUT #	Read data from a file.
JOYST	Read the joysticks' levers.
KEY	Read a key from the keyboard.
LET	Assign a value to a variable.
NEXT	End a FOR loop.
ON . . . GOSUB	Use the value of an expression to decide which subprogram to call.
ON . . . GOTO	Use the value of an expression to decide which statement to branch to.
OPEN #	Open a file.
OPTION BASE	Set the lowest array subscript to zero or one.
PRINT	Write to the screen.
PRINT #	Write to a file.
RANDOMIZE	Seed the random number generator.
READ	Read values from DATA statements.
REM	Put remarks in the program.
RESTORE	Select the DATA statement used by the next READ statement.
RESTORE #	Select the record in a file that will be processed by the next INPUT # or PRINT # statement.
RETURN	Return from a subprogram.
SCREEN	Change the color of the screen.
SOUND	Make up to three tones and one noise.
STOP	End a program.
VCHAR	Write a character at a specific row and column on the screen.

Although statements are generally included in programs, many can also be entered, without a line number, and run in Direct Mode. Table 3-4 lists the statements that can be executed in Direct Mode.

Table 3-4. Statement Available in Direct Mode

CALL **CLOSE** **DIM (Dimension)** **DISPLAY** **END**	LET (Assignment) OPEN PRINT RANDOMIZE	REM READ RESTORE STOP

When embedded in a program, TI BASIC statements are preceded by a line number:

<u>line#</u> <u>statement</u>

Line numbers must be whole numbers ranging from 1 to 32,767. If you enter a line number outside this range, you will see the message:

BAD LINE NUMBER

Most BASIC statements can be used in Direct Mode. For example, you use PRINT statements in Direct Mode, as in:

```
PRINT A + B
or
PRINT (750.59 - 255.36 + 45.93)*.34
```

You use PRINT statements in Program Mode, like this:

```
100  PRINT "HI THERE"
or
950  PRINT "THE TOTAL IS "; ANSWER
```

Some BASIC statements, such as FOR, GOSUB, and GOTO, cannot be used in Direct Mode because they are meaningless outside the context of a program. (Without a line number, where would you GOTO?)

Functions perform an operation and return a value as though they were variables in your program. YOU CANNOT ASSIGN A VALUE TO A FUNCTION.

Functions cannot stand alone. You must use them as part of a statement in the same way as you would use any other variable. Table 3-5 lists the

TI BASIC functions. Most functions operate equally well in Direct or Program Mode.

Table 3-5. TI BASIC Functions

Function	Returns
ABS	The absolute value of the argument.
ASC	The ASCII value of the first character of the argument character string.
ATN	The arctangent of the argument angle.
CHR$	The character whose value is equal to the value of the numeric argument.
COS	The cosine of the argument angle.
EOF	True (−1) if argument file is at end; otherwise false (0).
EXP	The value of "e" raised to the power of the argument.
INT	The integer part (largest whole number) of the numeric argument.
LEN	The length of the argument string.
LOG	The base 10 log of the argument value.
POS	The position in one string of another string.
RND	A random number between 0 and 1.
SEG$	A specified segment of the argument string.
SGN	−1 if the argument is negative; 0 if the argument is zero; and +1 if the argument is positive.
SIN	The sine of the argument angle.
SQR	The square root of the argument value.
STR$	The character string representation of the numeric argument.
TAN	The tangent of the argument angle.
VAL	The value, in numeric format, of the string argument.

A function reference may not appear on the left side of an equal sign in an assignment (LET) statement. For example:

```
20   SEG$(A$,2,6)="SMITHY"
```

is *NOT* a legal statement.

Functions can appear in statements executed in the Direct or Program Modes. For example, the square root function (SQR) can be used like this:

```
100   A = SQR (B^2 + C^2) (Program Mode)
                  or
   PRINT SQR(256 + 398) (Direct Mode)
```

ENTERING TI BASIC PROGRAMS

You *enter* a TI BASIC program when you type it into your computer's memory. You can enter a TI BASIC program in either of two ways:

1. Type a line number and one space followed by the BASIC statement, like this:

```
100   REM THIS IS A REMARK STATEMENT
```

or

```
1230 A = 4.5678
```

2. Type the NUMBER (or NUM) automatic line numbering command and, optionally, the starting line number and increment values, like this:

```
NUM
```

or

```
NUM 2000,20
```

Generally, you use the first method only to enter isolated lines into an existing program. After all, why should you key in line numbers when the computer will produce them for you automatically and without the chance of an error?

The NUM command generates statement numbers, saving you the considerable hassle of keying them in yourself. The NUM command has the general format:

```
NUM [starting-line# [,increment]]
```

Where:

starting-line# is the first line number you want to enter,

increment is the amount added to each line number to generate the next line number.

If you do not supply a starting line number, TI BASIC begins with line 100. If you do not specify an increment, 10 is used.

For example:

```
NUMBER
```

or

```
NUM
```

The first line number is 100 and the line numbers are incremented by 10.

Or, like this:

```
NUMBER 250,25
```

or

```
NUM 250,25
```

The first line number is 250 and the line numbers are incremented by 25.

We are going to go through an example showing how to enter a program. In the following sample dialog:

- Your commands are in *italics*
- The TI-99/4A's responses in **bold face**
- <ENTER> means you press the ENTER key.

If you are unfamiliar with the NUM command, try following this example on your TI. You do not have to put all the statements in, just end a section wherever you feel like. You begin by entering:

```
>NUM<ENTER>
100  REM PRESENT VALUE PROGRAM<ENTER>
110  DEF RD(X)=INT(X*100+.5)/100<ENTER>
120  CALL CLEAR<ENTER>
       .
       .
       .
```

TI BASIC continues to put statement numbers on the screen until you enter an empty line (just the ENTER key with no other letters on the line). If our sample program ends at line 250, you would finish like this:

```
       .
       .
       .
240  GOTO 160<ENTER>
250  STOP<ENTER>
260  <ENTER>
>
```

To prevent the loss of your program as a result of an error, you should make a habit of saving the program to tape, Wafertape, or diskette at regular intervals. The end of a logical program section is a good place to stop for a SAVE.

Whatever you do, at least *remember to save the program* on tape or disk if you want to use it again without re-typing the entire program.

CAUTION

Be careful when you press the (equal) = key! The = key can do two things, depending on whether you are holding down the SHIFT *key or the* FCTN *key when you press the = key. If you press* FCTN = *, you will return to the first (main title) screen and lose the program in memory. If you want to insert an = (as in A = B) make sure that you're holding down the* SHIFT *key when you press the* = *key.*

You will make typing errors as you enter your programs. To correct those errors, you use the *line editor* commands shown in Table 3-6. Notice that you hold down the FCTN key (just like a SHIFT key) while pressing another key to perform these editing functions.

Table 3-6. Line Editing Commands for Entering BASIC Programs

Key	Function
ENTER	Enter the program line. The line you are typing (line number and statement) is entered into the program currently in your computer's memory.
FCTN D (right-arrow)	Forwardspace one character. Move the cursor one character position to the right. No changes are made to any characters the cursor moves past. You use the FCTN D key to position your cursor when you want to add or delete characters on the line you're currently typing.
FCTN E (up-arrow)	Works just like the ENTER key. The program line you just typed is put into your computer's memory.
FCTN S (left-arrow)	Backspace one character. Move the cursor one character position to the left. No changes are made to any characters the cursor moves past. You use the FCTN S key to position your cursor when you want to add or delete characters on the line you're currently typing.
FCTN X (down-arrow)	Works just like the ENTER key. The program line you just typed is put into your computer's memory.
FCTN 1 (DEL)	Delete one character. Delete the character under the cursor. You usually use the FCTN S or FCTN D key to position the cursor to the character you want to delete.
FCTN 2 (INS)	Insert characters. Insert characters at the cursor position. You can use the FCTN S or FCTN D key to position the cursor to where you want to insert the characters. Unlike the other FCTN keys, INS puts you into *Insert Mode,* allowing you to insert as many characters as you need.
FCTN 3 (ERASE)	Erase the entire line. Does not erase the line number if you're in automatic line numbering mode (NUM command).
FCTN 4 (CLEAR)	Clear the current line. Cancels the line you are typing. If you are in automatic line numbering mode, FCTN 4 erases the current line and ends NUM command processing.
FCTN = (QUIT)	Quit. Leave BASIC and return to the main title screen. Memory is erased. If you have files opened, they are not closed. Use a **BYE** command if you want your files closed. Remember, you lose the program in memory if you haven't saved it.

EDITING TI BASIC PROGRAMS

Your TI-99/4A has a built-in editor so that you can make changes to your TI BASIC programs. Once a program, or part of a program is in memory, you can use the editor by entering one of the following:

1. EDIT line-number
2. line-number FCTN E (up-arrow)
3. line-number FCTN X (down-arrow)

After you enter one of these commands, you will see the line that you asked for *(line-number)* displayed on your screen. You can make any changes that you want to the line or delete the line.

Several keys have a special meaning when you use them as function keys (hold down the FCTN key and another key at the same time) while editing a program line. Table 3-7 shows the keys you use to edit TI BASIC programs.

For example, suppose you have entered the line:

560 NAME$ = "MRRY"

You notice that you have spelled the name wrong (it should be MARY). To correct it you enter:

```
560<FCTN E>
     or
EDIT 560<ENTER>
```

The computer responds with:

```
560 |N|AME$="MRRY"
```

with the cursor positioned on the N in NAME$. (The | | shows you where the cursor is.) To move the cursor to the R in MRRY, you must hold down the FCTN key while hitting the D (right arrow) key. Follow the cursor:

```
560  |N|AME$="MRRY"<FCTN D>
560  N|A|ME$="MRRY"<FCTN D>
560  NA|M|E$="MRRY"<FCTN D>
560  NAM|E|$="MRRY"<FCTN D>
560  NAME|$|="MRRY"<FCTN D>
560  NAME$|=|"MRRY"<FCTN D>
560  NAME$=|"|MRRY"<FCTN D>
560  NAME$="|M|RRY"<FCTN D>
560  NAME$="M|R|RY"
```

With the cursor over the incorrect R, you can enter the A to replace it, yielding:

```
560  NAME$="MA|R|Y"
```

Note that the cursor advances automatically to the next character. Now that you have made the correction, simply press the ENTER key to place the corrected line in the program.

Table 3-7. TI BASIC Editing Function Keys

Key	Function
ENTER	Enter the program line. The line you are editing (line number and statement) is entered into the program currently in your computer's memory.
FCTN D (right-arrow)	Forwardspace one character. Move the cursor one character position to the right. No changes are made to any characters the cursor moves past. You use the FCTN D key to position your cursor when you want to add or delete characters on the line you're currently editing.
FCTN E (up-arrow)	Enter the program line. The line you are editing (line number and statement) is entered into the program currently in your computer's memory. The statement with the next lower line number is then presented for editing.
FCTN S (left-arrow)	Backspace one character. Move the cursor one character position to the left. No changes are made to any characters the cursor moves past. You use the FCTN S key to position your cursor when you want to add or delete characters on the line you're currently editing.
FCTN X (down-arrow)	Enter the program line. The line you are editing (line number and statement) is entered into the program currently in your computer's memory. The statement with the next higher line number is then presented for editing.
FCTN 1 (DEL)	Delete one character. Delete the character under the cursor. You usually use the FCTN S or FCTN D key to position the cursor to the character you want to delete.
FCTN 2 (INS)	Insert characters. Insert characters at the cursor position. You can use the FCTN S or FCTN D key to position the cursor to where you want to insert the characters. Unlike the other FCTN keys, INS puts you into *Insert Mode,* allowing you to insert as many characters as you need.
FCTN 3 (ERASE)	Erase the entire line. Does not erase the line number.
FCTN 4 (CLEAR)	Clear the current line. Erases the current line and stops the editing process.
FCTN = (QUIT)	Quit. Leave BASIC and return to the main title screen. Memory is erased. If you have files opened, they are not closed. Use a **BYE** command if you want your files closed. Remember, you lose the program in memory if you have not saved it.

Renumbering the Lines in Your BASIC Program

After you make changes to your program, you will notice that your formerly orderly line numbers are now out of sequence. It is easier to make changes to a program with line numbers in sequence.

You can easily *renumber* your statements with a RESEQUENCE (RES) command. Renumbering, or resequencing your BASIC program, adjusts all the line numbers so that the line numbers begin with the number you want and will increase by the increment value that you specify.

You can resequence any BASIC program that is currently in your computer's memory by entering

```
RES
```

which starts the line numbers at 100 and increments them by 10.

If, instead, you want to start your line numbers at 500 and increment them by 50, you enter:

```
RES 500,50
```

All the line numbers in your program are adjusted to the new values. References to line numbers in statements (like IF, GOTO and GOSUB) are automatically adjusted to reflect the new (changed) line numbers.

Saving Your Programs

If you enter a program more than a few statements long that you expect to run again, you will want to save that program. To save a program for recall later enter:

SAVE <u>device-name.file-name</u>

Where:

device-name is the name of a storage device such as CS1 for a cassette, or DSK1 for disk drive 1.

file-name is the name of the file on the device. You don't use a *file-name* when saving to a cassette tape.

Of course, you want to save your program after you have finished entering it. But you should also save it from time to time as you enter it. A good time to save a backup copy (this is so you won't lose the complete program in the event of a problem) is at the end of a logical section of the program. This is not an iron-clad rule, but anytime you feel that you have invested so much time and effort in entering the program that you would really hate to lose it, make a duplicate copy.

A saved program is written to the device as a *program file*. When you do a directory listing of a diskette (cassettes do not have directories), your TI BASIC programs are listed as file type PROGRAM. When on disk, programs are written as fixed length, 256-byte (full sector) records. On cassette tape, programs are stored in fixed length 64-byte records, which have a maximum length of 12K (12,288) bytes.

If you are saving to a cassette tape, simply enter:

```
SAVE CS1
```

When you are entering a program, you can place a tape in your cassette recorder and save to it repeatedly. There is no need to rewind after every save, just write down the tape counter number each time so you can find the program if you have to recover it.

If you are saving to a disk, you use a command something like this:

```
SAVE DSK3.AUTOMAINT
```

This SAVE command saves the auto maintenance program to disk drive number 3, filing it under the name AUTOMAINT.

Saving to a Wafertape is similar to saving to a disk. The primary difference is in the device name.

Loading a Saved Program

Before a saved program can be run, it must be loaded into the computer's memory. The program may reside on cassette tape, Wafertape, or on a diskette, depending on which devices you have attached to your TI. In all cases, you use the OLD command to move the program from the device to Random Access Memory where it can be executed.

The format of the OLD command is:

OLD <u>device-name.file-name</u>

Where:

device-name is the name of a storage device like CS1 for a cassette, or DSK1 for disk drive 1.

file-name is the name of the file on the device. You do not use a *file-name* when loading a program from a cassette tape.

To load a program from a cassette tape, you enter the command:

```
OLD CS1
```

If you have a dual cassette cable you cannot load from device CS2—CS2 is a write-only device.

Loading from a diskette requires that you supply the name of the program file, as well as the device name. For example:

```
OLD DSK3.AUTOMAINT
```

requests loading of the programs stored on disk device 3 in the file named AUTOMAINT.

Debugging Programs

The process of getting a program to run successfully, producing the correct answer, is called *debugging*. Stories conflict concerning the source of this term but its persistence is well understood. It is simply intolerable to admit you may have made a mistake while entering or designing your program. Much better to blame its failure on "bugs" loose in the computer, eating your code (the letter missing from that variable name), or causing program logic to stumble.

You have typed in the program, saved it so you will not accidentally lose it, and entered the RUN command . . . And BOOP—it FAILS!

Some horrible error message comes up on your screen as the computer awaits your next command.

What should you do? Primarily, DON'T PANIC! It is only a machine and can be made to do what you want.

The first thing to do is *look at the statement where the error occurred* (if the message lists one). Compare this to what it is supposed to be. If you seen any obvious problems, correct them and RUN it again.

If you do not see a problem with the statement, *check to make sure you entered all of the statements in the program.*

If you find statement(s) missing, insert them where they belong. You may have to correct all the statements following any that are missing because GOTO, IF . . . THEN . . . ELSE, and GOSUB statements may refer to line numbers now on the wrong statements.

If it looks as though you have all the code entered in the right place, chances are you have a *misspelled variable name* somewhere. This can be very tedious to find.

You can use the brute force approach of looking through the whole program, searching for the misspelled, or just plain wrong, variable reference. And you may, in the end, have to do just that. However, this is not the most efficient way to find an error. At first, see if you can figure out what is wrong in the statement that failed.

One of the strongest, and simplest, debugging tools available in TI BASIC is the Direct Mode PRINT statement.

When your program fails, or is interrupted by pressing the **CLEAR** key (FCTN 3) or by encountering a BREAK point, you can PRINT the values of any variable in the program.

This can rapidly lead you to the problem, or at least localize the error in the program (reduce the probable cause to computation of a particular variable or to a particular section of code).

You can PRINT the value of the variables referred to in the statement. For example, if the statement that failed reads:

```
1020 CALL COLOR(I,FGCOLOR,BGCOLOR)
```

You can find out the value of the variables involved by entering:

```
PRINT I,FGCOLOR,BGCOLOR
```

Once you can see the value of these variables and can compare their values to those allowed, or those that are reasonable within the statement, you may be able to tell which variable is incorrect.

Having determined which variable has a bad value, you can restrict your search through the program to those statements which refer to that variable. If you find it calculated somewhere, PRINT the values of the variables involved in calculating it, just as you did for those in the statement that failed. If you keep tracing it back, you will eventually find the error.

Actually, this is not as lengthy a process as it sounds. You will find most of these sorts of errors in a few minutes of looking around the program.

You can also insert PRINT statements at strategic places in your program to display variable values. A good place for these is at entry to and exit from GOSUB routines and other well defined logic blocks (chunks of code that do some well defined thing) in your program.

Another cause of program failure, often less easy to find, is a *logic error.*

This kind of problem can show up as a bad value somewhere else in the program, or as a wrong answer. As you trace through the variables whose values appear to be wrong, pay attention to the IF statements, GOTO statements, and GOSUB statements that affect how those variables are calculated. You might find a statement that reads:

```
3050  IF K>0 THEN 3080
```

When it should, in fact, read:

```
3050  IF K<>0 THEN 3080
```

Logic errors that are designed into your program—Yes! You wrote it wrong!—are the hardest errors to find. This is especially true if you must look for them without help, as your own errors in logic are always difficult to spot. This is true even of seasoned professional programmers. If you are stuck on a problem, ask a friend who knows programming to help you. Or, take your problem to a local User's Group; there will always be a hot-shot there only too happy to dig through some code to find an error.

TI BASIC includes two statements that make it easier for you to find logic (and other kinds of) errors in your program.

TRACE writes each statement number to the screen as the statement is executed. This lets you watch your program execute, tracing the flow of statement execution and detecting any deviation from what it "should" be doing.

BREAK causes your program to stop at a statement, or statements called *breakpoints,* and revert to Direct Mode. You can then enter PRINT statements, for example, to "look around" at variable values in the program. You can even enter an assignment statement to set a variable to whatever value you want. This can be handy when you know a variable has been calculated wrong, but want to continue checking the rest of your program.

When you have finished at a breakpoint, you resume program execution with a CONTINUE (or CON) command. *YOU CANNOT CONTINUE EXECUTION IF YOU EDIT YOUR PROGRAM.* You must run it again.

FILES IN TI BASIC

A *file* is a collection of related data items written to or read from an external device. Sometimes the device is magnetic, like a cassette tape or disk, and can be both written to and read from. Files on other devices, such as a printer, can only be written to. In general, files are characterized by:

- Logical record size
- Physical block size
- Record format (FIXED or VARIABLE)
- Data format (DISPLAY or INTERNAL)
- Processing mode (INPUT, OUTPUT, UPDATE, or APPEND)
- Device (disk, cassette, Wafertape, printer, RS232)

Magnetic files present a way for you to expand the apparent size of the memory available to you. Rather than trying to keep all the data required by a program in memory at the same time, you can keep the data on an external storage device such as a cassette tape or disk. Then you can bring it into the TI-99/4A's RAM (Random Access Memory) only when it is needed.

Files also provide a way to store information (including programs) permanently. You know that when you turn off your computer whatever is in its memory is lost. It would be terribly inconvenient to have to re-enter programs and all their data each time you wanted to run them. You can avoid this by storing your programs magnetically on a cassette tape, Wafertape, or a disk.

These magnetic media retain the information written to them without the constant application of power, just as audio recordings on cassette tapes retain your favorite songs.

You process files in TI BASIC using the statements shown in Table 3-8.

Table 3-8. File Processing Statements*

Statement	Description
OPEN #	Prepares a file for processing and specifies the characteristics of the file.
PRINT #	Writes data to a file.
INPUT #	Reads data from a file.
CLOSE #	Ends processing of a file.
RESTORE #	Repositions a file to the beginning or, for RELATIVE files, to a particular record in the file.

*These statements are described in detail in Chapter 4.

Before you can do any processing of a file, you must *open* the file with an OPEN statement. This prepares the file for processing and performs an initialization or device positioning that may be required for access to the file. For example, if you are going to write to a printer through the RS232 interface, the operating parameters must be initialized to match those of your printer. This is done through various options in the OPEN statement.

File Characteristics

When you open a file (with an OPEN statement), you supply a set of keywords that define important characteristics of the file.

Identifying a File

To identify a file in an OPEN statement you specify:

device[.file-name[.options]]

The *device* you specify places constraints on the other file characteristics that you can request. For example, if you open an *RS232* device attached to a printer, it makes no sense to open it for INPUT processing.

You must supply a *file-name* for some devices and you must not supply a *file-name* for others. Disk files must be named, but cassette files are not named.

Some devices, like the RS232 interface, allow you to supply *options* specific to the device. Multiple *options* must be separated from one another by periods.

The device you specify actually causes TI BASIC to invoke the Device Service Routine (DSR) with that name. Each peripheral that can be attached to your TI-99/4A has a DSR associated with it that contains all the code required to handle the peripheral. The Expansion Box Disk Controller card, for example, contains a disk DSR capable of handling up to three disk drives.

When you turn the computer on it looks to see which devices are attached to it by locating their DSRs. It places this information in a table in memory. This infomation is then searched for by the device that you specify. The file-name and any options are interpreted by the DSR.

When you acquire a new peripheral, look at the documentation that comes with it to find out what it needs for a device name and whether it requires a file-name or options or both.

Consider this example of a file identifier for the RS232 interface card:

RS232.BA=9600.DA=8

This requests use of the RS232 device, with the option that it operate at 9600 baud and transfer 8 data bits per byte.

File Record Description

A TI BASIC file is composed of a sequence of *records* whose characteristics you specify when you open (prepare for processing) the file.

A *record* is entered into a file each time a PRINT # statement is executed, provided the PRINT # statement does not end with a semicolon (;). A PRINT # statement that ends with a semicolon causes a *pending write* condition. This holds the current record incomplete until either a PRINT # statement without an ending semicolon is executed, or until the file is closed. In both cases, the record is written to the file.

If the file is on tape or disk, records are inserted in *physical blocks* before being written to the device. On a cassette tape, only one record can be included in a physical block.

On disk the physical blocks are called *sectors*. These sectors are 256 bytes long and can contain only as many complete records as will fit. For example, only two 100-byte records fit into a 256-byte sector as records are not allowed to span sectors.

You must describe the records that you wish to write to a file.

Data Format

In TI BASIC you can write files with one of two data formats, *INTERNAL* or *DISPLAY*.

Internal format causes data items to be written to the file in the same format as they are stored internally. That is, a numeric data item occupies 8 bytes, and a character string data item is its length plus one byte.

Display format causes the data items written to the file to be converted to a character format according to the conversion rules that apply to PRINT statements directed to the screen.

You should use the DISPLAY format when writing to a device that will allow people to read it, such as a printer or a modem.

You should always use the INTERNAL format when writing to a tape, a disk or other storage device not intended to be read directly by people. Internal format requires less storage and the data is processed faster than in display format.

Record Format

Records can be in either of two formats, *FIXED* or *VARIABLE*. These formats determine whether the records in a file will all have the same fixed length, or whether they will have variable lengths.

Fixed length record files are simply a collection of the same size records, one following the other. Relative record disk files and cassette tape files must contain fixed length records.

If you write a short record into a fixed length record file, the short record is padded with null (zero value) characters to the length declared for the file.

Variable length records are only as long as the data contained in them. If you are writing to a printer for example, you should OPEN the printer file using the variable format. In disk files, unless you are certain that all your records will be the same size, variable length records make better use of disk space.

Record Length

For fixed length files, record length is the length of all records written to the file. For variable length files, the record length is the *maximum* length of the records written to the file.

You specify the record length immediately following the record format, as in:

```
OPEN #6:"DSK1.TEST", OUTPUT, VARIABLE 102
```

Where 102 is the maximum length of records written to the disk with filename TEST. If you do not specify a record length, TI BASIC will use 80 bytes.

Processing Sequence

There are two processing sequences (or file organizations) available in TI BASIC, *sequential* and *relative.*

In most cases, records are read from or written to a file in their physical order. This is called *sequential* processing and is the default processing sequence for all files.

If you have a disk drive, you can create a *relative* record file on disk. The records in a relative file must be a fixed length. You can directly access a record in a relative record file by specifying, in the REC option of an INPUT # statement, the *relative record number* of the record you want.

You can consider a relative file as a large array. The first record in the file is record number zero (0), the second record in the file is record number one (1), and so on for as many records as you have in your file. If you want data from a particular record, you code an INPUT # statement that looks like this:

```
INPUT #4,REC 23:A,B,C
```

This INPUT # statement reads values into variables A, B, and C from record 23 (the 24th record) in the file opened as #4.

For some devices, sequential is the only reasonable processing sequence. For example, a printer can hardly be processed as a relative file. Cassette files must be opened for sequential processing.

Processing Mode

When you open a file, you must tell TI BASIC how you will be processing the file. You have four choices:

- *INPUT* means you will be reading from an existing file.
- *OUTPUT* means you want to write records to an empty file.

- *UPDATE* means you can both read records from and write records to an existing file.
- *APPEND* means you want to add new records to the end of an existing file.

When you open a file for *INPUT* processing, you can use only the INPUT # and RESTORE # statements to process it. If it is a fixed length file on disk and you have opened it for relative record processing, you can include the REC option on your INPUT # and RESTORE # statements.

When you open a file for *OUTPUT* processing, you can use only the PRINT # and RESTORE # statements to process it. If it is a fixed length file on disk and you have opened it for relative record processing, you can include the REC option in your OUTPUT # and RESTORE # statements. If you open a disk file for output processing and the file already exists, the existing file is replaced by the new file.

A file opened for *UPDATE* processing may be both written to and read from. In general, during update processing you read a record, change it, and write it back to the file. Files opened for update processing must reside on a disk. Relative record files are often processed in update mode as it allows you to retrieve a particular record, place new information into it, and write it back to the file at the same position from which it was read.

APPEND allows you to add records to the end of an existing file. When you open an existing file for append processing, it is first positioned at the end of the file. You may then use PRINT # statements to add new records to the end of the file exactly as you do with a file opened for output processing.

Files opened for APPEND processing must reside on a disk and be of a variable length record format.

Files on Cassette

Cassette tape is the least expensive file storage medium. As a result, it is also the most common.

OPEN statements for cassette files offer very few options:

- The file must be named CS1 or, for output only, CS2.
- The file format must be FIXED.
- The file processing mode must be either INPUT or OUTPUT.
- File organization must be SEQUENTIAL.
- The cassette file format may be either DISPLAY or INTERNAL, but you should use INTERNAL unless you have reason not to. INTERNAL type file processing is faster and usually yields smaller records.
- The record size may be up to 192 bytes (characters) with a default of 64.

Record size is the most important choice you have to make when creating a cassette file. The record size determines the size of the *physical block* that is written on the tape. Only three physical block sizes are actually written to a cassette tape: 64, 128, or 192 bytes. The physical block size must be the same within a cassette file. You should choose a physical block size sufficient to accommodate the largest record you will write to the file.

Whatever record size you specify, it is rounded up to the next higher physical block size. If for example, you specify a record size of 158 bytes, TI BASIC actually writes a 192-byte physical record to the cassette tape. The part of the block between the 158 bytes you wrote and the 192 bytes actually written is filled with null (zero value) characters.

CHAPTER 4

TI BASIC Statements, Commands, and Functions

This chapter describes all the TI BASIC elements: statements, commands, and functions, presented in alphabetical order.

First, we describe the format and terms used in each description. In the main part of the chapter, we examine each TI BASIC element showing its format, operands, purpose, defaults, description, common errors, and one or more examples.

HOW EACH BASIC ELEMENT IS DESCRIBED

Each BASIC element's description contains the following information:

- *Type*—Tells whether the element is a statement, a command, or a function. Those elements shown without line numbers can be used only as commands. Those shown with a required line number can be used only as statements in a program. Those shown with an optional line number can be used as either statements (with the line number) or commands (without the line number).
- *Format*—Shows you what the element looks like with its operands. Descriptive names are used for the operands (for example, *column* or *file-name*). Some elements show two formats because you can use an abbreviation.
- *Purpose*—A brief definition of what the element does and why you use it.
- *Operands*—Shows the element's operands and gives a brief definition of their values.
- *Defaults*—What the values for the element's operands are if you do not supply a value yourself.
- *Description*—A complete description of the element and its common uses. We tell you why and when to use the statement, command, or

function. We give the links to other statements, commands and functions when appropriate (for example, READ and DATA are linked).
- *Common Errors*—What errors you can expect and the most common reasons that the errors happen.
- *Examples*—One or more examples showing how to use the element. The examples are usually short programs that you can enter and run. We often give suggestions on changes that you can make to the example program so that it will do something a little differently. Making changes that work will show you whether or not you really understand what the element does.

TABLES AND FIGURES

It is sometimes annoying and often frustrating to have to look elsewhere in a book for information. Such as searching for an Appendix or a description in another section of the book. We tried to put all the information that you will need with the element description, which means you will find duplicate information in this book.

You will not have to search throughout the descriptions to find a table. Tables and figures are included wherever, and whenever, they are needed.

The tables also appear in the Appendices so that once you know what the elements are and what they can do, you will have a quick reference section in the Appendices.

EXAMPLES

A common complaint about examples is that they do not always make much sense to the beginning programmer. You enter the statements and your computer will do something, but you're not always sure what it was *supposed* to do.

We made the examples a little longer than what you probably expected so that you have mini-programs that do something, as strange as that something may sometimes be. There is a short description of what you should expect the example program to do. A few examples have instructions for entering TI BASIC commands as well as the program.

Many examples begin with a CALL CLEAR. This simply clears your screen so that you will see only what the program (or TI BASIC) writes. You won't see any extra stuff left over from previous programs.

All the programs end with an END statement. This way, you will know when you have entered the complete program.

NOTE

Remember to use NEW before you enter a program or you may have other program statements that are already in your computer's memory included in the program you are entering. *LIST* the program after you are done entering it to see what you have entered.

Most examples start with line number 100. Those that do not are the ones that require commands outside the program. Use the NUM command to enter the programs, like this:

```
NUM <ENTER>
```

TI BASIC will automatically generate the line numbers for you, beginning with line number 100 and adding 10 to each new line number.

NOTE

Some examples may look like they have lines that start without a statement number. These lines are continuations of the previous line and are divided to make it easier for you to read. *ENTER THESE STATEMENTS AS PART OF THE STATEMENT WITH THE LINE NUMBER!*

Every example program has been tested on our own TI-99/4A. If you enter the statements correctly, the programs will work. If you make a mistake and your program doesn't work, look at the statement with the error and see if you have misspelled something. Check the Common Errors section for that statement if you still need help.

NOTATION

Whenever the format for a statement or command is given, the following rules apply:

1. Words in BOLDFACE AND CAPITALS are *keywords* that you enter exactly as they appear.
2. Words in reversed letters designate keystrokes.
3. *num-exp* means any *numeric expression*, like A+B, 42.34. Numeric expressions can be numeric variables (A, DOLLARS), numbers (−1, 2.45, 1.562E15), or numeric expression (HOURS+OTHOURS, SQR(VALUE), POINTS*50+250).
4. *num-var* means any *numeric variable*, like X, INTEREST.
5. *str-exp* means any *string expression*, like A$, "XYZ", FIRST$&MIDDLE$&LAST$. String expressions can be string variables (A$, MYNAME$), string data items ("ABC", "HELLO THERE!"), or string expressions (FIRST$&LAST$, SEG$(ABC$,1,1)).
6. *str-var* means any *string variable*, like Y$, NAME$.
7. *variable* means any variable, string or numeric, like YES$, PAYMENT.
8. *brackets* ([ ]) mean whatever is between the [ ] is *optional*. You do not have to use whatever's between the brackets if you don't want to.
9. *ellipsis* (. . .) means that the preceding item can be repeated as many times as necessary.

10. *device-filename* means the device for cassette files (like CS1). For disk files it means the name of the file on the disk as well as the device name (like DSK1.MYFILE).
11. When there is more than one format for a TI BASIC element (like optional operands), we show you all the formats.

COMMANDS, STATEMENTS, AND FUNCTIONS

The remainder of Chapter 4 lists the Commands, Statements, and Functions of TI BASIC in alphabetical order. Each element is listed and described separately.

ABS	Absolute value of an expression.
Type:	Function
Format:	ABS(*num-exp*)
Purpose:	ABS returns the ***absolute value*** (positive value) of the number represented by *num-exp*.
Operands:	*num-exp* is any number, numeric variable, or numeric expression.
Defaults:	None.

Description:

ABS is a numeric function that returns the absolute value (positive value) of *num-exp*. Table 4-1 shows you how ABS works on *num-exp*.

Table 4-1. ABS Results

num-exp	*Result*
less than zero	ABS returns the positive value of *num-exp*.
−12.345	ABS returns 12.345
zero	ABS returns zero.
greater than zero	ABS returns the value of *num-exp*.
987.654	ABS returns 987.654

ABS is a *function*. You use ABS to put a value into a variable, like this:

```
RDANS = ABS(ANS) + .5
DOLLARS = 2.50*ABS(XINT)
ANS = ABS(ASIDE - BSIDE)
```

Or, you use ABS as part of a numeric expression, like this:

```
ASIDE = SQR(ABS(CSIDE^2 - BSIDE^2))
RESULT = SQR(ABS(X*Y - Z^3 + R*Q/D)) + 75.689
IF ABS(ANS - 50)<500 THEN 1000 ELSE 2000
```

ABS is useful when you are evaluating an expression for an ON . . . GOSUB or ON . . . GOTO statement. Because these statements cannot use negative or zero values for their operands, you can use ABS to be sure the operand is positive. (Remember that you should also check for zero and whatever range you use in the statements.)

Common Errors:

STRING-NUMBER MISMATCH

You used a string expression instead of a numeric expression for *num-exp*.

Example:

The program in Listing 4-1 uses ABS to print the absolute value of whatever number you enter. Enter positive and negative numbers, large and small numbers, so that you can see just what happens when you use ABS.

```
100  CALL CLEAR
110  PRINT "I'LL PRINT THE POSITIVE":
     "FORM OF WHATEVER NUMBER":"YOU ENTER."
120  PRINT :
130  INPUT "YOUR NUMBER -> ":ANS
140  PRINT :"THE ABSOLUTE VALUE OF";ANS;"IS";ABS(ANS): :
150  INPUT "TRY AGAIN? (Y/N) -> ":Y$
160  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
170  PRINT :"GOODBYE."
180  END
```

Listing 4-1. ABS Example

ASC	ASCII value of first string character.
Type:	Function
Format:	ASC (*str-exp*)
Purpose:	ASC returns a number that is the ASCII code for the first character of the string *str-exp*. Table 4-2 gives the ASCII codes for the characters.
Operands:	*str-exp* is any string, string variable, or string expression.
Defaults:	None.

Description:

ASC is a function that returns a number that is the ASCII value of the first character of *str-exp*. Table 4-2 shows you what ASC returns as the ASCII code for the various characters.

Even though ASC returns a number, it is considered one of the string-related functions. It's very useful with the other string functions (SEG$, CHR$, LEN, POS, SEG$, STR$, VAL) when you want to do some special string processing. You can use ASC with HCHAR and VCHAR when you're writing to specific locations on your screen.

ASC can put a value into a variable, like this:

```
ASCCODE = ASC("HELLO")
ANS = ASC(SEG$(NAME$,N,1))
```

Table 4-2. ASC and ASCII Character Codes

ASCII Char	Decimal	ASCII Char	Decimal	ASCII Char	Decimal
(space)	32	@	64	`	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
−	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	|	124
=	61	]	93	}	125
>	62	^	94	~	126
?	63	_	95	(DEL)	127

Or, ASC can be part of an expression, like this:

```
IF (ASC(ANS$)>=97)*(ASC(ANS$)<=122) THEN 500
CALL HCHAR(10,15,ASC(SEG$(MSG$,N,1),5))
```

The first statement above uses ASC to see if the first character in the string ANS$ is a lower-case character (ASCII values 97 through 122 represent the lower-case letters a through z). The second statement uses ASC to translate a string value to a numeric value for the HCHAR statement because HCHAR needs a number, not a string.

Common Errors:

STRING-NUMBER MISMATCH

You used a numeric expression instead of a string expression for *str-exp*.

BAD ARGUMENT

The string expression *str-exp* is, or evaluates to, the null string (a string whose length is zero).

Example 1:

The program in Listing 4-2 uses ASC to encode a message into the numbers representing the ASCII values of the letters. Try entering the same message in uppercase only, lowercase only, and both uppercase and lowercase.

You can use this program as a beginning for a code generator program. Add or subtract a fixed amount (your secret code number) from the ASCII values and use CHR$ to rewrite the message. Be careful that you don't go beyond the ASCII values that represent printable characters (33 to 126) when you code your new value. You could even use different "secret codes" for different people.

```
100  CALL CLEAR
110  PRINT "HI THERE.":"I'M A MAGIC CODE GENERATOR"
120  PRINT :"ENTER A MESSAGE":"AND I'LL TRANSLATE"
130  PRINT "IT INTO A NUMERIC CODE.": :
140  PRINT : :
150  INPUT "YOUR MESSAGE -> ":MSG$
160  IF LEN(MSG$)=0 THEN 210
170  PRINT :"YOUR CODED MESSGE IS": :
180  FOR J=1 TO LEN(MSG$)
190  PRINT ASC(SEG$(MSG$,J,1));
200  NEXT J
210  PRINT : :
220  INPUT "TRY AGAIN? (Y/N) ->":Y$
230  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 140
240  PRINT :"GOODBYE":" OR ": :
250  MSG$="GOODBYE"
260  FOR J=1 TO LEN(MSG$)
270  PRINT ASC(SEG$(MSG$,J,1));
280  NEXT J
290  END
```

Listing 4-2. ASC Example 1

Example 2:

The program in Listing 4-3 uses ASC to translate lower-case letters (a->z) to upper-case letters (A->Z). The lower-case letters have ASCII values 97 (a) through 122 (z); upper-case letters, ASCII values 65 (A) through 90 (Z).

You subtract 32 from the ASCII value of a lower-case letter and use CHR$ with the result to get the upper-case letter. Try changing the program to convert in the other direction (uppercase to lowercase). Or, convert every other letter in the message.

```
100  CALL CLEAR
110  PRINT "I'LL TRANSLATE":"LOWERCASE TO UPPERCASE"
120  PRINT : :
130  INPUT "YOUR MESSAGE -> ":MSG$
140  IF LEN(MSG$)=0 THEN 260
150  OUTMSG$=""
160  FOR K=1 TO LEN(MSG$)
170  ASCVAL=ASC(SEG$(MSG$,K,1))
180  IF ASCVAL<97 THEN 210
190  ASCVAL=ASCVAL-32
200  OUTMSG$=OUTMSG$&CHR$(ASCVAL)
210  NEXT I
220  PRINT :"YOUR ORIGINAL MESSAGE WAS":MSG$
230  PRINT :"YOUR TRANSLATED MESSAGE IS":OUTMSG$: :
240  INPUT "ANOTHER MESSAGE? (Y/N) -> ":Y$
250  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
260  PRINT : :"GOODBYE."
270  END
```

Listing 4-3. ASC Example 2

ATN	Get the arctangent.
Type:	Function
Format:	ATN *(num-exp)*
Purpose:	ATN returns the trigonometric arctangent of *num-exp*. The arctangent is expressed as an angle in radians.
Operands:	*num-exp* is a number, numeric variable, or numeric expression.
Defaults:	None

Description:

ATN is a trigonometric function that returns the arctangent of *num-exp*. The arctangent is the angle, expressed in radians, whose tangent is *num-exp*.

The ATN function returns an angle (in radians) in the range (where PI = 3.14159265359):

$$-PI/2 < ATN(\textit{num-exp}) < PI/2$$

You can use ATN to convert angles between radians and degrees.

1. To convert angles from radians (RADS) to degrees (DEGS), multiply the radians by 180/(4*ATN(1)):

$$DEGS = RADS*180/(4*ATN(1))$$

2. To convert angles from degrees (DEGS) to radians (RADS), multiply the degrees by (4*ATN(1))/180:

$$RADS = DEGS*(4*ATN(1))/180$$

Common Errors:

STRING – NUMBER MISMATCH

You used a string expression instead of a numeric expression for *num-exp*.

Example 1:

The program in Listing 4-4 uses ATN to convert angles from degrees to radians. Try changing it to convert from radians to degrees.

```
100  CALL CLEAR
110  PRINT "I'LL CHANGE ANGLES":
     "FROM DEGREES TO RADIANS"
120  PRINT :
130  INPUT "YOUR ANGLE -> ":ANS
140  DEGS=ANS
150  IF DEGS<=360 THEN 180
160  DEGS=DEGS-360
170  GOTO 150
180  PRINT :ANS;"DEGREES IS "
190  PRINT DEGS*(4*ATN(1))/180;"RADIANS"
200  INPUT "TRY AGAIN? (Y/N) -> ":Y$
210  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
220  PRINT :"GOODBYE"
230  END
```

Listing 4-4. ATN Example 1

Example 2:

The program in Listing 4-5 prints the arctangent of whatever value you enter. Remember, the arctangent is the angle, expressed in radians, whose tangent is the value *num-exp*.

Try changing the program to also print the angle in degrees.

```
100  CALL CLEAR
110  PRINT "ENTER A NUMBER AND":
     "I'LL TELL YOU IT'S":"ARCTANGENT"
120  PRINT : :
130  INPUT "YOUR NUMBER -> ":ANS
140  PRINT "THE ARCTANGENT OF";ANS;"IS";ATN(ANS) : :
150  INPUT "TRY AGAIN? (Y/N) -> ":Y$
160  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
170  PRINT :"GOODBYE"
180  END
```

Listing 4-5. ATN Example 2

BREAK	Set one or more breakpoints.
Type:	Command
Format:	[*line#*] BREAK or [*line#*] BREAK *line-num-list*

BREAK	Set one or more breakpoints. *(continued)*
Purpose:	BREAK sets *breakpoints* (places where your program stops before executing the statement) at selected lines in your program. BREAK is very useful in debugging programs.
Operands:	*line#* is a BASIC statement line number that you need when you include BREAK in a program. You don't need *line#* when you use BREAK as a command. *line#* can be any number between 1 and 32767. *line-num-list* is a list of BASIC statement line numbers, separated by commas when you use more than one line number, where you want to set breakpoints in your program (where your program will stop). The line numbers can be any number between 1 and 32767.
Defaults:	When you use BREAK as a statement in a program and you don't use a *line-num-list*, TI BASIC sets the breakpoint at the *line#* of the BREAK statement.

Description:

BREAK sets breakpoints in your BASIC program. A breakpoint is a marker in the program set at the statements whose line numbers appear in *line-num-list*, or at the BREAK statement if you include no *line-num-list*.

When TI BASIC reaches a breakpoint, it doesn't execute the statement, but stops and prints the message

```
BREAKPOINT AT LINE XXX
```

then waits for you to enter a command. You can PRINT variable values, change variable values, or check some calculations. For example, you can look at values of some variables (like PRINT INAMT,OUTAMT); print calculated values (like PRINT NEWVAL/2); or change variable values (like RATE = .6). You resume execution with a CONTINUE command.

You cannot edit lines in your program (add, delete, or change lines). If you try to continue after editing your program at a breakpoint, you'll get the message:

```
CANT CONTINUE
```

When you reach a breakpoint, TI BASIC restores the standard character set (the one it starts with). If you've re-defined any characters with CALL CHAR, the new (changed) characters on your screen will be replaced by the standard characters. Re-defined ASCII characters in the range from 128 to 159 are not affected by the breakpoint processing. (See the section on CALL CHAR for details on making your own characters.)

You remove breakpoints by:

- Entering a NEW command
- Entering a SAVE command
- Entering an UNBREAK command
- Entering a CONTINUE command after you reach a breakpoint set by a BREAK command.

BREAK is very useful in debugging a program. You use the BREAK command to do different debugging tasks than you do with the BREAK statement. First, you must understand how BREAK works as a command and as part of a program.

1. When you use BREAK as a command, you must set at least one breakpoint by entering a line number in the *line-num-list*. When BASIC reaches this breakpoint, it *clears* the breakpoint when you enter a CONTINUE command. That means that you have to reset the breakpoint with another BREAK command if you want your program to stop at the line again.
2. When you use BREAK as a statement in a program, you don't have to enter any line numbers in the *line-num-list*. (You can, of course, enter as many line numbers as you need.) BASIC stops at the breakpoint set by the BREAK statement, in the same way as it does for a BREAK command. However, even though the breakpoint is cleared when you continue your program, the breakpoint is reset when BASIC next executes the BREAK statement.

NOTE

When you use BREAK as a statement in a program and you don't use a *line-num-list*, the breakpoint is set at the BREAK statement. To remove this type of breakpoint, you must delete (remove) the BREAK statement from the program.

You can check to see if your program is nearing a group of selected statements with the BREAK command. Set breakpoints for the statements that you want to check and RUN your program. Example 1 (below) shows you how to do this. If your program is supposed to be doing some processing and you never get to a breakpoint that you set, you know for certain that your program has at least one logic error.

With the BREAK statement you can set different breakpoints, depending on where you are in your program. For example, suppose you aren't sure if you're getting to either of two subprograms. You can set breakpoints at the beginning of both subprograms with a BREAK command and proceed as above.

But, suppose you know that at certain points in your program, you recalculate variables which affect your getting to these subprograms. You can set breakpoints with BREAK statements at various points in your program. The breakpoints can differ, depending on where you are in your program. This way when a breakpoint occurs, you know how you got to that point in your program. Example 2 (below) shows you how to set different breakpoints using BREAK statements.

Common Errors:

BAD LINE NUMBER

You used a line number in *line-num-list* that is less than or equal to zero or greater than 32767.

Or, you used a line number in *line-num-list* that is not a valid line number for a statement in your program.

CAN'T CONTINUE

You edited your program (changed a program statement) and tried to restart the changed program with a CONTINUE command.

INCORRECT STATEMENT

You forgot to put at least one line number in the *line-num-list* in your BREAK command. You don't need a *line-num-list* when you use BREAK as a statement.

Example 1:

The program in Listing 4-6 uses the BREAK command to set breakpoints at two different places in a program. The RND function is used to generate numbers.

It seems that the subprograms are never used. The BREAK command will show when a subprogram is reached. <ENTER> means press the **ENTER** key.

You'll see which subprogram, if any, the program uses. You have to enter another BREAK command if you want to stop the program at the same place after it reaches a breakpoint. You continue the program with a CONTINUE command.

```
NEW <ENTER>
NUM <ENTER>
100  CALL CLEAR
110  X=RND*10
120  IF X<5 THEN 140
130  GOSUB 200
140  PRINT X
150  IF X<2.5 THEN 170
160  GOSUB 230
170  INPUT "ENTER 0 TO STOP":ANS
180  IF ANS<>0 THEN 110
190  STOP
200  PRINT "HI THERE"
210  X=X*100
220  RETURN
230  PRINT "HI THERE"
240  X=X/500
250  RETURN
260  END
270  <ENTER>
BREAK 200,230 <ENTER>
RUN <ENTER>
```

Listing 4-6. BREAK Example 1

Example 2:

The program in Listing 4-7 uses BREAK as a program statement to set breakpoints at different places, depending on the value of a variable. The RND function generates random numbers.

When you run this example, you'll get breakpoints at different statements, depending on the value of X. You might want to use this technique when your program performs complicated calculations on variables that change in a number of places. At a breakpoint, PRINT the variables you want to check and see if they contain what you think they should.

```
100  CALL CLEAR
110  X=RND*100
120  IF X<50 THEN 150
130  BREAK 170,190
140  GOTO 160
150  BREAK 190,210
160  PRINT "X=";X
170  Y=X/50
180  X=25
190  Q=SQR(ABS(Y-X))
200  Y=Q-3500
210  X=75
220  INPUT "PRESS ENTER TO STOP":X$
230  END
```

Listing 4-7. BREAK Example 2

BYE	Close all files and leave TI BASIC.
Type:	Command
Format:	BYE
Purpose:	BYE closes all your *opened* files, erases your program and variables, and resets the computer before returning you to the main title screen.
Operands:	None.
Defaults:	None.

Description:

BYE closes all *opened* files, erases the program and the variable values, resets the computer, and leaves TI BASIC.

After you enter BYE, your computer will show the main title screen (the one you get when you turn it on).

You can also use a FCTN QUIT (FCTN =) to exit TI BASIC. However, FCTN QUIT does not close any files that you may have *opened*.

It's good practice to use a BYE command when you're done using TI BASIC.

Common Errors:

```
CANT DO THAT
```

You tried to use BYE as a program statement. BYE must be used as a command.

Example:

The program in Listing 4-8 shows you how to enter a short program, RUN it, and exit from BASIC through a BYE.

When you're done, you'll see the main title screen, the one that you get when you turn on your computer.

<ENTER> means to press the **ENTER** key.

```
NEW <ENTER>
NUM <ENTER>
100  PRINT "HELLO"
110  END
120  <ENTER>
RUN <ENTER>
     The computer prints:
HELLO
BYE <ENTER>
```

Listing 4-8. BYE Example

CALL CHAR	Define a character image.
Type:	Statement
Format:	[*line#*] CALL CHAR*(ASCII-code,pattern-string)*
Purpose:	CALL CHAR redefines the character *ASCII-code* to have the shape defined by dots represented by the hexadecimal information in *pattern-string*.
Operands:	*line#* is a BASIC statement line number that you need when you include CALL CHAR in a program. You don't need *line#* when you use CALL CHAR as a command. *line#* can be any number between 1 and 32767. *ASCII-code* is a number, numeric variable, or numeric expression that represents the ASCII value identifying the character that you want to redefine. *ASCII-code* can be any decimal value between 32 and 159. *pattern-string* is a string, string variable, or string expression that contains up to 16 hexadecimal digits (0 – >9, A – >F) defining the shape of the new character referenced as *ASCII-code*.
Defaults:	If the *pattern-string* contains fewer than 16 hexadecimal digits, TI BASIC uses zeros for the remaining digits. Any digit past the sixteenth is ignored.

Description:

CALL CHAR defines the pattern or image associated with the character represented by the decimal number *ASCII-code*. The new pattern is defined by up to 16 hexadecimal digits in the string *pattern-string*.

CALL CHAR only defines the form for the character. You write the new character to the screen with DISPLAY, PRINT, CALL HCHAR, or CALL VCHAR.

In TI BASIC, you can redefine any character with *ASCII-codes* 32 through 127 (the standard character set of upper-case and lower-case letters, numbers, special characters, and blank). Table 4-3 lists ASCII codes for the standard characters.

You can also define ASCII codes 128 through 159 for your own special characters so that you can use all the standard characters and 32 additional characters.

Table 4-3. CHAR and ASCII Character Codes

ASCII Char	Decimal	ASCII Char	Decimal	ASCII Char	Decimal
(space)	32	@	64	‘	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
–	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	|	124
=	61	]	93	}	125
>	62	^	94	~	126
?	63	_	95	(DEL)	127

If you redefine one of the standard characters (ASCII codes 32 through 127), the redefined character reverts to its standard (original) representation when your program ends. Any redefined standard characters on your screen will "blink" back to their original forms when your program stops.

In a similar way, any standard characters that you redefine with CALL CHAR that are already printed on the screen will "blink" to the new shape when your program executes the CALL CHAR. Another CALL CHAR for the same character will once again change the redefined characters already on the screen.

When your program ends, all ASCII characters defined in the range from 128 to 159 are reset to undefined. If you want to use them again, you have to CALL CHAR for them again.

NOTE

If you redefine any standard ASCII-code from characters 32 through 127 and your program stops for a breakpoint (through a BREAK command or statement), the redefined characters resume their standard forms. The definitions for ASCII codes 128 through 159 are not affected by a BREAK.

Before you can start redefining characters, you have to understand how to design characters. Every character (numbers, letters, punctuation, and special graphics characters) that TI BASIC draws on your TV screen is made up of "dots" or *pixels* (picture elements).

Each character is really a grid of eight rows of eight dots. Each row has two parts: a left part of four dots and a right part of four dots. Fig. 4-1 shows you:

- A single row with the left and right parts marked
- An entire character grid of eight rows and eight columns with the row numbers and columns marked
- A drawing of the dots TI BASIC uses for the character "A"

Characters are formed by turning the 64 dots "on" or "off". The blank or space character (ASCII code 32) has all the dots turned off. If you turned on all the dots, you would have a solid square character.

You define your new character by a 16-hexadecimal digit *pattern-string*. Each hexadecimal digit represents a left or right side of a row, beginning with row 1, left side. The hexadecimal digits, themselves, represent a binary code of four digits that can be ones ("on" dots) or zeros ("off" dots). Fig. 4-2 shows you the pattern-string diagram of the on/off dots that correspond to the 16 hexadecimal digits.

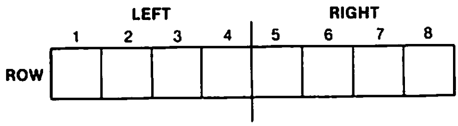
If you use fewer than 16 hexadecimal digits in your *pattern-string*, the remaining digits are set to zero. If you use more than 16 digits, any digit past the sixteenth is ignored.

NOTE

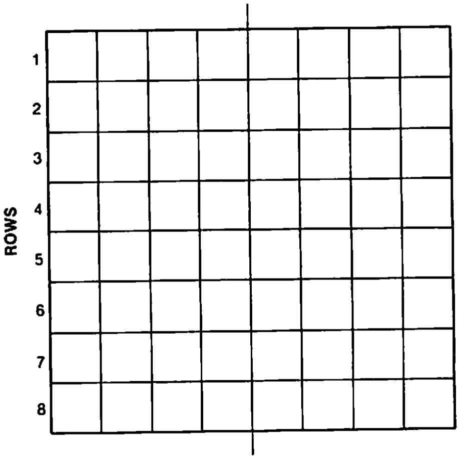
The hexadecimal digits are 0−>9 and A−>F. You *MUST* use *UPPER-CASE* letters for A−>F. If you use lower-case letters, you'll get an error.

If you don't want to remember the hexadecimal digits associated with the 16 possible on/off dot patterns, you can use the method shown in Fig. 4-3 to calculate each hexadecimal digit.

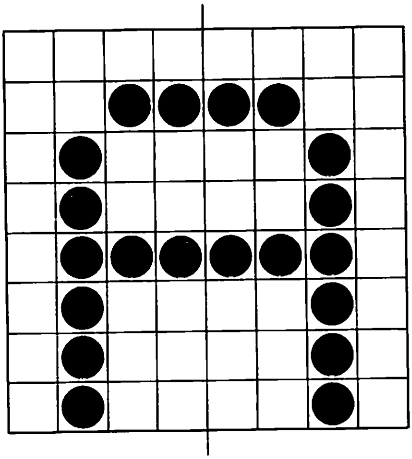
Each side (left and right) of a row consists of four dots. If you number the dot positions (8-4-2-1), as shown in Fig. 4-3, you simply add the



(A) Single row of eight dots.

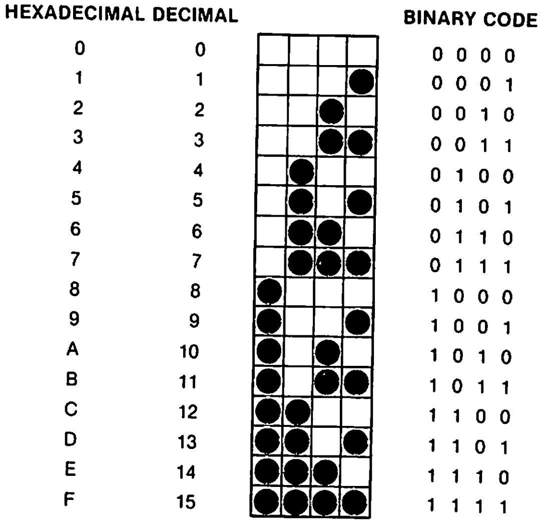


(B) Eight rows of eight dots.



(C) Dots used to make the character A.

Fig. 4-1. Character grid diagram.



HEXADECIMAL	DECIMAL		BINARY CODE
0	0		0 0 0 0
1	1	●	0 0 0 1
2	2	●	0 0 1 0
3	3	●●	0 0 1 1
4	4	●	0 1 0 0
5	5	● ●	0 1 0 1
6	6	●●	0 1 1 0
7	7	●●●	0 1 1 1
8	8	●	1 0 0 0
9	9	● ●	1 0 0 1
A	10	● ●	1 0 1 0
B	11	● ●●	1 0 1 1
C	12	●●	1 1 0 0
D	13	●● ●	1 1 0 1
E	14	●●●	1 1 1 0
F	15	●●●●	1 1 1 1

Fig. 4-2. Pattern-string diagram.

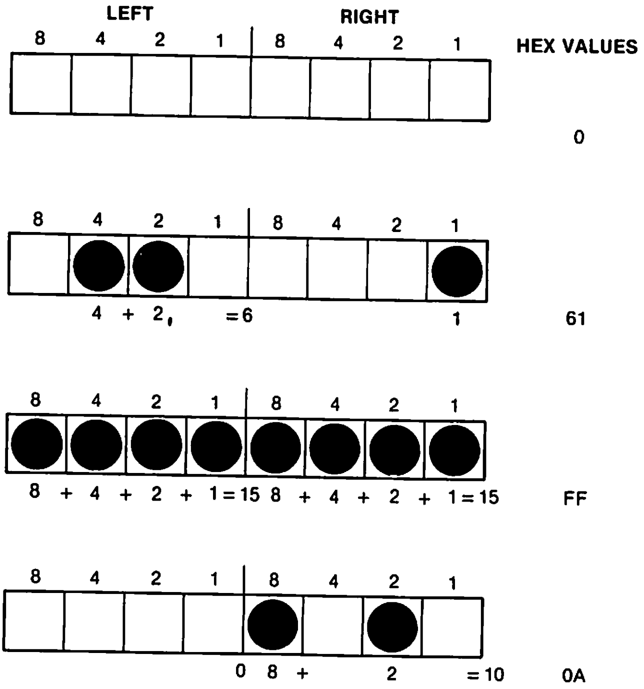
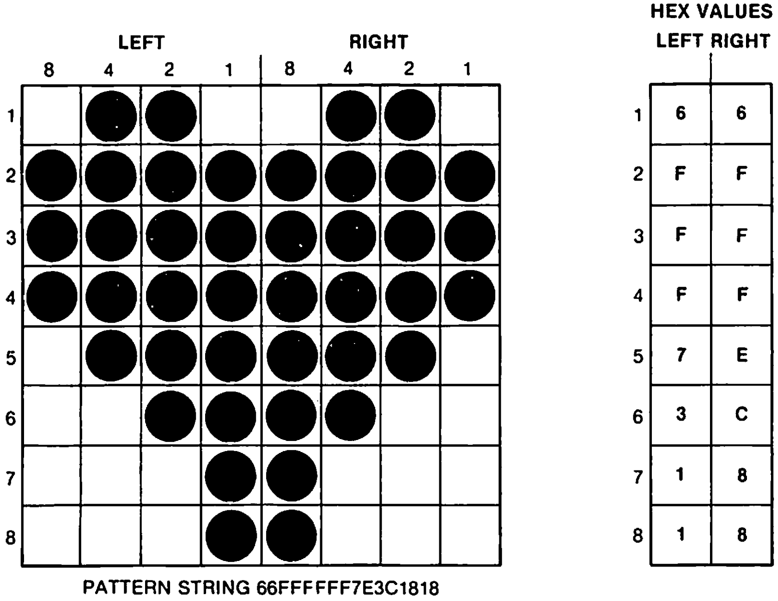


Fig. 4-3. Hexadecimal code diagram.

values where a dot is turned on to get the hexadecimal digit. Remember that you need *two* hexadecimal digits for each row. When all the dots are off, the digit is zero.

To see how to design your own characters, look at the heart-shaped figure in Fig. 4-4. The left drawing shows you which dots in the eight by eight character grid are turned on (set) to make the figure. The right drawing shows you the hexadecimal values corresponding to the dots for the figure's rows. You'll also see the *pattern-string* for the figure.



	LEFT				RIGHT			
	8	4	2	1	8	4	2	1
1		●	●			●	●	
2	●	●	●	●	●	●	●	●
3	●	●	●	●	●	●	●	●
4	●	●	●	●	●	●	●	●
5		●	●	●	●	●	●	
6			●	●	●	●		
7				●	●			
8				●	●			

PATTERN STRING 66FFFFFF7E3C1818

HEX VALUES	LEFT	RIGHT
1	6	6
2	F	F
3	F	F
4	F	F
5	7	E
6	3	C
7	1	8
8	1	8

Fig. 4-4. Heart-shaped figure example.

One final precaution. When you define characters with ASCII codes 128 through 159, you are using additional memory to store the new definitions. (You don't use additional memory when you *redefine* the standard characters with ASCII codes 32 through 127.) If your program is very large, you may get a MEMORY FULL error when you define your extra characters.

Common Errors:

BAD VALUE

You used a value for *ASCII-code* that is less than 32 or greater than 159.

Or, you have an invalid character in your *pattern-string*. The valid characters are 0123456789ABCDEF. You cannot use lower-case letters.

MEMORY FULL

There isn't enough memory left to contain the character definition. Try to make the program shorter if possible. (Try shortening messages, remarks, or extra-long variable names.)

Example 1:

The program in Listing 4-9 uses CALL CHAR to redefine the space character (*ASCII-code* 32) to a character made up of vertical stripes.

CALL CLEAR fills the screen with space characters. When you run the program the screen will fill with stripes. When the program ends the screen reverts to spaces because you redefined a standard character.

Try changing the pattern definition string in CALL CHAR to make the space character turn into horizontal or slanted stripes.

```
100  CALL CHAR(32,"F0F0F0F0F0F0F0F0")
110  CALL CLEAR
120  INPUT "PRESS ENTER TO STOP ME.":X$
130  END
```

Listing 4-9. CHAR Example 1

Example 2:

The program in Listing 4-10 redefines the letter O (*ASCII-code* 79) to horizontal stripes ("00FF00FF00FF00FF") and defines an additional character (*ASCII-code* 135) to a checkerboard pattern ("AA55AA55AA55AA55").

Then VCHAR fills the screen with the two characters. When the program stops you will see the stripes revert to capital O, because O is one of the standard characters. The checkerboard pattern remains on the screen.

Try other patterns for the two characters.

```
100  CALL CLEAR
110  CALL CHAR(79,"00FF00FF00FF00FF")
120  CALL CHAR(135,"AA55AA55AA55AA55")
130  FOR I=1 TO 32 STEP 2
140  CALL VCHAR(1,I,79,24)
150  CALL VCHAR(1,I+1,135,24)
160  NEXT I
170  INPUT "PRESS ENTER TO STOP":X$
180  END
```

Listing 4-10. CHAR Example 2

CHR$	Translate ASCII code to its character.
Type:	Function
Format:	CHR$ *(num-exp)*
Purpose:	The CHR$ function returns a one-character string which contains a character whose ASCII value is *num-exp*.
Operands:	*num-exp* is a number, numeric variable, or numeric expression whose value is between 0 and 32767.
Defaults:	None.

Description:

CHR$ is a string function that returns a one-character string containing the character whose ASCII value is *num-exp*. Table 4-4 lists the ASCII values for the standard character set.

The ASCII value of the character returned by CHR$ can range from 0 to 255. If you supply a *num-exp* that results in a value greater than 255, the ASCII value of the returned character is given by the expression:

$$\text{ASCII-value} = \textit{num-exp} - (\text{INT}(\textit{num-exp} / 256) * 256)$$

If *num-exp* is not an integer (whole number), it is *rounded* to the nearest integer value before it is used.

If you print a character that is outside the range of the normal character set (ASCII values 32 through 128) and you haven't defined the character with a CALL CHAR, you will see whatever is in the memory at the

Table 4-4. CHR$ ASCII Character Codes

ASCII Char	Decimal	ASCII Char	Decimal	ASCII Char	Decimal
(space)	32	@	64	‘	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
–	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	|	124
=	61	]	93	}	125
>	62	^	94	~	126
?	63	_	95	(DEL)	127

character's definition location. You may or may not get a printable character. Many times you will get what looks like a blank or a space character.

You often use CHR$ in games to write characters that you can't enter through the keyboard, like special graphics characters that you define. For example, to print a special graphics character defined as ASCII code 140, use

```
PRINT CHR$(140)
```

You can also use CHR$ to build a string containing characters that cannot be entered from the keyboard. This method uses CHR$ like this:

```
STRING$ = "ABCDEF"&CHR$(140)&CHR$(141)&"XYZ"
```

To print the value stored in STRING$ use:

```
PRINT STRING$
```

CHR$ is often used with CALL CHAR when designing special graphics characters or an alternate letter set.

Common Errors:

```
BAD VALUE
```

You used a value for *num-exp* that is less than zero or greater than 32767. Values between 32 and 127 give you the standard character set. Values between 128 and 159 give you your own, specially defined characters. Other values (less than 32 and greater than 159) give you a random character.

Example 1:

The program in Listing 4-11 uses CHAR to define a diagonally striped character and uses CHR$ to print the character across the screen.

Since the character is defined as ASCII code 129, the newly defined character remains unchanged on your screen when the program ends. Try defining the character as an ASCII code between 32 and 127 and see what happens.

```
100  CALL CLEAR
110  CALL CHAR(129,"AA552211AA552211")
120  FOR I=1 TO 10
130  PRINT CHR$(129);TAB(5);CHR$(79);TAB(10);
     CHR$(129);TAB(15);CHR$(90)
140  NEXT I
150  PRINT : : :
160  INPUT "PRESS ENTER TO STOP.":X$
170  END
```

Listing 4-11. CHR$ Example 1

Example 2:

The program in Listing 4-12 uses CHR$ to print the message "HELLO" using the ASCII values for the letters.

Try printing another greeting. Or, ask for some numbers between 32 and 128 and print the "secret" word you get by using CHR$ for the numbers.

```
100  CALL CLEAR
110  PRINT CHR$(72)&CHR$(69)&CHR$(76)&CHR$(76)&CHR$(79)
120  END
```

Listing 4-12. CHR$ Example 2

CALL CLEAR	Clear the screen.
Type:	Statement
Format:	[*line#*] CALL CLEAR
Purpose:	CALL CLEAR "clears the screen" by writing an entire screenful of blank characters (the character represented by *ASCII-code* 32).
Operands:	*line#* is a BASIC statement line number that you need when you include CALL CLEAR in a program. You don't need *line#* when you use CALL CLEAR as a command. *line#* can be any number between 1 and 32767.
Defaults:	None.

Description:

CALL CLEAR "clears the screen" to all blank characters (the character with ASCII code 32). When you clear the screen, the screen remains the background color that BASIC sets or that you set by CALL SCREEN.

CALL CLEAR erases everything that you wrote to the screen and fills the entire screen with blank characters. If you redefined the blank character using a CALL CHAR with an ASCII code of 32, the screen will be filled with the new blank character and not spaces. (This is a useful technique when you want to create a background effect.)

It's very useful to be able to clear the screen in your programs. You will notice that most of the examples in this book begin with a CALL CLEAR. This removes any previous messages, program lines, etc., from the screen. You know that anything printed is from the currently executing program.

You use CALL CLEAR in your program whenever you want a blank screen. It's common to clear the screen when you change processing, such as: from initialization to printing results, at a "new screen page," during error processing, etc.

Common Errors:

None.

Example 1:

The program in Listing 4-13 first fills the screen by printing the alphabet 23 times. Then, when you press the ENTER key, CALL CLEAR clears the screen to blanks.

```
100  FOR I=1 TO 23
110  PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
120  NEXT I
130  INPUT "PRESS ENTER TO CLEAR ME":X$
140  CALL CLEAR
150  INPUT "PRESS ENTER TO STOP":X$
160  END
```

Listing 4-13. CLEAR Example 1

Example 2:

The program in Listing 4-14 shows you what happens to CALL CLEAR when you redefine the blank character ASCII code 32.

First, redefine the blank character to a vertically striped character ("F0F0F0F0F0F0F0F0") by a CALL CHAR. Then, using a similar program to that in Example 1, you will see the screen turn striped instead of blank.

If you change the blank character definition to "00FF00FF00FF00FF", you will see horizontal lines instead of vertical stripes.

```
100  CALL CLEAR
110  CALL CHAR(32,"F0F0F0F0F0F0F0F0")
120  FOR I=1 TO 23
130  PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
140  NEXT I
150  INPUT "PRESS ENTER TO CLEAR ME":X$
160  CALL CLEAR
170  INPUT "PRESS ENTER TO STOP":X$
180  END
```

Listing 4-14. CLEAR Example 2

CLOSE #	Close a file.
Type:	Statement
Format:	[*line#*] CLOSE # *file-num* or [*line#*] CLOSE # *file-num* : DELETE
Purpose:	CLOSE # removes any association between a *file-num* and a file on a tape, disk, or other device. CLOSE # "closes a file."
Operands:	*line#* is a BASIC statement line number that you need when you include CLOSE # in a program. You don't need *line#* when you use CLOSE # as a command. *line#* can be any number between 1 and 32767. *file-num* is a number, numeric variable, or numeric expression that specifies a file number that appeared in a previous OPEN statement.
Defaults:	None.

Description:

CLOSE # closes the file OPENed as *file-num* and, when you add DELETE, removes any reference to the file from the device. You can delete files from a disk. If you say DELETE with a cassette file, the file is closed but not removed from the tape.

When you OPEN a file on a cassette, you see messages telling you to press buttons on your cassette recorder (CS1 or CS2) that look like this:

```
* REWIND CASSETTE TAPE   CSn
  THEN PRESS ENTER
* PRESS CASSETTE PLAY   CSn
  THEN PRESS ENTER
```

When you CLOSE # the cassette file, you will see the message:

```
* PRESS CASSETTE STOP   CSn
  THEN PRESS ENTER
```

This turns off the connection between your computer and your recorder and you won't be able to read from or write to the cassette until you again OPEN a cassette file.

To close the disk file you OPENed as *file-num* 45, use:

```
CLOSE # 45
```

To close this same disk file and *remove it from the disk,* use:

```
CLOSE # 45 : DELETE
```

One use for CLOSE # is OPENing and CLOSEing the same file for reading and writing. For example, first you OPEN the file for output and write to it. Then, when you're done writing, you CLOSE # the file and re-OPEN it for input so that you can read from it.

CLOSE # safeguards the data in your files. An improperly closed disk file may be unreadable. When an error occurs that stops your program, BASIC automatically closes all files which you OPENed in your program.

Your program can also stop with a BREAK command/statement or when you press the FCTN CLEAR key. When this happens, TI BASIC automatically closes your files if:

1. You end your TI BASIC session with a BYE command
2. You enter a NEW command to enter a new program
3. You RUN the program currently stopped
4. You edit one or more lines in the program

CAUTION
IF YOU USE FCTN QUIT TO END YOUR BASIC PROGRAM, YOUR FILES WILL NOT BE CLOSED AND YOU MAY LOSE YOUR DISK FILES!

You can OPEN a file with any *file-num* between 0 and 255. *file-num* 0 is reserved for the screen (output) and the keyboard (input). Other values can be used for any file on tape, disk, RS232, or other device.

OPEN sets up an association between the actual file on the device and the *file-num*. This association remains intact until you CLOSE # *file-num*. Then, you can reuse the file with another *file-num* or you can reuse the *file-num* with another file.

It's very convenient to use the DELETE option when you have OPENed a disk file, and then decide that you don't want to keep it after you've written to it. Or, maybe you've read the file and don't need it any more.

With the DELETE option you simply CLOSE # the *file-num* with a DELETE option. Suppose it's *file-num* 49. You use CLOSE # like this:

```
CLOSE # 49 : DELETE
```

The file associated with *file-num* 49 is now *deleted* and removed from the disk.

Common Errors:

BAD VALUE

You used a value for *file-num* that is less than zero or greater than 255.

FILE ERROR

You tried to CLOSE a file that you haven't OPENed yet.

INCORRECT STATEMENT

You don't have a number sign (#) before the *file-num* operand.

Or, you don't have a colon (:) before the DELETE operand.

Or, you have a colon (:) in your CLOSE statement but you don't have a DELETE operand.

I/O ERROR 13

You have an invalid CLOSE # command.

I/O ERROR 16

There is a device error. This can happen if you accidentally disconnect your cassette recorder or disk drive while your program is running. When you try to CLOSE # a file on the disconnected device, TI BASIC thinks there's a problem with the device since it's not properly connected.

STRING-NUMBER MISMATCH

You used a value for *file-num* that is not a number, numeric variable, or a numeric expression.

Example:

The program in Listing 4-15 OPENs a cassette file for output and writes whatever you enter to that file. Then it uses CLOSE # to close the file, clears the screen, and re-OPENs the file to read what was written. BASIC will tell you what to do with your cassette recorder when it gets to the OPEN and CLOSE # statements.

You will notice that the same file on the cassette is used with two different *file-num* values. You can use any value that you want, as long as you are consistent. When you OPEN a file with a specific *file-num*, you have to use that *file-num* for all INPUT # or PRINT # statements to it. Of course, you CLOSE # with the same value.

You can easily change the program to write the file to a disk file (if you have a disk). If you do, you can DELETE the file when you CLOSE # it after reading it. That way, you won't have junk files taking up space on your disks.

```
100  CALL CLEAR
110  PRINT "ENTER ANYTHING AND I'LL":
     "WRITE IT TO YOUR CASSETTE"
120  OPEN #9:"CS1",OUTPUT,INTERNAL,FIXED 64
130  PRINT "ENTER YOUR DATA": :
     "ENTER XXX WHEN YOU WANT":"TO STOP"
140  PRINT : :"DON'T FORGET QUOTES":
     "AROUND STRING DATA": :
150  INPUT "YOUR DATA -> ":X$
160  IF X$="XXX" THEN 190
170  PRINT #9: X$
180  GOTO 160
190  PRINT #9: X$
200  CLOSE #9
210  CALL CLEAR
220  PRINT "NOW, TO READ YOUR TAPE": :
230  OPEN #25:"CS1",INPUT,INTERNAL,FIXED 64
240  INPUT #25:X$
250  IF X$="XXX" THEN 280
260  PRINT X$
270  GOTO 240
280  PRINT : :"GOODBYE."
290  CLOSE #25
300  END
```

Listing 4-15. CLOSE # Example

CALL COLOR	Set foreground/background character colors.
Type:	Statement
Format:	[*line#*] CALL COLOR(*char-set, foreground-color, background-color*)
Purpose:	CALL COLOR sets the colors for the foreground of the character image and the background of the character. The colors are used to draw characters on your screen, it does not set the screen color.
Operands:	*line#* is a BASIC statement line number that you need when you include CALL COLOR in a program. You don't need *line#* when you use CALL COLOR as a command. *line#* can be any number between 1 and 32767. *char-set* is a number, numeric variable, or numeric expression that

CALL COLOR Set foreground/background character colors. *(continued)*

specifies which of the 16 sets of 8 characters you are setting for the foreground color and which you are setting for the background color. *char-set* may be any value between 1 and 16.

foreground-color is a number, numeric variable, or numeric expression that specifies the color for the dots (foreground color) used to draw the characters in *char-set*. *foreground-color* may be any of the valid colors between 1 and 16.

background-color is a number, numeric variable, or numeric expression that specifies the color for the dots that make up the background in drawing the characters in *char-set*. *background-color* may be any of the valid colors between 1 and 16.

Defaults: None.

Description:

Every character TI BASIC draws on your screen is made up of 64 dots (see the CHAR discussion for details). The dots that are used to draw the character itself are called the foreground dots. The other dots make up the background.

When a dot is set on (value 1), it is displayed in the foreground color. When a dot is off (value 0), it is displayed in the background color.

CALL COLOR sets the foreground and background colors for the characters in *char-set*. This is a very powerful tool for designing complex screens. Not only can you decide what color the entire screen will be, but you can also decide to set the characters themselves to different colors on different backgrounds. CALL COLOR is very effective when you are drawing special characters in games or border designs for screens.

The characters with ASCII values 32 through 159 are divided into 16 sets of 8 characters each, as shown in Table 4-5. The ASCII character

Table 4-5. COLOR Character Sets

char-set	*ASCII-values*
1	32-39
2	40-47
3	48-55
4	56-63
5	64-71
6	72-79
7	80-87
8	88-95
9	96-103
10	104-111
11	112-119
12	120-127
13	128-135
14	136-143
15	144-151
16	152-159

Note: Sets 1 through 12 are the characters in the standard character set. Sets 13 through 16 are the special character set.

codes are shown in Table 4-6. The 16 possible colors are shown in Table 4-7.

Each CALL COLOR sets the foreground and background colors for one character set. You do not change the *screen color*. You use CALL SCREEN to change the color of the entire screen.

Suppose you use CALL SCREEN to set the screen color to light blue (6) and use CALL COLOR to set all the character sets to a dark red (7) foreground on a light yellow (12) background. When you PRINT, DISPLAY, CALL VCHAR, or CALL HCHAR to write to the screen, you will see the characters written as dark red on light yellow squares. Any spaces that you don't write to will remain light blue.

Transparent (1) lets the screen color (set through CALL SCREEN) to show through as either the background or foreground color, depending on how you use it. The standard colors that TI BASIC uses are:

background-color = 1 (Transparent)
foreground-color = 2 (Black)

Table 4-6. COLOR and ASCII Character Codes

ASCII Char	Decimal	ASCII Char	Decimal	ASCII Char	Decimal
(space)	32	@	64	'	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
–	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	¦	124
=	61	]	93	}	125
>	62	^	94	~	126
?	63	_	95	(DEL)	127

Table 4-7. COLOR Codes

foreground-color *background-color*	*Color*
1	Transparent
2	Black
3	Medium Green
4	Light Green
5	Dark Blue
6	Light Blue
7	Dark Red
8	Cyan
9	Medium Red
10	Light Red
11	Dark Yellow
12	Light Yellow
13	Dark Green
14	Magenta
15	Gray
16	White

This combination draws black characters on whatever color the screen is (light green, or 4 as the BASIC standard). The color combinations reset to the BASIC standards when your program ends or when you BREAK your program.

NOTE

Remember that the space character (ASCII value 32) belongs to *char-set* 1. If you change this character set, any blanks that you write to the screen will be solid *background-color.*

Common Errors:

BAD VALUE

You used a value for *char-set* that is less than 1 or greater than 16.

Or, you used a value for *foreground-color* or *background-color* that is less than 1 or greater than 16.

Example 1:

The program in Listing 4-16 uses CALL COLOR to change the character sets 1 through 12 to give a rainbow effect when you print something. The screen is first set to gray (15).

```
100  CALL CLEAR
110  CALL SCREEN(15)
120  FOR I=1 TO 12
130  CALL COLOR(I,I,13-I)
140  NEXT I
150  INPUT "ENTER A MESSAGE -> ":X$
160  INPUT "PRESS ENTER TO STOP":X$
170  END
```

Listing 4-16. CALL COLOR Example 1

You will notice that some of the random color combinations are quite difficult to read. Try changing the program to make the combinations more readable.

Example 2:

The program in Listing 4-17 sets the upper-case letters (ASCII values 65 through 90, *char-set* 5 to 8) to dark blue (5) on cyan (8). The lower-case letters (ASCII values 97 through 122, *char-set* 9 to 12) are set to white (16) on dark green (13). Note that some special characters are also affected by the color changes.

When you enter a message, use both upper-case and lower-case letters to get a rainbow effect. Use **FCTN CLEAR** to stop the program.

Try different color combinations to see what you like best, as not all color combinations look good on every television.

```
100  CALL CLEAR
110  PRINT "ENTER A MESSAGE":"PRESS FCTN CLEAR TO STOP"
120  FOR I=5 TO 8
130  CALL COLOR(I,5,8)
140  CALL COLOR(I+4,16,11)
150  NEXT I
160  INPUT "YOUR MESSAGE -> ":X$
170  PRINT : X$ : :
180  GOTO 160
190  END
```

Listing 4-17. CALL COLOR Example 2

CONTINUE or CON	Continue an interrupted program.
Type:	Command
Format:	CONTINUE or CON
Purpose:	CONTINUE (or its abbreviation CON) resumes executing your program after the program stopped because of BREAK or because you pressed the **FCTN CLEAR** key.
Operands:	None.
Defaults:	None.

Description:

CONTINUE or CON resumes executing a BASIC program after:

- The program executes a BREAK statement/command
- The program stops because of an error
- You pressed **FCTN CLEAR**

It doesn't matter if you use CONTINUE or its abbreviation CON. Neither CONTINUE nor CON can be used as a statement in a program. CONTINUE is a command only.

When you CONTINUE the program, it begins executing the line immediately following the last line it executed before it stopped.

BREAK and CONTINUE are very useful in debugging programs. You use BREAK to set breakpoints (places in your program where it will stop) so that you can check whether your program's working correctly.

If you haven't set breakpoints through BREAK, you can press the FCTN CLEAR (FCTN 4) to make your program stop. If the program is waiting for you to enter data (with an INPUT statement), it will stop immediately. If it is running, you cannot be sure exactly where it will stop. You can be sure, however, that it will CONTINUE at the proper place.

NOTE

You cannot CONTINUE a program once you edit (change) any lines in the program.

You can look at any variable values (with PRINT or DISPLAY) and you can reset the values of any variables with an assignment statement (A = 5) before continuing program execution.

Common Errors:

CAN'T CONTINUE

You tried to CONTINUE after you edited (changed) one or more lines in your program. You can PRINT or change variables or LIST program statements, but you cannot change the program.

CAN'T DO THAT

You tried to use CONTINUE or CON as a statement in your program. You can use CONTINUE or CON only as a command.

Example 1:

The program in Listing 4-18 uses FCTN CLEAR (FCTN 4) to stop a listing. Then, you enter a CONTINUE (or, if you prefer, CON) command to resume the listing.

See how fast you can stop the program. Try to change a line and see

```
100  CALL CLEAR
110  PRINT "THIS PROGRAM SHOWS YOU":
     "  HOW TO USE CONTINUE"
120  PRINT :"PRESS FCTN 4 TO STOP IT": :
     "  THEN USE CON TO RESUME"
130  PRINT "TRY IT SEVERAL TIMES.": : :
140  INPUT "PRESS ENTER TO BEGIN.":X$
150  FOR I=1 TO 1000
160  PRINT I;I*.5;
170  NEXT I
180  END
```

Listing 4-18. CONTINUE Example 1

whether or not the program will CONTINUE. If it doesn't, RUN it again. Print some values, maybe PRINT I before you CON.

Example 2:

The program in Listing 4-19 uses a BREAK statement to set a breakpoint. When the program stops, use CON to resume execution. The program will stop every time it gets to the BREAK statement.

Try changing the position (line number) of the BREAK statement and see what happens.

```
100  CALL CLEAR
110  PRINT "YOU HAVE TO USE CON":
     "TO RESTART THE PROGRAM"
120  PRINT "EVERY TIME IT REACHES":"LINE 160."
130  PRINT :"IT WILL ONLY STOP 10 TIMES"
140  PRINT : :"GET READY."
150  FOR I=1 TO 10
160  BREAK
170  PRINT "I=";I
180  NEXT I
190  END
```

Listing 4-19. CONTINUE Example 2

COS	Cosine of an angle in radians.
Type:	Function
Format:	COS(*rad-angle*)
Purpose:	COS returns the trigonometric cosine of *rad-angle* where *rad-angle* is an angle expressed in radians.
Operands:	*rad-angle* is a number, numeric variable, or numeric expression that represents an angle expressed in radians.
Defaults:	None.

Description:

COS is a trigonometric function that returns the cosine of the angle *rad-angle*, where *rad-angle* is expressed in radians. COS returns a value between − 1 and + 1.

COS is used in games where you want to calculate a distance between two points on a grid. It's sometimes used to generate a random number between − 1 and + 1 (RND generates a random number between 0 and 1).

NOTE

rad-angle must be expressed in *radians*, not degrees. If you want to convert degrees to radians, use one of the following expressions (PI = 3.14159):

RADIANS = DEGREES * PI / 180

or

RADIANS = DEGREES * (4 * ATN(1))/180
or
RADIANS = DEGREES * .01745329251994

Common Errors:

BAD ARGUMENT

The value of *rad-angle* is greater than $1.5707963266375*10^{10}$ or less than $-1.5707963266375*10^{10}$

STRING-NUMBER MISMATCH

You used a string instead of a number for *rad-angle*.

Example:

The program in Listing 4-20 uses the (4 * ATN(1))/180 formula to convert degrees to radians. Then it prints the COS of the angle.

```
100  CALL CLEAR
110  PRINT "ENTER AN ANGLE IN DEGREES"
120  PRINT :"I'LL CONVERT THE ANGLE":"  TO RADIANS AND"
130  PRINT "  PRINT ITS COSINE.": :
140  INPUT "YOUR ANGLE (999 TO STOP)->":ANGLE
150  IF ANGLE<>999 THEN 180
160  PRINT :"GOODBYE"
170  STOP
180  RADS=ANGLE*(4 * ATN(1))/180
190  PRINT :ANGLE;"DEGREES IS";RADS;"RADIANS";
200  PRINT "AND ITS COSINE IS";COS(RADS)
210  PRINT
220  GOTO 140
230  END
```

Listing 4-20. COS Example

Description:

A DATA statement stores numeric data or string data or both in a program. You use a READ statement to put the values in the DATA statement *data-list* into variables in your program.

TI BASIC does not execute DATA statements; the DATA statements are simply ways to store information in your program. If you have to initialize a large array, it usually takes less space to store the values in DATA statements and use READ statements for the initialization than it takes to use assignment statements for each array value.

You can have as many DATA statements as you need in your program. And you can put them anywhere in the program. BASIC uses the DATA statements in line number order. All the values in the first DATA state-

DATA	Store program data for READ.
Type:	Statement
Format:	*line#* DATA *data-list*
Purpose:	DATA stores information in your program that you access through READ statements. You can select which DATA statement to READ with a RESTORE statement.
Operands:	*line#* is a BASIC statement line number that you need when you include DATA in a program. *line#* can be any number between 1 and 32767. *data-list* is a list of one or more data items (numbers or strings), separated by commas if there is more than one item in the list. String data items which contain commas, leading/trailing blanks, or double quotes must be enclosed in double quotes (").
Defaults:	None.

ment's *data-list* are used before any are used from the next DATA statement.

You can select a specific DATA statement that you want to use with a RESTORE statement when you select the DATA statement by line number. Once BASIC READs information from a DATA statement, you cannot reuse that DATA statement unless you RESTORE it.

READ statements take the information from a DATA statement's *data-list* and puts it into variables in your program. Numeric data must be READ into numeric variables; string data into string variables (string variable names end in $).

You will get an error if you try to read string data into numeric variables. However, numbers are valid string data items. If you miscount and READ numbers into string variables, BASIC won't care. The string will be the string representation of whatever number you have. You will probably find out what happened when you try to read the number and either have no data left or are misaligned and reading string data.

You enter numeric data items as values separated by commas. Most string data gets entered the same way. You don't need to put double quotes (") around string data in a *data-list* unless:

- The string data contains a comma (“PHILA, PA”)
- The string data contains trailing blanks (“New value is   ”)
- The string data contains leading blanks (“   title one”)
- The string data contains quotes—you use two adjacent double quotes for each quote contained in the string (“this string contains one double quote "" in it”)

If you want to put a null string (an empty string which contains no characters and has a length of zero) in a *data-list*, enter two adjacent commas (,,) for the string, like this:

```
DATA HI THERE,,12.345
```

When the above DATA statement is READ, there are two strings (HI THERE and a null string) and one numeric value (12.345).

Common Errors:

CAN'T DO THAT

You tried to use DATA as a command. DATA can be used only as a statement in your program.

DATA ERROR

You forgot one or more commas in your DATA statement. You must put commas between the values in the *data-list*. If your values are strings, remember to put the string data in quotes if the string data itself contains commas.

Example 1:

The program in Listing 4-21 uses DATA statements to store string data (month names). The READ statements put this information into the MONTH$ array.

The program asks you for a number representing a month and converts the month to its string format. Try changing the program to print full names for the months instead of abbreviations.

```
100  CALL CLEAR
110  DATA JAN,FEB,MAR,APR,MAY,JUN
120  DATA JUL,AUG,SEP,OCT,NOV,DEC
130  DIM MONTH$(12)
140  FOR I=1 TO 12
150  READ MONTH$(I)
160  NEXT I
170  PRINT : :"ENTER A NUMBER BETWEEN":
     "  1 AND 12.  I'LL TELL"
180  PRINT "  WHAT MONTH IT IS.": : :
190  INPUT "WHAT NUMBER (0 TO STOP)->":MTH
200  IF MTH<>0 THEN 200
210  PRINT : :"GOODBYE."
220  STOP
230  IF (MTH<1)+(MTH>12) THEN 140
240  PRINT "MONTH ";MTH;"IS ";MONTH$(MTH)
250  GOTO 160
260  END
```

Listing 4-21. DATA Example 1

Example 2:

The program in Listing 4-22 uses DATA statements to store string data (month names) mixed with numeric data (number of days each month). Each READ statement reads string data (a month name) and numeric data (how many days there are in the month).

Try changing the DATA statements so that all the months are read first and the days second. (Remember that you'll need different READ statements if you do this.)

```
100  CALL CLEAR
110  DATA JAN,31,FEB,28,MAR,31,APR,30,MAY,31,JUN,30
120  DATA JUL,31,AUG,31,SEP,30,OCT,31,NOV,30,DEC,31
130  DIM MONTH$(12),DAYS(12)
140  FOR I=1 TO 12
150  READ MONTH$(I),DAYS(I)
160  PRINT MONTH$(I);" HAS";DAYS(I);"DAYS."
170  NEXT I
180  PRINT : : :
190  INPUT "PRESS ENTER TO STOP.":X$
200  END
```

Listing 4-22. DATA Example 2

DEF	Define a user function.
Type:	Statement
Format:	*line#* DEF *fctn-name* = *expression* or *line#* DEF *fctn-name*[(*parameter*)] = *expression*
Purpose:	DEF lets you define your own numeric or string functions.
Operands:	*line#* is a BASIC statement line number that you need when you include DEF in a program. *line#* can be any number between 1 and 32767. *fctn-name* is a valid variable name that you use when you want to use your function in your program. You must use a string variable name (ending with a dollar sign, $) when your function returns a string value and a numeric variable name when your function returns a numeric value. *parameter* is a valid variable name that represents a variable used in the function's *expression*. The *parameter* name must be a numeric variable name when it represents a numeric value and a string variable name (ending with a dollar sign, $) when it represents a string value. The *parameter* name can be considered a place holder and does not affect any variable in your program which has the same name. You don't always need a *parameter*, depending on your function. *expression* is any valid string or numeric expression. This is evaluated and its value returned when you use the function.
Defaults:	None.

Description:

DEF defines a numeric or string function with the name *fctn-name*. You use the function just as you would any of BASIC's own functions (like ABS or VAL or STR$).

When you include the function's name, *fctn-name*, in an expression, assignment statement, or PRINT or DISPLAY statement, TI BASIC evaluates the function's *expression* and uses the result where you used *fctn-name*.

DEF statements can appear anywhere in your program. But, the DEF statement defining a function, *fctn-name*, must have a lower line number than any statement that uses *fctn-name*.

You often use a function when you are using an expression several times in a program, especially when the expression is long or complicated. Then, instead of re-entering the long expression every place you need it, you can simply use the *fctn-name*. This saves space in your program and makes it easy to change *expression*.

It's even possible to have functions reference other functions. Just as long as you don't, directly or indirectly, have a function reference itself. For example, suppose you use a DEF statement to define a function, ROUND(X), which you define as the value of X rounded to two decimal places. This statement is a useful function for printing numbers. You can use ROUND(X) in another DEF statement that makes the value you rounded into a string. Like this:

```
250  DEF ROUND(X)=INT(X*100)/100
260  DEF PRT$(Z)="$"&STR$(ROUND(Z))
```

The ROUND function in the above example takes a number, multiplies it by 100, makes it into an integer (strips off any decimal places), then divides by 100 to get two decimal places in the result. Then, the PRT$ function puts a dollar sign ($) before the string representation of the rounded number. This would be very useful when you're printing dollar values and you want to format the output. Suppose you are printing these values:

```
A=567.89235
B=987654.32132654
C=1.232323
```

You can use the two functions, ROUND and PRT$, to print these values, like this:

```
PRINT PRT$(A);TAB(10);PRT$(B);TAB(20);PRT$(C)
```

You would see this line (with the dollar signs automatically included):

```
$567.89   $987654.32   $1.23
```

You will notice there's an optional *parameter* shown in the DEF statement format. When you omit the *parameter,* BASIC uses whatever values are in the variables you used in *expression*. When you use *parameter*, BASIC uses the value of the variable that you substitute for *parameter* when you use *fctn-name*. BASIC uses the current values of any other variables (other than *parameter*) appearing in *expression*.

Functions can have parameters and arguments. The parameter, called *parameter* in the function's definition, is a valid variable name. This tells the function that you want it to use whatever variable you put in *parameter*'s place when you use the function.

The actual variable or value that you tell *fctn-name* to use is called an argument. This is sometimes confusing. Remember that you pass an argument which the function uses as a parameter. The parameter is only a place holder.

Parameter can be any valid variable name. However, if you want to use a number, you must use a numeric variable name. If you want to use a string, you must use a string variable name (ending in a dollar sign, $).

NOTE

> When you use a *parameter*, you are telling TI BASIC that you want to *substitute* a different variable for *parameter*'s place in *expression* when you use *fctn-name*. The actual variable name that you use for *parameter* is only used for the DEF statement. You can use the same variable in your program. TI BASIC can tell the difference.

When you use a parameter in your DEF statement, you must enclose the *parameter* in parentheses *(parameter)*, and, when you use the function itself, you must again use the parentheses surrounding the variable you want to use for the function argument. (Remember, you pass an argument to the function's parameter.)

If you don't use *parameter* when you define *fctn-name*, you CANNOT use an argument with your function. You either have a *parameter* or you don't. And the decision is made when you write your DEF statement. You get an error when you use an argument for a function without a parameter or when you don't supply an argument for a function that expects a parameter.

Common Errors:

CAN'T DO THAT

You tried to use DEF as a command. DEF can be used only as a statement in your program.

INCORRECT STATEMENT

You forgot to put a closing parenthesis ")" after the *parameter* in your DEF statement.

Or, you used a *parameter* that's not a valid variable name.

Or, you don't have an equals sign (=) before the *expression* in your DEF statement.

MEMORY FULL

Your program is too large and there is not enough memory left to allocate one or more functions defined in a DEF statement.

Or, the function that you defined in your DEF statement references itself (has its own name as part of its definition).

Or, your program is large and there are too many DEF statements that use functions defined in other DEF statements in your program.

Example 1:

The program in Listing 4-23 uses DEF to define the function AVERAGE which is (X + Y)/2. Since there are no parameters, the values of the variables X and Y are used when the program references the function AVERAGE.

You enter two numbers (X and Y) and the program prints the average. This is a very simple example of a function. Change the program to define another function called X (like DEF X = A + B), enter three numbers (A, B, and Y), and see what happens.

```
100  CALL CLEAR
110  DEF AVERAGE=(X+Y)/2
120  PRINT "ENTER TWO NUMBERS AND":
     " I'LL PRINT THEIR AVERAGE."
130  INPUT "YOUR NUMBERS (0,0 TO STOP) ->":X,Y
140  IF X<>0 THEN 180
150  IF Y<>0 THEN 180
160  PRINT : :"GOODBYE."
170  STOP
180  PRINT :"THE AVERAGE OF";X;"AND";Y;"IS";AVERAGE : :
190  GOTO 130
200  END
```

Listing 4-23. DEF Example 1

Example 2:

The program in Listing 4-24 uses two DEF statements. The first DEF defines a string function, FIVE$, with a string argument; it returns the first 5 characters of whatever string you pass it.

The second DEF defines a numeric function, RADS, also with an argument. The argument is an angle in degrees. The function converts the angle to radians. This function is used with the BASIC trigonometric functions (SIN, COS, ATN, and TAN) which require angles in radians, not degrees.

```
100  CALL CLEAR
110  DEF FIVE$(A$)=SEG$(A$,1,5)
120  DEF RADS(X)=X*(4*ATN(1))/180
130  PRINT "ENTER A STRING AND I'LL":
     "  TELL YOU ITS FIRST"
140  PRINT "  FIVE CHARACTERS.": :
     "ENTER AN ANGLE AND I'LL"
150  PRINT "  TELL YOU ITS COSINE":"  AND SINE.": :
160  INPUT "YOUR STRING -> ":IN$
170  PRINT :"THE FIRST 5 LETTERS ARE: ";FIVE$(IN$): :
180  INPUT "YOUR ANGLE (0-360) ->":ANGLE
190  IF (ANGLE<0)+(ANGLE>360) THEN 180
200  PRINT "THE COSINE OF";ANGLE;"IS";COS(RADS(ANGLE))
210  PRINT "THE SINE OF";ANGLE;"IS";SIN(RADS(ANGLE))
220  PRINT : : :"GOODBYE."
230  END
```

Listing 4-24. DEF Example 2

DELETE	Delete a file.
Type:	Command
Format:	[*line#*] DELETE "*device-filename*" or [*line#*] DELETE *str-exp*
Purpose:	DELETE removes ("deletes") a file called "filename" from the device "device."
Operands:	*line#* is a BASIC statement line number that you need when you include DELETE in a program. You don't need *line#* when you use DELETE as a command. *line#* can be any number between 1 and 32767. *device-filename* is a string enclosed in double quotes (") that represents a valid device attached to your computer and the name of a file (*filename*) stored on that device. *str-exp* is a string, string variable, or string expression that specifies the device (same as *device*) and the name of the file (same as *filename*) of the file to be deleted from the device.
Defaults:	None.

Description:

DELETE deletes file named *filename* from the physical device *device*. If the device is a disk (DKS1, DSK2, or DSK3), the file called *filename* is removed from the diskette currently in the disk drive. The file is gone when the DELETE command is executed.

CAUTION

A DELETED FILE IS GONE FOREVER. ONCE YOU DELETE IT, YOU CAN'T READ IT AGAIN.

DELETE lets you recover space from files on diskettes or Wafertapes. Every file, on any storage media, will remain intact until you DELETE it (for diskettes and Wafertapes) or write over it (for cassette tapes).

You can easily accumulate a lot of files in a short time. Once you are done with a file, you should either make a copy of it and store it in a safe place, or get rid of it. Otherwise, you will have the task of finding out what is in these files that are taking up all the storage space.

NOTE

If you attempt to delete a file from disk that does not exist, no error occurs and execution continues normally.

Other devices, such as the Hexbus Wafertape, also have limited capacity for storing information. When you know that you no longer need a file, you should delete it.

You must use double quotes (") around the *device-filename*. You don't need the double quotes when you use *str-exp*.

Using DELETE with a cassette file is meaningless. You cannot remove a file from a cassette. You don't have to. When you want to reuse the space taken by a cassette file you just position the tape and write over the old data. However, this is a valid command

```
DELETE "CS1"
```

which does nothing to any file on the cassette but does print this message:

```
PRESS CASSETTE STOP CS1
THEN PRESS ENTER
```

Remember, no action is taken when you DELETE from a cassette. When you DELETE from other devices, such as a disk or a Hexbus Wafertape, the file you DELETEd is gone forever.

Common Errors:

```
INCORRECT STATEMENT
```

Device-filename is not a valid string, or *str-exp* is not a valid string, string variable, or string expression.

```
I/O ERROR 70
```

You used a *device* that isn't a valid name for a device attached to your computer. Check the spelling for the device.

```
I/O ERROR 71
```

You tried to delete a file that's protected. You cannot delete a protected file.

```
I/O ERROR 76
```

There is a device error. The device is disconnected or is not working properly. This error can occur when you disconnect a device after you have started your program.

Example 1:

The example in Listing 4-25 uses DELETE as a command to remove two files (PROGRAM1 and NEWDATA) from the diskette on disk drive one, and one file (OLDDATA) from the diskette on disk drive 2.

```
DELETE "DSK1.PROGRAM1"
DELETE "DSK1.NEWDATA"
DELETE "DSK2.OLDDATA"
```

Listing 4-25. DELETE Example 1

Example 2:

The program in Listing 4-26 reads an input file and writes a copy of it to an output file. The program uses DELETE in a program to optionally delete the input file when the program is finished.

```
100  CALL CLEAR
110  INPUT "WHAT'S YOUR INPUT FILE ->":FILEIN$
120  INPUT "YOUR OUTPUT FILE ->":FILEOUT$
130  OPEN #6: FILEIN$,INPUT
140  OPEN #20: FILEOUT$,OUTPUT
150  IF EOF(6) THEN 200
160  INPUT #6: LINEIN$
170  PRINT #20: LINEIN$
180  LINES=LINES+1
190  GOTO 150
200  CLOSE #6
210  CLOSE #20
220  PRINT LINES;" RECORDS READ/WRITTEN."
230  INPUT "DELETE FILE "&FILEIN$&" (Y/N) ":Y$
240  IF (SEG$(Y$,1,1)="N")+(SEG$(Y$,1,1)="n") THEN 260
250  DELETE FILEIN$
260  PRINT "GOODBYE."
270  END
```

Listing 4-26. DELETE Example 2

DIM	Allocate an array.
Type:	Statement
Format:	[*line#*] DIM *array-name(dim1*[,*dim2*[,*dim3*]]) [, . . .]
Purpose:	DIM allocates memory for groups of data called arrays. DIM sets up the boundaries (dimensions) for the arrays at the *dim1* through *dim3* values.
Operands:	*line#* is a BASIC statement line number that you need when you include DIM in a program. You don't need *line#* when you use DIM as a command. *line#* can be any number between 1 and 32767. *array-name* is any valid string or numeric variable name. If you use a string *array-name*, the array must contain only string data. If you use a numeric *array-name*, the array must contain only numeric data. *dim1* is a positive number (NOT a variable or expression) that specifies the upper limit on the number of elements stored in the array's first dimension. *dim1* may be any value between 0 (or 1, if you use an OPTION BASE 1 statement) and 32767, remembering, of course, that you have a maximum of 16K of memory on your computer. *dim2* is similar to *dim1* but it specifies the maximum limit on the second dimension (ARRAYNAME(2,3), where 3 is the maximum second dimension, *dim2*). *dim3* is similar to *dim1* but it specifies the maximum limit on the third dimension (STRING$(5,25,19) where 5 is the maximum first dimension, 25 is the maximum second dimension, and 19 is the maximum third dimension, *dim3*).
Defaults:	If you don't use a DIM statement to dimension an array, BASIC uses 10 for *dim1*, *dim2*, and *dim3*. The upper bounds are set to 10 and the lower bound is set to 0 (if you don't use an OPTION BASE 1) or 1 (if you use OPTION BASE 1).

Description:

DIM allocates (dimensions) space for one or more arrays. Each array (*array-name*) gets elements allocated depending on the values you use for *dim1*, etc. You can have up to 3 dimensions for an array in TI BASIC. This description gives you some details on arrays, but not all. For more information concerning arrays, look in Chapter 2, DATA IN BASIC.

Arrays are simply collections of data with a single name. You can have string arrays (*array-name* ends with a dollar sign, $) or numeric arrays. You can have one-, two-, or three-dimension arrays, depending on how many *dim* operands you use.

An array has elements which are identified by a subscript, or number for each element. The subscript maximum values are given in the *dim1*, *dim2*, and *dim3* operands, depending on how many dimensions you choose. You can use any number, numeric variable, or numeric expression for the subscript as long as it is within the limits you set for the array.

Subscripts usually begin at zero. If it's inconvenient to begin the array at subscript 0, use an OPTION BASE 1 statement to make 1 the first subscript value.

Remember, each element of an array takes space in your program. String arrays can use a lot of memory if you fill the array with long strings. Unused elements take space. Use the OPTION BASE 1 statement when you don't intend to use the element zero in your arrays.

Listed below are a few rules you must follow when you use arrays.

1. You must DIMension your array in a statement whose *line#* is lower than any statements which reference the array.
2. You can DIMension an array only once in a program and you cannot re-execute the DIM statement. If you want to go back to the beginning of your program, you must GOTO a *line#* after the DIM *line#*.
3. Once you DIMension an array, you cannot use the *array-name* for a function (DEF) or a simple variable (variable without subscripts).
4. The number of subscripts that you use when you reference an array must agree with the number that you used in your DIM statement and cannot exceed the values you specified. If you allocate a one-dimension array, you use one subscript—VALUE(4); a two-dimension array gets two subscripts—NAME$(2,6); a three-dimension array gets three subscripts—SCORE(4,2,1).

Arrays make it easy for you to group similar data in your program. You can use the same variable name for all the data. You just have to change the subscript value you use to get the element you need. Look at the description of the FOR statement for more examples of array use.

You can think of a one-dimension array as a list with *dim1* entries. You don't have to fill the entire array. You fill just as many elements as you need, never, of course, going past the maximum number you chose as *dim1*.

Suppose you want a list of 20 names. You can dimension a string array of 20 names like this:

```
120   DIM NAME$(20)
```

Then, to PRINT the fifth name, you would:

```
200   PRINT NAME$(5)
```

If you don't use an array and you want to store 20 names in your program, you need 20 different variables. See how easy arrays are.

Now, for a two-dimension array, you have a table with *dim1* times *dim2* entries. Take our 20 names again. Now, you want to store first and last names. You can use two one-dimension arrays, FIRST$(20) and LAST$(20), or you can use a single two-dimension array, NAME$(20,2).

The NAME$(20,2) array has 40 elements (20 × 2). We'll use the elements NAME$(n,1) for the 20 first names and the remainder, NAME$(n,2), for the last names. (n goes from 1 to 20.) To allocate the array, you use:

```
150   DIM NAME$(20,2)
```

To print the first and last names for the twelfth entry, you use:

```
300   PRINT NAME$(12,1);NAME$(12,2)
```

Now, for three-dimension arrays. Think of three-dimension arrays as a group of tables where the first two dimensions (*dim1* and *dim2*) make up a table and the third (*dim3*) tells you how many tables you have.

Suppose you have a class of 15 students. Each student takes 5 tests in each of 3 subjects. A perfect example of a numeric array. In this DIM statement, *dim1* (15) tells how many students, *dim2* (5) tells which test, and *dim3* (3) tells which subject (the array contains 225 elements—15 × 5 × 3):

```
120   DIM SCORES(15,5,3)
```

After you fill the array with data, you can find out what student 7 scored in the third test for subject 2 by:

```
400  STUDENT=7
410  TEST=3
420  SUBJ=2
430  PRINT "STUDENT";STUDENT;"SCORED";
     SCORES(STUDENT,TEST,SUBJ);"IN TEST ";
     TEST;"AND SUBJECT";SUBJ
```

(You must enter the PRINT statement, 430, as a single statement without extra ENTERs in it. It looks like it does because it won't fit on a single line when printed here.)

You could also print the above information like this:

```
430   PRINT SCORES(7,3,2)
```

Once you get familiar with arrays, you will find a lot of uses for them. Use arrays to keep any similar data together, such as objects in an adventure game, positions on a board game like Tic-Tac-Toe, names and addresses, any data that you usually think of as a list or table.

Common Errors:

BAD SUBSCRIPT

This error occurs in references to array elements. It indicates an incorrect value assigned an array subscript.

You used an OPTION BASE 1 statement and now you used a subscript of zero.

Or, one or more of your subscripts is less than zero or greater than the maximum dimension value. For example, if you dimension an array, A(10,5), your subscripts cannot exceed 10 for the first dimension or 5 for the second dimension. A reference, as in A(12,3), will give you the BAD SUBSCRIPT error.

BAD VALUE

These errors are caused by bad dimension (*dim1*) values. An array dimension (*dim1*) is less than zero or greater than 32767.

Or, an array dimension is zero and you used an OPTION BASE 1 statement.

INCORRECT STATEMENT

These errors are caused by invalid *array-names* or incorrect formats. You used an *array-name* that has no dimension specified or has more than three dimensions specified.

Or, the value that you used for a dimension (*dim1*) is not a number.

Or, the format of a subscript is incorrect with the closing ")" missing.

Or, the *array-name* is not a valid string or numeric variable name.

Or, you specified several arrays in one DIM statement (more than one *array-name*) and you forgot either a comma between the array definitions or a closing ")".

MEMORY FULL

Your program is too large and you asked for an array to be dimensioned that cannot fit into the available memory.

NAME CONFLICT

These errors are caused when you use an *array-name* incorrectly such as using the same name (*array-name*) for both an array and a simple (nonarray) variable or for two arrays.

Or, you used the same name (*array-name*) for both an array and a function.

Or, you referenced an array incorrectly. Your array does not have the same number of dimensions in the reference and in the DIM statement for the array. If you dimension an array with two dimensions, as in AR$(5,2), you must use two dimensions in every reference—AR$(1,1) or AR$(4,2), etc.

Example 1:

The program in Listing 4-27 uses DIM to allocate a one-dimension numeric array (DAYS) with 12 elements and a two-dimension string array (MONTH$) with 24 elements (12 × 2). The OPTION BASE 1 statement makes all arrays start with element 1 instead of zero.

```
100  CALL CLEAR
110  DIM MONTH$(12,2),DAYS(12)
120  DATA JAN,FEB,MAR,APR,MAY,JUN
130  DATA JUL,AUG,SEP,OCT,NOV,DEC
140  DATA 31,28,31,30,31,30,31,31,30,31,30,31
150  DATA JANUARY,FEBRUARY,MARCH,APRIL,MAY,JUNE
160  DATA JULY,AUGUST,SEPTEMBER,OCTOBER
170  DATA NOVEMBER,DECEMBER
180  FOR I=1 TO 12
190  READ MONTH$(I,1)
200  NEXT I
210  FOR I=1 TO 12
220  READ DAYS(I)
230  NEXT I
240  FOR I=1 TO 12
250  READ MONTH$(I,2)
260  NEXT I
270  PRINT :"I'LL TELL YOU HOW DAYS THERE":
     "  ARE IN EACH MONTH."
280  PRINT : :
290  INPUT "DO YOU WANT FULL NAMES (Y/N)->":Y$
300  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 330
310  NAMES=1
320  GOTO 340
330  NAMES=2
340  CALL CLEAR
350  FOR I=1 TO 12
360  PRINT TAB(5);MONTH$(I,NAMES);" HAS";
     DAYS(I);"DAYS."
370  NEXT I
380  PRINT : :
390  INPUT "TRY AGAIN? (Y/N) -> ":Y$
400  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 290
410  PRINT : :"GOODBYE."
420  END
```

Listing 4-27. DIM Example 1

The program uses DATA statements to hold the information that gets READ into the arrays.

Example 2:

The program in Listing 4-28 uses a DIM statement to create a 20 element one-dimension numeric array, RANDOMS(20). RND fills the array with random numbers scaled between 1 and 100.

You try to guess one of the numbers in the array. Change the program to use numbers between − 100 and 100, or between 1 and 10.

```
100  CALL CLEAR
110  DIM RANDOMS(20)
120  RANDOMIZE
130  FOR I=1 TO 20
140  RANDOMS(I)=INT(RND*100)
150  NEXT I
160  PRINT :"I KNOW 20 SECRET NUMBERS."
170  PRINT "WHICH ONE DO YOU WANT TO"
180  INPUT "GUESS (1-20) ":CHOOSE
190  IF (CHOOSE<1)+(CHOOSE>20) THEN 170
200  PRINT : : "OK.  LET'S START.": :
210  TRIES=0
220  INPUT "YOUR GUESS (0-100) -> ":GUESS
230  TRIES=TRIES+1
240  IF GUESS<>RANDOMS(CHOOSE) THEN 300
250  PRINT : : "YOU WIN IN";TRIES;" GUESSES."
260  INPUT "TRY ANOTHER? (Y/N) -> ":Y4
270  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 170
280  PRINT : :"GOODBYE."
290  STOP
300  IF GUESS<RANDOMS(CHOOSE) THEN 330
310  PRINT "TOO HIGH."
320  GOTO 220
330  PRINT "TOO LOW."
340  GOTO 220
350  END
```

Listing 4-28. DIM Example 2

DISPLAY	Write to the screen.
Type:	Statement
Format:	[*line#*] DISPLAY or [*line#*] DISPLAY *list*
Purpose:	DISPLAY writes information to your television screen. It works the same as PRINT except that you can write only to the screen.
Operands:	*line#* is a BASIC statement line number that you need when you include DISPLAY in a program. You don't need *line#* when you use DISPLAY as a command. *line#* can be any number between 1 and 32767. *list* is a list of variable names, expressions, numbers, strings, and functions that you want to print on your screen.
Defaults:	If you don't use a *list*, BASIC displays (prints) a single blank line.

Description:

DISPLAY writes the data in *list* to the screen the same as a PRINT statement. You can write 24 rows, with 28 characters per row, on your screen with DISPLAY. The leftmost column is column 1. The top row is row 1.

list can be any one or more valid BASIC data items: number, string, expression, array, function. If you don't use a *list*, you get a blank line written on your screen:

```
150 DISPLAY
```

If you have two variables (A = 15 and B = 99), you can write them to your screen by using:

```
200 DISPLAY A,B
```

Lines on your screen are divided into two *zones* for printing. Zone 1 begins at column 1. Zone 2 begins at column 15. In the above DISPLAY statement, the value of A would begin at column 1; the value of B, in column 15. If you write more than two values, the third value would begin in column 1 of the next line.

TI BASIC will not split a data item over two lines (unless it is a string longer than 28 characters). If the data item can't fit into Zone 2, it gets put into Zone 1 on the next line. This usually happens when you are writing string data to the screen.

You can make TI BASIC write in formatted positions on your screen by using *print separators*. Table 4-8 shows you what the print separators do. The positions of the data written to your screen depend on the *print separators* you use between the data items.

Table 4-8. DISPLAY *print-separators*

Print-separator	Meaning
semicolon (;)	Write the next data item right next to the current data item. Do not leave any extra spaces (except for the leading and trailing spaces around numeric data items).
colon (:)	Skip to the next line.
comma (,)	Write the next data item at the next available zone. Zone 1 starts in column 1. Zone 2 starts in column 15.

You can also use the TAB function to position data on your screen. Remember that you can write in only 28 columns on your screen. If you use a TAB position past 28, TI BASIC continually subtracts 28 from your TAB position until it gets a value between 1 and 28.

TI BASIC defines file number 0 as the screen for output and the keyboard for input. You can replace any DISPLAY statement with a PRINT # 0 statement (providing you OPEN # 0 first) and get the same results.

Common Errors:

None.

Example 1:

The program in Listing 4-29 uses DISPLAY to write a message to your screen.

Try changing the DISPLAY statement to PRINT and see what happens.

```
100  CALL CLEAR
110  DISPLAY "HI THERE.": :"I'M A DISPLAY STATEMENT."
120  DISPLAY
130  DISPLAY TAB(5);"GOODBYE NOW."
140  DISPLAY : : :
150  INPUT "PRESS ENTER TO STOP.":X$
160  END
```

Listing 4-29. DISPLAY Example 1

Example 2:

The program in Listing 4-30 uses DISPLAY to write several different variables to your screen. PRINT works the same way. If you want to use OPEN # and PRINT #, PRINT # 0 writes to the screen.

```
100  CALL CLEAR
110  INPUT "WHAT'S YOUR NAME? -> ":NAME$
120  DISPLAY : : "HI "&NAME$: :
     "I'M USING DISPLAY STATEMENTS"
130  DISPLAY : :
140  NUMVAL=44
150  DISPLAY : :"HERE'S A NUMBER";NUMVAL
160  NUMVAL=9.876E45
170  DISPLAY : :"AND HERE'S A BIG NUMBER";NUMVAL
180  DISPLAY : : :"BYE NOW, ";NAME$
190  END
```

Listing 4-30. DISPLAY Example 2

EDIT	Edit a line in a program.
Type:	Command
Format:	EDIT *line-num* or *line-num* FCTN E (Up-arrow) or *line-num* FCTN X (Down-arrow)
Purpose:	EDIT lets you change existing lines in the BASIC program currently in your computer's memory.
Operands:	*line-num* is the line number of a statement in your BASIC program that you want to change. *line-num* can be any number between 1 and 32767.
Defaults:	None.

Description:

You use EDIT when you want to change a line number (*line-num*) in the BASIC program currently in your computer's memory. You must have a program in memory for BASIC to edit. You get the program into your computer's memory either by entering it (maybe with the help of a NUM command) or by reading it in (through an OLD command).

There are three forms of EDIT, which work in exactly the same way. To change line 290 in the program you have in memory, you can enter any of these:

EDIT 200
or
200 FCTN E (up-arrow)
or
200 FCTN X (down-arrow)

TI BASIC writes line 200 to the screen and positions the cursor at the first character of the statement, just past the line number. You then use the keys shown in Table 4-9 to change the line. If you just press ENTER when the line appears, no changes are made.

Table 4-9. TI BASIC Editing Function Keys

Key	Function
ENTER	Enter the program line. The line you are editing (line number and statement) is entered into the program currently in your computer's memory.
FCTN D (right-arrow)	Forwardspace one character. Move the cursor one character position to the right. No changes are made to any characters the cursor moves past. You use the FCTN D key to position your cursor when you want to add or delete characters on the line you're currently editing.
FCTN E (up-arrow)	Enter the program line. The line you are editing (line number and statement) is entered into the program currently in your computer's memory. The statement with the next lower line number is then presented for editing.
FCTN S (left-arrow)	Backspace one character. Move the cursor one character position to the left. No changes are made to any characters the cursor moves past. You use the FCTN S key to position your cursor when you want to add or delete characters on the line you're currently editing.
FCTN X (down-arrow)	Enter the program line. The line you are editing (line number and statement) is entered into the program currently in your computer's memory. The statement with the next higher line number is then presented for editing.
FCTN 1 (DEL)	Delete one character. Delete the character under the cursor. You usually use the FCTN S or FCTN D key to position the cursor to the character you want to delete.

Table 4-9. *(continued)*

Key	Function
FCTN 2 (INS)	Insert characters. Insert characters at the cursor position. You can use the FCTN S or FCTN D key to position the cursor to where you want to insert the characters. Unlike the other FCTN keys, INS puts you into *Insert Mode*, allowing you to insert as many characters as you need.
FCTN 3 (ERASE)	Erase the entire line. Does not erase the line number.
FCTN 4 (CLEAR)	Clear the current line. Erases the current line and stops the editing process.
FCTN = (QUIT)	Quit. Leave BASIC and return to the main title screen. Memory is erased. If you have files opened, they are not closed. Use a BYE command if you want your files closed. Remember, you lose the program in memory if you haven't saved it.

You use EDIT when you are entering programs and you make a typing mistake, or a logic mistake. EDIT makes it easy to correct these problems. Just remember to SAVE your program after you EDIT it.

Common Errors:

BAD LINE NUMBER

You used a value for *line-num* that is not the line number of a statement in your BASIC program.

CAN'T DO THAT

You tried to use EDIT as a statement in a program. You use EDIT only as a command.

Example 1:

The example in Listing 4-31 uses EDIT to change only one line in a program. A mistake is made in the 100 CALL CLEAR statement (CLEAR

```
NUM <ENTER>
100  CALL CLARE
110  PRINT "HI"
120  END
130  <ENTER>
EDIT 100 <ENTER>
100  CALL CLARE
     Use FCTN D to space over to the A in CLARE.
     Make the line look like this.
100  CALL CLEAR <ENTER>
RUN <ENTER>
     The program clears the screen and prints HI.
```

Listing 4-31. EDIT Example 1

is spelled CLARE). Use EDIT 100 to change CLARE to CLEAR. Then RUN the program.

Example 2:

The example in Listing 4-32 the *line-num* FCTN E form of EDIT to change two lines. Once again, CLEAR is spelled incorrectly. Statement 110 should really be a DISPLAY statement.

```
NUM <ENTER>
100  CALL CLARE
110  PRINT "HI"
120  END
130  <ENTER>
100  <FCTN E>
100  CALL CLARE
     Use FCTN D to space over to the A in CLARE.
     Make the line look like this.
100  CALL CLEAR <ENTER>
110  <FCTN E>
110  PRINT "HI"
     Use FCTN 1 (DELETE) to delete the 5 characters
     PRINT.
110  "HI"
     Now, use FCTN 2 (INSERT) to tell BASIC to begin
     inserting characters at the cursor position.
     Insert the word DISPLAY and press <ENTER>.
110  DISPLAY "HI"
     Finally, RUN the program.
RUN <ENTER>
     The program clears the screen and prints HI.
```

Listing 4-32. EDIT Example 2

END	End (stop) the program.
Type:	Statement
Format:	*line#* END
Purpose:	Your BASIC program stops executing when it reaches an END statement. END functions the same as STOP in TI BASIC.
Operands:	*line#* is a BASIC statement line number that you need when you include END in a program. *line#* can be any number between 1 and 32767.
Defaults:	None.

Description:

END ends your program and stops its execution. You usually use one END statement per program, as the last line in your program. You do not need to use an END statement in TI BASIC. TI BASIC will stop executing the program when it runs out of statements.

But it is a good practice to use END statements. By ending all your programs with an END statement, you can always be sure that you have read an entire program into memory. The END statement is your end-of-program marker.

END functions much like STOP. When BASIC reaches an END statement, it stops executing the program. If you want to stop your program in several places, use STOP statements within the program and an END statement at the end of the program.

Common Errors:

None.

Example:

The program in Listing 4-33 uses an END statement to mark the end of the program and a STOP statement at a logical termination point in the middle of the program.

```
100  CALL CLEAR
110  PRINT "I'LL PRINT NUMBERS":"  UNTIL YOU TELL ME"
120  PRINT "  TO STOP OR UNTIL":"  I REACH 100.": :
130  X=1
140  IF X<100 THEN 160
150  STOP
160  PRINT X
170  X=X+1
180  INPUT "STOP YET? (Y/N) -> ":Y$
190  IF (SEG$(Y$,1,1)="N")+(SEG$(Y$,1,1)="n") THEN 160
200  END
```

Listing 4-33. END Example

EOF	Check for end of file.
Type:	Function
Format:	EOF (*file-num*)
Purpose:	EOF is the end of file function that tells you whether you are at the logical end of the file OPENed as *file-num* or whether you are at the physical end of space available for the file on the device.
Operands:	*file-num* is a number, numeric variable, or numeric expression for the file OPENed with *file-num*. *file-num* may be any value between 1 and 255.
Defaults:	None.

Description:

The EOF function tells you if you are at a physical or a logical end of the file *file-num*. *file-num* corresponds to the *file-num* that you use when you OPENed the file.

When you read a file (INPUT #) and run out of records, you are at the *logical* end of file (logically there's no more file). When you write a file

(PRINT #) and run out of room on the device, you are at the *physical* end of file (there's no more physical room to put the file).

Table 4-10 shows you what values EOF returns, depending on where you are in a file.

When you are reading a file, EOF is zero until you reach the logical end of file. When you are writing a file, EOF is one until you reach a physical end of file.

Table 4-10. EOF Values

Value	Meaning
−1	You are writing a file and you are at a physical end of file for file *file-num*. This happens when you're writing to a file and there's no more room left on the device to write any more information for the file.
0	You are reading a file and you are not at the end of file *file-num*. There are still records left to read from the file.
1	You are reading a file and you are at a logical end of file for file *file-num*. This happens when you're reading a file and try to read past the last record in the file.

You *CANNOT* use EOF with a cassette file. You can make your own end of file marker for cassette files by writing a special record to mark the end of the file. You might write a final record with all the numeric variables set to 999 and all the string variables set to "ZZZZ". Then, when you read the file, check the values of one or more variables to see if they contain these special "end of file" values.

NOTE

EOF is a function. It returns a value when it gets executed. Make sure that you have the EOF where it gets executed before you read or write the file you want to check.

You should use EOF whenever you are reading a file (other than from a cassette) or writing a file (other than to the screen or printer). EOF will prevent your program from running out of records in the file when it is almost done processing. If it runs out of records it will stop with an error. Or, when it runs out of room on your disk it will also stop with an error.

Always CLOSE # the file in your EOF processing. This ensures that you will be able to read the file later. If you are writing a file, you will save what has been written.

Common Errors:

BAD VALUE

The value of *file-num* is less than zero or greater than 255.

STRING-NUMBER MISMATCH

You used a string instead of a number for *file-num*.

Example 1:

The program in Listing 4-34 uses EOF to see if you are at the end of an input file (have read all the records in the file). When the entire file is read, the program prints the number of records that it read and stops. Try changing the program to write the record that you read.

```
100  CALL CLEAR
110  INPUT "WHAT FILE -> ":FILEIN$
120  OPEN #29: FILEIN$,INPUT
130  RECS=0
140  IF EOF(29) THEN 180
150  INPUT #29: INDATA$
160  RECS=RECS+1
170  GOTO 140
180  PRINT : RECS;"RECORDS READ."
190  CLOSE #29
200  END
```

Listing 4-34. EOF Example 1

Example 2:

The program in Listing 4-35 uses EOF to see if there is any room left on the disk. If more data is to be written and there is not enough room left on the disk, the program closes the file and tells you where the file ended.

You can put in another disk and restart the program either at the beginning or at the point where it stopped. Your disk must run out of room before the program will take the EOF action.

```
100  CALL CLEAR
110  PRINT "I'LL WRITE TO A FILE ON":
     " DISK 1 UNTIL THERE'S NO"
120  PRINT " MORE ROOM ON THE DISK.": :
130  PRINT "YOU CAN START ME AT":" ANY VALUE.": :
140  INPUT "WHAT'S YOUR FILENAME ->":FILEOUT$
150  OPEN #66:"DSK1."&FILEOUT$,OUTPUT,FIXED 254
160  INPUT "WHAT'S THE FIRST NUMBER -> ":OUTNUM
170  IF EOF(66)=-1 THEN 220
180  RECS=RECS+1
190  PRINT #66: OUTNUM
200  OUTNUM=OUTNUM+1
210  GOTO 170
220  PRINT "NO MORE ROOM ON DISK 1":" FOR FILE ";
     FILEOUT$
230  PRINT "I'M CLOSING THE FILE NOW."
240  CLOSE #66
250  PRINT "YOU CAN USE ANOTHER":" DISK AND START ME"
260  PRINT " AT VALUE";OUTNUM
270  PRINT : :"BYE NOW."
280  END
```

Listing 4-35. EOF Example 2

EXP	Raise e to a power.
Type:	Function
Format:	EXP(*num-exp*)
Purpose:	The EXP function returns the value of e^x, where e = 2.718281828 and x = *num-exp*.
Operands:	*num-exp* is a number, numeric variable, or numeric expression that specifies the power to which you want to raise e.
Defaults:	None.

Description:

EXP is a numeric function that returns the value of $e^{num\text{-}exp}$, where e = 2.718281828. The EXP function is the inverse of the natural logarithm function, LOG.

You use EXP to assign a value to a variable like this:

```
100   ANS=EXP(X*Y+Z)
```

Or, you can use EXP as part of a numeric expression, like this:

```
200   ANS=SQR(Z+B)/EXP(R)
```

Common Errors:

None.

Example:

The program in Listing 4-36 uses EXP to print the value of e raised to whatever value you enter.

```
100  CALL CLEAR
110  PRINT "ENTER A NUMBER AND":" I'LL TELL YOU WHAT"
120  PRINT " E^NUMBER IS"
130  PRINT : :
140  INPUT "YOUR NUMBER -> ":POWER
150  PRINT :"E TO THE";POWER;"IS";EXP(POWER): :
160  INPUT "TRY AGAIN? (Y/N) -> ":Y$
170  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 130
180  PRINT : : "BYE."
190  END
```

Listing 4-36. EXP Example

FOR . . . TO . . . STEP	Execute statements repeatedly.
Type:	Statement
Format:	*line#* FOR *control* = *init-val* TO *end-val* [STEP *incr*]
Purpose:	A FOR statement repeatedly executes the statements between the FOR and its associated NEXT statement (the FOR loop) until the value of *control* is greater than *end-val*. *control* starts at *init-val* and gets incremented by *incr* (or 1, if you don't use *incr*).

FOR . . . TO . . . STEP Execute statements repeatedly. *(continued)*

Operands: *line#* is a BASIC statement line number that you need when you include FOR in a program. *line#* can be any number between 1 and 32767.
control is the name of a numeric variable. This variable is used in controlling the number of times the FOR loop is executed.
init-val is a number, numeric variable, or numeric expression that specifies the initial value used for *control.*
end-val is a number, numeric variable, or numeric expression that specifies the final value for *control.*
incr is a number, numeric variable, or numeric expression that gets added to *control* each time the FOR loop is executed. If you don't specify a value for *incr,* BASIC uses an increment of 1.

Defaults: If you don't supply an *incr* value for STEP, TI BASIC uses an increment of one (1).

Description:

FOR . . . TO . . . STEP works with a NEXT statement and repeatedly executes the statements between the FOR and NEXT statements. This group of statements is called a "FOR loop."

Fig. 4-5 shows you the limits of a simple FOR loop. A FOR loop begins with the FOR statement and ends with the NEXT statement. The statements between FOR and NEXT are executed until the *control* variable exceeds the *end-val,* 10 times in this example.

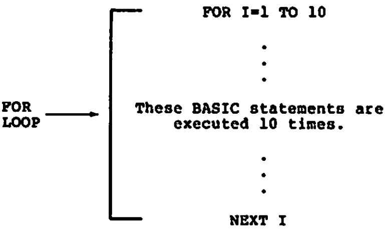


Fig. 4-5. FOR loop example.

Control is a variable used as a counter to keep track of how many times you execute the FOR loop. You can use any numeric variable name for *control.* You can use the same variable for as many FOR loops as you want. You can even use the *control* variable elsewhere in your program.

Even though you can change the values for *init-val, end-val,* or *incr* in the statements in the FOR loop, the original FOR loop values are used until the loop ends. The changed values have no effect on the current FOR loop execution.

If you change the value of the *control* variable, you affect the FOR loop's execution. The new (changed) value of the *control* variable is used when TI BASIC checks against *end-val.*

The *control* variable gets set to *init-val* when the FOR statement is executed the first time. Each time the associated NEXT statement is executed, *control* is incremented by *incr* or by one (if you don't use STEP). If *control* is less than or equal to *end-val*, the statements between FOR and NEXT are executed again.

The *control* variable is checked before the statements in the FOR loop are executed. You can have situations where the FOR loop is never executed. If *control* starts out higher than *end-val* (maybe you used an *init-val* that was higher than *end-val* and you used a positive *incr*), the FOR loop does not get executed.

You can use negative values for *init-val*, *end-val*, and *incr*. The FOR loop ends when *control* is greater than *end-val* so be careful with negative values.

NOTE

If you use a value for *incr*, the value may not be zero. It can be less than one (like −2) or a fraction (like .5).

FOR loops are very useful when you want to repeatedly execute statements. Depending on the values you set for *init-val*, *end-val*, and *incr* (any or all of which can be numeric variables or expressions, as well as numbers), you can even change the number of times the FOR loop is executed each time you execute it.

For example, you want to read numbers into an array. You can use a FOR loop to get the data from the keyboard, with the number of numbers you read changing each time you run the program. You can use this FOR loop:

```
100  DIM A(20)
110  INPUT "HOW MANY NUMBERS (1-20) ":MAX
120  FOR I=1 TO MAX
130  INPUT "ENTER YOUR NUMBER ":A(I)
140  NEXT I
150  PRINT I;"VALUES READ."
160  END
```

You can use an IF or GOTO statement to branch out of a FOR loop at any time. If you do branch out of the FOR loop, the *control* variable contains whatever value it had when you left the loop. The following short program shows you how to leave a FOR loop before it's finished:

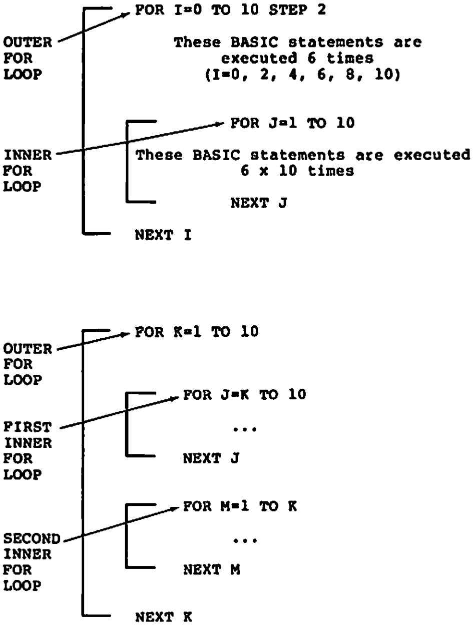
```
100  FOR I=1 TO 10
110  PRINT "IN THE LOOP":"CONTROL IS";I
120  IF I>5 THEN 140
130  NEXT I
140  PRINT :"OUT OF THE LOOP":"CONTROL IS";I
150  END
```

You can use GOSUB statements to leave a FOR loop and then return to it. There is one CAUTION here. Don't change the value of the *control* variable in your subprogram (where you GOSUB to) or you will have an incorrect *control* variable in your original FOR loop when you return. The following example shows you a subprogram and a FOR loop:

```
100   FOR I=1 TO 3
110   PRINT "I IS";I
120   GOSUB 170
130   PRINT "BACK FROM SUB"
140   NEXT I
150   PRINT :"DONE."
160   STOP
170   PRINT "IN SUB. I IS";I
180   RETURN
190   END
```

If you add another statement to this example, (175 I=5), you can see what happens if you change the *control* variable outside the FOR loop and then return to the FOR loop.

And then there are the "nested FOR loops" where you have a FOR loop inside another FOR loop, as shown in Fig. 4-6. Here's where you have to be careful to match your FOR and NEXT statements.



OUTER FOR LOOP
These BASIC statements are executed 6 times (I=0, 2, 4, 6, 8, 10)
INNER FOR LOOP
These BASIC statements are executed 6 x 10 times
OUTER FOR LOOP
FIRST INNER FOR LOOP
SECOND INNER FOR LOOP

```
FOR I=0 TO 10 STEP 2
   FOR J=1 TO 10
   NEXT J
NEXT I
```

```
FOR K=1 TO 10
   FOR J=K TO 10
   ...
   NEXT J
   FOR M=1 TO K
   ...
   NEXT M
NEXT K
```

Fig. 4-6. Nested FOR loops.

Follow these rules for nested FOR loops:

1. The outer loop (the first one you get to) must totally enclose all inner loops (those after the first FOR statement before its associated NEXT statement).

```
FOR I=1 TO 10
    . . .
    FOR J=1 TO 3
      . . .
    NEXT J
NEXT I
```

2. Use a different *control* variable for each nested loop. You can use the same *control* variable for inner loops if they aren't themselves nested.

```
FOR I=1 TO 10
    . . .
    FOR J=1 TO 3
      . . .
    NEXT J
    . . .
    FOR J=50 TO 100
      . . .
    NEXT J
NEXT I
```

3. If you nest to several levels, be certain that you keep the nested loops correctly paired.

```
FOR I=1 TO 10
    . . .
    FOR J=1 TO 3
      . . .
      FOR K=50 TO 100
        . . .
      NEXT K
    NEXT J
NEXT I
```

While nested FOR loops are very useful in programs, you must be careful that you always correctly match the FOR with its NEXT statement. Keep all inner loops inside the outer loop.

Common Errors:

```
BAD VALUE
```

The value of the STEP *incr* is zero.

```
CAN'T DO THAT
```

You tried to use FOR as a command. You can use FOR only as a statement in a BASIC program.

Or, you used a NEXT without first using a FOR.

Or, the *control* in the NEXT statement does not match the *control* in the FOR statement.

INCORRECT STATEMENT

These errors occur when you make a mistake in writing the FOR statement.

- *control* is not followed by an equals (=) sign
- Or, *control* is not a numeric variable
- Or, you forgot the TO keyword or you put something other than STEP or end of line after *end-val*.

Example 1:

The program in Listing 4-37 uses FOR to print the numbers between 10 and 100 by tens. Try changing the values on the FOR statement to print other sequences.

```
100  CALL CLEAR
110  PRINT "I'LL PRINT FROM 10 TO":"  100 BY 10."
120  PRINT : :
130  FOR I=10 TO 100 STEP 10
140  PRINT I
150  NEXT I
160  END
```

Listing 4-37. FOR Example 1

Example 2:

The program in Listing 4-38 uses FOR to create the sum of a series of numbers. You tell the first and last number in the series and the increment to use.

Try changing the FOR loop to include additional calculations, such as multiplying all the numbers together (MULT = MULT*I). Don't forget to set your initial value for MULT, etc.

```
100  CALL CLEAR
110  PRINT "HI THERE.": :"I'M AN ADDING PROGRAM."
120  PRINT : :"TELL ME THE FIRST AND":
     " LAST VALUES AND I'LL"
130  PRINT " TELL YOU THE SUM OF ALL":
     " THE NUMBERS IN BETWEEN."
140  PRINT :"YOU CAN EVEN SKIP VALUES.": :
150  PRINT "ENTER THE STARTING VALUE,":
     " ENDING VALUE, AND INCREMENT": :
```

```
160  PRINT : :
170  INPUT "YOUR NUMBERS (0,0,0 TO END) -> ":
     STVAL,ENDVAL,INC
180  IF (STVAL=0)*(ENDVAL=0)*(INC=0) THEN 300
190  IF INC<>0 THEN 210
200  PRINT :"INCREMENT CAN NOT BE ZERO."
210  GOTO 150
220  SUM=0
230  FOR I=STVAL TO ENDVAL STEP INC
240  SUM=SUM + I
250  NEXT I
260  PRINT :"THE SUM OF ALL THE NUMBERS":" BETWEEN";
270  PRINT STVAL;"AND";ENDVAL:" INCREMENTED BY";INC;
280  PRINT "IS";SUM
290  GOTO 160
300  PRINT : :"GOODBYE."
310  END
```

Listing 4-38. FOR Example 2

CALL GCHAR	Get a character from screen row, col.
Type:	Statement
Format:	[*line#*] CALL GCHAR *(row,col,num-var)*
Purpose:	GCHAR reads directly from the screen buffer and puts the ASCII value of the character at *row* and *col* on the screen into the variable *num-var.*
Operands:	*line#* is a BASIC statement line number that you need when you include CALL GCHAR in a program. You don't need *line#* when you use CALL GCHAR as a command. *line#* can be any number between 1 and 32767. *row* is a number, numeric variable, or numeric expression that specifies the row on the screen from which you want to read a character. *row* may be any value between 1 and 24. *col* is a number, numeric variable, or numeric expression that specifies the column on the screen from which you want to read a character. *col* may be any value between 1 and 32. *num-var* is the name of the numeric variable that will contain the ASCII value of the character at *row,col* on the screen.
Defaults:	None.

Description:

GCHAR reads a character from position *row, col* in the screen buffer and puts the ASCII code for the character into variable *num-var.* Table 4-11 lists the ASCII codes for the characters.

The screen has 24 rows and 32 columns, arranged like the grid shown in Fig. 4-7. Row 1, column 1 is the upper left corner of your screen. Row 24, column 1 is the lower left corner of your screen. Row 1, column 32 is the upper right corner of your screen. Row 24, column 32 is the lower right corner of your screen.

BASIC PRINT and DISPLAY statements use only columns 3 through 30 on the 32 column line. If you want the first character BASIC writes to

Table 4-11. GCHAR ASCII Codes

ASCII Char	Decimal	ASCII Char	Decimal	ASCII Char	Decimal
(space)	32	@	64	`	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
–	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	|	124
=	61	]	93	}	125
>	62	^	94	~	126
?	63	_	95	(DEL)	127

a line for example, you must request *col* 3 in your CALL GCHAR statement.

Remember that the screen "scrolls" upward as you write information to it with PRINT or DISPLAY statements. Thus, the same *row* and *col* may have different values, depending on what's on the line when you call GCHAR. GCHAR gets the value that's at *row, col when the GCHAR statement is executed.*

Common Errors:

BAD VALUE

row or *col* or both are less than 1 or greater than 32.

INCORRECT STATEMENT

num-var is not a numeric variable.

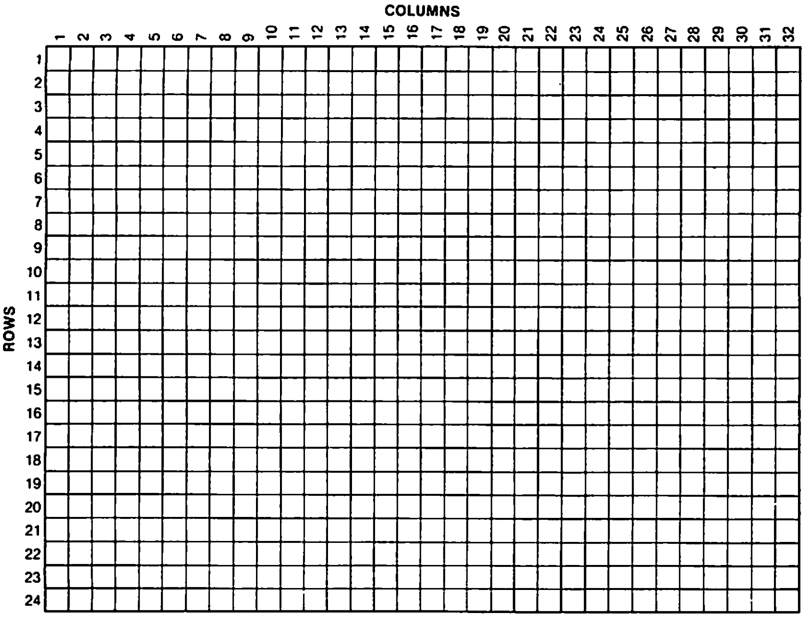


Fig. 4-7. Screen grid diagram.

Example:

The program in Listing 4-39 writes random characters all over the screen and uses GCHAR to read information directly from the screen.

You tell GCHAR which row and column to read. Notice that the same row and column will contain different values as the screen "scrolls" upward.

```
100  CALL CLEAR
110  RANDOMIZE
120  FOR I=2 TO 22
130  FOR J=1 TO 32
140  CH=INT(RND*96)
150  IF (CH<32)+(CH>95) THEN 140
160  CALL VCHAR(I,J,CH)
170  NEXT J
180  NEXT I
190  INPUT "WHICH ROW,COLUMN (0,0 TO END) -> ":ROW,COL
200  IF (ROW=0)*(COL=0) THEN 260
210  IF (ROW<1)+(ROW>24) THEN 190
220  IF (COL<1)+(COL>32) THEN 190
230  CALL GCHAR(ROW,COL,CHREAD)
240  PRINT "ROW";ROW;"COL";COL;"IS";CHREAD;
     "(";CHR$(CHREAD);")"
250  GOTO 190
260  END
```

Listing 4-39. GCHAR Example

Try changing the program so that you have rows of different letters printed across the screen before you use GCHAR.

GOSUB or GO SUB	Call a subprogram.
Type:	Statement
Format:	*line#* GOSUB *line-num* or *line#* GO SUB *line-num*
Purpose:	GOSUB executes a set of statements as a subprogram and returns to the statement after the GOSUB when the subprogram executes a RETURN statement.
Operands:	*line#* is a BASIC statement line number that you need when you include GOSUB in a program. *line#* can be any number between 1 and 32767. *line-num* is the line number of a statement in your program. This is considered to be the beginning of a subprogram (a set of statements that ends with a RETURN statement). *line-num* may be any value between 1 and 32767, as long as it is the line number of a statement in your program.
Defaults:	None.

Description:

GOSUB "calls a subprogram" or "transfers control" to the subprogram at line *line-num*. Like GOTO, GOSUB makes BASIC change the order in which it executes statements. You can use either GOSUB or GO SUB; the space between GO and SUB is optional.

When BASIC executes a GOSUB statement, it "calls a subprogram" or branches to the statement with the line number *line-num* and begins executing the statements at *line-num*. However, after BASIC executes a RETURN statement, BASIC executes the statement immediately after the GOSUB that called the subprogram.

The subprogram is defined as the set of statements that begin at *line-num* and end with a RETURN statement. A subprogram can have as many RETURN statements as you need. Full details on using RETURN statements are in the section describing the RETURN statement.

Subprograms can be considered small programs within your program. They can start with any statement and end with a RETURN statement.

Subprograms are very useful when you are executing the same statements more than once. If you have processing that is too long or too complicated to fit into a function (DEF statement), use a subprogram instead.

Once you get past writing short, simple programs, you will find yourself using a lot of subprograms. You can even call a subprogram from a subprogram. BASIC knows where to RETURN to if you don't make the mistake of using a GOTO instead of a RETURN to get out of a subprogram.

You may hear about "structured programs." Structured programs are programs written with a lot of subprograms, each subprogram doing a logical piece of the program's processing. The technique grew out of the complicated processing done by many large programs.

When you write a long program in this way, it's easy to find out where the errors occur. You can often isolate one or more subprograms and test them separately, since the program's processing is segmented that way.

It's easier to add new features to a program written with a lot of subprograms. If you segment each feature (like commands in a game) to individual subprograms, you can easily add more features by adding more subprograms. You won't have to worry about interfering with existing features because each feature is handled in one compact, unique place. More about this technique in the ON . . . GOSUB statement.

There is one CAUTION when you use GOSUB: Make sure that you don't GOSUB to the same line (100 GOSUB 100). If you do you will run out of memory before you can complete your program.

Common Errors:

BAD LINE NUMBER

The value for *line-num* is not a valid line number in your program.

CAN'T DO THAT

You tried to use GOSUB as a command. You can use GOSUB only as a statement in a program.

MEMORY FULL

The *line-num* in the GOSUB statement is the same as the *line#* for the statement (the GOSUB calls itself).

Or, you have executed too many GOSUBs without RETURNing.

Example 1:

The program in Listing 4-40 uses GOSUB to print the numbers between 1 and 5. This example shows you how to segment a program into a subprogram.

Try changing what the subprogram does.

```
100  CALL CLEAR
110  PRINT "I'M USING A SUBPROGRAM.":
     " TO PRINT THESE NUMBERS."
120  GOSUB 150
130  PRINT : : "BYE NOW."
140  STOP
150  FOR I=1 TO 5
160  PRINT I;
170  NEXT I
180  RETURN
190  END
```

Listing 4-40. GOSUB Example 1

Example 2:

The program in Listing 4-41 uses GOSUB to call two subprograms. The first reads information. The second prints the data. Notice that the input routine has a higher line number than the output routine. It doesn't matter what order your subprograms are in. As long as they end with a RETURN, BASIC knows their beginning and ending lines.

While these are very simple subprograms, the example shows you a technique that is used in many complicated programs. If possible, get your input data in one place and write your results from one place. When you want to change what you read in or write out, you have only one place to worry about.

```
100  CALL CLEAR
110  GOSUB 190
120  GOSUB 150
130  PRINT : :"GOODBYE."
140  STOP
150  PRINT :"I'M IN YOUR OUTPUT":"  SUBROUTINE."
160  PRINT "YOU ENTERED THIS NUMBER ":NUMIN
170  PRINT "AND THIS STRING ":STRINGIN$
180  RETURN
190  PRINT :"I'M IN YOUR INPUT":"  SUBROUTINE."
200  INPUT "ENTER A STRING -> ":STRINGIN$
210  INPUT "ENTER A NUMBER -> ":NUMIN
220  RETURN
230  END
```

Listing 4-41. GOSUB Example 2

GOTO or GO TO	Transfer control to a statement.
Type:	Statement
Format:	*line#* GOTO *line-num* or *line#* GO TO *line-num*
Purpose:	GOTO (or GO TO) "unconditionally branches" to the statement with line number *line-num*.
Operands:	*line#* is a BASIC statement line number that you need when you include GOTO in a program. *line#* can be any number between 1 and 32767. *line-num* is the line number of a statement in your program. This is the statement that BASIC executes after it executes the GOTO statement. *line-num* may be any valid line number between 1 and 32767, as long as there is a statement in your program with the value *line-num*.
Defaults:	None.

Description:

GOTO "branches" or "unconditionally transfers control" to the statement at line *line-num*. Instead of following the normal BASIC execution sequence, each line executed right after the one before it, GOTO makes BASIC jump to another place in your program.

Once the GOTO is executed, BASIC resumes its normal execution sequence at the line that you GOTO. The statement at *line-num* is executed, then the next, and the next, and the next, etc. Unless, of course, you have another GOTO to branch elsewhere.

You can use either GOTO or GO TO as your TI BASIC statement, the space between GO and TO doesn't matter. Use whichever form you prefer.

You can GOTO to any line in your program, before or after the line with the GOTO. It's extremely bad to GOTO the GOTO statement itself. Your program will run forever (an infinite loop), constantly executing a statement that says to branch to itself.

GOTOs are common in programs. You will see them in many examples in this book. In the following short example, BASIC will skip statements 150 and 160 and execute statement 170.

```
120  A=999
130  B=100
140  GOTO 170
150  A=5
160  B=6
170  PRINT A,B
```

Or, if you are setting a variable to different values and then executing an expression, you use GOTO like this:

```
100  INPUT "ENTER A NUMBER ":IN
110  IF IN<0 THEN 140
120  X=50
130  GOTO 150
140  X= -50
150  PRINT X*IN
160  END
```

GOTOs can be used to make a program structure called a "loop," statements that get executed over and over again. When you have a loop in your program, you have to be sure that you get out of it someway, perhaps with an IF statement. This short program shows you how to use a loop. Notice how easily you can make it an "endless" loop if you GOTO 100 instead of GOTO 110.

```
100  A=0
110  A=A+1
120  PRINT A
130  IF A=10 THEN 150
140  GOTO 110
150  END
```

It is not good programming practice to GOTO another GOTO. This makes it difficult to debug a program because it's not always obvious how you got to the point in the program where you are.

If you resequence your program with a RES or RESEQUENCE command, TI BASIC automatically adjusts all the *line-num* operands in your GOTO statements to reflect the new line number values.

Common Errors:

BAD LINE NUMBER

The value of *line-num* is not a valid line number in your program.

CAN'T DO THAT

You tried to use GOTO as a command. You can use GOTO only as a statement in a program.

Example 1:

The program in Listing 4-42 uses GOTO to branch back to a question if you don't answer correctly—you answer with a letter other than Y, y, N, or n.

```
100  CALL CLEAR
110  PRINT "I'LL PRINT THE NUMBERS":" FROM 1 TO 10."
120  PRINT : :
130  FOR I=1 TO 10
140  PRINT I;
150  NEXT I
160  PRINT : :
170  INPUT "WANT TO SEE THEM AGAIN? (Y/N) ->":Y$
180  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
190  IF (SEG$(Y$,1,1)="N")+(SEG$(Y$,1,1)="n") THEN 210
200  GOTO 160
210  PRINT "BYE NOW."
220  END
```

Listing 4-42. GOTO Example 1

Example 2:

The program in Listing 4-43 uses GOTO to unconditionally branch back to the beginning of a program. The only way to stop this program is to press FCTN 4 (CLEAR).

This example shows you a common problem that occurs when you use GOTO incorrectly. The situation is called an "endless loop," which means that the program will run forever, unless you stop it with an FCTN 4.

```
100  CALL CLEAR
110  PRINT "I'LL CONTINUE UNTIL":" YOU PRESS FCTN 4."
120  GOTO 110
130  END
```

Listing 4-43. GOTO Example 2

CALL HCHAR	Write character(s) at row, col.
Type:	Statement
Format:	[*line#*] CALL HCHAR *(row,col,ASCII-code)* or [*line#*] CALL HCHAR *(row,col,ASCII-code,repetitions)*
Purpose:	CALL HCHAR writes *repetitions* horizontal (across the screen) copies of the character represented by *ASCII-code* on your screen. The first character is written at row *row* and column *col*. The second character (if *repetitions* is greater than one) is written at row *row* and column *col* + 1.
Operands:	*line#* is a TI BASIC statement line number that you need when you include CALL HCHAR in a program. You don't need *line#* when you use CALL HCHAR as a command. *line#* can be any number between 1 and 32767. *row* is a number, numeric variable, or numeric expression that contains the row on your screen where you want to write the first character represented by *ASCII-code*. *row* may be any number between 1 and 24, the number of rows on your screen. *column* is a number, numeric variable, or numeric expression that contains the column on your screen where you want to write the first character represented by *ASCII-code*. *column* may be any number between 1 and 32, the number of columns on your screen. *ASCII-code* is a number, numeric variable, or numeric expression that contains the ASCII value of the character you want to write at *row,col*. Table 4-12 shows you the ASCII codes for the characters. *ASCII-code* must not be less than zero or greater than 32767. *repetitions* is a number, numeric variable, or numeric expression that tells HCHAR how many times you want the character repeated on the row on the screen. *repetitions* must not be less than one or greater than 32767.
Defaults:	If you don't supply a value for *repetitions*, HCHAR writes *one* character on the screen at *row* and *col*.

Description:

HCHAR writes the character with the ASCII value *ASCII-code* at row *row* and column *col*. If you use a value for *repetitions*, you'll get that many characters written across the screen (horizontally) beginning at *row,col*. Table 4-12 shows you the ASCII codes for the TI-99/4A standard characters. You can define characters for ASCII-codes 128 through 159 using CHAR.

With HCHAR, you can write one or more characters at any row and column on your screen. The top left corner of your screen is row 1, column 1. The bottom left corner is row 24, column 1. The upper right corner is row 1, column 32. The bottom right corner is row 24, column 32. Depending on the adjustment of your television set, you may not be able to see the characters in columns 1 and 32.

If the number of *repetitions* you specify exceeds the number of places remaining on the *row*, placement of characters proceeds to the next row on the screen (*row* + 1). If the last character of the last row is reached with *repetitions* still remaining, character placement continues with the first

Table 4-12. HCHAR and ASCII Character Codes

ASCII Char	Decimal	ASCII Char	Decimal	ASCII Char	Decimal
(space)	32	@	64	‘	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
–	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	|	124
=	61	]	93	}	125
>	62	^	94	~	126
?	63	_	95	(DEL)	127

character of row one (1). Because there are 768 character positions on the screen (24 rows times 32 columns), a *repetitions* factor greater than 768 causes the same position to be repeatedly overwritten by the same character.

If you specify an ASCII-code greater than 255, 256 is repeatedly subtracted from it until its value is less than or equal to 255. Thus an ASCII-code of 300 results in display of the comma (ASCII character 44 = 300 − 256).

While VCHAR, HCHAR, and PRINT all put characters on your screen, there is one important difference between PRINT and VCHAR/HCHAR. With PRINT you can put a maximum of 28 characters across the screen. With VCHAR and HCHAR you can put a maximum of 32 characters across the screen.

HCHAR, and its relative VCHAR, are very useful when you design screens. You can even design your own special graphics characters with

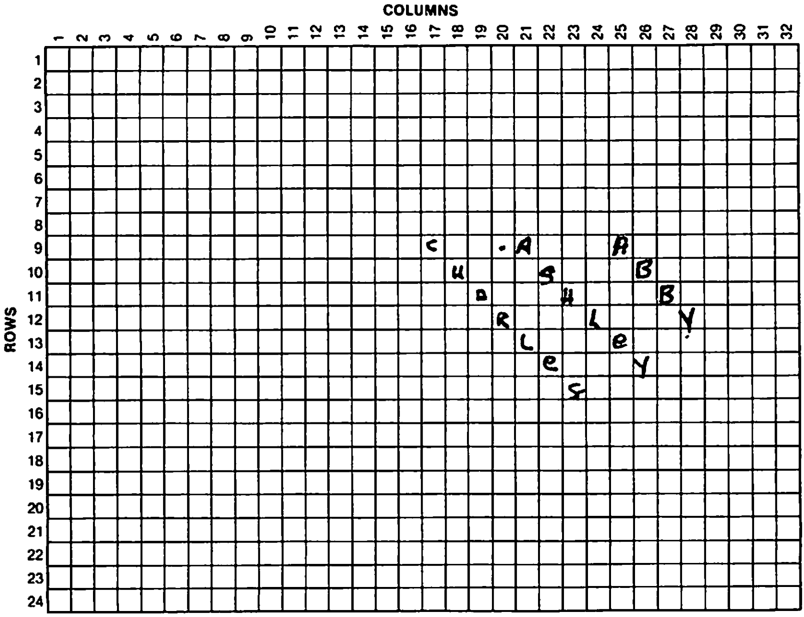


Fig. 4-8. HCHAR screen grid design.

CHAR. You will find it easy to design a screen if you use a grid like the one in Fig. 4-8. Make a 24-row by 32-column grid and fill in the squares with the characters you want to use. Example 1 (below) shows you how to do this.

Another use for HCHAR is writing messages in specific positions on your screen. You use SEG$ to get each letter of a message from a string variable and then print each letter using HCHAR. You can print messages at specific rows on the screen and the screen will not scroll upward. This technique is particularly useful when you want to write messages at a specific line on the screen without disturbing the remainder of the screen.

Suppose you're playing a game and you want the top 20 lines of the screen to remain intact (except, of course, for action from the game) while you write messages at the bottom of the screen. Example 2 (below) uses HCHAR in just this way.

Common Errors:

```
BAD VALUE
```

Row is less than 1 or greater than 24. Or, *col* is less than 1 or greater than 32.

Or, *ASCII-code* is less than 0 or greater than 32767. Remember that only values between 32 and 127 print the standard characters. Values between 128 and 159 print the characters you define through CHAR. Other values may or may not print a character, depending on what is in memory at the location used.

Repetitions is less than 0 or greater than 32767.

Example 1:

The program in Listing 4-44 uses HCHAR to write the mesage "HI" in big letters on your screen. Fig. 4-9 shows you how to fill in the grid.

Try filling in your name across the bottom of the screen and using HCHAR and VCHAR to write the characters.

```
100  CALL CLEAR
110  CALL CHAR(128,"AAAAAAAAAAAAAAAA")
120  FOR I=4 TO 8
130  CALL HCHAR(I,6,128)
140  CALL HCHAR(I,9,128)
150  CALL HCHAR(I,12,128)
160  NEXT I
170  CALL HCHAR(6,7,128,2)
180  END
```

Listing 4-44. HCHAR Example 1

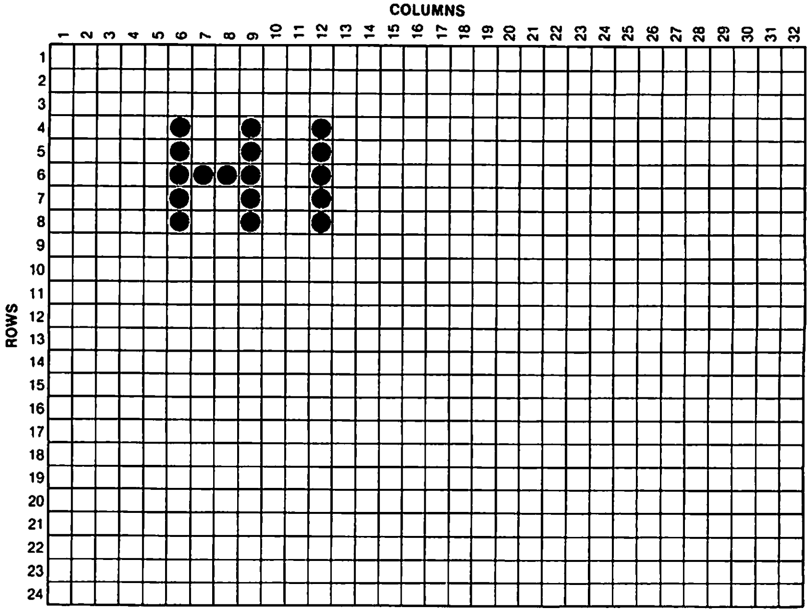


Fig. 4-9. HCHAR example.

Example 2:

The program in Listing 4-45 asks you for a message and at what row and column you want to print the message. Then, it clears the screen and uses a subprogram with HCHAR to write the message on the screen. The subprogram that begins at line 1000 can be included in your own programs.

If the message is longer than 32 characters (the number of columns on your screen), it will take more than one line to write the message.

Notice that the screen will not "scroll" (move upward for each line printed) while the message is being written through HCHAR. But, when the INPUT statement gets executed, the entire screen scrolls upward one line.

```
100  CALL CLEAR
110  PRINT "ENTER A MESSAGE AND":"  WHERE YOU WANT TO"
120  PRINT "  WRITE IT ON THE SCREEN": :
130  INPUT "YOUR MESSAGE -> ":MSG$
140  INPUT "WHAT ROW,COL -> ":ROW,COL
150  IF (ROW<1)+(ROW>24)+(COL<1)+(COL>32) THEN 140
160  CALL CLEAR
170  GOSUB 1000
180  INPUT "PRESS ENTER TO STOP":X$
190  STOP
1000 REM PRINT MSG$ STRING AT ROW,COL
1010 DONE=0
1020 IF LEN(MSG$)<=32-COL THEN 1060
1030 TMP$=SEG$(MSG$,32-COL,255)
1040 MSG$=SEG$(MSG$,1,32-COL)
1050 DONE=1
1060 FOR I=1 TO LEN(MSG$)
1070 CALL HCHAR(ROW,COL+I-1,ASC(SEG$(MSG$,I,1)))
1080 NEXT I
1090 IF DONE=0 THEN 1230
1100 ROW=ROW+1
1200 COL=1
1210 TMP$=MSG$
1220 GOTO 1020
1230 RETURN
1240 END
```

Listing 4-45. HCHAR Example 2

IF . . . THEN . . . ELSE	Evaluate condition, then branch.
Type:	Statement
Format:	*line#* IF *condition* THEN *then-line* or *line#* IF *condition* THEN *then-line* ELSE *else-line*
Purpose:	IF . . . THEN . . . ELSE performs a "conditional branch." TI BASIC evaluates the *condition* after IF and branches to the statement *then-line* when the *condition* is true or to the statement *else-line* when the *condition* is false (or to the statement following the IF statement if you don't use ELSE).

IF . . . THEN . . . ELSE	Evaluate condition, then branch. *(continued)*
Operands:	*line#* is a TI BASIC statement line number that you need when you include IF in a program. *line#* can be any number between 1 and 32767. *condition* is any numeric expression or relational expression that gets evaluated. If the *condition* is true (evaluates to a value other than zero), TI BASIC branches to *then-line*. If the *condition* is false (evaluates to zero), TI BASIC branches to *else-line* or the next statement (depending on whether or not you use ELSE). *then-line* is the line number of a statement in your program. When *condition* is true, TI BASIC branches to the statement with *then-line*. *else-line* is the line number of a statement in your program. When *condition* is false, TI BASIC branches to the statement with *else-line*.
Defaults:	If you don't use an ELSE keyword and *else-line*, TI BASIC executes the statement following the IF statement when *condition* is false.

Description:

IF . . . THEN . . . ELSE lets you change the normal sequential order in which TI BASIC executes statements.

IF performs a "conditional branch" in your program. While a GOTO unconditionally branches to a statement, IF first determines whether *condition* is true or false and then branches to line number *then-line* when the *condition* is true or line number *else-line* when it's false. If you don't use an ELSE and *else-line*, TI BASIC executes the statement immediately following the IF statement when *condition* is false.

Table 4-13 shows you what criteria TI BASIC uses to determine true or false for the IF *condition*.

Table 4-13. IF *condition* Results

condition	Result
Evaluates to zero	False
Evaluates to a value other than zero	True

Condition can be any test that you want to make. If you compare strings, all values must be strings. You can't mix numbers and strings in one test. Some simple tests *(condition)* using IF are:

```
ANS=100
NAME$="SMITH"
LEN(MSG$)<=32
SEG$(ANS$,1,1)="Y"
```

These tests use the relational operators shown in Table 4-14. You can also use the logical operators shown in Table 4-15 to make complex *con-*

ditions. If the result of the operation is not zero, the condition is true. Logical and relational operations and operators are discussed in detail in Chapter 2.

Table 4-14. IF Relational Operators

Operator	Meaning
A=B	A is equal to B.
A>B	A is greater than B.
A<B	A is less than B.
A<>B	A is not equal to B.
A<=B	A is less than or equal to B.
A>=B	A is greater than or equal to B.

Table 4-15. IF Logical Operators

Operator	Meaning
+ (plus)	Logical OR
(A=B)+(C=D)	A is equal to B OR C is equal to D.
* (multiplication)	Logical AND
(A=B)*(C=D)	A is equal to B AND C is equal to D.

When TI BASIC compares numbers or numeric expressions, the results of any expressions are compared algebraically.

When TI BASIC compares strings or string expressions, the strings are compared character by character, left to right. A character with a higher ASCII code is considered "larger" than one with a lower ASCII code, so you can easily sort strings. When one string is longer than the other, the comparison is performed for as many characters as there are in the shorter string. Table 4-16 shows the ASCII codes for the standard characters.

Some examples of simple relational conditions are:

```
IF ANS=0 THEN 400 ELSE 700
IF MSG$=“DONE” THEN 350
IF NAME$=“MARY” THEN 1020
IF ANS$<>“YES” THEN 500
```

Some examples using logical operators in the conditions are:

```
IF (ANS$=“Y”)+(ANS$=“y”) THEN 200
IF (ROW<1)+(ROW>24) THEN 400 ELSE 900
IF (RESULT<0)*(ANS$=“Y”) THEN 5000
```

You will notice that there are parentheses ( ) around parts of the conditions. The parentheses can be used for your own convenience, as they are in the above conditions, to make it easier to read. Or, you can use parentheses to group tests into larger tests, such as:

```
IF (A=B)*((C=D)+(X=Y)) THEN 100
IF ((A=B)*(C=D))+(X=Y) THEN 100
```

Table 4-16. ASCII Character Codes

ASCII Char	Decimal	ASCII Char	Decimal	ASCII Char	Decimal
(space)	32	@	64	`	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
–	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	|	124
=	61	]	93	}	125
>	62	^	94	~	126
?	63	_	95	(DEL)	127

These two statements are very different. The first statement says to branch to statement 100 when:

A = B AND either C = D or X = Y

The second statement says to take the branch when:

both A = B and C = D OR when X = Y

Suppose you want to branch to statement 150 when your answer (RESULT) is negative. You can use IF this way:

```
IF RESULT<0 THEN 100
```

When RESULT is negative, TI BASIC branches to statement 100. When it's zero or positive, TI BASIC executes the statement after the IF.

It is sometimes convenient to use a two-way branch. Suppose you want to rerun your program when the answer (ANS$) is "Y" or to ask another question when the answer is not "Y". Then use:

```
IF ANS$ = "Y" THEN 150 ELSE 900
```

When you use this format, you should remember that there is no way for TI BASIC to execute the statement after the IF statement unless you GOTO or GOSUB it. Beware of unexecutable sections of code that you may generate this way.

Common Errors:

CAN'T DO THAT

You tried to use IF as a command. You can only use IF as a statement in a program.

INCORRECT STATEMENT

THEN is not followed by a line number *(then-line)*. Or, you forgot the keyword THEN.

Example 1:

The program in Listing 4-46 is a version of the old "guess a number" game. Your guess is compared to the computer's number in IF statements.

The other use of an IF statement shows you a logical operator (+) which means OR. If you answer "Y" or "y" then TI BASIC branches to statement 120.

Try other logical IF statements.

```
100  CALL CLEAR
110  RANDOMIZE
120  COMPNUM=INT(RND*100)
130  PRINT "I HAVE A NUMBER BETWEEN":"  0 AND 100."
140  PRINT : :"TRY TO GUESS IT.": :
150  TRIES=0
160  INPUT "YOUR GUESS -> ":GUESS
170  TRIES=TRIES+1
180  IF GUESS<>COMPNUM THEN 250
190  PRINT :"CONGRATULATIONS! !"
200  PRINT "YOU GUESSED IT IN":TRIES;"TRIES.": :
210  INPUT "TRY AGAIN? (Y/N) -> ":Y$
220  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
230  PRINT :"GOODBYE."
240  STOP
250  IF GUESS<COMPNUM THEN 280
260  PRINT "TOO HIGH."
270  GOTO 160
280  PRINT "TOO LOW."
290  GOTO 160
300  END
```

Listing 4-46. IF Example 1

Example 2:

The program in Listing 4-47 uses IF statements to see if the values you enter are out of range.

This program uses a complex logical IF statement with both logical operators + (OR) and * (AND) to check whether the row and column values you enter are within range. When they are in range, TI BASIC branches to statement 190; if not, TI BASIC executes the next statement.

Try changing the range of valid numbers.

```
100  CALL CLEAR
110  PRINT "ENTER A ROW, COL, AND":
     " ASCII VALUE AND I'LL"
120  PRINT " WRITE THAT CHARACTER.":
     " FOR 5 ROWS AND COLUMNS."
130  PRINT
140  INPUT "ROW (1-24), COL (1-32) -> ":ROW,COL
150  INPUT "ASCII VALUE (30-126) -> ":ASCII
160  IF (ROW>0)*(ROW<25)*(COL>0)*(COL<33) THEN 190
170  PRINT "USE A ROW BETWEEN 1 AND 24":
     " AND A COL BETWEEN 1 AND 32."
180  GOTO 130
190  IF (ASCII>29)*(ASCII<127) THEN 220
200  PRINT "USE AN ASCII VALUE BETWEEN":
     "  30 AND 126.": :
210  GOTO 150
220  CALL CLEAR
230  CALL VCHAR(ROW,COL,ASCII,5)
240  CALL HCHAR(ROW,COL,ASCII,5)
250  INPUT "TRY ANOTHER? (Y/N) ->":Y$
260  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 130
270  PRINT :"BYE."
280  END
```

Listing 4-47. IF Example 2

INPUT	Get information from keyboard.
Type:	Statement
Format:	*line#* INPUT *prompt: variable-list* or *line#* INPUT *variable-list*
Purpose:	INPUT writes a message to the screen *(prompt)* and waits for you to enter data from the keyboard. You must enter enough data to fill all the variables in *variable-list*.
Operands:	*line#* is a TI BASIC statement line number that you need when you include INPUT in a program. *line#* can be any number between 1 and 32767. *prompt* is a string, string variable, or string expression that TI BASIC prints on the screen when it executes the INPUT statement. *prompt* must be followed by a colon (:). *variable-list* is a list of one or more string or numeric variables. When you enter data from your keyboard, TI BASIC puts the data into the variables in the order in which they appear in *variable-list*. You must enter enough data to give all the variables in *variable-list* a value.
Defaults:	If you don't use a *prompt*, TI BASIC writes a question mark character (?) and one space when it executes the INPUT statement. If you do not use *prompt*, do not use the colon (:).

Description:

INPUT statements are a link between your program's variables and the outside world. You use INPUT statements to read values entered from the keyboard into variables in your program.

With INPUT, TI BASIC writes a message *(prompt)* to the screen and reads data entered from the keyboard into the variables in *variable-list*, in the order in which the variables appear. TI BASIC won't execute any more statements until you enter enough data to give every variable in *variable-list* a value.

You enter values separated by commas. The total number of characters in the data (including the separating commas) that you enter for the variables must be less than a total of 112 characters, which is the number of characters that will fit onto four screen lines (4 x 28). TI BASIC defines four screen lines as its maximum keyboard data input line. If you have to enter more than 112 characters, use several INPUT statements. If you try to enter more than 112 characters, TI BASIC refuses to accept more input.

The values you enter must be appropriate for the variables in the *variable-list*. TI BASIC checks every value you enter to determine whether it is valid for the type of variable it's going into. You must enter numeric data for numeric variables and string data for string variables.

NOTE

Strings must be enclosed in double quotes (") if they contain leading blanks, trailing blanks, or commas. For example:

“   this string contains leading blanks”
“this string contains trailing blanks   ”
“and now, commas(,) and periods (. . .)”

Remember that numbers represent valid string data. If you enter numbers for a string, TI BASIC puts the string format of the number into the variable. However, if you enter string data (characters other than the valid digits 0 – >9 and decimal point) for a number, TI BASIC writes this warning message to the screen and gives you a chance to re-enter the data:

```
WARNING: INPUT ERROR, TRY AGAIN
```

prompt is the message that TI BASIC writes to the screen when you are supposed to enter data for the INPUT statement *variable-list* variables. *prompt* usually tells you what to enter and can be a string, string variable, or string expression, as shown in these examples:

```
100  MSG$ = "ENTER YOUR NAME "                    (string)
110  INPUT MSG$:NAME$                             (string variable)
100  INPUT "ENTER YOUR NAME ":NAME$               (string)
110  INPUT "HOW MANY NUMBERS, "&NAME$:ANS         (string expression)
```

You do not need a *prompt* when you use an INPUT statement. But, with no *prompt*, TI BASIC writes “? ” to tell you that you are supposed to be entering data. Your INPUT statement could look like these without a *prompt:*

```
100  INPUT X,Y,Z$                  (enter two numbers, one string)
100  INPUT NAME$                   (enter one string)
100  INPUT ANS                     (enter one number)
100  INPUT ARRAY(I) ,X$            (enter one number, one string)
```

While it appears no *prompt* means you have to guess what to enter, this form of the INPUT statement is really quite useful.

Suppose you want to clear the screen and write a long message, with values from several variables and written on several lines, and enter data for a varying number of array elements. You would have to include extra spaces in *prompt* to make your message print in the form you want. Your message alone might be longer than a single INPUT statement could be (approximately four lines on your screen). You could:

```
300  CALL CLEAR
310  PRINT "YOU HAVE TO ENTER";NUMVAL;"ENTRIES."
320  PRINT :"YOU CAN ENTER VALUES":" BETWEEN;STVAL;
330  PRINT "AND";ENDVAL
340  PRINT "PLEASE PRESS ENTER AFTER":" EACH ENTRY."
350  FOR I=1 TO NUMVAL
360  INPUT ARRAY(I)
370  IF (ARRAY(I)> = STVAL)*(ARRAY(I)< = ENDVAL) THEN 400
380  PRINT "USE A VALUE BETWEEN";STVAL;"AND";ENDVAL
390  GOTO 360
400  NEXT I
```

In the above example, you print a long message with variable values in it and get a different number of entries each time you execute the statements. The INPUT statement at line 360 will print "? " for each entry. You can even do your own range validation and re-execute the INPUT for an invalid entry. TI BASIC will, of course, make sure that you enter only numbers since you are asking for data for a numeric array.

Since TI BASIC reads the data into the variables in the order in which they appear in the variable list, you must be careful when you are entering both a subscript and a value for an array. The subscript must appear first, like this:

```
100  INPUT "ENTER SEQ, VALUE":I,ARRAY(I)
```

In this example, TI BASIC first reads the variable I and then uses the value you enter as the subscript for ARRAY. If you enter 4,300 then TI BASIC puts the value 300 in ARRAY(4).

Common Errors:

```
CAN'T DO THAT
```

You tried to use INPUT as a command. You can only use INPUT as a statement in a program.

INCORRECT STATEMENT

The *prompt* is not a valid string, string variable, or string expression.

INPUT ERROR

You tried to enter non-numeric data for a numeric variable. One or more characters is not a valid digit 0 – >9 or a decimal point (.). This is only a warning. You get another chance to enter the data.

Or, the data you entered was not properly separated by commas. This is only a warning. Your program will not stop. You get another chance to enter the number.

Or, the line you entered was longer than 112 characters. This is only a warning. You get another chance to enter the data.

Or, the number you entered is too large. This is only a warning. You get another chance to enter the data.

Or, the number of variables in *variable-list* does not match the number of data items read from the file. This is only a warning. You get another chance to enter the data.

Example 1:

The program in Listing 4-48 uses two INPUT statements. The first asks you to enter your name. The second makes the computer wait until you press the **ENTER** key before it continues with the program.

This second use of INPUT lets you "pause" before continuing with your program, a valuable technique when you want someone to read what is on the screen before it "scrolls" off the top of the television screen.

Try adding more INPUT and PRINT statements to get numbers and other string data.

```
100  CALL CLEAR
110  INPUT "WHAT'S YOUR NAME? ":NAME$
120  PRINT "HELLO, ";NAME$
130  PRINT : :
140  INPUT "PRESS ENTER TO STOP.":X$
150  PRINT : :"BYE NOW, ";NAME$
160  END
```

Listing 4-48. INPUT Example 1

Example 2:

The program in Listing 4-49 shows you how to use INPUT with arrays. You enter five strings and the program prints them in reverse order.

Try using different types of arrays (like numbers) or printing the strings in random order.

```
100  CALL CLEAR
110  DIM ANS$(5)
120  PRINT "HI THERE."
130  PRINT : :"IF YOU ENTER 5 STRINGS"
140  PRINT " I'LL PRINT THEM BACKWARDS.": :
150  FOR Q=1 TO 5
160  INPUT "STRING "&STR$(Q)&" -> ":ANS$(Q)
170  NEXT Q
180  PRINT
190  FOR I=1 TO 5
200  PRINT :"STRING";I;"IS:"
210  FOR X=LEN(ANS$(I)) TO 1 STEP -1
220  PRINT SEG$(ANS$(I),X,1);
230  NEXT X
240  NEXT I
250  PRINT
260  INPUT "TRY AGAIN? (Y/N) -> ":Y$
270  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 150
280  PRINT : :"GOODBYE."
290  END
```

Listing 4-49. INPUT Example 2

INPUT #	Get information from a file.
Type:	Statement
Format:	*line#* INPUT# *file-num* : *variable-list* or *line#* INPUT# *file-num*, REC *rec-num* : *variable-list*
Purpose:	INPUT # reads data from the file OPENed as *file-num* into the variables in *variable-list*. If the file is a RELATIVE file on a disk, the data is read from record *rec-num*.
Operands:	*line#* is a TI BASIC statement line number that you need when you include INPUT# in a program. *line#* can be any number between 1 and 32767. *file-num* is a number, numeric variable, or numeric expression that corresponds to the value used when you OPENed the file. *file-num* can be any value between 0 and 255, where 0 is always the keyboard. *variable-list* is a list of one or more string or numeric variables. When you read data from your file, TI BASIC puts the data into the variables in the order in which they appear in *variable-list*. *rec-num* is a number, numeric variable, or numeric expression that corresponds to the record number of the record that you want to read from a disk file that you OPENed as a RELATIVE file.
Defaults:	None.

Description:

INPUT # reads data from the file you OPENed as *file-num* into the variables in *variable-list*. Chapter 3 has a detailed description of files and file structures on your TI-99/4A.

file-num must be the number of a file that you have already OPENed. A *file-num* of zero (0) means read from the keyboard which works exactly like an INPUT statement (except you cannot write a message like you do with the INPUT *prompt*).

variable-list is a list of one or more numeric or string variables or both. When INPUT # reads a record from *file-num,* the values are put into the variables in *variable-list* in the order in which they are read. You must read numeric data into numeric variables and string data into string variables.

REC and *rec-num* are used only with disk files that you OPENed as RELATIVE. RELATIVE files are random access files. This means that you can ask for a record by its *rec-num*. RELATIVE files and the INPUT # statement are discussed below. Random access files are discussed in Chapter 3.

You normally use an EOF function to determine when you reach end of file in a disk file.

You can put an IF statement with an EOF function before your INPUT # statement, like this:

```
100  IF EOF(55) <> 0 THEN 500
110  INPUT #55: A, B, C$
```

These statements check for the end of the file opened as #55. If EOF is zero, you still have records to read in the file. If EOF is not zero, there are no records left to read and you branch to statement 500. Don't try to read the file past its end or you will get an error.

While the INPUT # statement is very similar to the INPUT statement, there are some important differences. TI BASIC allocates a special area in memory, called an I/O buffer, for each file that you OPEN.

When you INPUT # a record from the file, TI BASIC reads a record into the file's I/O buffer. Then values are assigned to the variables in *variable-list*. If there are more variables in *variable-list* than there are in the record that TI BASIC read into the I/O buffer, TI BASIC reads another record from the file into the I/O buffer and continues assigning values to the variables.

For example, suppose your file has records with three numeric values per record and your INPUT # statement looks like this:

```
100  INPUT # 4: HOURS, RATE, OTHOURS, OTRATE
```

When TI BASIC executes this INPUT # statement, it first reads a record from the file you OPENed as #4 into the I/O buffer for the file. Since there are three values in the buffer, variables HOURS, RATE, and OTHOURS they get values assigned to them. But, your INPUT # statement's *variable-list* still has one variable, OTRATE, which doesn't have values.

TI BASIC reads another record from file #4. Now, there are three more values in the I/O buffer. The first value gets assigned to OTRATE. All processing for the INPUT # statement is complete and TI BASIC executes the next statement in your program.

But what does TI BASIC do with any data left in the I/O buffer after all the variables have values assigned? That depends on whether your INPUT

# statement ends with a comma (,). In the above example, the INPUT # statement did not end in a comma. When the four variables have values assigned, there are still two values left in the I/O buffer. TI BASIC ignores them. The next INPUT # statement gets a new record from the file.

Pending INPUT # Statements—TI BASIC also has *pending* INPUT # statements. A pending INPUT # statement has a *variable-list* that ends in a comma (,). This makes TI BASIC use any values left in the file's I/O buffer for the next INPUT # statement's *variable-list* variables. You can not use a pending INPUT # with file #0, the keyboard.

If your INPUT # statement looked like this (notice the ending comma), TI BASIC would process the extra two values in a different way:

```
100   INPUT # 4: HOURS, RATE, OTHOURS, OTRATE,
```

With this INPUT # statement (ending with a comma), TI BASIC keeps the two values left in its I/O buffer for the file and uses these values the next time it executes an INPUT # statement for file #4.

NOTE

The EOF function checks to see if there is another record to be read from the file. It does not look for a pending read. Therefore, you may have data remaining to be read, but get a nonzero return from EOF.

If you OPENed the file as UPDATE and you use a pending INPUT # statement, a PRINT # to the same file will cancel the pending INPUT # processing. The next INPUT # will read another record from the file instead of using the data already in the I/O buffer.

Relative Files and INPUT #—Relative files (disk files OPENed with a RELATIVE attribute) are random access files where each record has a unique number, *rec-num*. The first record in the file is record 0. The second is record 1, etc.

TI BASIC sets up a record counter for each RELATIVE file when you OPEN it. Each time you INPUT # or PRINT # (without a *rec-num*) to the file, TI BASIC increases the file's record counter by one.

With INPUT # and a RELATIVE file, you can read the file randomly (by asking for a record by its specific number) or sequentially (beginning at record zero and getting the next record for the next INPUT #).

When you want to INPUT # the RELATIVE file sequentially, TI BASIC reads record 0 for your first INPUT #. The next INPUT # increases the record counter by one and TI BASIC gets record 1. Each INPUT # and PRINT # to the file increases the record counter by 1. To read values for variables A and B from the RELATIVE file you OPENed as 33, use:

```
200   INPUT # 33: A, B
```

You can also INPUT # records randomly (by specific record numbers) by using the REC *rec-num* operand. TI BASIC gets you the record you ask for as *rec-num*. To read values for variables A and B from the record with record number 99 in the file you OPENed as 214, use:

```
250   INPUT # 214,REC 99 : A, B
```

When you use a pending INPUT # statement and your next INPUT # statement uses a REC option, TI BASIC ignores the pending INPUT # statement and reads the requested record *(rec-num)* from the file.

Once you have read a record from a file using a *rec-num*, you have positioned the file to that record. Each subsequent INPUT # or PRINT # statement without a REC option increases TI BASIC's record counter by one, as though you were sequentially processing the file from that point.

Common Errors:

CAN'T DO THAT

You tried to use INPUT # as a command. You can only use INPUT # as a statement in a program.

FILE ERROR

You tried to INPUT # data from a file that you OPENed as OUTPUT or APPEND.

Or, you tried to INPUT # from a file that you have not OPENed.

INCORRECT STATEMENT

There isn't any number sign (#) before or colon (:) after the *file-num*.

INPUT ERROR

You tried to read non-numeric data into a numeric variable. One or more characters is not a valid digit 0 – >9 or a decimal point (.). You can get this error when you read pad characters in a file.

Or, a numeric data item caused an overflow.

Or, the number of variables in *variable-list* does not match the number of data items read from the file.

I/O ERROR 23

You tried an illegal operation with your INPUT # statement. Check that your file is OPENed as INPUT.

I/O ERROR 25

You read past the end of your file. Use the EOF function to see when you are at the end of the file.

I/O ERROR 26

The device with the file you are trying to INPUT # has an error. Perhaps it became disconnected after you started your program.

STRING-NUMBER MISMATCH

You used a non-numeric value for *file-num*.

Example 1:

The program in Listing 4-50 reads records containing string data from a cassette file. Each record contains three values: NAME$, STREET$, and STATE$. There's a final record (an end of file marker record) with all three strings set to "ZZZZ", an unlikely name, address, and state.

Before you can run this program, you have to create a cassette file. There's a program to do this in the PRINT # section.

Try changing the program to read other types of data from a cassette file that you create. Read more string data, such as first and last names. Add numeric data, such as telephone numbers or birthdays.

```
100  CALL CLEAR
110  OPEN #200: "CS1",INPUT,SEQUENTIAL,
     INTERNAL,FIXED 192
120  PRINT "HERE ARE YOUR NAMES":"  AND ADDRESSES.": :
130  INPUT # 200:NAME$,STREET$,STATE$
140  RECS=RECS+1
150  IF (NAME$="ZZZZ")*(STREET$="ZZZZ")*
     (STATE$="ZZZZ") THEN 180
160  PRINT :"NAME:";NAME$:TAB(3);STREET$:TAB(3);STATE$
170  GOTO 130
180  PRINT :RECS-1;"RECORDS READ."
190  END
```

Listing 4-50. INPUT # Example 1

Example 2:

The program in Listing 4-51 reads numbers from a disk file. The numbers are written to the file in one of the example programs in the PRINT # description.

```
100  CALL CLEAR
110  PRINT "I'LL READ FROM A FILE ON DSK1."
120  INPUT "WHAT'S YOUR FILE NAME ->":FILEIN$
130  OPEN # 155:"DSK1."&FILEIN$,INPUT,INTERNAL,
     VARIABLE,SEQUENTIAL
140  RECS=0
150  IF EOF(155)<>0 THEN 200
160  INPUT #155: VALUE,
170  PRINT "NUMBER IS";VALUE
180  RECS=RECS+1
190  GOTO 150
200  CLOSE #155
210  PRINT :RECS;"NUMBERS READ."
220  END
```

Listing 4-51. INPUT # Example 2

The PRINT # program writes five random numbers per record. This program reads a single value with each INPUT # statement. Since this is a file on a disk, you can use the EOF function to see when you're at the end of file. You don't need a special record to mark the end of file, as you did in the previous cassette file example.

INT	Return an integer value.
Type:	Function
Format:	INT*(num-exp)*
Purpose:	The INT function returns an integer value of *num-exp* (the form of *num-exp* with no decimal places).
Operands:	*num-exp* is a number, numeric variable, or numeric expression.
Defaults:	None.

Description:

INT returns the largest integer (whole number) that is not greater than *num-exp*.

The integer form of a number is the number without any decimal places. INT(123.45) is 123.

For negative values, INT returns the next smallest value. INT(− 123.45) is − 124.

If you use a value for *num-exp* that is already an integer (it has no numbers after its decimal place), you get the same number from INT. INT(111) is 111.

INT is useful when you want to eliminate unnecessary decimal places from numbers. For example, if you are calculating dollar amounts and you don't need or want to print past the second decimal place ($123.45 instead of $123.4532123). If you want to round a variable to two decimal places, use (adding .5 rounds the second decimal place):

```
DOLLARS=INT(DOLLARS*100+.5)/100
```

Since TI BASIC carries decimal places in its calculations, you can use INT to see if a value is evenly divisible by a number. To see if the value of the variable TEST is evenly divisible by 25, use:

```
IF (INT(TEST/25)*25)=(TEST/25)*25 THEN 500
```

If TEST is evenly divisible by 25 (as 25, 50, 150, etc.), your program will branch (GOTO) statement 500. If TEST isn't evenly divisible by 25 (as 12, 27, etc.), your program will execute the statement after the IF statement.

Common Errors:

```
STRING-NUMBER MISMATCH
```

You passed a string data argument as *num-exp*.

Example 1:

The program in Listing 4-52 uses INT to round a dollar amount to two decimal places and uses STR$ to write the value to the screen.

Try changing the program to round the division result (add 0.5 instead of just dividing by 100).

```
100  CALL CLEAR
110  PRINT "GIVE ME TWO NUMBERS":
     "  AND I'LL DIVIDE AND"
120  PRINT "  MULTIPLY THEM.": :
     "THEN I'LL MAKE THE RESULTS"
130  PRINT "  INTO A DOLLAR FORMAT."
140  PRINT : :
150  INPUT "ENTER YOUR TWO NUMBERS (0,0) TO END -> ":
     NUM1,NUM2
160  IF (NUM1=0)*(NUM2=0) THEN 220
170  DIV=INT(100*NUM1/NUM2)/100
180  MULT=INT(100*NUM1*NUM2)/100
190  PRINT :NUM1;"DIVIDED BY";NUM2;"IS";"$"&STR$(DIV)
200  PRINT :NUM1;"MULTIPLIED BY";NUM2;"IS";
     "$"&STR$(MULT)
210  GOTO 150
220  PRINT : :"BYE."
230  END
```

Listing 4-52. INT Example 1

Example 2:

The program in Listing 4-53 uses INT to see if a number is evenly divisible by 10 and 48.

Try changing the program to use numbers other than 10 and 48 and see what happens.

```
100  CALL CLEAR
110  PRINT "ENTER A NUMBER AND I'LL":
     "  TELL YOU WHETHER IT'S"
120  PRINT "  EVENLY DIVISIBLE BY":"  10 AND 48."
130  PRINT : :
140  INPUT "YOUR NUMBER (0 TO STOP) -> ":NUMIN
150  IF NUMIN<>0 THEN 180
160  PRINT : "BYE."
170  STOP
180  IF (INT(NUMIN/10)*10)=(NUMIN/10)*10 THEN 210
190  PRINT : NUMIN;"IS NOT";
200  GOTO 220
210  PRINT : NUMIN;"IS";
220  PRINT " EVENLY DIVISIBLE BY 10."
230  IF (INT(NUMIN/48)*48)=(NUMIN/48)*48 THEN 260
240  PRINT : NUMIN;"IS NOT";
250  GOTO 270
260  PRINT : NUMIN;"IS";
270  PRINT " EVENLY DIVISIBLE BY 48."
280  GOTO 130
290  END
```

Listing 4-53. INT Example 2

CALL JOYST Read the joystick.

Type:	Statement
Format:	[*line#*] CALL JOYST(*key-unit,x-return,y-return*)
Purpose:	CALL JOYST tells you the position of the joystick lever for the joystick *key-unit* (one or two). You get the status of the joystick fire buttons with KEY.
Operands:	*line#* is a BASIC statement line number that you need when you include CALL JOYST in a program. You don't need *line#* when you use CALL JOYST as a command. *line#* can be any number between 1 and 32767. *key-unit* is a number, numeric variable, or numeric expression that specifies which joystick to read. *key-unit* can be 1 or 2. *x-return* is a numeric variable whose value is set to −4, 0, or 4, depending on the x position of the joystick *key-unit*. Fig. 4-10 shows you the *x-return* and *y-return* values. *y-return* is a numeric variable whose value is set to a value of −4, 0, or 4, depending on the y position of the joystick *key-unit*. Fig. 4-10 shows you the *x-return* and *y-return* values.
Defaults:	None.

Description:

CALL JOYST puts the x and y positions of the lever for joystick *key-unit* into the two variables *x-return* (x position) and *y-return* (y position). *key-unit* can be either 1 (for joystick one) or 2 (for joystick two). You use KEY to get the status of the joystick FIRE buttons.

The values returned in *x-return* and *y-return* are −4, 0, or 4, depending on the position of the joystick. The center position (with the joystick lever centered) is 0,0. Fig. 4-10 shows you the values returned for each of the nine possible joystick lever positions.

CALL JOYST returns the position of the *key-unit* joystick lever at the time the CALL JOYST statement is executed. The x and y positions (as shown in Fig. 4-10) are put into your *x-return* and *y-return* variables so that you can use the values in your program.

NOTE

When you use the joysticks, you must have the ALPHA LOCK key unlocked (or up). When the ALPHA LOCK key is locked in the down position, you can't use the joysticks to move up (towards the top) on your screen.

Common Errors:

BAD VALUE

key-unit is less than 1 or greater than 4. Values 1 and 2 return the joystick position for joysticks 1 and 2. Values 3 and 4 are reserved for future use.

INCORRECT STATEMENT

Either *x-return* or *y-return* is not a valid numeric variable name. You cannot use a string variable for either *x-return* or *y-return*.

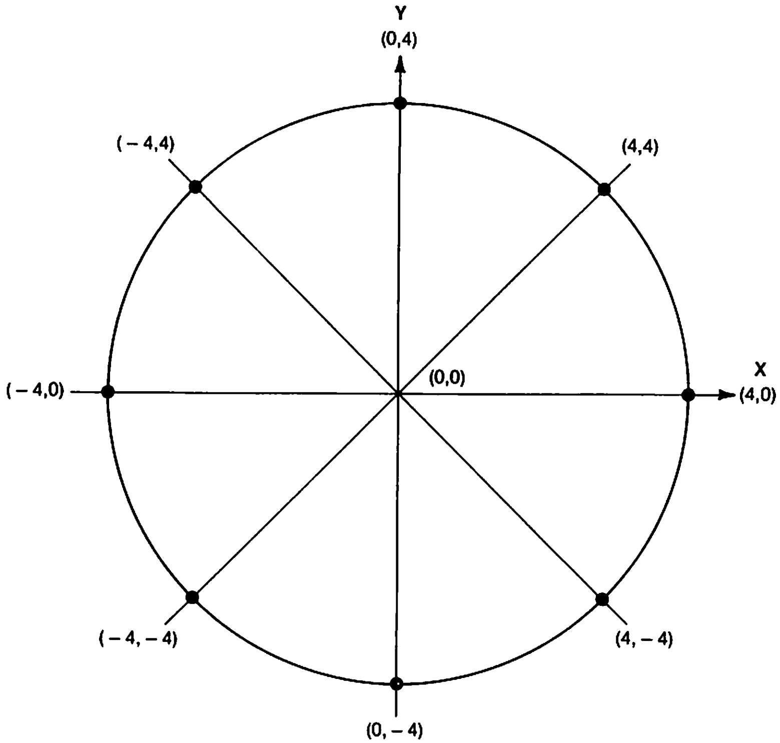


Fig. 4-10. JOYST x-return and y-return values.

Example 1:

The program in Listing 4-54 first reads the lever on joystick one and prints its x and y positions. Then it reads joystick two and prints its x and y positions. You can use this program to test the joysticks. If you don't get a correct reading when you hold the joysticks in a certain position, you will know why you are having problems playing some games.

After a pause (using the FOR loop) it reads the joysticks again. Ten positions for each joystick are printed.

Try changing the program by increasing or decreasing the FOR loop ending value to increase or decrease the pause between reading the joysticks. Or try adding a CALL KEY to get the status of the FIRE buttons.

Example 2:

The program in Listing 4-55 uses HCHAR to write a block at the center of the screen. You use joystick one to move the block around on the screen. The direction to move is determined by the values returned by JOYST.

```
100  CALL CLEAR
110  PRINT "I'LL TELL YOU THE X AND Y":
     "POSITIONS OF YOUR JOYSTICKS."
120  PRINT : :"FIRST, JOYSTICK 1":" THEN 2.": :
     "DON'T FORGET THE ALPHA LOCK!!!":
130  PRINT " JOYSTICK 1";TAB(15);"JOYSTICK 2"
140  PRINT TAB(4);"X";TAB(10);"Y";TAB(16);"X";
     TAB(23);"Y"
150  FOR I=1 TO 20
160  CALL JOYST(1,X1,Y1)
170  FOR L=1 TO 200
180  NEXT L
190  CALL JOYST(2,X2,Y2)
200  PRINT TAB(3);X1;TAB9;Y1;TAB(15);X2;TAB(22);Y2
210  FOR J=1 TO 50
220  NEXT J
230  NEXT I
240  PRINT :"BYE."
250  END
```

Listing 4-54. JOYST Example 1

```
100  CALL CLEAR
110  PRINT "USE THE JOYSTICK":
     " TO WRITE BLOCKS OF COLOR":
     "DON'T FORGET THE ALPHA LOCK!!"
120  PRINT : :"PRESS FCTN 4 TO STOP."
130  CALL COLOR(129,1,14)
140  CALL CHAR(129,"A5A5A5A5A5A5A5A5")
150  ROW=12
160  COL=16
170  CALL HCHAR(ROW,COL,129)
180  CALL JOYST(1,X,Y)
190  IF (X=0)*(Y=0) THEN 180
200  ON SGN(X)+2 GOTO 230,240,210
210  COL=COL+1
220  GOTO 240
230  COL=COL-1
240  ON SGN(Y)+2 GOTO 250,280,270
250  ROW=ROW+1
260  GOTO 280
270  ROW=ROW-1
280  IF (ROW>=1)*(ROW<=24) THEN 330
290  IF ROW<1 THEN 320
300  ROW=1
310  GOTO 330
320  ROW=24
330  IF (COL>=1)*(COL<=32) THEN 170
340  IF COL<1 THEN 370
350  COL=1
360  GOTO 170
370  COL=32
380  GOTO 170
390  END
```

Listing 4-55. JOYST Example 2

Try adding code to use both joysticks. Or change the character that's written by changing the pattern string in the CALL CHAR statement. Or use a CALL KEY to use the FIRE button status to stop the program. Or change the color of the character written.

CALL KEY	Read the keyboard.
Type:	Statement
Format:	[*line#*] CALL KEY (*key-unit,return-var,status-var*)
Purpose:	CALL KEY reads a character from the keyboard directly into a variable in your program. You can section the keyboard into right and left sides, read control and function key values, and read upper- and lower-case letters. CALL KEY can also read the status of the joystick FIRE buttons.
Operands:	*line#* is a BASIC statement line number that you need when you include CALL KEY in a program. You don't need *line#* when you use CALL KEY as a command. *line#* can be any number between 1 and 32767. *key-unit* is a number, numeric variable, or numeric expression that tells TI BASIC how you want to configure your keyboard. *key-unit* can be 0 to 5. *key-unit* 1 or 2 is used to read the joystick FIRE buttons. *return-var* is the name of a numeric variable into which KEY places the ASCII code of the character read (for *key-unit* 0, 3, 4, or 5) or a code between 0 and 19 (for *key-unit* 1 or 2). *status-var* is the name of a numeric variable into which KEY places an indicator (−1, 0, or +1) telling you whether a key was pressed and whether the same key was pressed on the last CALL KEY.
Defaults:	None.

Description:

CALL KEY reads the keyboard and puts the ASCII value of the key pressed into the variable you use for *return-var.* CALL KEY does not wait for you to press a key. You can tell if a key was pressed by the value CALL KEY puts into *status-var.*

CALL KEY reads the key pressed. It does not "echo" the key (write it on the screen). This means that you can get information from the keyboard without changing what's on the screen.

With INPUT, your program waits for data. You can, however, see on the screen what you are entering and the screen also scrolls upward. With CALL KEY, the program does not wait for you to press a key, nor do you see what you are entering on the screen and the screen does not scroll upward. If you want to display the characters entered, use HCHAR or VCHAR.

This combination (CALL KEY with HCHAR or VCHAR) lets you format the screen to whatever you want. You may, for instance, want to put messages out only on line 1 and keep the remainder of the screen active for a game.

Table 4-17 shows you the *key-unit* values. Table 4-18 shows you what CALL KEY returns in your *status-var.* Table 4-19 lists the ASCII codes returned in *return-var* for the numbers, punctuation, and upper- and lower-case letters.

Table 4-17. KEY *key-unit* Values

key-unit	Meaning
0	Console keyboard. Use the *key-unit* from the last CALL KEY.
1	Left side of the console keyboard or joystick one. (See Fig. 4-11)
2	Right side of the console keyboard or joystick two. (See Fig. 4-11)
3	Standard TI-99/4A console keyboard scan. Upper- and lower-case alphabetic characters returned as upper case. Function codes 1 through 15 returned. No control codes returned. (See Fig. 4-12)
4	Pascal console keyboard scan. Upper- and lower-case alphabetic characters returned. Function codes 129 through 143 returned. Control codes 1 through 31 returned.
5	BASIC console keyboard scan. Upper- and lower-case alphabetic characters returned. Function codes 1 through 15 returned. Control codes 128 through 159 and 187 returned. (See Fig. 4-13)

Table 4-18. KEY *status-var* Values

status-var	Meaning
1	A key was pressed since the last CALL KEY and the key is different from the last CALL KEY read.
0	No key was pressed.
−1	A key was pressed since the last CALL KEY and the key is the same as the last CALL KEY read.

Table 4-19. ASCII *return-var* Character Codes

ASCII Char	Decimal	ASCII Char	Decimal	ASCII Char	Decimal
(space)	32	@	64		96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
–	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115

Table 4-19. *(continued)*

ASCII Char	Decimal	ASCII Char	Decimal	ASCII Char	Decimal
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	|	124
=	61	]	93	}	125
>	62	^	94	~	126
?	63	–	95	(DEL)	127

You use the *key-unit* operand to tell TI BASIC whether you want to:

- Divide the keyboard into left and right sides (Fig. 4-11)
- Translate all lower-case letters to upper-case letters (Fig. 4-12)
- Read upper- and lower-case letters (Fig. 4-13)

Fig. 4-11 shows you the keyboard mapping for *key-unit* = 1 (left) and *key-unit* = 2 (right). The numeric values written on the lower right of each key shows you the values that CALL KEY returns. Table 4-20 lists the values that CALL KEY returns in your *return-var* when *key-unit* is 1 or 2. This mapping is used for many of the games where you can use either the keyboard or joysticks to play.

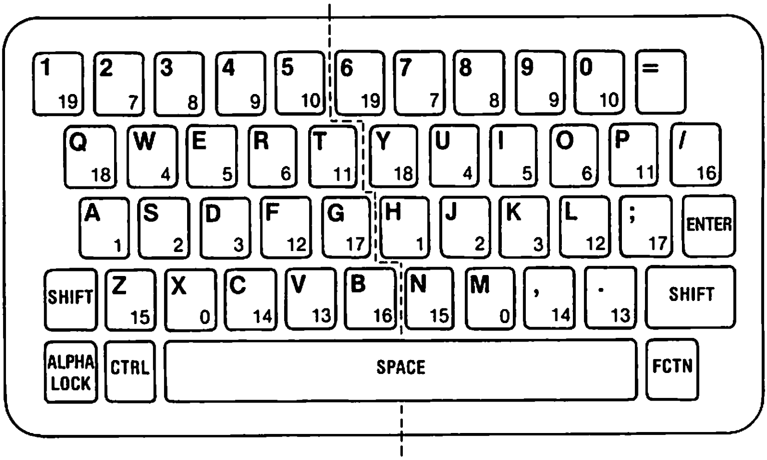


Fig. 4-11. KEY keyboard mapping (key-unit = 1 or 2).

Table 4-20. KEY Values for *key-unit* = 1 or 2

key-unit = 1		*key-unit=2*	
Key	*return-var* value	Key	*return-var* value
X	0	M	0
A	1	H	1
S	2	J	2
D	3	K	3
W	4	U	4
E	5	I	5
R	6	O	6
2	7	7	7
3	8	8	8
4	9	9	9
5	10	0	10
T	11	P	11
F	12	L	12
V	13	.	13
C	14	,	14
Z	15	N	15
B	16	/	16
G	17	;	17
Q	18	Y	18
1	19	6	19

Fig. 4-12 shows you the standard TI-99/4A keyboard mapping, where lower-case letters are automatically translated to their upper-case forms. The numbers in the lower right corner show you the values that CALL KEY returns for the FCTN keys.

You can read the FCTN keys (like FCTN 4 or FCTN CLEAR) but not the

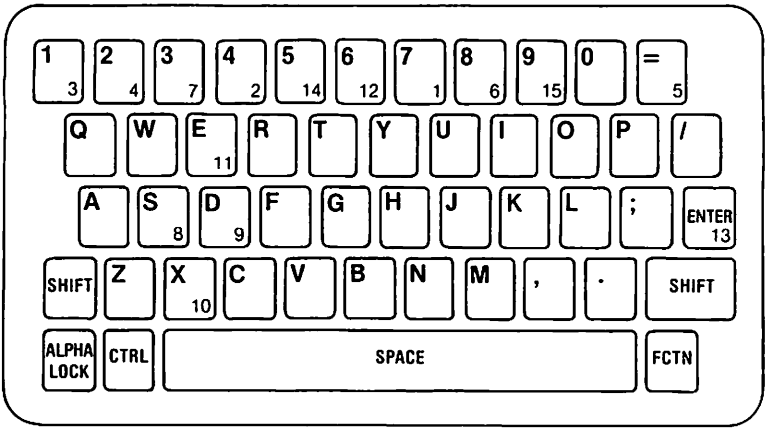


Fig. 4-12. KEY keyboard mapping (key-unit = 3).

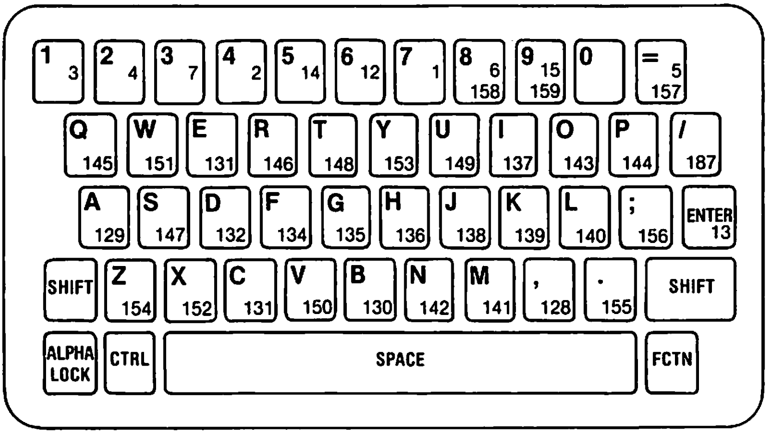


Fig. 4-13. KEY keyboard mapping (key-unit = 5).

CTRL keys when you use *key-unit* = 3. Table 4-21 lists the values for the FCTN keys that CALL KEY returns in your *return-var*. The ASCII values for the upper-case letters (Table 4-20) are also returned.

Table 4-21. KEY Function Key Values for *key-unit* = 3

Key	*return-var* value
FCTN 7	1
FCTN 4	2
FCTN 1	3
FCTN 2	4
FCTN =	5
FCTN 8	6
FCTN 4	7
FCTN S	8
FCTN D	9
FCTN X	10
FCTN E	11
FCTN 6	12
FCTN ENTER	13
FCTN 5	14
FCTN 9	15

Fig. 4-13 shows you the BASIC mode (*key-unit* = 5) where you can read both upper- and lower-case letters, punctuation, numbers, FCTN keys, and CTRL keys. The values in the lower right corners of the keys show you what CALL KEY returns for the CTRL keys. The values in the middle of the keys are CALL KEY values for the FCTN keys. Table 4-22 lists the values CALL KEY returns for the FCTN and CTRL keys.

You use CALL KEY to read the status of the FIRE button on the two joysticks. Table 4-23 lists the *key-unit* values for the joysticks. Table 4-24 lists the values CALL KEY returns in *return-var* when it reads the joysticks. Table 4-25 lists the CALL KEY *status-var* values when you read the joystick FIRE buttons.

Table 4-22. KEY Function and Control Key Values for *key-unit* = 5

Key	*return-var* value	Key	*return-var* value
FCTN 7	1	**CTRL H**	136
FCTN 4	2	**CTRL I**	137
FCTN 1	3	**CTRL J**	138
FCTN 2	4	**CTRL K**	139
FCTN =	5	**CTRL L**	140
FCTN 8	6	**CTRL M**	141
FCTN 4	7	**CTRL N**	142
FCTN S	8	**CTRL O**	143
FCTN D	9	**CTRL P**	144
FCTN X	10	**CTRL Q**	145
FCTN E	11	**CTRL R**	146
FCTN 6	12	**CTRL S**	147
FCTN ENTER	13	**CTRL T**	148
FCTN 5	14	**CTRL U**	149
FCTN 9	15	**CTRL V**	150
		CTRL W	151
CTRL ENTER	13	**CTRL X**	152
CTRL ,	128	**CTRL Y**	153
CTRL A	129	**CTRL Z**	154
CTRL B	130	**CTRL .**	155
CTRL C	131	**CTRL ;**	156
CTRL D	132	**CTRL =**	157
CTRL E	133	**CTRL 8**	158
CTRL F	134	**CTRL 9**	159
CTRL G	135	**CTRL /**	187

Table 4-23. KEY *key-unit* Values for the Joysticks

key-unit	Meaning
1	Read the fire button on joystick 1.
2	Read the fire button on joystick 2.

Table 4-24. KEY *return-var* Values for the Joysticks

return-var	Meaning
0	The fire button was not pressed.
18	The fire button was pressed.

Table 4-25. KEY *status-var* Values for the Joysticks

status-var	Meaning
−1	The fire button is pressed and was also pressed the last time you used CALL KEY for this joystick.
0	The fire button is not pressed when you CALL KEY for the joystick.
+1	The fire button is pressed now but was not pressed the last time you used CALL KEY for this joystick.

Common Errors:

BAD VALUE

You used a value for *key-unit* that is less than 0 or greater than 5.

INCORRECT STATEMENT

Status-var or *return-var* is not the name of a valid numeric variable. You cannot use a string variable.

Example 1:

The program in Listing 4-56 shows you how to use CALL KEY in a subprogram to make your computer pause. As soon as you press any key, your program continues.

```
100  CALL CLEAR
110  PRINT "I'LL SHOW YOU HOW":"  TO USE CALL KEY TO"
120  PRINT "  MAKE ME PAUSE.": :
130  GOSUB 160
140  PRINT : :"BYE."
150  STOP
160  REM USE KEY TO PAUSE
170  PRINT "PRESS ANY KEY TO CONTINUE";
180  CALL KEY(0,K,S)
190  IF S=0 THEN 180
200  RETURN
210  END
```

Listing 4-56. KEY Example 1

Example 2:

The program in Listing 4-57 shows you the different values you get when you use different *key-units*. CALL KEY is used five times, each with a different *key-unit*. Press the same key four times to see what CALL KEY would give you in *return-var* for the different *key-unit* values.

Notice that you do not see what key you pressed until it is written in the messages. CALL KEY reads the key but doesn't "echo" (write to the screen) its value. Change the program to echo the key you press by using HCHAR or VCHAR.

LEN	Get the length of a string.
Type:	Function
Format:	LEN (*str-exp*)
Purpose:	LEN tells you how many characters there are in the string *str-exp*.
Operands:	*str-exp* is a string, string variable, or string expression.
Defaults:	None.

```
100  CALL CLEAR
110  PRINT "I'LL SHOW YOU WHAT KEY":" RETURNS."
120  PRINT : :"FIRST, PRESS A KEY ON":
     "  THE LEFT SIDE (1)"
130  CALL KEY(1,V1,ST)
140  IF ST=0 THEN 130
150  PRINT :"YOU PRESSED ";" ASCII CODE";V1
160  PRINT : :"NOW, PRESS A KEY ON":
     "  THE RIGHT SIDE (2)"
170  CALL KEY(2,V2,ST)
180  IF ST=0 THEN 170
190  PRINT :"YOU PRESSED ";" ASCII CODE";V2
200  PRINT : :"PRESS ANY KEY (3)"
210  CALL KEY(3,V2,ST)
220  IF ST=0 THEN 210
230  PRINT :"YOU PRESSED ";" ASCII CODE";V3
240  PRINT : :"FINALLY, ANY KEY (5)"
250  CALL KEY(5,V5,ST)
260  IF ST=0 THEN 250
270  PRINT :"YOU PRESSED ";CHR$(V5);" ASCII CODE";V5
280  PRINT : : :
290  INPUT "TRY AGAIN? (Y/N) -> ":Y$
300  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 100
310  PRINT : : :"BYE."
320  END
```

Listing 4-57. KEY Example 2

Description:

The LEN function returns the number of characters in *str-exp*. The length of the null string is zero since the null string has no characters.

LEN is used when you want to do special formatting, or if you want to make sure that someone entered a string answer. You can use LEN to assign a value to a variable, like this:

100 STRLGTH = LEN(STRIN$)
150 INCHARS = INCHARS + LEN(ANS$)

Or, you can use LEN as part of an expression, like this:

150 FOR I = 1 TO LEN(ANS$)
150 LAST$ = SEG$(ANS$,LEN(FIRST$),255)
600 IF LEN(ANS$) = 0 THEN 500

This last example checks to see if the answer (ANS$) was a null string (only the **ENTER** key was pressed).

Common Errors:

STRING-NUMBER MISMATCH

You passed a numeric instead of a string argument to the LEN function.

Example 1:

The program in Listing 4-58 uses LEN to tell you how many characters there are in the string you enter.

Change the program to see how many characters you read for all the strings entered. (Hint: add the LEN to a counter for each string.)

```
100  CALL CLEAR
110  PRINT "ENTER A STRING AND":
     " I'LL TELL YOU HOW MANY"
120  PRINT "  CHARACTERS IT HAS IN IT."
130  PRINT : :"DON'T FORGET QUOTES IF YOU":
     "  NEED THEM!!"
140  PRINT : : :
150  INPUT "YOUR STRING -> ":ANS$
160  PRINT :"YOUR STRING HAD";LEN(ANS$);
     "CHARACTERS.": :
170  INPUT "TRY ANOTHER? (Y/N) -> ":Y$
180  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 140
190  PRINT : :"BYE."
200  END
```

Listing 4-58. LEN Example 1

Example 2:

The program in Listing 4-59 uses LEN to see if you entered an answer. If the length of the answer is zero, you pressed the ENTER key without entering any other characters.

```
100  CALL CLEAR
110  PRINT "ENTER A STRING OR":
     " JUST PRESS ENTER.": : :
120  INPUT "A STRING OR ENTER-> ":ANS$
130  IF LEN(ANS$)=0 THEN 160
140  PRINT :"YOU ENTERED";LEN(ANS$);" CHARACTERS.": :
150  STOP
160  PRINT : :"YOU ONLY PRESSED ENTER!!"
170  END
```

Listing 4-59. LEN Example 2

LET	Assign a value to a variable.
Type:	Statement or Command
Format:	[*line#*] *variable* = *expression* or [*line#*] LET *variable* = *expression*
Purpose:	LET places the value of *expression* into *variable*.
Operands:	*line#* is a BASIC statement line number that you need when you include LET in a program. You don't need *line#* when you use LET as a command. *line#* can be any number between 1 and 32767.

LET	Assign a value to a variable. *(continued)*
	variable is a valid variable name. String variable names must end with a dollar sign ($). Numeric variable cannot end with a $.
	expression is any valid expression, including simple constants (1.23 or “ABC”) and TI BASIC functions. You must have a string *expression* assigned to a string *variable* and a numeric *expression* for a numeric *variable*.
Defaults:	You can omit the keyword LET. TI BASIC supplies the keyword LET when you use an assignment statement.

Description:

LET evaluates the *expression* and assigns the value to *variable*. You must use a string *variable* (ending in $) with a string *expression* and a numeric *variable* with a numeric *expression*. Variables and expressions are discussed in detail in Chapter 2.

The LET statement is the simple assignment statement that you use quite often. The LET keyword is left over from past implementations of BASIC; you don't need it when you write assignment statements. If you do use it, note that it consumes an additional byte of memory.

These assignment statements mean the same thing to TI BASIC:

```
100  LET A$=“XYZ”
```

is the same as

```
100  A$=“XYZ”
```

```
250  LET SIDEC=SQR(SIDEA*SIDEA+SIDEB*SIDEB)
```

is the same as

```
250  SIDEC=SQR(SIDEA*SIDEA+SIDEB*SIDEB)
```

When you use a numeric assignment statement, TI BASIC checks the result to see if you generated an overflow (a number greater than 9.9999999999999E127) or an underflow (a number less than −9.9999999999999E−128). TI BASIC sets the result of an underflow to zero (0) and continues with the program. Overflows cause this warning to appear on your screen before the program continues:

```
NUMBER TOO BIG
```

TI BASIC has a maximum string length of 255 characters, the most characters you can put into a string *variable*. If your *expression* evaluates to more than 255 characters, TI BASIC truncates (ignores) all characters past the 255th and does not write any message telling you that the string is shortened.

LET and Relational Operators—You can also use the relational operators shown in Table 4-26 in your *expression*. A TRUE relation returns the value −1; a FALSE relation returns a zero.

Table 4-26. Relational Operators

Operator	Meaning
A=B	A is equal to B.
A>B	A is greater than B.
A<B	A is less than B.
A<>B	A is not equal to B.
A<=B	A is less than or equal to B.
A>=B	A is greater than or equal to B.

The relational operators can be the total *expression*, like this:

```
150   RESULT=ANS<>GUESS
```

Or, the relational operators can be part of a more complicated *expression*, like this:

```
250   ANS=(PRIOR<>CURRENT)*NEWVAL
```

Common Errors:

```
INCORRECT STATEMENT
```

You forgot the equals sign (=) in the assignment statement.

Example 1:

The program in Listing 4-60 shows you some simple assignment statements. Notice that the keyword LET is not used.

There are many other examples of assignment statements in the other programs in this book.

```
100  CALL CLEAR
110  INPUT "ENTER TWO NUMBERS (N,N) -> ":N1, N2
120  ANS=N1+N2
130  PRINT : :N1;"+";N2;"=";ANS
140  MSG$="GOODBYE."
150  PRINT : :MSG$
160  END
```

Listing 4-60. LET Example

LIST	List some program lines.
Type:	Command
Format:	LIST or LIST *start-line* – *end-line* or LIST *start-line* – or LIST –*end-line* or

LIST	List some program lines. *(continued)*
	LIST *line-num* or LIST "*device*" [: [*start-line*] [– [*end-line*]]]
Purpose:	LIST prints program lines beginning with *start-line* and ending with *end-line* to the screen or to the *device*.
Operands:	*line-num* is the line number of a statement in your BASIC program. When you use this form, TI BASIC lists only the statement at *line-num*. *start-line* is the line number of a statement in your BASIC program. This is the first statement that TI BASIC lists. *end-line* is the line number of a statement in your BASIC program. This is the last statement that TI BASIC lists. *device* is the name of a device attached to your system. TI BASIC will "list" your program to the device instead of the screen.
Defaults:	If you don't use a *device*, TI BASIC lists your program lines to the screen. If you don't use a *start-line*, TI BASIC begins with the first line in the program. If you don't use an *end-line*, TI BASIC lists to the last line in the program.

Description:

LIST prints lines from the BASIC program in memory, beginning with *start-line* and ending with *end-line*.

You may notice that your program listing looks slightly different from what you entered. When TI BASIC lists a statement, it removes all unnecessary blanks. Blanks (space characters) in strings (within double quotes) are not affected.

LIST has a number of different operand combinations. Which ones you use determines which line or lines are listed. You don't need to use any line numbers, like this:

LIST (list the entire program)

The *start-line* and *end-line* operands let you list only a selected range of program lines.

LIST 350 – 450 (list lines 350 to 450)

If you don't use *start-line*, the first line in the program is the first line listed. Notice that a hyphen (-) is used before the *end-line* operand.

LIST – 200 (list all lines to line 200)

If you don't use *end-line*, the last line in the program is the last line listed. Notice the hyphen (-) after the *start-line* operand.

LIST 500 – (list all lines beginning at line 500)

If you use a single line number *(line-num)* without any hyphen, TI BASIC will list only the line with that line number. There is a difference between the following two LIST statements and the ones above. Each of these lists only one statement from the program; the previous LIST statements listed ranges or groups of lines.

LIST 200 (list only line 200)
LIST 500 (list only line 500)

LIST follows these rules if the line numbers you specify in your operands are not line numbers of statements in your program:

1. If you use a line number larger than any line number in your program, TI BASIC uses the largest line number in your program.
2. If you use a line number lower than any line number in your program, TI BASIC uses the smallest line number in your program.
3. If you use a line number between line numbers in your program, TI BASIC uses the line number of the next higher statement in your program.
4. If you use a line number less than or equal to 0 or greater than 32767, TI BASIC writes the error message:

BAD LINE NUMBER

LISTing to a File or Peripheral—You can LIST a program to a device other than the screen by using a *device* operand before your line numbers.

A *device* can be as simple as the name of an RS232 port ("RS232") or as complex as a disk identifier and filename ("DSK2.MYLIST"). Commonly used *devices* are shown in Table 4-27.

Table 4-27. LIST *device* Values

device	Meaning
none	List to the screen.
RS232	List to the RS232 interface serial port (when you use a single serial port).
RS232/1	List to the RS232 interface serial port 1 (when you use a Y-connector to get two serial ports).
RS232/2	List to the RS232 interface serial port 2 (when you use a Y-connector to get two serial ports).
PIO	List to the RS232 parallel port.
DSK1.PGMLIST	List to the file "PGMLIST" on disk 1.

NOTE: You cannot list to a cassette file (CS1 or CS2).

NOTE

You can LIST a program to a file on a disk. You CANNOT LIST to a file on a cassette tape.

To LIST the entire program, don't specify any *line-num, start-line,* or *end-line* operands. You must use double quotes (") around the *device* operand.

LIST "RS232"
(list entire program to device attached to RS232 serial port)
LIST "DSK1.PGMLST"
(list entire program to file PGMLST on disk 1)

To LIST selected lines, specify *line-num*, or *start-line* and *end-line* operands. This format differs from the previous one. Not only is the *device* enclosed in double quotes (") but you also need to use a colon (:) before the line numbers.

LIST "RS232":-500 (list to line 500 on the RS232 device)
LIST "DSK2.NEWLST":100-900
(list lines 100 to 900 to file NEWLST on disk 2)

If you have another peripheral, such as a Hexbus peripheral, check the manual you get with the device to see if you can LIST to it.

Common Errors:

BAD LINE NUMBER

You used a *line-num*, *start-line*, or *end-line* that is 0, less than 0, or greater than 32767.

CAN'T DO THAT

You tried to use LIST as a statement in a program. You can use LIST only as a command.
Or, you entered a LIST command and you don't have any program in memory yet.

INCORRECT STATEMENT

You have a character other than a hyphen (-) between the *start-line* and *end-line* values.
Or, you used a *line-num*, *start-line*, or *end-line* which is not an integer (has decimal places—as 12.34).

Example 1:

The example in Listing 4-61 shows you how to enter a small BASIC program (5 lines) and then list it in several ways.
<ENTER> means press the ENTER key.

```
NEW <ENTER>
NUM <ENTER>
100  A=5
110  B=25
120  PRINT A,B
130  CALL SCREEN(14)
140  PRINT "BYE"
150  <ENTER>
LIST 120 <ENTER>            (List only line 120.)
120  PRINT A,B
LIST -120                   (List up to and including
100  A=5                     line 10.)
110  B=25
120  PRINT A,B
LIST 140- <ENTER>           (List from line 140 to the
140  PRINT "BYE"             end of the program.)
LIST 100-110                (List line 100 to line 110.)
100  A=5
110  B=25
```

Listing 4-61. LIST Example 1

Example 2:

The example in Listing 4-62 first reads a program into memory from a cassette tape and then uses LIST to write a listing to the printer. The printer is attached to an RS232 interface card.

```
OLD CS1 <ENTER>
LIST "RS232" <ENTER>
          The program is listed at the
          printer attached to the RS232
          interface.
```

Listing 4-62. LIST Example 2

LOG	Get the natural logarithm.
Type:	Function
Format:	LOG(*num-exp*)
Purpose:	LOG returns the natural logarithm of *num-exp* or LOG_e(*num-exp*).
Operands:	*num-exp* is a number, numeric variable, or numeric expression. *num-exp* must be greater than zero.
Defaults:	None.

Description:

LOG returns the natural logarithm of *num-exp* which is:

$$\log_e(\textit{num-exp})$$

The LOG function assigns a value to a variable, like this:

```
100   ANS=LOG(X+Y)
```

Or, LOG can be part of a longer numeric expression, like this:

```
100  PRINT ANS + LOG(X*Y*Z)
500  RESULT = 500*SQR(Z^3) + LOG(Z*FACTOR)
```

If you want the logarithm (log) of a number in another base, as 10, you can use the LOG function this way

$$\log_{10}(X) = \mathrm{LOG}(X)/\mathrm{LOG}(10)$$

where LOG is the TI BASIC LOG function and log represents the logarithm in another base (10 in this example). You could code this as a user-defined function like this:

```
DEF LOG10(X) = LOG(X) / LOG(10)
```

Common Errors:

```
BAD ARGUMENT
```

You used a negative or zero value for *num-exp*.

Example 1:

The program in Listing 4-63 asks you for a number and then prints the LOG of it.

Try printing the LOG in another base.

```
100  CALL CLEAR
110  PRINT "ENTER A NUMBER AND I'LL":
     "  TELL YOU ITS LOG."
120  PRINT : : :
130  INPUT "YOUR NUMBER -> ":ANS
140  IF ANS>0 THEN 170
150  PRINT : :"ONLY NUMBERS LARGER THAN":"  ZERO!!"
160  GOTO 120
170  PRINT : :"THE LOG OF";ANS;"IS";LOG(ANS): :
180  INPUT "TRY AGAIN? (Y/N) -> ":Y$
190  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 130
200  PRINT : :"BYE."
210  END
```

Listing 4-63. LOG Example 1

Example 2:

The program in Listing 4-64 asks you for a number and prints its LOG and its logarithm in base 10.

Try changing the program to print the logarithm in other bases.

```
100  CALL CLEAR
110  PRINT "ENTER A NUMBER AND I'LL":
     "  TELL YOU ITS LOG IN"
120  PRINT "  BASE E AND BASE 10."
130  PRINT : : :
140  INPUT "YOUR NUMBER -> ":ANS
150  IF ANS>0 THEN 180
160  PRINT : :"ONLY NUMBERS LARGER THAN":"  ZERO!!"
170  GOTO 120
180  PRINT : :"THE LOG OF";ANS;"IS";LOG(ANS)
190  PRINT "AND THE LOG IN BASE 10 IS";LOG(ANS)/LOG(10)
200  INPUT "TRY AGAIN? (Y/N) -> ":Y$
210  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 130
220  PRINT : :"BYE."
230  END
```

Listing 4-64. LOG Example 2

NEW	Erase memory, reset BASIC.
Type:	Command
Format:	NEW
Purpose:	NEW erases the TI BASIC program currently in your computer's memory and resets memory.
Operands:	None.
Defaults:	None.

Description:

NEW erases the TI BASIC program currently in memory and:

- Cancels all TRACE commands
- Cancels all BREAK commands
- Erases all variables
- Erases all entries in the variable name table
- Closes any OPENed files
- Releases any space allocated for special characters (through CHAR)
- Clears the screen

If you are working on a program make sure that you SAVE it before you enter a NEW command. A NEW command erases the program in memory. You cannot recover the program after a NEW command without re-entering the entire program (unless you have it stored on a tape or disk).

Common Errors:

CAN'T DO THAT

You tried to use NEW as a variable name or as a statement in a program. NEW can be used only as a command.

Example:

The example in Listing 4-65 shows you how the NEW command works. First, you enter a small program and LIST and RUN it. Then, you enter a NEW command and the program gets erased.

<ENTER> means press the **ENTER** key.

```
NEW <ENTER>
     The screen clears.
NUM <ENTER>
100  CALL CLEAR
110  FOR I=1 TO 3
120  PRINT TAB(5);"HELLO."
130  NEXT I
140  END
150  <ENTER>
LIST <ENTER>
     Your computer lists lines 100 to 140.
RUN <ENTER>
     Your computer prints:
     HELLO
     HELLO
     HELLO
NEW<ENTER>
     The screen clears.
LIST <ENTER>
     CAN'T DO THAT
     (You don't have any program left in memory.)
```

Listing 4-65. NEW Example

NEXT	Mark the end of a FOR loop.
Type:	Statement
Format:	*line#* NEXT *control*
Purpose:	NEXT marks the end of a FOR loop.
Operands:	*line#* is a BASIC statement line number that you need when you include NEXT in a program. *line#* can be any number between 1 and 32767. *control* is the name of the control variable used in the FOR statement associated with this NEXT statement. The FOR statement marks the beginning of the loop and the NEXT statement marks the end of the loop.
Defaults:	None.

Description:

NEXT is associated with a FOR . . . TO . . . STEP statement and marks the end of the FOR loop. A FOR loop is the group of statements between the FOR and NEXT statements that get executed until the *control* variable is greater than *end-val*.

NOTE

TI BASIC checks the *end-val* and *control* variables *before* it executes any statements in the loop.

If *end-val* starts out higher than *start-val,* no statements in the FOR loop are executed. For example, the statements in this loop will never be executed:

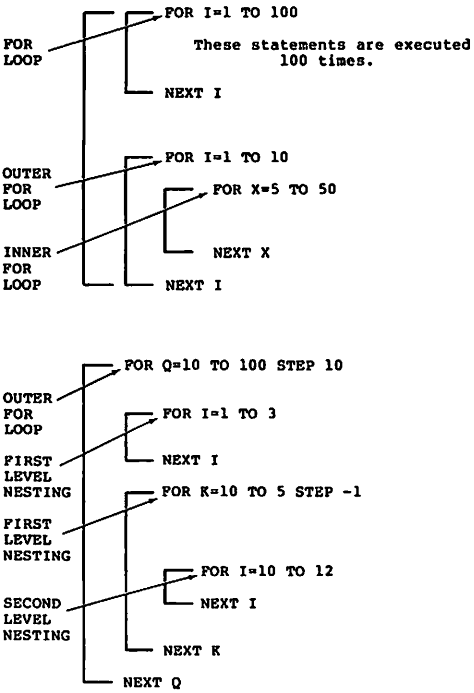
FOR I = 100 TO 1 STEP 10

When TI BASIC executes a NEXT statement, it branches to the associated FOR statement. The FOR statement increments the *control* variable by the amount specified in STEP operand (or 1 if your FOR statement doesn't use a STEP) and takes one of these actions:

- If *control* is less than or equal to *end-val* then the statements between the FOR and NEXT are executed.
- If *control* is greater than *end-val* then the statement *after* the NEXT statement is executed.

Nested FOR Loops—Since each NEXT statement is associated with a FOR statement, you can "nest" your loops by having an entire loop enclosed in another loop.

Fig. 4-14 shows you some examples of nested FOR loops. Example 2, below, uses nested FOR loops.



```
FOR I=1 TO 100
    These statements are executed
           100 times.
NEXT I

FOR I=1 TO 10
    FOR X=5 TO 50

    NEXT X
NEXT I

FOR Q=10 TO 100 STEP 10
    FOR I=1 TO 3
    NEXT I
    FOR K=10 TO 5 STEP -1

        FOR I=10 TO 12
        NEXT I
    NEXT K
NEXT Q
```

Fig. 4-14. NEXT statements and FOR loops.

CAUTION
When you "nest" your FOR loops, you must have the inner FOR . . . NEXT loop completely inside the outer loop.

Common Errors:

CAN'T DO THAT

You tried to use NEXT as a command. NEXT must be used as a statement in a program.

FOR NEXT ERROR

You have mismatched FOR and NEXT statements; the *control* operands do not match.

INCORRECT STATEMENT

The *control* operand is missing in a NEXT statement.

Example 1:

The program in Listing 4-66 uses NEXT to close a simple FOR loop.
Try changing the limits of the loop or adding more statements inside the FOR loop.

```
100  CALL CLEAR
110  PRINT "HELLO"
120  PRINT : :"I'LL COUNT WITH A FOR LOOP.": : :
130  FOR I=10 TO 100 STEP 10
140  PRINT I
150  NEXT I
160  PRINT : : :"BYE."
170  END
```

Listing 4-66. NEXT Example 1

Example 2:

The program in Listing 4-67 shows you how to use NEXT statements with nested FOR loops. Notice that the inner loop must be entirely contained in the outer loop. (The NEXT statement for the inner FOR statement must be before the NEXT statement for the outer FOR statement.)
Try adding more loops. Maybe one inside the inner loop and another inside the outer loop. Make sure that your nesting is correct.

```
100  CALL CLEAR
110  PRINT "I'M USING NESTED LOOPS.": :
120  PRINT "OUT IS THE OUTER LOOP":
     "IN IS THE INNER LOOP."
130  FOR OUT=1 TO 3
140  PRINT "OUT IS";OUT
150  FOR IN=5 TO 7
160  PRINT "IN IS";IN
170  NEXT IN
180  PRINT "OUT OF INNER LOOP"
190  NEXT OUT
200  PRINT : : "DONE"
210  END
```

Listing 4-67. NEXT Example 2

NUMBER or NUM	Provide program line numbers.
Type:	Command
Format:	NUMBER or NUM or NUMBER *start-line* or NUM *start-line,incr*
Purpose:	NUMBER or NUM automatically provides line numbers for the TI BASIC program that you are entering.
Operands:	*start-line* is the first line number displayed for the program. *start-line* can be any valid line number between 1 and 32767. *incr* is the increment added to each successive line number. *incr* can be any value greater than zero.
Defaults:	TI BASIC uses 100 for *start-line* and 10 for *incr.*

Description:

NUMBER or NUM generates sequenced line numbers for entering a BASIC program. *start-line* is 100 and *incr* is 10 if you don't specify values for these operands. Chapter 3 contains a discussion of methods to use in entering and editing TI BASIC programs.

NUM is very useful when you are entering programs. You do not have to worry about remembering to enter line numbers. TI BASIC automatically prints the line numbers for you when you say NUM. When you press ENTER without entering a statement on the line, TI BASIC gets out of its automatic numbering mode.

Remember, TI BASIC will execute some statements as soon as you enter them. Other statements will get an error if you do not use a line number to say that the statement is part of a program.

If you want your program to begin at line 100 and have each line number incremented by 10 (the line numbers are 110, 120, 130, etc.), use:

Other forms of NUM let you start at a specific line number and even adjust the increment. Suppose you have a long program (about 100 lines) that you want to enter from a book. You, of course, are careful and SAVE your program from time to time (just to make sure you don't lose the whole thing).

The line numbers begin at 100 and increase by 10. No problem. You start by entering:

```
NUM
```

You enter 25 lines, ending with:

```
340  X=SQR(Z*FACTR)
350  <ENTER>
SAVE CS1
```

And you save what you have entered to the cassette tape on your recorder. Now, you want to continue entering the program, beginning at line 350. It's easy. Use:

```
NUM 350
```

The line numbers begin at 350, get 10 added each time you press **ENTER** and you can continue until you finish entering the program or SAVE it again.

Sometimes you might want to use a different *incr.* Suppose you entered a program and you have to insert four lines between line 200 and 210. Instead of entering the line numbers along with the statements, try:

```
NUM 200,2
```

Some programs are written in modules where sections begin at convenient line numbers, maybe 1000, 2000, 3000, etc. To enter a program with this line numbering scheme, use a NUM command for each section.

Common Errors:

```
CANT DO THAT
```

You tried to use NUMBER or NUM as a statement in a program. You can use NUMBER or NUM only as a command.

Or, you tried to use NUMBER or NUM as a variable name in a program statement.

```
INCORRECT STATEMENT
```

You used a character other than a comma (,) between the *start-line* and *incr* operands.

Example 1:

The example in Listing 4-68 shows you how to use NUM to automatically number the lines in your program. Try using different values for *start-line* and *incr.*

<ENTER> means press the **ENTER** key

```
NEW <ENTER>
NUM <ENTER>
100  MSG$="HELLO"
110  PRINT MSG$
120  <ENTER>
LIST <ENTER>
     You'll see the program listed like this.
100  MSG$="HELLO"
110  PRINT MSG$
RUN <ENTER>
     Your computer prints
HELLO
     Now, add some lines to the program like this:
NUM 200,25 <ENTER>
200  MSG$="BYE"
225  PRINT MSG$
250  <ENTER>
LIST <ENTER>
     The program has four lines now.
100  MSG$="HELLO"
110  PRINT MSG$
200  MSG$="BYE"
225  PRINT MSG$
RUN <ENTER>
HELLO
BYE
```

Listing 4-68. NUM Example 1

Example 2:

The example in Listing 4-69 uses NUM to start the line numbers at 5000 and increments them by 10 (since there's no *incr* operand).

```
NEW <ENTER>
NUM 5000 <ENTER>
5000 PRINT "HI"
5010 PRINT "THERE"
5020 <ENTER>
RUN <ENTER>
     HI
     THERE
```

Listing 4-69. NUM Example 2

OLD	Read a program into memory.
Type:	Command
Format:	OLD *device* or OLD *device.program-name*
Purpose:	OLD reads a program from a storage medium (such as a cassette tape or disk) into memory.
Operands:	*device* is the name of a device attached to your computer. *program-name* is the name of a file containing a program stored on the *device*. You do not use a *program-name* for programs stored on cassette tape.
Defaults:	None.

Description:

OLD loads the BASIC program called *program-name* on device *device* into memory. After the program is in memory, you LIST, RUN, EDIT, or SAVE it.

Device is a storage device that holds information that can be read by your computer, such as, a cassette recorder, a disk drive, or a Hexbus peripheral, as shown in Table 4-28. You cannot use a peripheral like a printer because you cannot store a program on a printer in a way that it can be read by your computer.

Table 4-28. OLD *device* Values

device	Meaning
CS1	Cassette recorder 1
DSK1	Disk drive 1
DSK2	Disk drive 2
DSK3	Disk drive 3
HEXBUS1	Hexbus peripheral 1

If you want to load a program from a cassette tape on cassette recorder 1, use:

```
OLD CS1
```

Your computer will tell you when and how to operate the recorder, such as, rewinding the tape, starting the recorder, and stopping the recorder. You do not use a *program-name* for programs stored on cassette tape.

To load the program called "NEWPGM" that is on the diskette in disk drive 1, use:

```
OLD DSK1.NEWPGM
```

Notice that you need a *program-name* operand for programs loaded from a disk. That is because you must name the files that you store on your disks.

Common Errors:

CAN'T DO THAT

You tried to use OLD as a statement in a program. You can use OLD only as a command.

Or, you tried to use OLD as a variable name in a program statement.

INCORRECT STATEMENT

program-name is not valid.

I/O ERROR 50

You used a *device* that is not the name of a device on your system. Check the spelling for *device*.

I/O ERROR 56

There is a device error. Your device may be disconnected or not functioning properly.

I/O ERROR 57

There is a file error. Check the spelling for your *program-name*.

Example 1:

The example in Listing 4-70 reads a program from a cassette tape.

```
TI 99/4A BASIC REF
```

Listing 4-70. OLD Example 1

Example 2:

The example in Listing 4-71 reads a program in the file called "TESTPGM" that is on the diskette in disk drive 1.

```
TI 99/4A BASIC REF
```

Listing 4-71. OLD Example 2

ON . . . GOSUB	Call selected subprogram.
Type:	Statement
Format:	*line#* ON *num-exp* GOSUB *line-num-list*
Purpose:	ON . . . GOSUB evaluates *num-exp*, rounds the value to the nearest integer, and uses the result to determine which subprogram (from *line-num-list*) to transfer control to.
Operands:	*line#* is a BASIC statement line number that you need when you include ON . . . GOSUB in a program. *line#* can be any number between 1 and 32767.

ON . . . GOSUB	Call selected subprogram. *(continued)*
	num-exp is a numeric variable or numeric expression that TI BASIC evaluates and rounds to the nearest integer. This result is used as an index into the subprograms in *line-num-list*. *line-num-list* is a list of line numbers, separated by commas, of subprograms in your program. When *num-exp* is one, TI BASIC calls the subprogram at the first line number in *line-num-list;* when it's two, the second, etc.
Defaults:	None.

Description:

ON . . . GOSUB calls one of the subprograms in *line-num-list* based on the value of *num-exp*. TI BASIC first evaluates *num-exp* and rounds the result to the nearest integer. If the fractional part of *num-exp* is less than 0.5, it is rounded down; if 0.5 or greater, it is rounded up. This integer is used as an index into the list of subprogram line numbers.

For example, suppose you have subprograms at lines 1000, 2000, 3000, and 4000. You can use ON . . . GOSUB to call one of these subprograms based on the value in the variable ANS, like this:

```
100  ON ANS GOSUB 1000,2000,3000,4000
```

Depending on the value of ANS, TI BASIC calls one of the four subprograms:

- When ANS = 1, TI BASIC calls the subprogram at line 1000
- When ANS = 2, TI BASIC calls the subprogram at line 2000
- When ANS = 3, TI BASIC calls the subprogram at line 3000
- When ANS = 4, TI BASIC calls the subprogram at line 4000

You will notice that there are only four line numbers in the *line-num-list*. If you had a value greater than four or less than one in your ANS variable, you would get this error and your program would stop:

```
BAD VALUE
```

You should check that your *num-exp* value is not out of range for your subprogram list. You could use code like this:

```
100  IF (ANS<1)+(ANS>4) THEN 999
110  ON ANS GOSUB 1000, 2000, 3000, 4000
```

TI BASIC will check the value of ANS in statement 100. If ANS is out of range (less than one or greater than four), TI BASIC branches to statement 999 (where you process the out of range value).

After executing a RETURN statement in the called subprogram, TI BASIC returns to the statement after the ON . . . GOSUB and continues executing your program with that statement. You should not exit from a subprogram except through a RETURN statement.

Common Errors:

BAD VALUE

num-exp is less than 1 or larger than the number of line numbers in your statement.

BAD LINE NUMBER

One or more of the line numbers in *line-num-list* is not a line number of a statement in your program.

CAN'T DO THAT

You tried to use ON . . . GOSUB as a command. You can use ON . . . GOSUB only as a statement in a program.

INCORRECT STATEMENTS

The ON keyword is not followed by a valid numeric variable or numeric expression (*num-exp*).

MEMORY FULL

One or more of the *line-num-list* values is the line number of the ON . . . GOSUB statement. (The ON . . . GOSUB calls itself.)

Example 1:

The program in Listing 4-72 uses an ON . . . GOSUB statement to call one of three subprograms, depending on the value you enter. Notice that the program checks to see that you entered a value between 1 and 3 since there are only three subprograms in the ON . . . GOSUB list.

Try changing the program to do other processing in the subprograms. Or add more subprograms. This example shows you how to get to a specific subprogram. In your own programs, of course, the subprograms do something.

```
100  CALL CLEAR
110  PRINT "HELLO.  I'LL BRANCH TO ONE":
     " OF 3 SUBPROGRAMS."
120  PRINT : : :
130  INPUT "ENTER A NUMBER (1-3) ->":ANS
140  IF (ANS<1)+(ANS>3) THEN 220
150  PRINT : :"CALLING A SUBPROGRAM"
160  ON ANS GOSUB 240,270,300
170  PRINT :"NOW, I'M BACK.": :
180  INPUT "TRY AGAIN? (Y/N) -> ":Y$
190  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
200  PRINT : : :"BYE."
210  STOP
220  PRINT : : "I CAN ONLY CALL 3":" SUBPROGRAMS"
230  GOTO 120
240  REM SUBPROGRAM ONE
```

```
250  PRINT : : "HI THERE.":"I'M SUBPROGRAM ONE."
260  RETURN
270  REM SUBPROGRAM TWO
280  PRINT : : "HI THERE.":"I'M SUBPROGRAM TWO."
290  RETURN
300  REM SUBPROGRAM THREE
310  PRINT : : "HI THERE.":"I'M SUBPROGRAM THREE."
320  RETURN
330  END
```

Listing 4-72. ON...GOSUB Example 1

Example 2:

The program in Listing 4-73 prints a menu and then uses an ON . . . GOSUB to call one of the three subprograms.

Try adding more options to the menu and adding more subprograms. Remember to change the IF statement that checks the range. Or change what's done in the subprograms. Maybe round the answer to two decimal places.

```
100  CALL CLEAR
110  GOSUB 1000
120  PRINT : :"HERE ARE YOUR CHOICES.":
     " 1 ADD THE TWO NUMBERS"
130  PRINT " 2 SUBTRACT FIRST FROM":"  SECOND"
140  PRINT " 3 MULTIPLY THE TWO":
     " 4 DIVIDE FIRST BY SECOND"
150  PRINT " 5 ENTER NEW NUMBERS":" 6 STOP": :
160  INPUT "YOUR CHOICE (1-6) -> ":CHOICE
170  IF (CHOICE>=1)*(CHOICE<=6) THEN 200
180  PRINT : : "PLEASE PICK A CHOICE BETWEEN 1 AND 6"
190  GOTO 120
200  IF CHOICE<6 THEN 230
210  PRINT :"BYE"
220  STOP
230  ON CHOICE GOSUB 1500,2000,2500,3000,1000
240  GOTO 120
1000 REM GET THE TWO NUMBERS
1010 INPUT "ENTER TWO NUMBERS (X,X) -> ":NUM1, NUM2
1020 RETURN
1500 REM ADD THE TWO
1510 PRINT : :NUM1;"+";NUM2;"=";NUM1+NUM2
1520 RETURN
2000 REM SUBTRACT ONE FROM TWO
2010 PRINT : :NUM2;"-";NUM1;"=";NUM2-NUM1
2020 RETURN
2500 REM MULTIPLY THE TWO
2510 PRINT : :NUM1;"*";NUM2;"=";NUM1*NUM2
2520 RETURN
3000 REM DIVIDE ONE BY TWO
3100 PRINT : :NUM1;"/";NUM2;"=";NUM1/NUM2
3120 RETURN
3130 END
```

Listing 4-73. ON...GOSUB Example 2

ON . . . GOTO	Branch to selected statement.
Type:	Statement
Format:	*line#* ON *num-exp* GOTO *line-num-list*
Purpose:	ON . . . GOTO evaluates *num-exp*, rounds the value to the nearest integer, and uses the result to determine which statement (*line-num*) to branch to.
Operands:	*line#* is a BASIC statement line number that you need when you include ON . . . GOTO in a program. *line#* can be any number between 1 and 32767. *num-exp* is a numeric variable or numeric expression that TI BASIC evaluates and rounds to the nearest integer. This result is used as an index into the line numbers in *line-num-list*. *line-num-list* is a list of line numbers of statements in your program. When *num-exp* is one, TI BASIC branches to the first line number; when it's two, the second, etc.
Defaults:	None.

Description:

ON . . . GOTO unconditionally transfers control to one of the statements in *line-num-list*. TI BASIC starts executing your program at the statement to which your ON . . . GOTO branched. Unlike ON . . . GOSUB, TI BASIC does not return to your ON . . . GOTO statement.

When TI BASIC reaches an ON . . . GOTO statement, it first evaluates the *num-exp* and then rounds the result to the nearest integer. This integer result is used as an index into the line numbers in *line-num-list*.

Suppose you have an ON . . . GOTO statement that looks like this:

```
100   ON ANS GOTO 200,300,400
```

When TI BASIC gets to line 100 (your ON . . . GOTO statement), it rounds the value in the variable ANS. Then, TI BASIC branches to one of the three statements, like this:

- When ANS = 1, TI BASIC branches to statement 200
- When ANS = 2, TI BASIC branches to statement 300
- When ANS = 3, TI BASIC branches to statement 400

If ANS is less than one or greater than three, you will see this error and your program will stop:

```
BAD VALUE
```

If the line number that you branch to is not a valid line number for a statement in your program, you will see this error and your program will stop:

```
BAD LINE NUMBER
```

You can see that it is a good practice to check the range of your *num-exp* before you execute your ON . . . GOTO statement. Using the above example, you could code:

```
100  IF (ANS<1)+(ANS>3) THEN 999
110  ON ANS GOSUB 200,300,400
```

If your ANS variable is out of range (less than one or greater than three), TI BASIC branches to line 999 where you take care of the error.

Or maybe you don't really have an error if *num-exp* is out of range. Then, you could use code like this (where your program continues at line 120):

```
100  IF (ANS<1)+(ANS>3) THEN 120
110  ON ANS GOSUB 200,300,400
120  REM
```

Common Errors:

BAD VALUE

Num-exp is less than 1 or larger than the number of line numbers in your statement.

CAN'T DO THAT

You tried to use ON . . . GOTO as a command. You can use ON . . . GOTO only as a statement in a program.

BAD LINE NUMBER

One or more of the line numbers in your *line-num-list* is not a line number of a statement in your program.

INCORRECT STATEMENTS

The ON keyword is not followed by a valid numeric variable or numeric expression (*num-exp*).

Example 1:

The program in Listing 4-74 uses an ON . . . GOTO to print the results of an addition. Notice that the *num-exp* in this statement is a lengthy numeric expression.

Change the program to write the sum of the numbers as well as the message.

```
100   CALL CLEAR
110   INPUT "ENTER TWO NUMBERS (X,X) -> ":VAR1, VAR2
120   ON SGN(VAR1+VAR2)+2 GOTO 170,190,210
130   INPUT "TRY AGAIN (Y/N) -> ":Y$
140   IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 110
150   PRINT : :"BYE"
160   STOP
170   PRINT: :"THE SUM OF ";VAR1;"AND";VAR2;
      "IS NEGATIVE."
180   GOTO 130
190   PRINT: :"THE SUM OF ";VAR1;"AND";VAR2;"IS ZERO."
200   GOTO 130
210   PRINT: :"THE SUM OF ";VAR1;"AND";VAR2;
      "IS POSITIVE."
220   GOTO 130
230   END
```

Listing 4-74. ON...GOTO Example 1

Example 2:

The program in Listing 4-75 uses an ON . . . GOTO statement in solving another version of guess a number. The computer gets a random number and you enter your guess. The SGN function tells whether a value is negative, zero, or positive.

Try changing the program to get a number between 1 and 100.

```
100   CALL CLEAR
110   PRINT "I HAVE A NUMBER BETWEEN":"  1 AND 100." : : :
120   RANDOMIZE
130   CHOICE=INT(RND*100)
140   TRIES=0
150   INPUT "YOUR GUESS -> ":GUESS
160   TRIES=TRIES+1
170   ON SGN(GUESS-CHOICE)+2 GOTO 180,200,260
180   PRINT : "YOUR GUESS IS TOO LOW": :
190   GOTO 150
200   PRINT : :"CONGRATULATIONS!"
210   PRINT :"YOU GUESSED IT IN";TRIES;"GUESSES.": :
220   INPUT "PLAY AGAIN? (Y/N) -> ":Y$
230   IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 130
240   PRINT : :"BYE"
250   STOP
260   PRINT : "YOUR GUESS IS TOO HIGH": :
270   GOTO 150
280   END
```

Listing 4-75. ON...GOTO Example 2

OPEN	Open a file.
Type:	Statement
Format:	[*line#*] OPEN#*file-num*:*device*[.*filename*] [,*open-mode*][,*file-org*][,*file-type*] [,*record-type* [*record-size*]]
Purpose:	OPEN sets up an association between a file on a physical device and a logical file number (*file-num*) used in your program.
Operands:	*line#* is a BASIC statement line number that you need when you include OPEN in a program. You don't need *line#* when you use OPEN as a command. *line#* can be any number between 1 and 32767. *file-num* is a number, numeric variable, or numeric expression whose value is between 1 and 255. It is the *logical file number* by which you identify the file in your program's INPUT # and PRINT # statements. *device* is a string, string variable, or string expression that specifies the physical device where the file resides. Typical *device* values are shown in Table 4-29. *filename* is a string, string variable, or string expression that specifies the name of the file on the *device*. You use a *filename* with a disk file. You do not use a *filename* with a cassette file. *open-mode* is INPUT, OUTPUT, UPDATE, or APPEND and tells TI BASIC how to process the file. *file-org* is either sequential or relative and specifies how TI BASIC will access the records in the file. Sequential file records are read or written one after another. Relative file records are uniquely identified by a record number and can be read or written in whatever order you choose. *file-type* is either display or internal and tells TI BASIC the format of the data in the records. Display format records are in ASCII code and can be read by people as well as the computer. Internal format records are in internal machine format and can be read only by the computer. *record-type* is fixed or variable and tells TI BASIC if the file's records are all the same size (fixed) or of varying sizes (variable). *record-size* tells TI BASIC the maximum size of each record. Table 4-34 shows the typical record sizes for cassette files. Disk files can have any *record-size* between 2 and 254 (variable), or 1 and 255 (fixed).
Defaults:	TI BASIC uses these defaults: Sequential, and display. Other defaults depend on the device.

Description:

You can store data in files on tape or diskette using cassette recorders or disk drives. You use an OPEN statement to tell TI BASIC where the files are and what to expect in the files.

OPEN associates a file with a logical file number, *file-num*, that you use to identify the file in your program's PRINT # and INPUT # statements. This association enables TI BASIC to read data from INPUT # or write data to (PRINT #) the file.

You must include a *file-num* and *device* in every OPEN statement and they must appear in correct order. The other OPEN operands may appear in any order or not at all.

To open file number 11 on cassette recorder number 1 use:

```
100   OPEN #11:"CS1", INPUT
```

However, to open the same type of file on a disk, you must also give a *filename*, like this:

```
100   OPEN #11:"DSK1.MYDATA", INPUT
```

Which of the other operands you must use depends on where the file is and how you want to use it. For example, you need a *filename* to uniquely identify a disk file because of the way disks keep track of files. You do not need a *filename* for cassette files because you identify which tape the file is on and at what tape counter value.

We'll talk about the OPEN statement operands in the order in which they appear in the OPEN format.

You use a *file-num* to associate a logical file number (whatever you use as *file-num*) with a file. As we already said, your program uses INPUT # statements to read data from a file and PRINT # statements to write data to a file. The value that you use in the OPEN *file-num* is also used when you want to read from or write to the file. It's TI BASIC's way of identifying which file to use.

file-num can be any value between 0 and 255. You must, of course, use a different *file-num* for each file that you OPEN. *file-num* 0 is a special case. It refers to the keyboard (for input) or the screen (for output). You should not OPEN file number 0 in your program. TI BASIC has the keyboard and screen files always open.

OPEN's *device* operand tells TI BASIC where to look for the file. You can use any of the *device* values in TABLE 4-29. Other devices will become available as new products are developed. You must use double quotes (") around the *device* or *device.filename* operands if you are using the string form. You don't need the double quotes when you use a string variable to specify the device.

Table 4-29. OPEN *device* Values

device	Meaning
CS1	Cassette recorder 1. Can be used for input or output.
CS2	Cassette recorder 2. Can be used for output only.
DSK1	Disk drive 1. Requires a filename.
DSK2	Disk drive 2. Requires a filename.
DSK3	Disk drive 3. Requires a filename.
RS232	A peripheral attached to the RS232 interface.
RS232/1	A peripheral attached to the first serial port on the RS232 interface when you have a Y-connector on the RS232 serial port.
RS232/2	A peripheral attached to the second serial port on the RS232 interface when you have a Y-connector on the RS232 serial port.
HEXBUS	A Hexbus peripheral.

Some examples of *device* operands are:

- Open file number 2 for output on cassette recorder 1

```
100   OPEN #2:"CS1",OUTPUT"
```

- Open file number 44 for output as file NEWDATA on disk drive 3

```
150   OPEN #44:"DSK3.NEWDATA",OUTPUT
```

- Open file number 250 for input. The device is stored in the string DSKIN$. The file name is stored in the string FILEIN$.

```
120   OPEN #250:DSKIN$&"."&FILEIN$,INPUT
```

Device values are associated with physical units, such as cassette recorders and disk drives. *filenames* are associated with particular sets of data stored on the devices.

Some devices keep track of the files that are stored on them. Disk drives are one example of this type of device. Each diskette (also referred to as a disk) keeps a directory of the files on the diskette and how much space is left for other files. You have to tell TI BASIC which file to use by specifying a *filename* operand. You must use a period (.) between the *device* and *filename* operands, like this:

DSK1.NEWDATA
DSK2.NAMEFILE
DISK$&"."$&FILE$
"DSK"&STR$(DSKNUM)&"."&FILE$(DSKNUM)

Tapes, on the other hand, have no directory. You must keep track of what is on the tapes yourself. You never use a *filename* with a tape because it doesn't have any meaning. There are only two possible *device* values for tapes:

"CS1"
"CS2"
"CS"&STR$(CSNUM)

Still other devices that don't need a *filename* are those attached to the RS232 interface. These devices are usually printers and have *device* values like this:

"RS232"
"RS232/1"

Now that you know where the file is and its name (if necessary), you tell TI BASIC how you want to use the file through the *open-mode* operand. There are only four values that you can use for *open-mode*, as shown in Table 4-30.

Table 4-30. OPEN *open-mode* Operand

open-mode	Meaning
INPUT	You can only read the data in the file (using INPUT # statements).
OUPUT	You can only write data to the file (using PRINT # statements). Any OPEN operands that you don't supply will be taken from the file and from the OPEN statement defaults.
UPDATE	You can both read data from and write data to the file.
APPEND	You can only write data to the file. The data is added to the end of the existing file. This lets you expand a file by adding records that start at the end of all the records currently in the file.

NOTE: If you do not specify an *open-mode* value, TI BASIC uses UPDATE.

Some examples of the *open-mode* operand are:

- Open file number 66 on cassette one for input

```
100  OPEN #66:"CS1",INPUT
```

- Open file number 87 on disk one in the file called "MASTER" and update [read and write] the file

```
120  OPEN #87:"DSK1.MASTER",UPDATE
```

- Open file number 160 on disk one in the file called "DATAFILE" and position at the end of the file

```
500  OPEN #160:"DSK1.DATAFILE",APPEND
```

NOTE

You can use OUTPUT for cassette files on either device CS1 or CS2. You can use INPUT for cassette files only on device CS1.

Now it's time to tell TI BASIC how the file is organized. Organization simply means how the records are arranged in the file and how you want to use them. Table 4-31 shows you the TI BASIC *file-org* choices.

Some examples of the *file-org* operand are:

```
100  OPEN #5:"CS1",SEQUENTIAL
100  OPEN #11:"DSK1.OLDATA",SEQUENTIAL
```

You can have only SEQUENTIAL files on tape. Relative files can be only on disk because there is no way to randomly access a cassette file. You can process RELATIVE files sequentially or randomly, depending on your INPUT # or PRINT # operands. When you first OPEN a RELATIVE file for OUTPUT, you can tell TI BASIC a value for the number of records in the file, like this

```
100  OPEN #19:"DSK1.RELFILE",OUTPUT,RELATIVE 250
```

which tells TI BASIC that there will be 250 records in the file to start with.

Table 4-31. OPEN *file-org* Operand

file-org	Meaning
SEQUENTIAL	The records are arranged in sequential order, one right after the other. When TI BASIC reads a SEQUENTIAL file, it gets the first record, then the second, etc., in the order in which the records appear. When TI BASIC writes a SEQUENTIAL file, it writes the records in the order in which your program PRINT #s them.
RELATIVE	Each record has a unique identifier and TI BASIC can access the records by that identifier. TI BASIC can read a RELATIVE file in sequential order, beginning with record 0, then record 1, etc., or in random order by record number. TI BASIC can write a RELATIVE file in sequential order or random order.

NOTE: If you don't specify a *file-org* value, TI BASIC uses SEQUENTIAL.

Now TI BASIC knows how to identify the file (*file-num*), where the file is (*device* and *filename*), and how you want to access the file (*file-org*). You can also specify a *file-type* that tells TI BASIC what format is used to store the data in the file. *file-type* values are shown in Table 4-32.

Table 4-32. OPEN *file-type* Operand

file-type	Meaning
DISPLAY	The data is stored in printable ASCII characters. People and computers can read this format. DISPLAY data is usually printed. Each character takes one byte in the record.
INTERNAL	The data is stored in machine readable, internal format. Only computers can read this format. INTERNAL format data takes much less space than DISPLAY format data and is usually used for data that only the computer reads.

NOTE: When you don't specify a value for *file-type*, TI BASIC uses DISPLAY.

DISPLAY format data is easy for people to read. It looks like the characters you see on your screen. Usually, DISPLAY is used for output that you write to a printer attached to the RS232 interface card. For example,

```
100 OPEN #87:"RS232",OUTPUT,DISPLAY
```

DISPLAY format data takes more room on a tape or disk than does INTERNAL format data. In addition, DISPLAY format data must be converted to INTERNAL format data before your computer can use it. If you have data that you are not going to be reading but your computer will be reading and writing, use INTERNAL format, like this:

```
400  OPEN #213:"DSK1.MASTER",OUTPUT,INTERNAL
420  OPEN #66:"DSK1.OLDMASTER",INPUT,INTERNAL
```

Finally, you can choose the file's *record-type*. This tells TI BASIC whether the file's records are all the same length (FIXED) or whether they have different lengths (VARIABLE). Table 4-33 tells you how TI BASIC treats the different *record-type*.

Table 4-33. OPEN *record-type* Operand

record-type	Meaning
FIXED	All records in the file are the same length. If the data in the record is shorter than the *record-size,* TI BASIC "pads" the record to the *record-size.* Display data is padded with spaces. Internal data is padded with binary zeroes.
VARIABLE	The records in the file can have different lengths, with a maximum length of *record-size.* The maximum *record-size* depends on the device.

NOTE: If you don't specify a *record-type* operand, TI BASIC uses FIXED for relative files and VARIABLE for sequential files.

Depending on the *record-type,* you can also choose a *record-size.* Not all *record-sizes* are valid for all *record-types.* Those *record-sizes* valid for cassette files are shown in Table 4-34.

Table 4-34. OPEN Cassette file *record-size* Values

record-size	Meaning
64	Each record can contain a maximum of 64 characters. Used for fixed files, especially on cassettes.
128	Each record can contain a maximum of 128 characters. Used for fixed files, especially on cassettes.
192	Each record can contain a maximum of 192 characters. Used for fixed files, especially on cassettes.

Remember that TI BASIC will "pad" (extend the data to *record-size*) any unfilled FIXED records. If you are using short records, use a smaller *record-size.* If you try to write data that is longer than *record-size,* you will get an error and your program will stop.

Some examples of *record-type* and *record-size* operands are:

```
100  OPEN #101:"CS1",INPUT,FIXED 192,SEQUENTIAL
150  OPEN #2:"DSK1.MYDATA",UPDATE,VARIABLE
     125,RELATIVE
160  OPEN #8:"CS2",OUTPUT,FIXED 64
```

While OPEN looks imposing with its large number of operands, it's not that difficult to use. Decide what you want to do with the file and choose the appropriate operands.

Common Errors:

FILE ERROR

You tried to OPEN a file that is already OPEN.

INCORRECT STATEMENT

You used the same keyword more than once in the OPEN statement or you spelled one or more keywords incorrectly.

Or, you used a FIXED record length that is less than zero or greater than 255.

Or, you specified a negative number of records in a SEQUENTIAL or RELATIVE option.

I/O ERROR 00

You used an invalid *device* name.

I/O ERROR 02

You have one or more invalid operand values in your OPEN statement.

Or, you are using a file that already exists and one or more of the values that you used in your OPEN statement operands do not match the characteristcs of the file.

I/O ERROR 06

There is a device error. Perhaps your device became disconnected after you started your program. First check that everything is connected properly and rerun your program. If you get the error again, you may have a device that is not functioning properly.

STRING-NUMBER MISMATCH

File-num is not a valid number, numeric variable, or numeric expression.

Example 1:

The program in Listing 4-76 OPENs a file on cassette and writes whatever you enter to the tape.

The file has these characteristics:

- It's on cassette recorder one (CS1)
- Records are written to the file (OUTPUT)
- The largest record is 192 characters
- The records are all the same size (FIXED)

- The file is written sequentially (SEQUENTIAL)
- The data is written in internal format (INTERNAL)

Try changing the program to use other types of cassette files.

```
100  CALL CLEAR
110  PRINT "I'LL WRITE WHATEVER YOU WANT":
     " TO CASSETTE ONE."
120  OPEN #2:"CS1",OUTPUT,FIXED 192,SEQUENTIAL,INTERNAL
130  RECS=0
140  PRINT
150  INPUT "YOUR DATA (XXX TO END) -> ":ANS$
160  RECS=RECS+1
170  PRINT #2:ANS$
180  IF ANS$<>"XXX" THEN 140
190  CLOSE #2
200  PRINT : :RECS;"RECORDS WRITTEN.": : "BYE"
210  END
```

Listing 4-76. OPEN Example 1

Example 2:

The program in Listing 4-77 OPENs a disk file for INPUT and prints what it reads. Because TI BASIC uses the file characteristics from the disk directory information, you don't need all the OPEN operands.

The EOF function tells you when you get to the end of the disk file.

Try changing the program to write to the disk file.

```
100  CALL CLEAR
110  PRINT "I'LL PRINT YOUR DISK FILE"
120  INPUT "WHAT DISK (1,2,3) ->":DISKID
130  IF (DISKID<1)+(DISKID>3) THEN 120
140  INPUT "WHAT FILE NAME -> ":FILEIN$
150  OPEN #22:"DSK"&STR$(DISKID)&"."&FILEIN$,
     INPUT,INTERNAL
160  RECS=0
170  IF EOF(22) THEN 220
180  INPUT #22: STRIN$
190  PRINT STRIN$
200  RECS=RECS+1
210  GOTO 170
220  CLOSE #22
230  PRINT : :RECS;"RECORDS READ.": :"BYE."
240  END
```

Listing 4-77. OPEN Example 2

OPTION BASE	Set the first subscript value.
Type:	Statement
Format:	*line#* OPTION BASE 0 or *line#* OPTION BASE 1

OPTION BASE	Set the first subscript value. *(continued)*
Purpose:	OPTION BASE sets the lowest subscript for an array to zero or one.
Operands:	*line#* is a BASIC statement line number that you need when you include OPTION BASE in a program. *line#* can be any number between 1 and 32767.
Defaults:	OPTION BASE 0 is the default when you don't use an OPTION BASE statement.

Description:

OPTION BASE sets the lowest valid subscript for all arrays to zero (OPTION BASE 0) or one (OPTION BASE 1). The lowest value is set for every array in the program.

You can use only one OPTION BASE statement in your program and it must have a lower *line#* than any DIM statements in the program.

If you don't use an OPTION BASE statement, TI BASIC starts each array with element zero. You can use zero as a valid subscript value.

It's not always convenient to start numbering at zero, so you can use an OPTION BASE 1 statement to set the first valid subscript to one.

Another reason for using OPTION BASE 1 is to save the memory space taken by the zero subscript element. If you aren't going to use a subscript of zero and you have a lot of arrays, use OPTION BASE 1 to save the memory required for all the zero subscript elements.

If your program contains an OPTION BASE 1 statement, you can no longer use a subscript of zero.

Common Errors:

BAD SUBSCRIPT

You used an OPTION BASE 1 and you have a subscript of zero.

CAN'T DO THAT

You tried to use OPTION BASE as a command. You can use OPTION BASE only as a statement in a program.

Or, you have more than one OPTION BASE statement in your program.

Or, the OPTION BASE statement's *line#* is higher than the line numbers of the DIM statements in your program.

INCORRECT STATEMENT

The keyword OPTION is not followed by the keyword BASE. Or, you used a value other than 0 or 1 after OPTION BASE.

Example 1:

The program in Listing 4-78 uses an OPTION BASE 1 statement to set the lowest array subscript to one. If you try to use a subscript of zero, you would get an error.

Try changing the program to include a RANDOMIZE statement and see what happens to your random numbers.

```
100   CALL CLEAR
110   OPTION BASE 1
120   DIM RANDOMS(15)
130   PRINT "HERE ARE 15 RANDOM NUMBERS":
      " BETWEEN 1 AND 100.": :
140   FOR I=1 TO 15
150   RANDOMS(I)=INT(RND*100)
160   NEXT I
170   FOR L=1 TO 15
180   PRINT RANDOMS(L)
190   NEXT L
200   END
```

Listing 4-78. OPTION BASE Example 1

Example 2:

The program in Listing 4-79 uses an OPTION BASE 0 statement to set the lowest array subscript to zero. This program is very similar to the program in Listing 4-78 but notice that there are now 16 random values in the array. When you start counting at 0, you have one more value in the array.

You could use a subscript of zero with this program and not get an error.

```
100   CALL CLEAR
110   OPTION BASE 0
120   DIM RANDOMS(15)
130   PRINT "HERE ARE 16 RANDOM NUMBERS":
      " BETWEEN 1 AND 100.": :
140   FOR I=0 TO 15
150   RANDOMS(I)=INT(RND*100)
160   NEXT I
170   FOR L=0 TO 15
180   PRINT RANDOMS(L)
190   NEXT L
200   END
```

Listing 4-79. OPTION BASE Example 2

POS	Find a string in another string.
Type:	Function
Format:	POS (*search-str,find-str,num-exp*)
Purpose:	The POS function searches the string *search-str*, beginning at position *num-exp*, for the first occurrence of the string *find-str*.
Operands:	*search-str* is a string, string variable, or string expression that you want to search, beginning at *num-exp* characters from the beginning of the string. *find-string* is a string, string variable, or string expression that specifies the string that you want to search for.

POS	Find a string in another string. *(continued)*
	num-exp is a number, numeric variable, or numeric expression that specifies the position of the first character in *search-str* that gets searched for the *find-str* characters.
Defaults:	None.

Description:

POS is a string function that searches one string (*search-str*) for the occurrence of another string (*find-str*). POS returns the position of the first occurrence of *find-str* in *search-str* starting at character *num-exp* in *search-str*. Fig. 4-15 shows you how POS works.

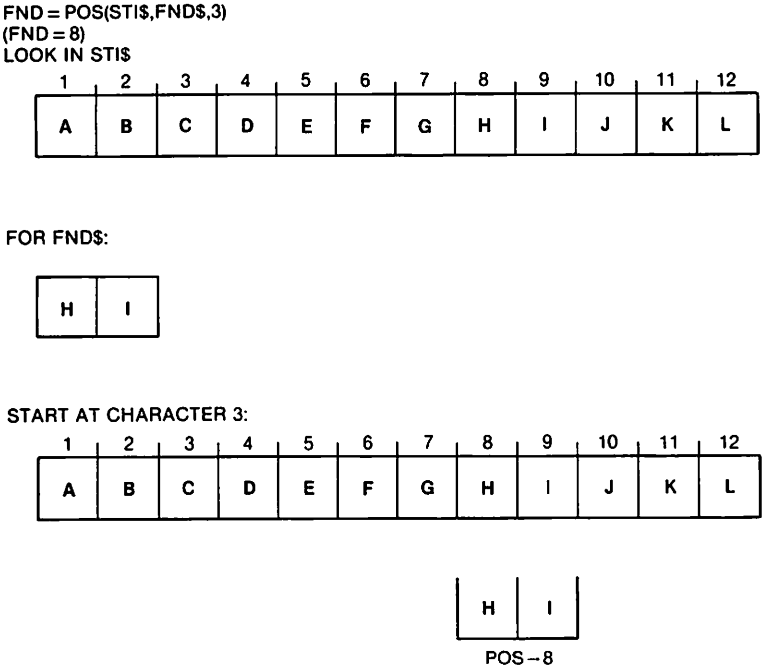


Fig. 4-15. POS example.

The value that POS returns tells you whether the *find-str* occurs in *search-str*, like this:

- zero (0) means that *find-str* is not contained in *search-str*
- a nonzero value means that *find-str* begins at that position in *search-str*

You use *num-exp* to tell TI BASIC at which character in *search-str* to start looking for your *find-str.* If you use a *num-exp* that is less than zero, your program stops and you get the error message:

BAD VALUE

If you use a value for *num-exp* that is larger than the length of *search-str,* POS returns a zero.

POS assigns a value to a variable, like this:

```
100   POSITION=POS(ANS$,“Y”,1)
```

Or, POS can appear in string expressions, like this:

```
120   IF POS(ANS$,“Y”,1)=0 THEN 500
130   RES$=SEG$(ANS$,POS(ANS$,Y$,1),20)
```

POS, when used with the other TI BASIC string functions, can help you to perform complicated string manipulations.

Common Errors:

BAD VALUE

Num-exp is less than zero, zero, or larger than 32767.

Example 1:

The program in Listing 4-80 uses POS to see if a numeric answer contains a digit 5. The numeric answer is converted to a string before POS can be used.

Try changing the program to see how many 7's there are in the answer.

```
100  CALL CLEAR
110  PRINT "ENTER A NUMBER AND I'LL":
     "  TELL YOU HOW MANY 5'S"
120  PRINT "  ARE IN IT."
130  PRINT : : :
140  INPUT "YOUR NUMBER -> ":ANS
150  FIVES=0
160  ANS$=STR$(ANS)
170  START=1
180  N=POS(ANS$,"5",START)
190  IF N=0 THEN 230
200  FIVES=FIVES+1
210  START=N+1
220  GOTO 180
230  PRINT : :"THERE ARE";FIVES;"5'S IN";ANS: : :
240  INPUT "TRY AGAIN? (Y/N) -> ":Y$
250  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 130
260  PRINT : :"BYE"
270  END
```

Listing 4-80. POS Example 1

Example 2:

The program in Listing 4-81 uses POS to find every "A" in a string and replace the upper-case "A" with a lower-case "a".

```
100   CALL CLEAR
110   PRINT "I'LL CHANGE CAPITAL A TO":
      " LOWERCASE a FOR YOU"
120   PRINT : : :
130   INPUT "ENTER A STRING -> ":ANS$
140   N=1
150   APOS=POS(ANS$,"A",N)
160   IF APOS=0 THEN 230
170   ANS$=SEG$(ANS$,1,APOS-1)&"a"&SEG$(ANS$,APOS+1,255)
180   N=APOS+1
190   IF A<LEN(ANS$) THEN 150
200   IF A>LEN(ANS$) THEN 230
210   IF POS(ANS$,A,1)<>"A" THEN 230
220   ANS$=SEG$(ANS$,1,A-1)&"a"
230   PRINT : :ANS$: :
240   INPUT "TRY AGAIN? (Y/N) -> ":Y$
250   IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
260   PRINT : : "BYE"
270   END
```

Listing 4-81. POS Example 2

PRINT	Write to the screen.
Type:	Statement
Format:	[*line#*] PRINT or [*line#*] PRINT *list*
Purpose:	PRINT writes the information in *list* or, if there is no *list*, a blank line to the screen.
Operands:	*line#* is a BASIC statement line number that you need when you include PRINT in a program. You don't need *line#* when you use PRINT as a command. *line#* can be any number between 1 and 32767. *list* is a list of string or numeric variables, constants, or expressions separated by the *print-separators* described below. This information is printed on one or more lines at the screen.
Defaults:	If you don't use *list*, TI BASIC prints a single blank line on the screen.

Description:

PRINT writes the information in *list* to the screen. Each item in the list, starting with the first item on the left and proceeding to the right, is evaluated, converted to character format if necessary, and displayed on the screen.

list is a list of items (variables, numbers, expressions, functions, strings) and *print-separators* that you want to display on your screen. When you have more than one item in your *list*, you must separate the items with one or more of the *print-separators* shown in Table 4-35. (The TAB function

Table 4-35. PRINT *print-separators*

print-separator	Meaning
semicolon (;)	Write the next data item right next to the current data item. Do not leave any extra spaces (except for the leading and trailing characters around numeric data items).
colon (:)	Skip to the next line.
comma (,)	Write the next data item at the next available zone. Zone 1 starts in column 1. Zone 2 starts in column 15.

is discussed in detail in its own section.) You can end your *list* with one or more *print-separators*.

Here are some examples of valid PRINT statements to show you what can be done. You can see how strings, variables, numbers, expressions, and *print-separators* are used in the *list*.

```
100   PRINT "4+5=";4+5
200   PRINT "MY NAME IS ";NAME$
300   PRINT : :"HELLO"
400   PRINT Z+SQR((A^5)+(X*FX))-.567
500   PRINT "ABC"&" DEF ";VARIABLE33
```

TI BASIC uses rules to determine where a value is printed and how the value is printed. First, we'll consider how TI BASIC prints string and numeric values.

Each screen line contains 28 possible print positions. That is, each TI BASIC screen print line is 28 characters long. To avoid losing some of the image on some television sets, TI BASIC does not PRINT in the first two or last two columns of the potential 32 columns on your screen.

The 28 print positions on a screen line are divided into two *zones* for printing:

Zone 1 begins in column 1.
Zone 2 begins in column 15.

Depending on which *print-separator* you use, TI BASIC either prints in the next available zone or at the location specified by the *print-separator*.

But TI BASIC also has to decide if a value can fit on a line:

- Items are not split across lines, unless the item is a string that contains more than 28 characters. In that case, the string begins on the next line and is printed on as many lines as necessary.
- If a number is too long to fit on the current line, it is printed on the next line.
- But, if the only character of a number that will not fit on the line is the trailing blank, TI BASIC does not print the trailing blank and prints the number at the end of the line.

TI BASIC follows these rules when printing data on your screen:

1. String variables and expressions are evaluated. The result, as well as any string constants that you include in *list,* is printed to the screen as it appears. TI BASIC does not include any extra blank characters (spaces) to the beginning or end of the string.
2. Numeric variables and expressions are evaluated. The result, as well as any numeric constants that you have in *list,* is printed to the screen with a trailing blank character. If the value is positive, TI BASIC prints a leading blank character. If the value is negative, TI BASIC replaces the leading blank with a minus sign (−).
3. Numbers that contain 10 or fewer digits are printed as they appear, in normal decimal format (like 1.23, 235.99, or −.025). The absolute value (ignoring the sign) of these numbers must be larger than 10^{-11} and smaller than 10^{11}.
4. Numbers which have more than 10 digits are printed in scientific notation. For example, 100000000000 is printed as 1.0E11. the absolute value (ignoring the sign) of these numbers must be larger than 10^{10} or smaller than 10^{-10}.
5. Numbers printed in scientific notation show only six significant digits, plus the exponent:

   n.nnnnnE{+|−}ee

   Chapter 2, Data in BASIC, contains a discussion of scientific notation.
6. Since TI BASIC maintains 13 or 14 digits in precision internally, numbers are rounded at the last displayed digit (the 10th or, in the case of scientific notation, the 6th).

To show you how PRINT places items on your screen, enter the following statements:

```
100  CALL CLEAR
110  A=12.34
120  B= −56.78
130  S$="A STRING"
140  REM USE COMMAS AND ZONES
150  PRINT A,B,S$
160  REM USE SEMICOLONS
170  PRINT A;B;S$
180  REM USE TABS
190  PRINT A;TAB(7);B;TAB(18);S$
200  REM BLANK LINES
210  PRINT : : :
220  PRINT "BYE"
230  END
```

When you RUN this program, you will see the three values printed in various formats on your screen. The spaces between the colon (:) *print-separator* are there so that you can also run this example in Extended BASIC.

Usually each PRINT statement prints at the beginning of a new line (Zone 1). If you end your *list* with a *print-separator,* the *print-separator* gets evaluated and the first item in your *next* PRINT statement gets printed in the position determined by that final *print-separator* of the previous PRINT statement.

For example, to print 10 numbers using semicolon (;) separators, you can use:

```
100  FOR I=1 TO 10
110  PRINT I;
120  NEXT I
```

Notice that the PRINT statement ended with a semicolon. The numbers are printed one right after the other, depending, of course, on whether it can fit onto a line. If you add this statement to the above example, you'll notice that the string printed in the final PRINT statement doesn't start on a new line because of the semicolon in the previous PRINT statement (line 110).

```
130  PRINT "GOODBYE"
```

To make the parting message print on its own line, use:

```
130  PRINT :"GOODBYE"
```

Print statements are one way that your computer communicates information to you. It's up to you to make the information readable. When you are writing programs, try to print your data in a well organized, pleasantly readable fashion.

Use strings in your PRINT statements to tell what it is you're printing:

```
190  PRINT "NEW INCOME = ";NETINC
```

Or, print instructions using several PRINT statements, like this:

```
250  PRINT "THIS PROGRAM ASKS YOU":" TO ENTER TWO VALUES."
260  PRINT :"THE FIRST VALUE IS":" THE NUMBER OF HOURS."
270  PRINT :"THE SECOND VALUE IS":" THE RATE PER HOUR."
```

Try different PRINT statements, using different *print-separators,* until you get the information on your screen so that it's easily readable.

Common Errors: None.

Example 1:

The program in Listing 4-82 uses PRINT statements to write the same data to the screen using different *print-separators*.

```
100  CALL CLEAR
110  PRINT "I'LL USE DIFFERENT SEPARATORS"
120  PRINT " TO WRITE TWO NUMBERS AND":
     " ONE STRING ON YOUR SCREEN."
130  STRING$="ABC DEF"
140  VAR=1.23
150  OTHER=-32.65
160  PRINT :"USING COMMAS": :
170  PRINT VAR,OTHER,STRING$
180  PRINT :"USING SEMICOLONS": :
190  PRINT VAR;OTHER;STRING$
200  PRINT :"USING TAB(5) AND (15)": :
210  PRINT VAR;TAB(5);OTHER;TAB(15);STRING$
220  PRINT :"USING TAB(15)": :
230  PRINT VAR;TAB(15);OTHER;TAB(15);STRING$
240  PRINT :"BYE"
250  END
```

Listing 4-82. PRINT Example 1

Try changing the *print-separators* to get different output.

Example 2:

The program in Listing 4-83 asks you for a string and prints it beginning at column 15. If the string won't fit on the line, it gets printed on the next line.

Try changing the program to ask for other information and print it in various formats.

```
100  CALL CLEAR
110  PRINT "ENTER A STRING AND":
     " I'LL PRINT IT STARTING"
120  PRINT " IN COLUMN 15 IF I CAN."
130  PRINT : : :
140  INPUT "YOUR STRING -> ":IN$
150  PRINT TAB(15);IN$
160  INPUT "TRY AGAIN? (Y/N) -> ":Y$
170  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 130
180  PRINT : :"GOODBYE."
190  END
```

Listing 4-83. PRINT Example 2

PRINT #	Write to a file.
Type:	Statement
Format:	[*line#*] PRINT #*file-num* or [*line#*] PRINT #*file-num* : *list* or [*line#*] PRINT #*file-num* ,REC *rec-num* : *list*
Purpose:	PRINT # writes the information in *list* or, if there is no *list*, a blank record to the file OPENed as *file-num*.

PRINT #	Write to a file. *(continued)*
Operands:	*line#* is a BASIC statement line number that you need when you include PRINT in a program. You don't need *line#* when you use PRINT as a command. *line#* can be any number between 1 and 32767. *file-num* is a number, numeric variable, or numeric expression that evaluates to a value between 0 and 255 and represents a file that you have already OPENed with the same *file-num*. *list* is a list of string or numeric variables, constants, or expressions separated by the *print-separators* described below. This information is written to the file OPENed as *file-num*. *rec-num* is a number, numeric variable, or numeric expression that specifies the record to write into a RELATIVE disk file. TI BASIC writes the record number *rec-num*.
Defaults:	TI BASIC writes empty record if you don't use a *list*.

Description:

PRINT # writes the data in *list*, or a blank record if you don't ust *list*, to the file OPENed as *file-num*. You use PRINT # to write to any external device, like a cassette tape, disk, or printer. You can even PRINT # to the screen if you use a *file-num* of zero. (PRINT #0 works the same as PRINT.)

When you use PRINT #, you must:

- Use a *file-num* that represents a currently open file.
- PRINT # to a file that was OPENed as OUTPUT, APPEND, or UPDATE.

You cannot use a PRINT # statement to process a file opened for INPUT.

When you OPEN your file, TI BASIC sets up a special area in memory called an *I/O buffer* as a temporary storage area for your file. A PRINT # statement puts the data from the *list* into the I/O buffer and, when all the items in the *list* are written, and the record is complete, BASIC writes the I/O buffer to the device. The sections below tell you how DISPLAY and INTERNAL format data is handled.

Usually you end your *list* with a data item (variable, expression, number, or string). This signals TI BASIC that your record is complete and can be written. This type of PRINT # statement looks like:

```
100  PRINT #98: A,B,C
200  PRINT #4:NAME$,ADDRESS$
```

If you end your *list* with a *print-separator*, you create a "pending print" which is described below. TI BASIC knows that more information will get put into the record and holds the record in the I/O buffer. The next PRINT # statement puts its data at the end of the data currently in the I/O buffer. A pending PRINT # statement looks like:

```
100  PRINT #22: A, B
500  PRINT #65: NAME$;
```

Depending on how you OPEN your file, you can write data in INTERNAL format or DISPLAY format. PRINT # works differently for each of these data formats.

PRINT # and DISPLAY Format Data—You usually use DISPLAY format when you are writing data to a device where you (not the computer) will be reading the information, like a printer or the screen.

DISPLAY format data takes more room than INTERNAL format data on a disk or tape and must be converted to INTERNAL format before your computer can use it. There are times when you must use DISPLAY format data for a tape or disk file. However, TI BASIC does not include punctuation (separating commas or double quotes) in its PRINT # output. Only the information that you specify in *list* is written.

In order to later read the data with an INPUT # statement, you must make the information stored in your file look exactly as you would enter it from the keyboard. This means that you must explicitly write comma separators between data items and double quotes (”) around string data:

- You include a separator comma in the *list* like this:

“,”

- You include a double quote (for the beginning or ending double quote around string data) in the *list* like this:

“”“”

For example, to print a number, a separating comma, leading double quotes, a string, and trailing double quotes, use a PRINT # like this:

```
100   PRINT #45: DOLLARS;“,”;“”“”;NAME$;“”“”
```

To print three numbers separated by commas, use a PRINT # like this:

```
200   PRINT #22:A(1);“,”;A(2);“,”;NEWVAL
```

When you PRINT # DISPLAY format data, the *print-separators* that you use between items in your *list* make a difference. Table 4-36 shows you the *print-separators* and their meanings. Use the *print-separators* that you need to format your output so that you can read it easily. You might, for example, use the TAB function with the semicolon *print-separator* to write data in columns. Don't forget that you usually have more than 28

Table 4-36. PRINT # *print-separators*

print-separator	Meaning
semicolon (;)	Write the next data item right next to the current data item. Do not leave any extra spaces (except for the leading and trailing characters around numeric data items).
colon (:)	Skip to the next line.
comma (,)	Write the next data item at the next available zone. Zone 1 starts in column 1. Zone 2 starts in column 15.

columns to PRINT # your data. Printers, for example, often have 80 or 132 columns.

DISPLAY data takes as much space in your file as it does on your screen—each character or space takes one byte of your record. PRINT # has to fit your data into the records in the file.

When you OPENed the file, you gave a maximum record length. After TI BASIC is done formatting the data to be written, it may have more characters than you specified in your maximum record length. Only strings longer than the maximum record length are allowed to split across record boundaries. TI BASIC follows these rules in filling records:

- If an item in the *list* makes the record longer than the maximum record length, TI BASIC writes the item as the first item in a new record.
- If a string is too long to fit into a single record, it is split and written in as many records as necessary.

PRINT # and INTERNAL Format Data—INTERNAL format data takes much less space on a tape or disk than does DISPLAY format data. Instead of one byte per character (as for DISPLAY format data), INTERNAL format data uses:

- Nine bytes per numeric value: 1.23, −5555.666, 2.345E77 each occupy nine bytes (8 for the number and one for the length, which is always 8).
- The length of a string plus one length byte for string value: "THIS STRING CONTAINS 34 CHARACTERS" takes 35 bytes (34 for the data plus one for the length), "ABC" takes 4 bytes (three for the data plus one for the length).

You can see how much less space is used, especially for numeric data.

There's another advantage to using INTERNAL format data. Your computer "thinks" in INTERNAL format. When it gets DISPLAY format data from the keyboard or from a file, it must convert the information into INTERNAL format before the data can be used. If your data is going to be used only by the computer (such as your master file of names and addresses stored on your tape or disk), it is better to use INTERNAL format to store it in your file.

You do not have to include any explicit separators when you write INTERNAL format data. TI BASIC doesn't have to translate the data. It "knows" where each data item begins and ends because it is stored in the format that is TI BASIC's "native language."

TI BASIC ignores any *print-separator* action when it writes INTERNAL format data with PRINT #. All the *print-separators* have exactly the same effect. They act as item separators in the *list*, not as formatting items to indicate spacing.

```
100  PRINT #123: A,B,STRINGIN$,X
```

gets written in exactly the same format as:

```
100  PRINT #123: A:B:STRINGIN$:X
```

or

```
100 PRINT #123: A;B;STRINGIN$;X
              or
100 PRINT #123: A:B,STRINGIN$;X
```

or any other combination of *print-separators*

TI BASIC follows these rules when you use PRINT # to write INTERNAL format data to records in your file:

- For FIXED length records, TI BASIC "pads" (includes extra characters) the data with binary zeros if the data is less than the record length.
- For VARIABLE length records, TI BASIC writes a length indicator before each record and does not "pad" the data.
- If you attempt to write data that is longer than the maximum record length you used in your file's OPEN statement, TI BASIC stops your program and prints:

```
FILE ERROR
```

PRINT # and RELATIVE Files—PRINT # also works with RELATIVE files, those files that you can read or write either sequentially (one record right after the one before it) or randomly (by unique record number).

Each record in a RELATIVE file has a unique record number. TI BASIC starts numbering records at record number zero and adds one for each successive record.

- If you PRINT # to a RELATIVE file and you don't use a REC *rec-num* operand, TI BASIC starts with record zero and adds one to the record number for each PRINT # to the file.
- If you use a REC *rec-num* operand in your PRINT # statement, TI BASIC writes the record with *rec-num* as its identifier.

TI BASIC adds one to its record counter every time it reads (INPUT #) or writes (PRINT #) a record to the RELATIVE file. You may not always get the record that you expect when you both read and write to a RELATIVE file without using a REC *rec-num* operand.

PRINT # and Pending Prints—A *pending print* occurs when you end your *list* with a *print-separator.* We already talked about records and the I/O buffer in the beginning of this section.

TI BASIC always writes the record you create with a PRINT # to the file when the PRINT # statement ends with no trailing *print-separator.*

When you cause a pending print by ending your PRINT # with a *print-separator,* TI BASIC holds the data in its I/O buffer until it gets the next PRINT # or INPUT # to the file. Then,

- If there is a pending print and you have an INPUT # to the file, the pending record is written and the pending print condition no longer exists.
- If there is a pending print and you use a PRINT # without a REC *rec-num* operand, the data from the new PRINT # statement gets put into

the I/O buffer immediately after the last character of the previous PRINT # statement's data. The pending print condition continues only if the new PRINT # statement has a trailing *print-separator;* otherwise the record is written to the file and the pending print condition is cleared.

- If there is a pending print and you use a PRINT # with a REC *rec-num* operand, TI BASIC writes the pending record to the file, using the *rec-num* from the PRINT # that caused the pending print. The new data starts its own record and the pending print condition no longer exists.

TI BASIC also clears all pending print conditions (writes the final data in the I/O buffer to the file) when you:

- CLOSE # the file
- RESTORE # the file
- STOP your program
- END your program

Common Errors:

FILE ERROR

You tried to write (PRINT #) to a file that you OPENed as INPUT (read only).

INCORRECT STATEMENT

You forgot the number sign (#) before the *file-num* or the colon (:) before the *list*.

I/O ERROR 36

There is a device error. The device that you PRINT # to may be disconnected or not functioning properly.

STRING-NUMBER MISMATCH

File-num is not a valid number, numeric variable, or numeric expression.

Example 1:

The program in Listing 4-84 asks you for a name, address, and state and writes the information to a file on a cassette.

Use this program with the program describing INPUT #.

Try asking for and writing more data. Remember to use the "ZZZZ" record as an end of file marker for your cassette file. You won't need this record if you use a disk file and use the EOF function to find the end of file when you INPUT # the disk file's data.

```
100  CALL CLEAR
110  PRINT "YOU ENTER NAME, ADDRESS, AND STATE"
120  PRINT :"I'LL WRITE THE INFORMATION":
     " TO YOUR CASSETTE"
130  PRINT : :
140  OPEN #99:"CS1",OUTPUT,FIXED 192,
     SEQUENTIAL,INTERNAL
150  INPUT "NAME -> ":NAME$
160  INPUT "ADDRESS -> ":ADDRESS$
170  INPUT "STATE -> ":STATE$
180  PRINT #99: NAME$,ADDRESS$,STATE$
190  INPUT "ANOTHER RECORD (Y/N) -> ":Y$
200  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 150
210  PRINT #99:"ZZZZ","ZZZZ","ZZZZ"
220  CLOSE #99
230  PRINT : : "BYE"
240  END
```

Listing 4-84. PRINT # Example 1

Example 2:

The program in Listing 4-85 writes 100 random numbers to a disk file. There are five numbers per record.

Use this program with the program describing INPUT #.

```
100  CALL CLEAR
110  RANDOMIZE
120  INPUT "ENTER DSKN.FILENAME -> ":DISKOUT$
130  OPEN #22:DISKOUT$,OUTPUT,INTERNAL,
     VARIABLE,SEQUENTIAL
140  PRINT : :"HERE ARE THE 100 RANDOM"
150  PRINT " NUMBERS THAT I'LL WRITE TO":
     " YOUR DISK FILE.": :
160  FOR I=1 TO 20
170  FOR J=1 TO 5
180  RANDOM(J)=INT(100*RND)
190  NEXT J
200  PRINT RANDOM(1);RANDOM(2);RANDOM(3);
     RANDOM(4);RANDOM(5)
210  PRINT #22: RANDOM(1);RANDOM(2);RANDOM(3);
     RANDOM(4);RANDOM(5)
220  NEXT I
230  PRINT : : "BYE"
240  END
```

Listing 4-85. PRINT # Example 2

RANDOMIZE Seed the random number generator.

Type: Statement

Format: [*line#*] RANDOMIZE

or

[*line#*] RANDOMIZE *num-exp*

Purpose: RANDOMIZE seeds the random number generator function RND.

RANDOMIZE	Seed the random number generator. *(continued)*
Operands:	*line#* is a BASIC statement line number that you need when you include RANDOMIZE in a program. You don't need *line#* when you use RANDOMIZE as a command. *line#* can be any number between 1 and 32767. *num-exp* is a number, numeric variable, or numeric expression that is used as the *seed* for the random numbers generated by the RND function.
Defaults:	If you don't supply a value for *num-exp*, TI BASIC generates an unpredictable sequence of random numbers when you use the RND function.

Description:

RANDOMIZE is used with the RND function to generate random numbers. RND returns a value between 0 and 1 each time you use it. The numbers RND returns and the sequence in which they appear are determined by RANDOMIZE. RANDOMIZE is very useful in those situations (like games) where you want a different series of numbers generated at random.

RANDOMIZE "seeds" the random number generator. This means that you control the numbers that RND returns. When you use a certain RANDOMIZE "seed," RND returns a specific sequence of numbers. This sequence is always the same for the same seed.

If you don't use a RANDOMIZE statement, TI BASIC seeds the random number generator with the same value each time you start your program. You will notice that you always get the same series of numbers from RND.

For example, no matter how many times you run this program, you'll always see the same values in the same order. The random numbers are scaled to values between 0 and 100. Try RUNning the program several times to see what happens.

```
100  FOR I=1 TO 10
110  PRINT I;INT(RND*100);
120  NEXT I
130  END
```

If you used this technique with a guess a number game, you would always get the same numbers in the same order. Not much of a challenge. But, if you're coding a secret message and using random numbers in the coding, you would want to make sure that you could decode the message by getting the same series of random values.

By using RANDOMIZE with a *num-exp*, you can make TI BASIC create a particular series of random numbers. Add this statement to the previous program and see what happens to your series of numbers:

```
90  RANDOMIZE 5
```

A word of warning about seeds. RANDOMIZE uses only the first two bytes of the internal representation of a seed. (See Chapter 5 for technical details on internal number representation.) You might therefore get the same sequence with different seeds. If it's necessary for you to have different sequences, check the numbers you get from the seeds.

Common Errors:

STRING-NUMBER MISMATCH

Your *num-exp* is not a valid number, numeric variable, or numeric expression.

Example 1:

The program in Listing 4-86 prints 10 random numbers scaled between 1 and 100. Then, it uses RANDOMIZE and prints 10 more random numbers.

If you run this program more than once, you'll see that the first 10 numbers are in the same sequence and the second set of 10 numbers is always different because of RANDOMIZE.

```
100  CALL CLEAR
110  PRINT "HERE ARE 10 NUMBERS": :
120  FOR I=1 TO 10
130  PRINT INT(RND*100);
140  NEXT I
150  PRINT
160  PRINT : : "AND NOW USING RANDOMIZE": :
170  RANDOMIZE
180  FOR I=1 TO 10
190  PRINT INT(RND*100);
200  NEXT I
210  PRINT : : :"BYE"
220  END
```

Listing 4-86. RANDOMIZE Example 1

Example 2:

The program in Listing 4-87 uses RANDOMIZE and RND to get a random number that you have to guess. If you take out the RANDOMIZE statement, you'll always get the same numbers (and you can amaze your friends with your good guesses!).

You get to enter a "seed" for the random number generator. Try different values and see what happens. (The same seed will generate the same sequence of random numbers.)

```
100  CALL CLEAR
110  INPUT "ENTER A NUMBER -> ":SEED
120  RANDOMIZE SEED
130  COMP=INT(RND*100)
140  PRINT : :"I HAVE A NUMBER": :
150  TRIES=0
160  INPUT "YOUR GUESS -> ":GUESS
170  TRIES=TRIES+1
180  ON SGN(GUESS-COMP)+2 190,210,260
190  PRINT :"TOO LOW"
200  GOTO 160
210  PRINT : :"YOU GUESSED IT IN";TRIES;"GUESSES."
220  INPUT "PLAY AGAIN? (Y/N) -> ":Y$
230  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 130
240  PRINT : : "BYE."
250  STOP
260  PRINT :"TOO HIGH"
270  GOTO 160
280  END
```

Listing 4-87. RANDOMIZE Example 2

READ	Read information from DATA statements.
Type:	Statement
Format:	[*line#*] READ *variable-list*
Purpose:	READ puts values from DATA statements into the variables in *variable-list*.
Operands:	*line#* is a BASIC statement line number that you need when you include READ in a program. You don't need *line#* when you use READ as a command. *line#* can be any number between 1 and 32767. *variable-list* is a list of string or numeric or both variable names, separated by commas. When you READ data from one or more DATA statements, the values are placed in the variables in *variable-list* in the order in which the variables appear.
Defaults:	None.

Description:

READ assigns values from a DATA statement to the string and numeric variables in *variable-list*. Values are taken from the DATA statements and assigned to the variables from left to right in the order in which they appear in the *variable-list*.

DATA statements are processed sequentially, in line number order. When your program executes a READ statement, it assigns values to the *variable-list* variables beginning with the first value in the lowest numbered DATA statement. If you want to change the order in which the DATA statements are used, you need to use a RESTORE statement.

It's easy to store values in DATA statements and READ them, like this:

```
100  DATA 123,STRING DATA,456.78
110  READ VALUE1
```

```
120  READ STRINGIN$,VALUE2
130  PRINT VALUE1,STRINGIN$,VALUE2
140  END
```

You have to READ numeric data into numeric variables and string data into string variables. Remember that numbers represent valid string data and can be assigned to string variables. It doesn't work the other way. If you try to put string data into a numeric variable, your program stops running and you get the error:

DATA ERROR

You don't need to read all the values in a DATA statement with a single READ statement. You assign as many values as you need with each READ statement.

You cannot, of course, READ when there is no more data left in DATA statements. If you attempt to READ beyond the last item in your DATA statements, you again get the error:

DATA ERROR

DATA and READ statements are often used when you want to initialize variables in your program, especially array variables. You can even store the subscript and data right next to each other, like this:

```
100  DIM TREAS$(50),SCORE(50)
110  DATA 5,"A LARGE GEM",100,10,"A GOLD SWORD",500
120  DATA 22,"TWO MOLDY MARSHMALLOWS",-25
130  DATA 999
140  READ I
150  IF I=999, THEN 180
160  READ TREAS$(I),SCORE(I)
170  GOTO 140
180  continue the program here
```

If you use this technique of reading subscripts and array values, make sure that your subscripts do not go out of range.

Common Errors:

DATA ERROR

You used a READ statement and there are no DATA statements in your program.

Or, there are more variables left in a READ statement's *variable-list* but there is no more data left in DATA statements.

Or, you tried to assign a string value to a numeric variable in the READ *variable-list*.

NUMBER TOO BIG

You read a value into a numeric variable. When TI BASIC stored the value in the variable, an overflow occurred (the value was larger than 9.9999999999999E127).

STRING-NUMBER MISMATCH

You tried to read a string value into a numeric variable.

Example 1:

The program in Listing 4-88 reads the names of the months into an array using READ statements.

Try changing the program to have other information in the DATA statements.

```
100   CALL CLEAR
110   DATA JANUARY,FEBRUARY,MARCH,APRIL,MAY,JUNE
120   DATA JULY,AUGUST,SEPTEMBER,OCTOBER
130   DATA NOVEMBER,DECEMBER
140   DIM MONTH$(12)
150   FOR I=1 TO 12
160   READ MONTH$(I)
170   NEXT I
180   PRINT :"HERE ARE THE MONTHS"
190   FOR K=1 TO 12
200   PRINT "MONTH";K;"IS ";MONTH$(K)
210   NEXT K
220   PRINT : :"DONE"
230   END
```

Listing 4-88. READ Example 1

Example 2:

The program in Listing 4-89 uses READ statements to initialize some string variables. Try changing the program to include other information in the DATA statements.

REM	Include a remark in your program.
Type:	Statement
Format:	[*line#*] REM or [*line#*] REM *string*
Purpose:	REM puts remarks, nonexecutable information, in your program.
Operands:	*line#* is a BASIC statement line number that you need when you include REM in a program. You don't need *line#* when you use REM as a command. *line#* can be any number between 1 and 32767. *string* is any information that you want to have in your program that is not an executable statement. REM stands for REMARK. What you put in *string* is your remark.
Defaults:	TI BASIC uses a null string if you don't supply a *string*.

```
100  CALL CLEAR
110  DATA "HELLO THERE, ","HOW ARE YOU,  "
120  DATA GOODBYE,"SEE YOU SOON."
130  READ GREET1$,GREET2$
140  READ BYE1$,BYE2$
150  RANDOMIZE
160  USE=1
170  IF RND<.5 THEN 190
180  USE=2
190  INPUT "WHAT'S YOUR NAME? ":NAME$
200  IF USE=2 THEN 230
210  PRINT GREET1$;NAME$
220  GOTO 240
230  PRINT GREET2$;NAME$
240  IF USE=2 THEN 270
250  PRINT : : :BYE1$;"   ";NAME$
260  STOP
270  PRINT : : :BYE2$;"   ";NAME$
280  END
```

Listing 4-89. READ Example 2

Description:

REM lets you include remarks or comments (nonexecutable statements) in a TI BASIC program. REMarks can tell what your program is doing, how it operates, and what your variables are.

TI BASIC does not try to execute REM statements. It ignores the *string* after the REM keyword. *string* can be up to 112 characters of information, including spaces and special characters. You don't need any double quotes around *string*.

REM statements without any *string* are used to make your program easier to read, like this:

```
100  REM CHECKBOOK BALANCER
110  REM
120  REM GET INITIAL BALANCE, ALL DEPOSITS
130  REM THEN GET ALL CHECKS
140  REM
150  REM FINAL BALANCE = INITIAL BALANCE + DEPOSITS – CHECKS
160  REM
```

You can put any nonprogram information in REM statements in your TI BASIC programs. You might want to keep track of when you wrote the program and use something like this:

```
100  REM PROG WRITTEN ON 6/10/83
100  REM PGM WRITTEN BY AMC
250  REM NEW MENU ITEM (REVISE VALUES) ADDED 7/25/83
```

Use REM statements whenever your program performs some processing that you might not easily remember, in case you have to go back to change it later. Though it may seem like a lot of trouble when you are writing the program, you will appreciate the effort you took to include these comments when you try, months later, to correct a program or add some new features to your program.

One caution about REM statements. Each character in a REM statement takes one byte of memory. When your program gets large, you may have to shorten your REMarks. If possible, don't remove them, just make them shorter.

Common Errors:

None.

Example 1:

The program in Listing 4-90 uses REM statements to tell you what is going on in the program. Add more REMarks.

```
100  REM CLEAR THE SCREEN
110  CALL CLEAR
120  REM GET THE PERSON'S NAME
130  INPUT "WHAT'S YOUR NAME? -> ":NAME$
140  REM IF NO NAME IS ENTERED, ASK AGAIN
150  IF LEN(NAME$)=0 THEN 130
160  REM
170  REM PRINT A GREETING
180  REM
190  PRINT "HELLO ";NAME$
200  REM DONE
210  END
```

Listing 4-90. REM Example 1

Example 2:

The program in Listing 4-91 plays guess a number. REM statements tell you what is happening at each stage in the program.

RESEQUENCE or RES	Resequence the lines in a program.
Type:	Command
Format:	RESEQUENCE or RES or RES *initial* or RES *initial ,incr* or RES *,incr*

RESEQUENCE or RES	Resequence the lines in a program. *(continued)*
Purpose:	RES renumbers the lines in your BASIC program.
Operands:	*line#* is a BASIC statement line number that you need when you include RESEQUENCE in a program. You don't need *line#* when you use RESEQUENCE as a command. *line#* can be any number between 1 and 32767. *initial* is a number that specifies the new line number for the first line in the resequenced program. *initial* can be any value between 1 and 32767. *incr* is a number that specifies the increment added to each line number to get the next line number. *incr* can be any value between 1 and 32767.
Defaults:	If you don't use an *initial* value, TI BASIC uses 100; if you don't use an *incr* value, TI BASIC uses 10.

Description:

RES or RESEQUENCE renumbers the lines in the TI BASIC program currently in memory.

```
100  REM CLEAR THE SCREEN
110  CALL CLEAR
120  REM GET THE SEED FOR THE RANDOM NUMBER GENERATOR
130  INPUT "ENTER A NUMBER -> ":SEED
140  RANDOMIZE SEED
150  REM
160  REM GET A RANDOM NUMBER AND SCALE IT BETWEEN 1 AND 100
170  REM
180  COMP=INT(RND*100)
190  PRINT : :"I HAVE A NUMBER": :
200  REM BEGIN THE GAME
210  TRIES=0
220  REM
230  REM  GET A GUESS AND COUNT THE TRIES
240  REM
250  INPUT "YOUR GUESS -> ":GUESS
260  TRIES=TRIES+1
270  REM  IS IT RIGHT?
280  REM  SGN IS -1 IF NEGATIVE (GUESS TOO HIGH)
290  REM  0 IF 0  (GUESS CORRECT),
300  REM   +1 IF POSITIVE  (GUESS TOO LOW)
310  ON SGN(GUESS-COMP)+2 320,340,420
320  PRINT :"TOO LOW"
330  GOTO 250
340  PRINT : :"YOU GUESSED IT IN";TRIES;"GUESSES."
350  REM
360  REM  SEE IF ANOTHER GAME
370  REM
380  INPUT "PLAY AGAIN? (Y/N) -> ":Y$
390  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 180
400  PRINT : : "BYE."
410  STOP
420  PRINT :"TOO HIGH"
430  GOTO 250
440  END
```

Listing 4-91. REM Example 2

When TI BASIC resequences your program, it uses *initial* for the line number of the first statement and increases the line number by *incr* for each successive statement. If you don't specify a value for *initial*, TI BASIC uses 100. If you don't specify an *incr* value, TI BASIC uses 10.

To renumber the lines in your program so that the line numbers begin at 100 and increase by 10, use:

```
RES
```

Suppose you want your program's line numbers to begin at 600 and increase by 15. Then use:

```
RES 600,15
```

To begin your program at line 1200 and increment the line numbers by 10, enter:

```
RES 1200
```

To renumber your program using whatever value you already have for the first line number and increasing the line numbers by 20, you use only the *incr* operand. Remember to use the comma (,) before the *incr* or else TI BASIC will think you are specifying the *initial* operand:

```
RES,20
```

You can use any combination of values that you want for the RES operands. If you are working on a program and expect to be adding a lot of new lines, try:

```
RES 100,50
```

Then, after you add your statements between the current program lines, resequence your program with:

```
RES
```

Line numbers that appear as operands in other statements (such as GOTO and GOSUB) are adjusted to reflect the new values. Line numbers in REM statements are not changed.

If you have a statement with a line number operand and the line number is not a line number of a statement in your program, TI BASIC substitutes 32767 for the unreferenced, resequenced line number. You will not get an error when the line is resequenced. You will get an error when you RUN your program and it executes the statement with the unreferenced line number.

RES is very handy when you enter a program and make changes to it, then want a neat listing with nicely sequenced line numbers. When you enter and debug a program, you often add statements between other statements (such as adding lines 151, 155, and 157 between 150 and 160) or delete statements.

Common Errors:

BAD LINE NUMBER

You resequenced your program and you got a line number that is greater than 32767. This usually happens when you use a large *incr* value and have a very large program.

CAN'T DO THAT

You tried to use RES as a statement in a program. RES can be used only as a command.

Example 1:

The example in Listing 4-92 shows you how RES works. First, you enter a small program (using the NUM command to get line numbers starting at 1000 and incremented by 100). Then, use the RES command to resequence the program to start at 100 and increment by 10.

Try different starting values and increments.

<ENTER> means press the ENTER key.

```
NEW <ENTER>
NUM 1000,100 <ENTER>
1000 CALL CLEAR
1100 PRINT "HELLO"
1200 INPUT "WHAT'S YOUR NAME? ":NAME$
1300 PRINT : :"SEE YOU LATER, ";NAME$
1400 END
1500 <ENTER>
RUN <ENTER>
     Your computer prints "HELLO", asks for your
     name, and tells you goodbye.
RES <ENTER>
LIST <ENTER>
100 CALL CLEAR
110 PRINT "HELLO"
120 INPUT "WHAT'S YOUR NAME? ":NAME$
130 PRINT : :"SEE YOU LATER, ";NAME$
140 END
```

Listing 4-92. RES Example 1

Example 2:

The example in Listing 4-93 shows you how RES adjusts line numbers in GOTO statements. First you enter a program that plays guess a number.

You realize that you forgot to tell what range the number can be in and you insert two lines between 180 and 190. Then, to get the program neatened up, you RES. But you also want the program to start at line 500. You will see that all the line numbers are adjusted to their correct values.

```
NUM <ENTER>
100  CALL CLEAR
110  INPUT "WHAT'S YOUR NAME? ":NAME$
120  PRINT : :"HELLO ";NAME$
130  INPUT "WANT TO GUESS A NUMBER? (Y/N) -> ":Y$
140  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 170
150  PRINT : : :"GOODBYE"
160  STOP
170  RANDOMIZE
180  COMP=INT(RND*100)
190  TRIES=0
200  INPUT "YOUR GUESS-> ":GUESS
210  TRIES=TRIES+1
220  IF GUESS<>COMP THEN 260
230  PRINT :"CONGRATULATIONS.":"YOU TOOK";TRIES;"TURNS."
240  INPUT "PLAY AGAIN (Y/N) -> ":Y$
250  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 170
260  IF GUESS>COMP THEN 290
270  PRINT :"TOO LOW"
280  GOTO 200
290  PRINT :"TOO HIGH"
300  GOTO 200
310  END
320  <ENTER>
182  PRINT : "I HAVE A NUMBER BETWEEN 1 AND 100."
185  PRINT : :
RES 500,10 <ENTER>
LIST <ENTER>
500  CALL CLEAR
510  INPUT "WHAT'S YOUR NAME? ":NAME$
520  PRINT : :"HELLO ";NAME$
530  INPUT "WANT TO GUESS A NUMBER? (Y/N) -> ":Y$
540  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 570
550  PRINT : : :"GOODBYE"
560  STOP
570  RANDOMIZE
580  COMP=INT(RND*100)
590  PRINT : "I HAVE A NUMBER BETWEEN 1 AND 100."
600  PRINT : :
610  TRIES=0
620  INPUT "YOUR GUESS-> ":GUESS
630  TRIES=TRIES+1
640  IF GUESS<>COMP THEN 680
650  PRINT :"CONGRATULATIONS.":"YOU TOOK";TRIES;"TURNS."
660  INPUT "PLAY AGAIN (Y/N) -> ":Y$
670  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 570
680  IF GUESS>COMP THEN 710
690  PRINT :"TOO LOW"
700  GOTO 620
710  PRINT :"TOO HIGH"
720  GOTO 620
730  END
```

Listing 4-93. RES Example 2

RESTORE	Reset a DATA statement.
Type:	Statement
Format:	[*line#*] RESTORE or [*line#*] RESTORE *line-num*
Purpose:	RESTORE sets the line number of the DATA statement used by the next READ statement in your program.
Operands:	*line#* is a BASIC statement line number that you need when you include RESTORE in a program. You don't need *line#* when you use RESTORE as a command. *line#* can be any number between 1 and 32767. *line-num* is the line number of a DATA statement. Data from this DATA statement is used the next time your program executes a READ statement. *line-num* can be any number between 1 and 32767.
Defaults:	If you don't give a value for *line-num*, BASIC uses the line number of the first DATA statement in your program.

Description:

RESTORE works with DATA and READ statements in your program. RESTORE sets the line number *(line-num)* for the DATA statement used by the next READ statement in your program.

NOTE

You can use RESTORE as a command only when you have a BASIC program already in your computer's memory.

DATA statements store data in your program and READ statements put the data from DATA statements into your program's variables. See the sections describing DATA and READ statements for details on these statements.

When you use a RESTORE statement, you tell BASIC that you want the information in the DATA statement with line number *line-num* to be used when your program gets to the next READ statement. This means you can store several different sets of data and, depending on program variables, read any one of the different sets.

Once data is read from a DATA statement, you cannot reread it unless you RESTORE the DATA statement.

If you want to restart the program without stopping and RUNning it again, simply include a RESTORE statement in one of these places:

- At the beginning of the program's initialization processing section
- Somewhere in the section where you find out if you want to restart the program

There are many ways to use RESTORE statements. You might want to store different error messages as string data in your DATA statements. Then, depending on what type of error you detect, you can RESTORE to the appropriate DATA statement and use a READ statement to get the right message.

You can place DATA statements anywhere in your program so you can put different information at different places. You can see in Example 1 (below) that the information to be read is stored in DATA statements placed right before the READ statement for the variables. Remember, DATA statements only store information to be used by READ statements.

Another common use for RESTORE statements is shown in Example 2 (below) where different forms of the same information are stored in DATA statements. Depending on what you want to do when you run the program, you can get different names printed (abbreviations or full names). Your choice determines which RESTORE statement gets used.

NOTE

If you use a *line-num* that is not the line number for a DATA statement in your program, BASIC uses the first DATA statement following *line-num* when the next READ statement is reached.

Common Errors:

BAD LINE NUMBER

You entered a *line#* or *line-num* that is less than 1 or greater than 32767.

DATA ERROR

You use a *line-num* that is larger than the largest line number in your program.

Example 1:

The program in Listing 4-94 uses RESTORE statements to read information from selected DATA statements. You can read words, numbers, or letters, depending on which DATA statement is used. Notice that the DATA statements are placed near the READ statements that use the information in the DATA statements.

This program uses a *menu* to let you select which type of data you want to read. The program tells you what the valid answers are for each question. If you make an invalid choice from the menu, the program reminds you of the valid choices and gives you another chance.

Once you have read data and printed what has been read, you can read other data by selecting menu choices 1 through 3, or stop by selecting menu choice 4.

Example 2:

The program in Listing 4-95 uses DATA statements to store different forms of the same information and uses RESTORE statements to decide which form to use.

```
100  REM RESTORE STATEMENT EXAMPLE
110  CALL CLEAR
120  PRINT : :"SELECT THE TYPE OF":" DATA TO READ"
130  PRINT "1 WORDS":"2 NUMBERS":"3 LETTERS":"4 STOP"
140  PRINT : :
150  INPUT "YOUR CHOICE (1-4) -> ":CHOICE
160  IF (CHOICE<1)+(CHOICE>4) THEN 350
170  ON CHOICE GOTO 190,240,290,340
180  DATA FIRST,SECOND,THIRD,FOURTH
190  RESTORE 180
200  READ A$,B$,C$,D$
210  PRINT "YOU CHOSE THESE WORDS":A$,B$,C$,D$
220  GOTO 320
230  DATA 100,200,300,400
240  RESTORE 230
250  READ A,B,C,D
260  PRINT "YOU CHOSE THESE NUMBERS":A,B,C,D
270  GOTO 320
280  DATA A,B,C,D
290  RESTORE 280
300  READ A$,B$,C$,D$
310  PRINT "YOU CHOSE THESE LETTERS":A$,B$,C$,D$
320  INPUT "TRY AGAIN? (Y/N) -> ":Y$
330  IF (Y$="Y")+(Y$="y") THEN 120
340  STOP
350  PRINT "PLEASE PICK A CHOICE":"BETWEEN 1 AND 4"
360  GOTO 130
370  END
```

Listing 4-94. RESTORE Example 1

The program prints a date using either the month or an abbreviation for the month. You decide which to use when you run the program.

You get the chance to change the format after processing each date.

RESTORE #	Reset a file.
Type:	Statement
Format:	[*line#*] RESTORE *#file-num* or [*line#*] RESTORE *#file-num* REC *rec-num*
Purpose:	RESTORE # is used with files. It sets the record to be processed by the next PRINT # or INPUT # statement (to the file opened as *file-num*) in your program.
Operands:	*line#* is a BASIC statement line number that you need when you include RESTORE # in a program. You don't need *line#* when you use RESTORE # as a command. *line#* can be any number between 1 and 32767. *file-num* is the file number used in the OPEN statement for the file you want to RESTORE. *file-num* may be any number between 1 and 255. *rec-num* is a number, numeric variable, or numeric expression that is evaluated and used as the pointer to a specific record in a RELATIVE file. (RELATIVE files cannot be cassette files.)
Defaults:	If you are using a RELATIVE file and you do not supply a value for *rec-num*, TI BASIC uses record 0.

```
100  REM FIRST THE FULL MONTHS
110  DATA JANUARY, FEBRUARY, MARCH, APRIL
120  DATA MAY, JUNE, JULY, AUGUST
130  DATA SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
140  REM NOW,  THE ABBREVIATIONS
150  DATA JAN, FEB, MAR, APR, MAY, JUN
160  DATA JUL, AUG, SEP, OCT, NOV, DEC
170  DIM MONTH$(12)
180  CALL CLEAR
190  PRINT : :"DO YOU WANT TO USE":
     "ABBREVIATIONS FOR THE"
200  INPUT "MONTHS? (Y/N) -> ":Y$
210  IF (Y$="Y")+(Y$="y") THEN 240
220  IF (Y$="N")+(Y$="n") THEN 270 ELSE 180
230  REM READ EITHER ABBREVIATIONS OR FULL MONTHS
240  REM USE THE ABBREVIATIONS
250  RESTORE 150
260  GOTO 290
270  REM USE FULL MONTHS
280  RESTORE 110
290  REM READ THE MONTHS
300  FOR I=1 TO 12
310  READ MONTH$(I)
320  NEXT I
330  REM GET THE DATE
340  PRINT : :"ENTER THE DATE AS":"MONTH,DAY,YEAR"
350  PRINT : "LIKE THIS:":"12,25,1983": : :
360  INPUT "YOUR DATE -> ":MON,DAY,YEAR
370  IF (MON<1)+(MON>12)+(DAY<1)+(DAY>31) THEN 490
380  OUT$=MONTH$(MON)&" "&STR$(DAY)&", "&STR$(YEAR)
390  PRINT "YOUR DATE IS: ";OUT$
400  INPUT "TRY AGAIN? (Y/N) -> ":Y$
410  IF (Y$="Y")+(Y$="y") THEN 440
420  IF (Y$="N")+(Y$="n") THEN 510 ELSE 400
430  REM DO YOU WANT TO CHANGE TO/FROM ABBREVIATIONS?
440  PRINT : :"CHANGE MONTH TO/FROM"
450  INPUT "ABBREVIATIONS? (Y/N) -> ":Y$
460  IF (Y$="Y")+(Y$="y") THEN 190
470  IF (Y$="N")+(Y$="n") THEN 340 ELSE 440
480  REM TELL THAT THE DATE'S FORMAT IS WRONG
490  PRINT :"PLEASE ENTER A DATE":"THAT IS REASONABLE"
500  GOTO 340
510  END
```

Listing 4-95. RESTORE Example 2

Description:

RESTORE # is used only with files. RESTORE # tells TI BASIC which record in the file (that you OPENed as *file-num*) is to be processed by the next PRINT # or INPUT # statement.

RESTORE # resets the pointer that TI BASIC uses to tell where it is in a file. When you use a file called *file-num* in your program, BASIC knows if it is at the beginning or at the end of the file, or somewhere in the middle. The next time you read from the file (with INPUT #) or write to the file (with PRINT #), TI BASIC reads or writes the information at the next available record.

NOTE

RESTORE # can be used only on files opened for INPUT or UPDATE processing.

When you RESTORE # a file, TI BASIC sets its pointer to the beginning of the file, to the first record. After you RESTORE # a file, you can reread the file (with INPUT # statements) or write over the file (with PRINT # statements). RESTORE # lets you, at any time, reposition to the beginning of your cassette or disk file.

NOTE

When you use a RESTORE # with a cassette file, TI BASIC does not tell you to stop the recorder and rewind the tape. You must do this yourself before you RESTORE # the file. If you do not stop and rewind the tape, TI BASIC will continue "reading" the tape.

TI BASIC also uses RELATIVE files, where each record has its own unique identifier. REC lets you position anywhere in a RELATIVE file—at record *rec-num*. If you use REC but do not give a value for *rec-num*, TI BASIC points to record 0, or the beginning of the RELATIVE file. Remember to include the comma (,) before the REC operand. REC cannot be used with cassette files.

If you are reading from a RELATIVE file with INPUT # statements, the next INPUT # will read record *rec-num*. If you're writing to a RELATIVE file with PRINT # statements, the next PRINT # will write to record *rec-num*.

Common Errors:

BAD VALUE

You used a *file-num* that is less than zero or greater than 255.
You forgot the # (number sign) before the *file-num* operand.

FILE ERROR

You tried to RESTORE # to a file *(file-num)* that isn't open. This happens when you forget to open a file (with an OPEN statement) or when you close a file (with a CLOSE statement) before you RESTORE # it.

I/O ERROR 43

You used RESTORE # on a file opened for INPUT.

STRING-NUMBER MISMATCH

You used a string instead of a number for the *file-num* operand.

Example 1:

The program in Listing 4-96 uses RESTORE # to position at the beginning of a cassette file to reread the data when restarting the program.

```
100  OPEN #5:"CS1",INTERNAL,INPUT,FIXED 64
110  INPUT #5:STRING$
120  IF STRING$="XXX" THEN 150
130  PRINT STRING$
140  GOTO 110
150  REM RE-START THE PROGRAM?
160  INPUT "RE-READ THE TAPE? (Y/N) -> ":Y$
170  IF (Y$="Y")+(Y$="y") THEN 200
180  IF (Y$="N")+(Y$="n") THEN 190 ELSE 160
190  CLOSE #5
200  STOP
210  RESTORE #5
220  GOTO 110
230  END
```

Listing 4-96. RESTORE # Example 1

Example 2:

The program in Listing 4-97 uses RESTORE # to restore to the beginning of a file on disk 1 called "DATAFILE" and writes over any information written on the disk.

The first 100 numbers are random numbers between 1 and 100. You will not see any number greater than 100. Then, after using RESTORE #, 100 more random numbers are printed to the file, overwriting the 100 already there. The second set of random numbers is scaled between 1 and 1000.

If you INPUT # and PRINT the contents of the disk data file, you will see numbers greater than 100, showing you that the second set of random numbers overwrote the first set.

```
100  OPEN #20:"DSK1.DATAFILE",INTERNAL,
     VARIABLE 254,UPDATE
110  CALL CLEAR
120  FOR I=1 TO 10
130  A=INT(RND*100)
140  PRINT #20: A
150  PRINT A;
160  NEXT I
170  RESTORE #20
180  PRINT : : :"NEXT LOOP": :
190  FOR I=1 TO 10
200  A=INT(RND*1000)
210  PRINT #20: A
220  PRINT A;
230  NEXT I
240  CLOSE #20
250  PRINT : :"DONE"
260  END
```

Listing 4-97. RESTORE # Example 2

RETURN	Return from a subprogram.
Type:	Statement
Format:	*line#* RETURN
Purpose:	RETURN in a subprogram transfers program control to the statement after the GOSUB or ON . . . GOSUB that called the subprogram.
Operands:	*line#* is a BASIC statement line number. *line#* can be any number between 1 and 32767.
Defaults:	None.

Description:

When you transfer control to a subprogram ("call a subprogram") with a GOSUB statement, you use a RETURN statement to go back to the statement after the GOSUB. This technique lets you write small sections of code that do specific tasks. Then, you GOSUB to the section. When the task is complete, you RETURN to a "higher level" in the program.

You can use more than one RETURN statement in a single subprogram. If your subprogram's logic needs several different ways to exit back to where it was called, you can use several RETURN statements. Example 2 (below) shows one subprogram with several RETURNs.

You cannot use a RETURN unless you are in a subprogram that you "called" with a GOSUB or ON . . . GOSUB statement.

NOTE

RETURN must be used with a GOSUB or ON . . . GOSUB statement.

Common Errors:

CAN'T DO THAT

You tried to use a RETURN statement as a command (without a *line#*). RETURN statements can be used only in subprograms.

You used a RETURN without a previous GOSUB or ON GOSUB statement.

MEMORY FULL

This error happens only during program execution when there are too many branches to the subprograms or when a GOSUB calls itself. That means there are too many GOSUBs that have not RETURNed.

INCORRECT STATEMENT

You put a word or character after the word RETURN in your BASIC statement.

Example 1:

The program in Listing 4-98 uses RETURN in two subprograms. The first subprogram prints a greeting. The second subprogram waits until you press any key before it returns. (This is a useful way to make your program wait before writing to the screen.)

```
100   REM USE SUBPROGRAMS AND RETURNS
110   CALL CLEAR
120   PRINT "USING FIRST GOSUB"
130   GOSUB 180
140   PRINT :"JUST AFTER THE RETURN": : :
150   GOSUB 230
160   PRINT :"BACK AGAIN.  GOODBYE"
170   STOP
180   REM SUBPROGRAM THAT GREETS YOU
190   PRINT : :"HELLO THERE."
200   PRINT "I'M YOUR SUBPROGRAM."
210   PRINT "NOW, I'LL RETURN.": :
220   RETURN
230   REM SUBPROGRAM THAT PAUSES
240   REM UNTIL YOU PRESS A KEY
250   PRINT "PRESS ANY KEY TO CONTINUE."
260   CALL KEY (0,KY,ST)
270   IF ST=0 THEN 260
280   RETURN
290   END
```

Listing 4-98. RETURN Example 1

Example 2:

The program in Listing 4-99 uses RETURN at three different places in a single subprogram. If you want to use a single RETURN statement, you can GOTO the RETURN statement instead of using several RETURNs.

```
100   REM SHOW SEVERAL RETURNS IN ONE SUBPROGRAM
110   CALL CLEAR
120   INPUT "ENTER A NUMBER (0 TO END)->":NUMBER
130   IF NUMBER<>0 THEN 160
140   PRINT : : "GOODBYE."
150   STOP
160   GOSUB 180
170   GOTO 120
180   REM RETURN FROM DIFFERENT PLACES, DEPENDING
190   REM ON THE SIGN (+/-) OF NUMBER
200   IF NUMBER<0 THEN 230
210   PRINT :"YOUR NUMBER WAS POSITIVE."
220   RETURN
230   PRINT :"YOUR NUMBER WAS NEGATIVE."
240   RETURN
250   END
```

Listing 4-99. RETURN Example 2

RND Get a random number.

Type: Function
Format: RND
Purpose: RND generates a pseudo-random number that is greater than or equal to zero and less than one.
Operands: None.
Defaults: None.

Description:

RND generates a random number between 0 and 1. The computer picks a number greater than 0 and less than 1 and uses it in place of RND.

The RND function actually generates a series of numbers. The numbers in this series are always the same unless you use the RANDOMIZE statement. See the section on RANDOMIZE for details.

You can assign the random number to a variable like this:

```
NUMBER = RND
```

Or, you can use RND in place of a variable, like this:

```
VALUE = 7 * RND
```

RND is often used in programming games where you want to have something happen by chance. Once a value is chosen through RND, you can decide whether or not an event should occur depending on the value. For example, if you want something to happen 75% of the time, you can compare the value of RND to 0.75 like this:

```
IF RND<=.75 THEN 500
```

This statement says that the code at line 500 will be executed only when the random number generated by RND is less than or equal to 0.75. Since there is an equal chance of getting any number between 0 and 1, the code at line 500 will be executed approximately 75% of the time you check RND.

RND is also useful when you want to make a choice of several alternatives at random. If you have four possible choices, you can use this code to generate a random choice:

```
100  CHOICE=INT(4*RND)+1
110  ON CHOICE GOSUB 1000,2000,3000,4000
```

Multiplying RND by 4 scales the value to between 0 and 3.999 (since RND is more than 0 and less than 1). Taking the integer value of the result (see the section on INT for details) gives you a number between 0 and 3. But, you want a number between 1 and 4. Just add 1 to the result and you can use the answer in an ON . . . GOSUB statement.

Common Errors:

None.

Example 1:

The program in Listing 4-100 uses RND to generate 20 random numbers. Then, after a RANDOMIZE statement is used to generate unpredictable numbers, 20 more random numbers are printed.

If you run this program more than once, you will get the same sequence of numbers each time for the first set of 20 numbers and a different sequence for the second set of 20 numbers.

```
100  CALL CLEAR
110  PRINT "20 RANDOM NUMBERS"
120  FOR I=1 TO 20
130  PRINT RND;
140  NEXT I
150  RANDOMIZE
160  PRINT :"NOW, 20 DIFFERENT NUMBERS"
170  FOR I=1 TO 20
180  PRINT RND;
190  NEXT I
200  END
```

Listing 4-100. RND Example 1

Example 2:

The program in Listing 4-101 uses RND to get the number that the computer uses for the old favorite "guess a number."

```
100  CALL CLEAR
110  RANDOMIZE
120  NUMBER=INT(RND*100)
130  PRINT "I HAVE A NUMBER":"  BETWEEN 1 AND 100"
140  N=0
150  INPUT "YOUR GUESS -> ":GUESS
160  N=N+1
170  IF GUESS<>NUMBER THEN 230
180  PRINT "YOU GUESSED IT IN";N;" TRIES!"
190  INPUT "TRY AGAIN? (Y/N) -> ":Y$
200  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
210  PRINT "GOODBYE"
220  STOP
230  IF GUESS<NUMBER THEN 260
240  PRINT "YOUR GUESS IS TOO HIGH."
250  GOTO 150
260  PRINT "YOUR GUESS IS TOO LOW."
270  GOTO 150
280  END
```

Listing 4-101. RND Example 2

Remember, RANDOMIZE makes a different sequence of numbers. Try this program with the RANDOMIZE a few times and write down the numbers. Then, remove the RANDOMIZE statement (simply omit statement 110 or make it a REM) and see what happens. If you run it several times without RANDOMIZE, you will get the same sequence of random numbers each time.

RUN	Run the program in memory.
Type:	Command
Format:	RUN or RUN *line-num*
Purpose:	RUN executes the BASIC program currently stored in memory. If you use *line-num*, the BASIC program starts executing at that line number.
Operands:	*line-num* is the line number of one of the statements in the BASIC program currently in memory. *line-num* can be any number between 1 and 32767.
Defaults:	If you don't supply a *line-num* value, TI BASIC starts with the first statement in the BASIC program currently in memory.

Description:

Once you have a BASIC program in your computer's memory, you use a RUN statement to execute it.

NOTE

You can only use RUN as a command. You cannot use RUN in a program.

RUN executes the BASIC statements beginning at the one with the line number *line-num*. If you use RUN without a *line-num*, the program in memory begins executing at its first program line.

The *line-num* operand lets you start a program at any line. Be careful when you use *line-num*. If your program is dimensioning an array to more than 10 elements (see the section on DIM for details) and *line-num* is past the DIM statement, you will get an error if you use an array element past 10.

More cautions for using *line-num*. If you are setting values in an early part of your program and you start past these statements, your program will not set the values. You may not get the results you expect. If you start at a *line-num* that is in a subprogram, you will get an error when your program attempts to RETURN.

Common Errors:

```
CAN'T DO THAT
```

You tried to use RUN in a program. You can only use RUN as a command.

You entered a RUN command and there is no BASIC program in your computer's memory.

Example 1:

The example in Listing 4-102 first asks you to enter a small program then RUNs it in several ways. <ENTER> means press the ENTER key.

The first time the program is RUN, the screen gets cleared and the two messages are printed. The second RUN statement begins executing at line 120. Notice that the screen is not cleared and the first line is not printed.

The third RUN statement begins executing at line 110. The screen is not cleared and two lines are printed. The fourth RUN statement begins executing at the beginning of the program, clears the screen, and prints both lines.

```
NUM <ENTER>
100  CALL CLEAR
110  PRINT "HI THERE."
120  PRINT "HOW ARE YOU."
130  END
<ENTER>
RUN <ENTER>
RUN 120 <ENTER>
RUN 110 <ENTER>
RUN <ENTER>
```

Listing 4-102. RUN Example 1

Example 2:

The example in Listing 4-103 shows you how to load a program from a cassette and run it.

Use your cassette and load any BASIC program. Then enter RUN to execute it.

```
OLD CS1 <ENTER>
RUN <ENTER>
```

Listing 4-103. RUN Example 2

SAVE	Save a program.
Type:	Command
Format:	SAVE *device* or SAVE *device.filename*
Purpose:	SAVE stores the BASIC program currently in your computer's memory on *device*. When you store your program on a device other than a cassette (CS1 or CS2), you must give a name for the file on the disk or other device.
Operands:	*device* is a string, string variable, or string expression that contains the name of the device where you want to store the program. *device* is CS1 (cassette 1), CS2 (cassette 2), DSK1 (disk drive 1), DSK2 (disk drive 2), DSK3 (disk drive 3), or another device, such as, a Hexbus peripheral.

SAVE Save a program. *(continued)*

filename is a string, string variable, or string expression that contains the name of the file on *device* where you are storing your BASIC program. You can use up to 10 characters in your disk filenames. You cannot use a *filename* for a cassette file.

Defaults: None.

Description:

SAVE writes the BASIC program currently in your computer's memory to the *device*. *device* can be any of the values shown in Table 4-37. SAVEd programs are read back into your computer's memory through OLD commands.

Table 4-37. SAVE *device* Operand

device	Meaning
CS1	Cassette recorder 1 (the lead with the three connectors if you have a dual cassette cable). You can both read from and write to CS1.
CS2	Cassette recorder 2 (the lead with two connectors if you have a dual cassette cable). You can use CS2 only with dual cables. You can only write to CS2.
DSK1	Disk drive 1. You also need a filename.
DSK2	Disk drive 2. You also need a filename.
DSK3	Disk drive 3. You also need a filename.
HEXBUS1.	Hexbus peripheral 1. You also need a filename.

You do not need a *filename* when you SAVE to a cassette tape since your cassette does not name its files. Your computer has no way to tell what is on a cassette tape. You have to keep track of what's where on each tape. To save the program that is currently in your computer's memory to a cassette tape, on cassette recorder 1, just enter:

```
SAVE CS1
```

TI BASIC will print instructions about rewinding the tape, pressing RECORD and STOP, and if you want to check what has been written. You don't have to rewind the tape unless you want to put your program at the beginning of the tape. Just keep track of what you put on each tape. Keep a log of the tape number, the tape counter (if you have one on your recorder), and what the program is.

You can check whether the program has been written to the cassette tape correctly. You should do this the first few times you use your recorder to make sure that you have the volume and tone set correctly.

NOTE

You must enter upper-case letters to answer the questions that TI BASIC asks during writing to a cassette. Either hold the SHIFT key and the letter or make sure the ALPHA LOCK key is down.

It is different with other devices, such as disks or Hexbus Wafertapes. These devices associate a *filename* with each file and they keep a *directory* of what files are on a disk or Wafertape. When you SAVE your program to a disk or Wafertape, you also need a *filename*.

To save your BASIC program to the diskette that is on disk drive 1 and put it into the file called "MYPROGRAM", you use:

```
SAVE DSK1.MYPROGRAM
```

Unlike writing to a cassette tape, you do not check whether the program has been correctly written to a disk. The disk controller makes sure that it writes what it is supposed to. You don't do anything except SAVE your program.

There is another very good use for SAVE besides storing entire programs—storing your program regularly while you are entering the statements.

When you're entering a large program, you should make it a practice to regularly SAVE the program, unfinished as it may be, to either cassette tape or disk. (We often SAVE at the end of a section or subprogram.) Then, if you should somehow lose what is in memory, you can restore the last copy that you saved.

You can SAVE the intermediate program to the same part of a cassette tape or to the same disk file. It doesn't matter. What you are really doing is protecting yourself from having to re-enter many lines of code. When you're finished, you can SAVE your program to its final cassette tape or disk file. Remember to make a copy and store it somewhere. Particularly if it is an important program that would be difficult to re-enter if you should lose your only copy.

NOTE

Remember to have your cassette recorder connected to your computer or your disk drives turned on BEFORE you turn on your computer.

Common Errors:

```
CAN'T DO THAT
```

You tried to use a SAVE command as a statement in a program.

You entered a SAVE command and you don't have any BASIC program in your computer's memory yet.

```
I/O ERROR 60
```

You gave a *device* that is not available. You may have spelled the *device* incorrectly (like C1S instead of CS1) or you may have used a disk and not turned on the disk drives.

```
I/O ERROR 63
```

There wasn't enough memory available to allocate to the Input/Output buffer that was needed to SAVE your program.

I/O ERROR 66

A device error occurred. This often happens when you accidentally disconnect or turn off a device while a program is running. You will not be able to re-connect the device without turning off your computer and losing your program. Try to save the program to another (connected) device if possible.

Example 1:

The example in Listing 4-104 shows you how to enter a short program and save it to a cassette tape. <ENTER> means press the ENTER key.

```
NUM <ENTER>
100  CALL CLEAR
110  PRINT "HI THERE"
120  END
<ENTER>
SAVE CS1 <ENTER>
```

Listing 4-104. SAVE Example 1

Example 2:

The example in Listing 4-105 shows you how to enter a short program and save it to a disk file called "MYPROG" on the diskette on disk drive 1. <ENTER> means press the ENTER key.

```
NUM <ENTER>
100  CALL CLEAR
110  PRINT "HI THERE"
120  END
<ENTER>
SAVE DSK1.MYPROG <ENTER>
```

Listing 4-105. SAVE Example 2

CALL SCREEN	Change the screen color.
Type:	Statement
Format:	[*line#*] CALL SCREEN *(color-code)*
Purpose:	SCREEN changes the background color of the screen to the color represented by *color-code*.
Operands:	*line#* is a BASIC statement line number that you need when you include CALL SCREEN in a program. You don't need *line#* when you use CALL SCREEN as a command. *line#* can be any number between 1 and 32767. *color-code* is a number, numeric variable, or numeric expression that defines the screen color. *color-code* can be any value between 1 and 16 as shown in Table 4-38.
Defaults:	None.

Description:

SCREEN changes the color of your television screen from the standard color you get when you are running a TI BASIC program (light green or 4) to any color shown in Table 4-38.

The screen color is a *background color* that shows through when you select transparent (*color-code* 1) as the color of something on the screen. For example, the characters in TI BASIC are defined as black on transparent so the light green screen color shows through the transparent part of the character.

Table 4-38. SCREEN *color-code* Values

color-code	Color
1	Transparent
2	Black
3	Medium Green
4	Light Green
5	Dark Blue
6	Light Blue
7	Dark Red
8	Cyan
9	Medium Red
10	Light Red
11	Dark Yellow
12	Light Yellow
13	Dark Green
14	Magenta
15	Gray
16	White

When you change the screen color you do not change the character color. Characters appear in black unless you color code otherwise (in a CALL COLOR statement). You will not be able to read what you write if you change the screen to black without first changing the character colors.

You might want to change the screen color to show a different processing phase in your program. Suppose your program is getting information entered through the keyboard, doing some lengthy calculations, and then printing results. You can change the screen color to light blue (6) when you are getting information, to light green (4) while your program's calculating, and then to magenta (14) when you are writing the results. This way, you can tell when your program's calculating—and it's nice to be able to see your computer do something so you know it's working.

You can also use color to tell when an error happens. For example, change the screen color to dark blue (5) when you need to revise a value because of an error. The user can tell immediately that an error occurred.

Or, when you are programming games, change the screen colors to represent different actions or situations. Dark red (7) is good for emergencies. Black (2) is great for cave games.

Common Errors:

BAD VALUE

You used a *color-code* that is less than 1 or greater than 16.

Example 1:

The program in Listing 4-106 changes the screen color to magenta and writes a message when your program begins.

When you run this, your screen will go blank and magenta, then the message "HI THERE!" appears, followed by the instruction "HIT RETURN TO STOP ME."

```
100   CALL CLEAR
110   CALL SCREEN(14)
120   PRINT "HI THERE!"
130   PRINT : : : : :
140   INPUT "HIT ENTER TO STOP ME. " : X$
150   END
```

Listing 4-106. SCREEN Example 1

Example 2:

The program in Listing 4-107 shows how to modify the program in Example 1 to get a new screen color and then branch back to change the screen color to the one you want.

Remember that you can only use values between 1 and 16 for your screen colors. This program makes sure you will have a correct value for *color-code*.

You stop this program by pressing **FCTN 4** (CLEAR).

```
100   COLORCODE = 14
110   CALL CLEAR
120   CALL SCREEN(COLORCODE)
130   PRINT "HI THERE!"
140   PRINT : : : : :
150   INPUT "WHAT COLOR DO YOU WANT TO   SEE?(1-
      16) FCTN 4 TO STOP":COLORCODE
160   IF (COLORCODE>=1) * (COLORCODE<=16) THEN 110
170   PRINT "YOU ENTERED A BAD":"COLOR CODE!!"
180   PRINT :"COLOR CODES MUST BE":"BETWEEN 1 AND 16."
190   PRINT :"TRY AGAIN."
200   GOTO 150
210   END
```

Listing 4-107. SCREEN Example 2

SEG$	Get a substring (part of a string).
Type:	Function
Format:	SEG$ *(str-exp,position,length)*
Purpose:	SEG$ takes a *substring* (part of a string) of the string defined by *str-exp*. The first character of the substring is character *position* of *str-exp*. The substring has *length* characters in it.
Operands:	*str-exp* is any string constant, string variable, or string expression. *position* is a number, numeric variable, or numeric expression that tells SEG$ where in *str-exp* to begin the substring. *position* cannot be less than one (1) or greater than 32767. *length* is a number, numeric variable, or numeric expression that defines the number of characters in *str-exp* to put into the substring. *length* cannot be less than zero (0) or greater than 32767.
Defaults:	None.

Description:

SEG$ returns a substring (a string segment) of *str-exp*. The returned string is *length* characters long and begins at character *position* in *str-exp*.

When you use SEG$, the original string, *str-exp*, is not changed. The substring of *str-exp* that you define by *position* and *length* can be assigned to a string variable or used in a string expression.

SEG$ can be used in an assignment statement like this:

```
100  ALPHA$=“ABCDEFGHI”
110  STRVAR$=SEG$(ALPHA$,8,2)
```

SEG$ is used as part of a string expression like this:

```
110  STRING1$=“HELLO HI ”
110  STRING2$=“THERE I AM”
120  PUNCT$=”,.?!”
130  PRINT SEG$(STRING1$,9,3)&SEG$(STRING2$,1,5)&
     SEG$(PUNCT$,4,1)
```

When you use a value for *position* that is past the end of the string *str-exp* or if you use a *length* of zero, you get the null string as a result. The null string is a character string of length zero. It contains no characters.

When you use a value for *length* that takes you past the end of the string *str-exp*, you get as many characters as there are in *str-exp* beginning at character *position*.

It is an error to use a negative value or zero for *position* or to supply a negative *length*.

Common Errors:

BAD VALUE

The value of *position* is negative (less than zero), zero, or greater than 32767.

Or, the value of *length* is negative (less than zero) or greater than 32767.

STRING-NUMBER MISMATCH

You used a numeric variable, expression, or constant for *str-exp*. *str-exp* must be a string expression, a string constant, or the name of a string variable.

You tried to assign a string (SEG$ result) to a nonstring variable.

Example 1:

The program in Listing 4-108 asks you for a string and uses SEG$ to print various portions of the string you enter. If you want to include special characters in the string you enter, remember to put the entire string in double quotes (").

Once you have seen the substring you asked for, you can get another substring, enter another string, or stop.

```
100  CALL CLEAR
110  PRINT "PUT YOUR ANSWER IN":"DOUBLE QUOTES IF"
120  PRINT "YOU WANT TO INCLUDE":"CHARACTERS THAT ARE"
130  PRINT "NOT NUMBERS, LETTERS,":"OR BLANKS": :
140  INPUT "YOUR STRING -> ":ANS$
150  MAXL=LEN(ANS$)
160  PRINT :"NOW FOR YOUR SEG$":"ENTER BEGINNING "
170  PRINT "AND LENGTH": :
180  INPUT "BEGINNING, LENGTH -> ":BEGIN,LENGTH
190  IF BEGIN>0 THEN 220
200  PRINT "YOU CAN'T START BEFORE":
     "THE START OF THE STRING"
210  GOTO 160
220  IF LENGTH>=0 THEN 250
230  PRINT :"YOU CAN'T GET LESS":"THAN 0 CHARACTERS"
240  GOTO 160
250  PRINT :"YOUR SUBSTRING IS":SEG$(ANS$,BEGIN,LENGTH)
260  INPUT "ANOTHER SUBSTRING? (Y/N)->":Y$
270  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 290
280  STOP
290  INPUT "ANOTHER STRING? (Y/N)->":Y$
300  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 140
     ELSE 160
310  END
```

Listing 4-108. SEG$ Example 1

Example 2:

The program in Listing 4-109 uses SEG$ to print only 3 characters of the first names in a card list. The names are stored on a cassette tape.

SGN	Get the sign of a number.
Type:	Function
Format:	SGN *(num-exp)*
Purpose:	SGN tells you if a number, numeric variable, or numeric expression is negative, zero, or positive.
Operands:	*num-exp* is a number, numeric variable, or numeric expression.
Defaults:	None.

```
100   CALL CLEAR
110   DIM LAST$(15),FIRST$(15)
120   OPEN #55:"CS1",SEQUENTIAL,INTERNAL,INPUT,FIXED 64
130   NAMES=0
140   FOR I=1 TO 15
150   IF EOF(55) THEN 190
160   INPUT #55:FIRST$(I),LAST$(I)
170   NAMES=NAMES+1
180   NEXT I
190   PRINT NAMES;" NAMES READ."
200   CALL CLEAR
210   PRINT "LAST NAME";TAB(20);"FIRST"
220   FOR I=1 TO NAMES
230   PRINT LAST$(I);TAB(22);SEG$(FIRST$(I),1,3)
240   NEXT I
250   END
```

Listing 4-109. SEG$ Example 2

Description:

SGN tells you the algebraic sign of a number. SGN returns a one if *num-exp* is positive, a zero if *num-exp* is zero, and a minus one if *num-exp* is negative. Table 4-39 lists the SGN results.

Table 4-39. SGN Results

Value	Meaning
−1	The value of *num-exp* is negative (less than zero).
0	The value of *num-exp* is zero.
+1	The value of *num-exp* is positive (greater than zero).

The result of the SGN function can be assigned to a variable, like this:

ANS = SGN(X * 45 − 23)

Or, used in an expression, like this:

RESULT = ANS * SGN(Y^2 − X*4)

If you want to branch to one of three areas, depending on whether a value is negative, zero, or positive, SGN gives you a three-way branch in an ON . . . GOTO or ON . . . GOSUB statement, like this:

ON SGN(ANS)+2 GOSUB 1000,2000,3000

Common Errors:

STRING-NUMBER MISMATCH

You used a string variable, expression, or constant for *num-exp*. *num-exp* must be a numeric expression, a numeric constant (a number), or the name of a numeric variable.

You tried to assign a number (SGN result) to a string variable.

Example 1:

The program in Listing 4-110 uses the SGN function to determine which subprogram to use.

```
100  CALL CLEAR
110  INPUT "ENTER A NUMBER -> ":ANS
120  ON SGN(ANS)+2 GOSUB 160,180,200
130  INPUT "TRY AGAIN? (Y/N) -> ":Y$
140  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 110
150  STOP
160  PRINT :"YOUR NUMBER WAS NEGATIVE."
170  RETURN
180  PRINT :"YOUR NUMBER WAS ZERO."
190  RETURN
200  PRINT :"YOUR NUMBER WAS POSITIVE."
210  RETURN
220  END
```

Listing 4-110. SGN Example 1

Example 2:

The program in Listing 4-111 uses SGN to decide whether to take the square root or not of a number.

```
100  CALL CLEAR
110  INPUT "ENTER A NUMBER -> ":ANS
120  ON SGN(ANS)+2 GOTO 160,190,210
130  INPUT "ANOTHER NUMBER? (Y/N) -> ":Y$
140  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 110
150  STOP
160  PRINT :"YOU CAN'T GET THE":"SQUARE ROOT OF A"
170  PRINT "NEGATIVE NUMBER.": : :
180  GOTO 130
190  PRINT :"SQUARE ROOT IS 0": : :
200  GOTO 130
210  PRINT :"SQUARE ROOT OF";ANS;" IS ";SQR(ANS): : :
220  GOTO 130
230  END
```

Listing 4-111. SGN Example 2

SIN	Get the sine of an angle.
Type:	Function
Format:	SIN *(rad-angle)*
Purpose:	SIN gives you the trigonometric sine of *rad-angle* where *rad-angle* is expressed in radians.
Operands:	*rad-angle* is a number, numeric variable, or numeric expression that represents an angle expressed in radians.
Defaults:	None.

Description:

SIN returns the trigonometric sine of the angle *rad-angle* where *rad-angle* is an angle expressed in radians.

You use the SIN function in calculating distances. Many games use the SIN and COS functions to get X- and Y-distances on a grid.

NOTE

> *rad-angle* must be expressed in radians, not degrees. If you want to convert degrees to radians, use one of the following expressions (PI=3.14159):
>
> RADIANS = DEGREES * PI / 180
>
> or
>
> RADIANS = DEGREES * (4 * ATN(1))/180
>
> or
>
> RADIANS = DEGREES * .01745329251994

Common Errors:

BAD ARGUMENT

You used a value for *rad-angle* that is greater than $1.5707963266375 * 10^{10}$ or less than $-1.5707963266375 * 10^{10}$.

STRING-NUMBER MISMATCH

You used a string variable, expression, or constant for *rad-angle*. *rad-angle* must be a numeric expression, a numeric constant (a number), or the name of a numeric variable.

You tried to assign a number (SIN result) to a non-numeric variable.

Example:

The program in Listing 4-112 uses SIN to print the trigonometric sine of an angle that you enter in degrees. The angle is converted to radians using the ATN function.

```
100  CALL CLEAR
110  PRINT "THIS PROGRAM PRINTS":"THE SINE OF AN ANGLE"
120  INPUT "YOUR ANGLE -> ":ANGLE
130  IF ABS(ANGLE)<360 THEN 160
140  ANGLE=ANGLE/360
150  GOTO 130
160  RADS=ANGLE*(4*ATN(1))/180
170  PRINT : :"THE SINE OF ";ANGLE;
180  PRINT " DEGREES (";RADS;")";"RADIANS IS";
190  PRINT SIN(RADS): :
200  INPUT "ANOTHER ANGLE (Y/N)? -> ":Y$
210  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
220  END
```

Listing 4-112. SIN Example

CALL SOUND Play a tone or make a noise.

Type:	Statement
Format:	[*line#*] CALL SOUND *(duration,freq1,vol1*[, ... , *freq4,vol4*]*)*
Purpose:	CALL SOUND plays one or more tones or a "noise." You can play up to three tones and one noise with a single CALL SOUND.
Operands:	*line#* is a BASIC statement line number that you need when you include CALL SOUND in a program. You don't need a *line#* when you use CALL SOUND as a command. *line#* can be any number between 1 and 32767. *duration* is a number, numeric variable, or numeric expression that represents how long the sound is played. *duration* is expressed in thousandths of a second (milliseconds) and may range from −4250 to 4250. *freq1* is a number, numeric variable, or numeric expression that represents the frequency of the first tone played. *freq1* is expressed in hertz and can range from 110 to 44,733 for a tone. *freq1* for noises ranges from −1 to −8. *vol1* is a number, numeric variable, or numeric expression that represents the volume of the first tone played. *vol1* may range from 0 to 30, where 0 is the loudest. *freq2* is similar to *freq1* but represents the second tone. *vol2* is similar to *vol1* but represents the volume for the second tone. *freq3* is similar to *freq1* but represents the third tone. *vol3* is similar to *vol1* but represents the volume for the third tone. *freq4* is similar to *freq1* but represents the fourth tone. *vol4* is similar to *vol1* but represents the volume for the fourth tone.
Defaults:	None.

Description:

CALL SOUND controls the tone and noise generator in your computer. You can generate up to three simultaneous tones and one "noise" for a specified *duration*. Each tone/noise can have its own volume. The valid ranges for the CALL SOUND operands are shown in Table 4-40.

Table 4-40. CALL SOUND Operands

Operand	Valid Range
duration	1 to 4250, −1 to −4250 (milliseconds)
freq1 through *freq4*	tones: 110 to 44,733 (hertz) noises: −1 to −8
vol1 through *vol4*	0 (loudest) to 30 (quietest)

duration is the length of the tone or noise in milliseconds: one second is 1000 milliseconds. You can play a sound for a maximum of 4250 milliseconds, or 4.25 seconds.

duration can be positive (greater than zero) or negative (less than zero). A positive *duration* plays the tone/noise only after any currently playing tone/noise is finished. A negative *duration* interrupts any current sound and plays the new tone/noise immediately.

NOTE

All the tones and the noise in a CALL SOUND are played for the same *duration*.

CALL SOUND tells the difference between a tone and a noise by looking at the value for the frequency (*freq1* through *freq4*). If you use a negative number in the range −1 to −8, you get a "noise." If you use a positive number in the range 110 to 44,733, you get a tone with that frequency (hertz).

NOTE

You can play a maximum of three tones and one noise at the same time.

Each frequency has its own volume, *vol1* through *vol4*. You can mix and match these volumes in any way. Maybe play the first tone at full volume, lowering the volume for each succeeding tone. Or, you might want the noise to be loudest. Try a variety of sounds to see what you can do.

Once CALL SOUND starts playing a tone or noise, your computer continues executing the current BASIC program. It does not wait for the tone/noise to finish.

It is almost impossible to describe noises. Try them to see what they sound like. Table 4-42 shows you what kinds of noises you can generate.

You are probably familiar with musical notes. Table 4-41 shows you what frequencies to request to get the musical notes you want. There is a relationship between the seemingly random numbers that produce musical notes as shown in Table 4-41. If you number the entries in the table (as we have under the N# column) beginning with 110 as entry zero (0), 116 as entry one (1), and so on, you can calculate the frequency of any note in the table using the expression:

FREQ = 110 * (2^(1/12)) ^ NOTE_NUMBER

Where the NOTE_NUMBER is the sequential number appearing in the column marked N# in Table 4-41. Thus, the first octave goes from 0 to 11, the second octave from 12 to 23, and so on. Middle C, for example, is note 15. To calculate the frequency value for middle C, use:

MIDDLE_C = 110 * (2^(1/12)) ^ 15

To illustrate this technique, the program below generates random musical notes, not just random sounds. (Use **FCTN CLEAR** to stop this program.)

```
100  RANDOMIZE
110  PRINT TAB(8);"RANDOM MUSIC"
120  MFACTOR=2^(1/12)
130  FREQ=110*MFACTOR^(RND*64)
140  DURATION=RND*RND*4250
150  CALL SOUND(DURATION,FREQ,0)
160  GOTO 130
```

Table 4-41. Frequencies for Musical Notes

N#	Freq	Note	N#	Freq	Note
0	110	A	32	698	F
1	116	A flat, B sharp	33	739	F sharp, G flat
2	123	B	34	783	G
3	130	C (low C)	35	830	G sharp, A flat
4	138	C sharp, D flat	36	880	A (above high C)
5	146	D	37	923	A sharp, B flat
6	155	D sharp, E flat	38	987	B
7	164	E	39	1046	C
8	174	F	40	1108	C sharp, D flat
9	185	F sharp, G flat	41	1174	D
10	196	G	42	1244	D sharp, E flat
11	207	G sharp, A flat	43	1318	E
12	220	A (below middle C)	44	1396	F
13	233	A sharp, B flat	45	1479	F sharp, G flat
14	246	B	46	1567	G
15	261	C (middle C)	47	1661	G sharp, A flat
16	277	D sharp, D flat	48	1760	A
17	293	D	49	1864	A sharp, B flat
18	311	D sharp, E flat	50	1975	B
19	329	E	51	2093	C
20	349	F	52	2217	C sharp, D flat
21	369	F sharp, G flat	53	2349	D
22	392	G	54	2489	D sharp, E flat
23	415	G sharp, A flat	55	2637	E
24	440	A (above middle C)	56	2793	F
25	466	A sharp, B flat	57	2959	F sharp, G flat
26	493	B	58	3135	G
27	523	C (high C)	59	3322	G sharp, A flat
28	554	C sharp, D flat	60	3520	A
29	587	D	61	3729	A sharp, B flat
30	622	D sharp, E flat	62	3591	B
31	659	E	63	4186	C
			64	4434	C sharp, D flat

NOTE: The value in the N# (note number) column is the factor used to get a true musical note with this calculation:

FREQUENCY = 110 * (2^(1/12)) ^ N#

Table 4-42. Frequencies for Noises

Frequency	Noise Description
−1	Periodic noise type 1
−2	Periodic noise type 2
−3	Periodic noise type 3
−4	Periodic noise that varies with the frequency of the third tone in the CALL SOUND
−5	White noise type 1
−6	White noise type 2
−7	White noise type 3
−8	White noise that varies with the frequency of the third tone in the CALL SOUND

While your computer can generate a very wide range of sounds, few people can hear tones much above 10,000 or 11,000 hertz. Experiment with CALL SOUND to determine the limits of your hearing. Generally, women can hear higher frequencies than men.

Common Errors:

BAD VALUE

One or more of the following happened:

- *duration* is less than −4250 or greater than 4250
- *freq1* through *freq4* is less than 110 or greater than 44,733
- A noise is less than −8 and greater than −1
- *vol1* through *vol4* is less than 0 or greater than 30.

INCORRECT STATEMENT

You specified more than three tones or more than one noise in a CALL SOUND.

Example 1:

The program in Listing 4-113 uses SOUND to make a crash sound.

```
100   FOR I=1 TO 4
110   FOR S=500 TO 800 STEP 40
120   CALL SOUND(-200,S,0,S+75,0)
130   NEXT S
140   FOR S=800 TO 500 STEP -40
150   CALL SOUND(-200,S,0,S+75,0)
160   NEXT S
170   NEXT I
180   FOR S=790 TO 110 STEP -30
190   A=.0435*(800-S)
200   CALL SOUND(-200,S,A,S+75,A)
210   NEXT S
220   FOR I=1 TO 2
230   CALL SOUND(33,-5,4)
240   CALL SOUND(370,-6,0)
250   CALL SOUND(333,-6,4)
260   CALL SOUND(303,-6,8)
270   CALL SOUND(1,-6,12)
280   CALL SOUND(67,-6,16)
290   CALL SOUND(33,-7,18)
300   CALL SOUND(33,-6,18)
310   NEXT I
320   END
```

Listing 4-113. SOUND Example 1

Example 2:

The program in Listing 4-114 plays any one tone or noise.

```
100  CALL CLEAR
110  PRINT "YOU CAN PLAY ANY ONE":"TONE OR NOISE"
120  INPUT "HOW LONG (-4250 TO 4250) -> ":DUR
130  IF (ABS(DUR)>4250)+(DUR=0) THEN 240
140  PRINT "WHAT FREQUENCY(110 TO 44733)"
150  INPUT "(-1 TO -8 NOISES) -> ":FREQ
160  IF (FREQ>=110)*(FREQ<=44733) THEN 180
170  IF (ABS(FREQ)<1)+(ABS(FREQ)>8) THEN 270
180  INPUT "WHAT VOLUME (0 TO 30) -> ":VOL
190  IF (VOL<0)+(VOL>30) THEN 300
200  CALL SOUND (DUR,FREQ,VOL)
210  INPUT "ANOTHER TONE? (Y/N) -> ":Y$
220  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
230  STOP
240  REM DURATION OUT OF RANGE
250  PRINT "YOUR VALUE";DUR;"IS OUT OF RANGE"
260  GOTO 120
270  REM FREQ OUT OF RANGE
280  PRINT "YOUR VALUE";FREQ;"IS OUT OF RANGE"
290  GOTO 140
300  REM VOL OUT OF RANGE
310  PRINT "YOUR VALUE";VOL;"IS OUT OF RANGE"
320  GOTO 180
330  END
```

Listing 4-114. SOUND Example 2

SQR	Get the square root of a number.
Type:	Function
Format:	SQR *(num-exp)*
Purpose:	SQR returns the positive square root of *num-exp*.
Operands:	*num-exp* is a number, numeric variable, or numeric expression that contains the value that you want the square root of. *num-exp* may not be less than zero.
Defaults:	None.

Description:

SQR returns the positive square root of *num-exp*. This is the same as raising *num-exp* to the ½ power (*num-exp*^(½)).

You cannot use SQR to take the square root of a negative number (*num-exp* cannot be less than zero).

You can use SQR to assign a value to a variable, like this:

DIST = SQR(A*A + B + B)

Or, you can use SQR in an expression, like this:

ANS = DISTANCE + SQR(X*Y + 53)

Common Errors:

STRING-NUMBER MISMATCH

You used a string value, variable, or expression for *num-exp*.

Example 1:

The program in Listing 4-115 uses SQR to calculate the hypotenuse of a right triangle (A squared = B squared + C squared, where B and C are the sides of the triangle).

```
100   REM CALCULATE HYPOTENUSE
110   PRINT : :"ENTER THE TWO SIDES"
120   INPUT "OF THE TRIANGLE (N,N) -> ":BSIDE,CSIDE
130   IF (BSIDE<0)+(CSIDE<0) THEN 200
140   PRINT : :"THE HYPOTENUSE OF THE":
      "RIGHT TRIANGLE WITH"
150   PRINT "SIDES ";BSIDE;" AND ";CSIDE
160   PRINT "IS ";SQR(BSIDE^2+CSIDE^2): : :
170   INPUT "TRY AGAIN? (Y/N) -> ":Y$
180   IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 110
190   STOP
200   PRINT "SIDES MAY NOT BE NEGATIVE !!"
210   GOTO 110
220   END
```

Listing 4-115. SQR Example 1

Example 2:

The program in Listing 4-116 uses SQR to calculate the shortest distance between two points on a square grid. This technique is often used in games where you move, or want to find out if you have enough fuel or energy to move, between two positions on a grid.

The program uses the absolute value (ABS) of the difference between the x and y coordinates in calculating the distance.

```
100   REM DISTANCE BETWEEN TWO POINTS
110   CALL CLEAR
120   PRINT : :"ENTER TWO POINTS":"AND I'LL TELL YOU"
130   PRINT "HOW FAR APART THEY":"ARE ON YOUR GRID": :
140   INPUT "POINT 1 (X,Y) IS -> ":X1,Y1
150   INPUT "POINT 2 (X,Y) IS -> ":X2,Y2
160   DIST=SQR(ABS((X1-X2)^2+(Y1-Y2)^2))
170   PRINT "THE POINTS ARE";DIST;"UNITS APART.": :
180   INPUT "TRY AGAIN? (Y/N) -> ":Y$
190   IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
200   END
```

Listing 4-116. SQR Example 2

STOP	Stop executing a program.
Type:	Statement
Format:	[*line#*] STOP
Purpose:	STOP stops executing BASIC statements in your program and returns you to command mode.
Operands:	*line#* is a BASIC statement line number that you need when you include STOP in a program. You don't need *line#* when you use STOP as a command. *line#* can be any number between 1 and 32767.
Defaults:	None.

Description:

STOP terminates TI BASIC program execution. STOP stops executing your TI BASIC basic program, closes any files that you may have OPENed in your program, and returns you to direct mode.

You do not need to use a STOP statement if your program ends after it executes its highest numbered line. TI BASIC automatically stops executing a BASIC program after executing its highest numbered line if the highest numbered line is not a GOSUB or GOTO statement.

STOP statements are very useful if you have several different ways of ending your program. You can include as many STOP statements in a program as you need.

Common Errors:

None.

Example 1:

The program in Listing 4-117 uses two STOP statements to end when you tell it to stop or when it reaches the end of its processing.

```
100  CALL CLEAR
110  FOR I=1 TO 100
120  PRINT I
130  INPUT "STOP YET? (Y/N) -> ":Y$
140  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 180
150  NEXT I
160  PRINT "YOU NEVER STOPPED ME!"
170  END
180  PRINT "OK."
190  STOP
```

Listing 4-117. STOP Example 1

Example 2:

The program in Listing 4-118 uses a STOP statement to end the program after you guess its number.

STR$	Convert numeric to string format.
Type:	Function
Format:	STR$ *(num-exp)*
Purpose:	STR$ "translates" numbers to string format, returning the string representation of the number given by *num-exp*.
Operands:	*num-exp* is a number, numeric variable, or numeric expression whose string representation is returned by STR$.
Defaults:	None.

```
100  CALL CLEAR
110  RANDOMIZE
120  NUMBER=INT(RND*100)
130  PRINT "I HAVE A NUMBER"
140  N=0
150  INPUT "YOUR GUESS -> ":GUESS
160  N=N+1
170  IF GUESS<>NUMBER THEN 230
180  PRINT "YOU GUESSED IT IN";N;" TRIES!"
190  INPUT "TRY AGAIN? (Y/N) -> ":Y$
200  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
210  PRINT "GOODBYE"
220  STOP
230  IF GUESS<NUMBER THEN 260
240  PRINT "YOUR GUESS IS TOO HIGH."
250  GOTO 150
260  PRINT "YOUR GUESS IS TOO LOW."
270  GOTO 150
280  END
```

Listing 4-118. STOP Example 2

Description:

You use STR$ when you want to change numeric data from its numeric format to a string format. Then you can use any string functions (SEG$, POS, LEN, etc.) to manipulate the formerly numeric data. You cannot use the result of a STR$ function as a *number;* the result is a *string*.

The string that results from STR$ can be assigned to a variable, like this:

ANS$ = STR$(NUMANS)

Or, you can use the result of STR$ in a string expression, like this:

OUTDOL$ = “$”&STR$(INT(NUMDOLS*100)/100)

STR$ is often used when you want to format a line to write to the screen in a special way. You use STR$ and LEN to see how many characters are in the number you want to print. Then, you put all the pieces of the line together as a string and write it neatly on the screen.

Because TI BASIC writes a space before and after it PRINTs a numeric variable or expression result, you can use STR$ to format numeric output without the spaces (e.g., to print a dollar sign ($) before a numeric value without a space between the dollar sign and the number). You use STR$ to make a string out of the number and then print the dollar sign and number with no spaces. The second example above does just this processing, while also using the INT function to round the numeric dollar value to two decimal places ($nnn.nn).

Common Errors:

```
STRING-NUMBER MISMATCH
```

You used a string instead of a numeric value, variable, or expression for *num-exp*.

Example 1:

The program in Listing 4-119 rounds a number to two decimal places and uses STR$ to make a string out of a number and prints the number with a leading dollar sign.

```
100  CALL CLEAR
110  PRINT "ENTER A NUMBER AND":" I'LL WRITE IT WITH"
120  PRINT " A LEADING DOLLAR SIGN"
130  INPUT "YOUR NUMBER -> ":ANS
140  DOL$=STR$(ANS)
150  IF SEG$(DOL$,1,1)<>" " THEN 170
160  DOL$=SEG$(DOL$,2,255)
170  PRINT : :"YOUR VALUE WAS ";"$";DOL$
180  END
```

Listing 4-119. STR$ Example 1

Example 2:

The program in Listing 4-120 uses STR$ to make a string out of a number so you can use POS to see if there are any nines in the number.

```
100  CALL CLEAR
110  PRINT "ENTER A NUMBER":"AND I'LL TELL YOU"
120  PRINT "HOW MANY 9'S ARE IN IT"
130  INPUT "YOUR NUMBER -> ":NUMANS
140  STRANS$=STR$(NUMANS)
150  BEG=1
160  NINES=0
170  POS9=POS(STRANS$,"9",BEG)
180  IF POS9=0 THEN 220
190  NINES=NINES+1
200  BEG=POS9+1
210  GOTO 170
220  PRINT :"THERE ARE";NINES;"""9'S"" IN ";NUMANS : :
230  INPUT "ANOTHER NUMBER? (Y/N) -> ":Y$
240  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 130
250  END
```

Listing 4-120. STR$ Example 2

TAB	Tab to a column and print.
Type:	Function
Format:	TAB *(column)*
Purpose:	TAB is used with PRINT and DISPLAY statements to write the next character at position *column* on the screen or paper.
Operands:	*column* is a number, numeric variable, or numeric expression that tells BASIC where to put the next character on the screen or paper.
Defaults:	None.

Description:

TAB lets you specify a *column* where the next data from a PRINT or DISPLAY will be printed.

TI BASIC keeps track of where it is on a print line. If you do not use TAB, BASIC prints data using the algorithms for the print separators (see the sections on PRINT and DISPLAY for full details on print separators).

The TAB function is considered a print item and must be enclosed in print separators in the same way as any other print item. Any print separator preceding the TAB is acted upon before the TAB *column* is evaluated.

Table 4-43 shows you the print separators and their meanings. Usually the semicolon (;) print separator is used around the TAB function.

Table 4-43. *print-separators*

print-separator	Meaning
semicolon (;)	Print the next data item right next to the current data item. Do not leave any extra spaces (except for the leading and trailing spaces around numeric data items).
colon (:)	Skip to the next line.
comma (,)	Print the next data item at the next available zone. Zone 1 starts in column 1. Zone 2 starts in column 15.

When you use TAB, you are telling TI BASIC in which column you want the next data item to be printed. But, you cannot print past the end of your television screen, off the edge of your thermal printer, or past the end of a record on an RS232 printer, cassette file, or disk file. TI BASIC follows these printing rules:

1. If you use a value for *column* that is less than one, BASIC does not back up. Instead, BASIC replaces negative *column* values with a one.
2. If you are past position *column*, BASIC moves to *column* on the next line.
3. If you use a value for *column* that is past the end of your device (more than 28 for a television screen, for example), BASIC keeps subtracting the maximum length from *column* until the *column* value is between 1 and the maximum length for the device. Table 4-44 shows the maximums for some devices.

Table 4-44. TAB *column* Maximums

Device	*column* Maximum
TV screen	28 characters per line
cassette file	maximum record length in the OPEN statement for the file
disk file	maximum record length in the OPEN statement for the file
RS232 file	maximum record length in the OPEN statement for the file (usually 80 for RS232 printers)
thermal printer	32 characters per line

NOTE

Remember that data items will not be split over screen lines unless the data item is a string longer than 28 characters. If you use TAB for positioning and your data item is too long to fit onto the line, BASIC prints the data item on the next line, regardless of what you said through TAB.

Common Errors:

BAD VALUE

You used a value for *column* that is larger than 32767.

STRING-NUMBER MISMATCH

You used a non-numeric value, variable, or expression for *column*.

Example 1:

The program in Listing 4-121 uses TAB to print 30 numbers at columns 5, 15, and 25.

Try changing the program to print at other columns.

```
100  CALL CLEAR
110  FOR I=1 TO 30 STEP 3
120  PRINT TAB(5);I;TAB(15);I+1;TAB(25);I+2
130  NEXT I
140  END
```

Listing 4-121. TAB Example 1

Example 2:

The program in Listing 4-122 uses TAB to print 20 random numbers in two different columns, depending on whether the number is positive (column 5) or negative (column 17).

The RND function is used for two purposes: (1) to generate 20 numbers between 1 and 1000; (2) to put a sign (+/−) on the generated number. If

```
100  CALL CLEAR
110  RANDOMIZE
120  PRINT TAB(3);"POSITIVE";TAB(15);"NEGATIVE" : :
130  FOR I=1 TO 15
140  TABPOS=5
150  NUM1=INT(RND*1000)
160  IF RND>=.432 THEN 180
170  TABPOS=17
180  PRINT I;TAB(TABPOS);NUM1
190  NEXT I
200  END
```

Listing 4-122. TAB Example 2

RND is less than 0.432, the number is called negative. If RND is greater than or equal to 0.432, the number is called positive.

TAN	Get the tangent of an angle.
Type:	Function
Format:	TAN *(rad-angle)*
Purpose:	TAN gives you the trigonometric tangent of the angle *rad-angle* expressed in radians.
Operands:	*rad-angle* is a number, numeric variable, or numeric expression that represents an angle expressed in radians.
Defaults:	None.

Description:

TAN returns the trigonometric tangent of the angle *rad-angle* where *rad-angle* is expressed in radians.

NOTE

Rad-angle must be expressed in radians, not degrees. If you want to convert degrees to radians, use one of the following expressions (PI=3.14159):

RADIANS = DEGREES * PI / 180
or
RADIANS = DEGREES * (4 * ATN (1))/180
or
RADIANS = DEGREES * .01745329251994

Common Errors:

```
STRING-NUMBER MISMATCH
```

You used a string variable, expression, or constant for *num-exp*. *num-exp* must be a numeric expression, a numeric constant (a number), or the name of a numeric variable.

You tried to assign a number (TAN result) to a non-numeric variable.

Example:

The program in Listing 4-123 prints the TAN of an angle that you enter in degrees. The angle is converted to radians using the ATN function.

```
100  CALL CLEAR
110  PRINT "THIS PROGRAM PRINTS":
     "THE TANGENT OF AN ANGLE"
120  INPUT "YOUR ANGLE -> ":ANGLE
130  IF ABS(ANGLE)<360 THEN 160
140  ANGLE=ANGLE/360
150  GOTO 130
160  RADS=ANGLE*(4*ATN(1))/180
170  PRINT :"THE TANGENT OF ";ANGLE;
180  PRINT " DEGREES (";RADS;")";"RADIANS IS";
190  PRINT TAN(RADS): :
200  INPUT "ANOTHER ANGLE (Y/N)? -> ":Y$
210  IF (SEG$(Y$,1,1)="Y")+(SEG$(Y$,1,1)="y") THEN 120
220  END
```

Listing 4-123. TAN Example

TRACE	Monitor program execution.
Type:	Command
Format:	[*line#*] TRACE
Purpose:	TRACE writes the line number of each BASIC statement before BASIC executes the statement.
Operands:	*line#* is a BASIC statement line number that you need when you include TRACE in a program. You don't need *line#* when you use TRACE as a command. *line#* can be any number between 1 and 32767.
Defaults:	None.

Description:

TRACE lists the line numbers of TI BASIC statements before the statements are executed. The list of line numbers is very useful when you are debugging a program.

The line numbers are printed at your screen. If your program also prints information to your screen, you will see your program's information mixed in with the TRACE line numbers.

Once you use TRACE, it stays in effect until you enter a NEW command or an UNTRACE command/statement.

TRACE used as a command will print the line numbers for every statement. Example 1 (below) shows you how to TRACE a small program.

Be careful when you are TRACEing a very large program. You will see the line numbers for every statement that gets executed. This can be confusing to read. If you must TRACE a large program, you can use BREAK and CONTINUE commands to stop (BREAK) and restart (CONTINUE) execution when your screen gets filled and you want to read the numbers.

If you are having a problem with only a section of your program, you can use TRACE as a program statement. If you also use an UNTRACE statement somewhere after the TRACE statement, you will see only those statements that are executed between the TRACE and UNTRACE statements. Example 2 (below) shows you how to do this.

Common Errors:

None.

Example 1:

The example in Listing 4-124 uses TRACE to trace an entire program. The program is short so that you can see all of the line numbers on your screen.

The program contains a FOR loop so that certain lines will be executed more than once. There is a PRINT statement at the beginning and another at the end of the program so that you can see how the TRACE information mixes in with your program's information on the screen.

```
TRACE
100  CALL CLEAR
110  PRINT "HI THERE.  I'M STARTING."
120  FOR I=1 TO 4
130  NEXT I
140  PRINT "GOODBYE."
150  END
```

Listing 4-124. TRACE Example 1

Example 2:

The program in Listing 4-125 uses TRACE and UNTRACE as program statements. Only the inner FOR loop statements are TRACEd.

```
100  CALL CLEAR
110  PRINT "HI THERE."
120  FOR I=1 TO 10
130  PRINT "I=";I
140  REM EXECUTE INNER LOOP ONLY
150  REM FOR EVEN I VALUES
160  IF (I/2)<>INT(I/2) THEN 240
170  PRINT "STARTING TRACE"
180  TRACE
190  FOR J=1 TO 3
200  PRINT "J=";J
210  NEXT J
220  PRINT "ENDING TRACE"
230  UNTRACE
240  NEXT I
250  PRINT "GOODBYE."
260  END
```

Listing 4-125. TRACE Example 2

UNBREAK	Remove all program breakpoints.
Type:	Command
Format:	[*line#*] UNBREAK or [*line#*] UNBREAK *line-num-list*
Purpose:	UNBREAK removes one or more *breakpoints* set by BREAK commands or statements. If you use *line-num-list*, breakpoints are removed from only those line numbers in the list.
Operands:	*line#* is a BASIC statement line number that you need when you include UNBREAK in a program. You don't need *line#* when you use UNBREAK as a command. *line#* can be any number between 1 and 32767. *line-num-list* is a list of BASIC statement line numbers (values between 1 and 32767) which you want to remove from the BREAK list.
Defaults:	If you use UNBREAK without a *line-num-list*, all breakpoints are removed from the program.

Description:

UNBREAK removes the breakpoints for the lines in *line-num-list* or, if you don't use a *line-num-list*, UNBREAK removes all program breakpoints.

You use BREAK to set breakpoints in your program. A breakpoint is a special marker that BASIC puts in your program at whatever line number(s) you specify with BREAK. When BASIC is executing your program and reaches one of these special markers, BASIC stops executing your program. You restart execution with a CONTINUE command.

Breakpoints are useful when your program is not running correctly. You may be getting a strange answer for a calculation. Or you may not be seeing what you think you told your program to write.

At a breakpoint, you can print or change variable values and continue program execution with a CONTINUE. If you edit any program lines, you cannot CONTINUE.

UNBREAK and BREAK can be very useful as statements when you are debugging a section of your program. You can set a series of breakpoints (with BREAK) and, in your program, selectively UNBREAK some of the line numbers. Example 2 (below) shows you this technique.

Common Errors:

```
BAD LINE NUMBER
```

This message is only a warning. Your program will continue to execute. You used a line number in the *line-num-list* that is not a line number of a statement in your BASIC program.

Example:

The example in Listing 4-126 uses BREAK and UNBREAK as commands. First, set breakpoints at lines 120 and 150. Now, run the program.

At the first breakpoint, use a CONTINUE command to resume execution. At the second breakpoint, use an UNBREAK command to remove all breakpoints and a CONTINUE command to resume execution.

```
BREAK 120,150
100  CALL CLEAR
110  PRINT "LINE 110"
120  PRINT "LINE 120"
130  T=T+1
140  PRINT "LINE 140"
150  PRINT "LINE 150"
160  IF T>1 THEN 180
170  GOTO 110
180  END

     To run this example, use:

BREAK 120,150 <ENTER>
Enter the program lines 100 to 180
RUN <ENTER>
LINE 110
BREAKPOINT AT 120
CONTINUE <ENTER>
LINE 120
LINE 140
BREAKPOINT AT 150
UNBREAK <ENTER>
LINE 150
LINE 110
LINE 120
LINE 140
LINE 150
READY
```

Listing 4-126. UNBREAK Example

UNTRACE	Remove program statement monitoring.
Type:	Command
Format:	[*line#*] UNTRACE
Purpose:	UNTRACE reverses the TRACE actions. The line numbers of the BASIC statements are not written to the screen before the statements are executed.
Operands:	*line#* is a BASIC statement line number that you need when you include UNTRACE in a program. You don't need *line#* when you use UNTRACE as a command. *line#* can be any number between 1 and 32767.
Defaults:	None.

Description:

UNTRACE cancels a TRACE command/statement. When you use TRACE, BASIC prints the line number of a statement before executing it. UNTRACE reverses this processing and BASIC no longer prints line numbers; BASIC simply executes the statements.

TRACE and UNTRACE are useful when you are debugging a program.

You can see exactly which statements are executed and in exactly what order.

Common Errors:

None.

Example:

The example in Listing 4-127 uses TRACE to write all line numbers and UNTRACE to reverse the TRACE processing.

In the following example, <ENTER> means press the ENTER key.

```
NEW <ENTER>
NUM <ENTER>
100  CALL CLEAR
110  FOR I=1 TO 5
120  PRINT "I=";I
130  NEXT I
140  PRINT "DONE"
150  END
160  <ENTER>
TRACE <ENTER>
RUN <ENTER>
     Here  you'll  see  the  FOR  loop  values   listed
     interspersed with the TRACE line number list.
UNTRACE <ENTER>
RUN <ENTER>
     Here you'll see the FOR loop values listed without
     the TRACE line number list.
```

Listing 4-127. UNTRACE Example

VAL	Translate a string to numeric format.
Type:	Function
Format:	VAL *(str-exp)*
Purpose:	VAL "translates" string data to internal numeric format.
Operands:	*str-exp* is a string constant, string variable, or string expression that represents a valid BASIC number.
Defaults:	None.

Description:

VAL converts the number stored as a string in *str-exp* to internal numeric format. The value in *str-exp* must represent a correctly formatted BASIC number. (Chapter 2 talks about BASIC numbers and their character string formats.)

STR$ is the complement function to VAL, converting numeric data into string format.

You often use VAL after you have extracted, using the SEG$ function, numeric characters from a string containing non-numeric characters. For example:

AMOUNT = VAL(SEG$("$45.65",2,5))

places the value 45.65 into numeric variable AMOUNT.

Common Errors:

BAD ARGUMENT

Str-exp either is a null string (a string with no characters) or is longer than 254 characters.

Or, the number represented by *str-exp* does not represent a valid BASIC number. For example, it may contain non-numeric characters such as $ or @.

Example 1:

The program in Listing 4-128 uses VAL to change a string to numeric format.

```
100  CALL CLEAR
110  INPUT "ENTER A NUMBER -> ":STRIN$
120  ANS=VAL(STRIN$)
130  PRINT :"YOU ENTERED ";ANS
140  END
```

Listing 4-128. VAL Example 1

Example 2:

The program in Listing 4-129 asks you for a string that contains a number with a leading dollar sign ($). It uses SEG$ to strip off the dollar sign and VAL to convert the value to a numeric format.

```
100  CALL CLEAR
110  PRINT "ENTER A NUMBER WITH A"
     " LEADING DOLLAR SIGN ($)."
120  INPUT "YOUR NUMBER ($XXX.XX) -> ":DOL$
130  N=POS(DOL$,"$",1)
140  IF N>0 THEN 170
150  PRINT : :"USE A DOLLAR SIGN!!": :
160  GOTO 120
170  TMP$=SEG$(DOL$,N+1,255)
180  ANS=VAL(TMP$)
190  PRINT :"2 X YOUR VALUE IS";2*ANS
200  PRINT :"DONE"
210  END
```

Listing 4-129. VAL Example 2

CALL VCHAR	Write character(s) at screen row, col.
Type:	Statement
Format:	[*line#*] CALL VCHAR *(row,col,ASCII-code*[*,repetitions*]*)*
Purpose:	CALL VCHAR writes *repetitions* vertical (down the screen) copies of the character represented by *ASCII-code* to your screen. The first character is written at row *row* and column *col*. The second character (if *repetitions* is greater than one) is written at row *row* + 1 and column *col*.
Operands:	*line#* is a BASIC statement line number that you need when you include CALL VCHAR in a program. You don't need *line#* when you use CALL VCHAR as a command. *line#* can be any number between 1 and 32767. *row* is a number, numeric variable, or numeric expression that contains the row on your screen where you want to begin writing the character represented by *ASCII-code*. *row* may be any number between 1 and 24, the number of rows on your screen. *column* is a number, numeric variable, or numeric expression that contains the column on your screen where you want to write the first character represented by *ASCII-code*. *column* may be any number between 1 and 32, the number of columns on your screen. *ASCII-code* is a number, numeric variable, or numeric expression that contains the ASCII value of the character you want to write at *row,col*. Table 4-45 shows you the ASCII codes for the characters. *ASCII-code* must not be less than zero or greater than 32767. *repetitions* is a number, numeric variable, or numeric expression that tells VCHAR how many characters you want to write in a column on the screen. *repetitions* must not be less than one or greater than 32767.
Defaults:	If you do not supply a value for *repetitions*, VCHAR writes one character on the screen at *row* and *col*.

Definition:

VCHAR writes the character with the ASCII value *ASCII-code* at row *row* and column *col*. If you use a value for *repetitions*, you will get that many characters written down the screen (vertically) beginning at *row,col*. Table 4-45 shows you the ASCII codes for the TI-99/4A characters. You can define characters for ASCII-codes 128 through 159 with CHAR.

Using VCHAR, you can write one or more characters anyplace on the screen. The top left corner of your screen is row 1, column 1. The bottom left corner is row 24, column 1. The upper right corner is row 1, column 32. The bottom right corner is row 24, column 32. Depending on the adjustment of your television set, you may not be able to see the characters in columns 1 and 32.

If the number of *repetitions* you specify exceeds the number of positions remaining down column *col*, placement of characters continues with the first row of the next column (*col* + 1). If the last character of the last row is reached (row 24, column 32) with *repetitions* still remaining, character placement continues within column one, row one. Since there are 768 character positions on the screen (24 rows time 32 columns), a *repetitions* factor greater than 768 causes the same position to be repeatedly overwritten by the same character (*ASCII-code*).

Table 4-45. VCHAR and ASCII Character Codes

ASCII Char	Decimal	ASCII Char	Decimal	ASCII Char	Decimal
(space)	32	@	64	‘	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
’	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
−	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	|	124
=	61	]	93	}	125
>	62	^	94	~	126
?	63	_	95	(DEL)	127

If you specify an ASCII-code greater than 255, 256 is repeatedly subtracted from it until its value is less than or equal to 255. Thus an ASCII-code of 300 results in display of the comma (ASCII character 44 = 300 − 256).

While VCHAR, HCHAR, and PRINT all put characters on your screen, there is one important difference between PRINT and VCHAR/HCHAR:

- With PRINT you can put a maximum of 28 characters across the screen. With VCHAR and HCHAR you can put a maximum of 32 characters across the screen.

VCHAR and its relative HCHAR are very useful when you design screens. You can even design your own special graphics characters with CHAR. You will find it easy to design a screen if you use a grid like the one in Fig. 4-16. Make a 24-row by 32-column grid and fill in the squares

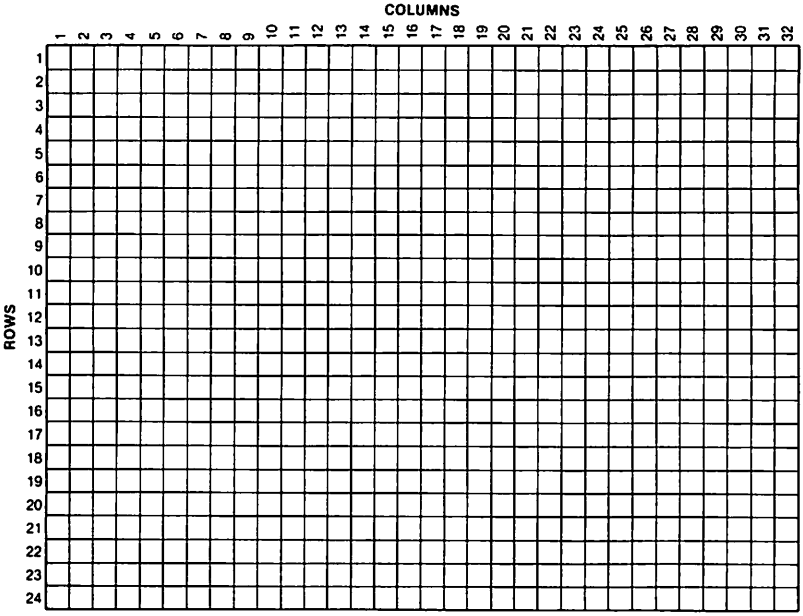


Fig. 4-16. VCHAR grid diagram.

with the characters you want to use. Example 1 (below) shows you how to do this.

Another use for VCHAR is writing messages in specific positions on your screen. You use SEG$ to get each letter of a message from a string variable and then print each letter using VCHAR. You can print words in columns this way. Example 2 (below) uses VCHAR this way.

Common Errors:

BAD VALUE

Row is less than 1 or greater than 24. Or, *col* is less than 1 or greater than 32.

Or, either *ASCII-code* or *repetitions* is less than zero or greater than 32767.

Example 1:

The program in Listing 4-130 uses SCREEN to change the screen color to magenta (14) and writes the message "HI" in BIG letters using VCHAR. Fig. 4-17 shows you the grid used to design this screen. The program uses an INPUT statement that waits for you to press **ENTER**

```
100  CALL CLEAR
110  CALL SCREEN(14)
120  CALL VCHAR(6,12,72,7)
130  FOR I=1 TO 3
140  CALL VCHAR(9,I+12,72,1)
150  NEXT I
160  CALL VCHAR(6,16,72,7)
170  CALL VCHAR(6,19,73,7)
180  INPUT X$
190  END
```

Listing 4-130. VCHAR Example 1

COLUMNS

ROWS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1																																
2																																
3																																
4																																
5																																
6												H				H			I													
7												H				H			I													
8												H				H			I													
9												H	H	H	H	H			I													
10												H				H			I													
11												H				H			I													
12												H				H			I													
13																																
14																																
15																																
16																																
17																																
18																																
19																																
20																																
21																																
22																																
23																																
24																																

Fig. 4-17. VCHAR screen grid design.

before it stops (so you can see what the screen looks like before the program ends and returns to command mode). Try modifying the program to put your name in big letters across the bottom of the screen.

Example 2:

The program in Listing 4-131 asks you for your name and then writes it diagonally across the screen.

Change the program to write the name in different places on your screen. Remember that you cannot write past column 32 or row 24 or before column 1 or row 1.

```
100  CALL CLEAR
110  INPUT "WHAT'S YOUR NAME? -> ":NAME$
120  IF LEN(NAME$)=0 THEN 110
130  CALL CLEAR
140  MSG$="HI "&NAME$
150  FOR I=4 TO 16 STEP 4
160  ROW=4
170  FOR J=1 TO LEN(MSG$)
180  IF I+J>32 THEN 220
190  IF ROW+J>24 THEN 220
200  CALL VCHAR(ROW+J,J+I,ASC(SEG$(MSG$,J,1)))
210  NEXT J
220  NEXT I
230  MSG$="PRESS ANY KEY TO STOP"
240  FOR I=1 TO LEN(MSG$)
250  CALL VCHAR(23,I+3,ASC(SEG$(MSG$,I,1)))
260  NEXT I
270  CALL KEY (0,K,S)
280  IF S=0 THEN 260
290  END
```

Listing 4-131. VCHAR Example 2

CHAPTER 5

TI BASIC Technical Information

Some TI-99/4A owners are interested in what happens to their TI BASIC program once they enter it from the keyboard. In this chapter, we provide a brief review of how TI BASIC stores numeric and string variables and constants, and we discuss what TI BASIC does to the statements and commands in your BASIC programs.

INTRODUCTION TO HEXADECIMAL

In order to fully understand what follows, you must know something of the hexadecimal numbering system—and to understand that, you must be familiar with binary.

The TI-99/4A, along with nearly all modern computers, has a memory system based on the 8-bit byte. Each bit (binary digit) in the 8-bit byte can be either on (value 1) or off (value 0). Numbers built from ones and zeros are called binary numbers and form a base 2 counting system. Our normal numbering system—the decimal system—is based on 10. Counting in binary is similar to counting in decimal, but without the digits 2 through 9.

Counting the same numbers, side by side, in binary and decimal looks like this:

0000 = 0
0001 = 1
0010 = 2
0011 = 3
0100 = 4
0101 = 5
0110 = 6
0111 = 7

1000 = 8
1001 = 9
1010 = 10
1011 = 11
1100 = 12
1101 = 13
1110 = 14
1111 = 15

Ordinary decimal numbers do not correspond nicely with the values you can store in one byte. Consider, for example, these two values:

11000101 binary = 197 decimal
and
00011111 binary = 31 decimal

As you can see, there is no natural correspondence between the 8-bit binary representation and the decimal representation of the numbers. To make it convenient to deal with bytes, it would be nice to use a numbering system more compact than binary, but that also corresponds directly to the bits in a byte.

That is done with *hexadecimal numbers*. The binary numbering system is based on 2; the familiar decimal numbering system is based on 10; the hexadecimal numbering system is based on 16.

When you count in binary, you have only two digits to count with: 0 and 1. In decimal, you have ten digits: the ordinary 0 through 9.

To count in hexadecimal, you must use 16 unique digits, starting with the familiar 0 through 9. But that is only ten digits, we still need six more. To maintain compatibility with existing printers, keyboards, and display screens, the designers of the hexadecimal system chose to use the letters A through F to represent the additional hexadecimal digits. Counting in hexadecimal looks like this:

0 1 2 3 4 5 6 7 8 9 A B C D E F hexadecimal
is equal to
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 decimal

So what has this got to do with representing the value of a byte? Well, it turns out that a single hexadecimal digit can represent any 4-bit value, as in:

0000 = 0
0001 = 1
0010 = 2
0011 = 3
0100 = 4
0101 = 5
0110 = 6
0111 = 7
1000 = 8
1001 = 9
1010 = A
1011 = B
1100 = C
1101 = D
1110 = E
1111 = F

As you can see, one hexadecimal digit neatly represents any 4-bit value, called a *nibble*. (What else would you call a half a byte?) It follows then that any two hexadecimal digits can represent the value of an 8-bit byte. Consider these byte-to-decimal-to-hexadecimal examples:

11000101 = 197 decimal = C5 hexadecimal
00011111 = 31 decimal = 1F hexadecimal

Thus we arrive at a concise and consistent two-digit representation of any one-byte value that is readily translated into the underlying bit pattern (sequence of ones and zeros) in the byte. The standard TI notation for hexadecimal numbers is to precede the value with a greater than sign (>), as in:

>C5
>BF1A

INTERNAL DATA STORAGE FORMATS

TI BASIC supports two data types—numeric data and character string data. You can include these two data types in programs as variables (NETINC, NAME$) or as constants (3.14159, "GEORGE WASHINGTON"). Variables and constants are stored in a different way in your TI BASIC program. In this section, we will discuss the storage format employed for variables. If you want to see how constants are treated, look below in the section INSIDE A TI BASIC PROGRAM.

Numeric Variables

TI BASIC stores all numeric variable values in the format called *radix-100 notation*. As implemented in TI BASIC, this storage format provides 13 or 14 decimal digits of precision in 8 bytes of memory.

Radix-100 notation means that the numbers are stored as powers of 100. The following are examples of radix-100 numbers:

3 is stored as 3×100^0
58,623 is stored as 5.8623×100^2
186,238 is stored as 18.6238×100^2
15,162,923 is stored as 15.162923×100^3
115,162,923 is stored as 1.15162923×100^4

The numeric value is composed of two parts. The first byte is an exponent, followed by seven bytes of mantissa as shown in Fig. 5-1.

The exponent is a power of 100 and is stored in *excess 64* notation. You obtain the actual exponent value by subtracting 64 (>40) from the exponent stored value. For example, the exponent +5 is stored as:

+5 = 64 + 5 = 69 decimal = >45 hexadecimal

The mantissa is coded as a sequence of pairs of decimal digits. The significant digits (not counting trailing zeros) of the numeric value are parsed, beginning on the right, into a series of two digit pairs, like this:

decimal value 94,568,258 (94.568458×100^3)

is split into four pairs:

94 56 82 58

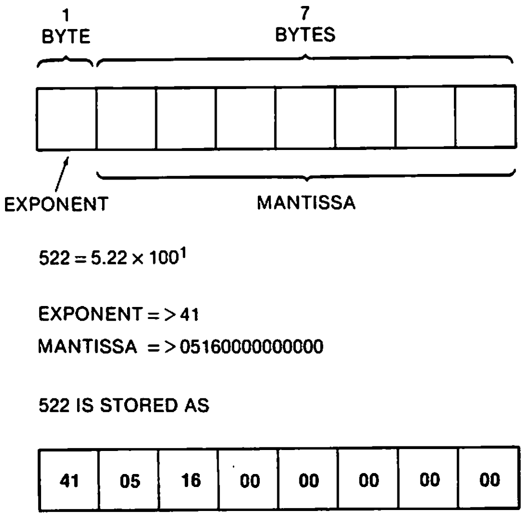


Fig. 5-1. Numeric data format.

or, in hexadecimal:

>5E >38 >52 >3A

The exponent is computed as:

>40 + >3 = >43

This results in the number being stored in 8 bytes as:

>43 >5E >38 >52 >3A >00 >00 >00

or, in decimal:

67 94 56 82 58 00 00 00

In this encoding scheme, the implied decimal point always follows the first two digits of the mantissa. This means that if you have an odd number of significant digits, you must pad it on the left with a zero. Consider, for example:

$$5 = 5 \times 100^0$$

This is stored as:

>40 >05 >00 >00 >00 >00 >00 >00

or, in decimal

64 05 00 00 00 00 00 00

Padding in a larger number looks like this:

115,162,923 is stored as 1.15162923×100^4
or
>44 >01 >0F >10 >1D >17 >00 >00
or
68 01 15 16 29 23 00 00 decimal

NOTE

The value zero (0) is always stored as:

>00 >00 >hh >hh >hh >hh >hh >hh

Where the >hh are undefined hexadecimal values. If the first two bytes are zero, TI BASIC does not look at the rest of the number.

A negative number is indicated by storing the first two bytes of the value in *2s complement* format. The 2s complement format is computed by taking the first two bytes of the number, turning all the one bits to zero, all the zero bits to one (complementing the value), then adding one (1) to the two-byte result. For example, the value 5 is stored as:

>40 >05 >00 >00 >00 >00 >00 >00

While −5 is stored as:

>BF >FB >00 >00 >00 >00 >00 >00

This is computed from the first two bytes as follows:

>4005 = 0100 0000 0000 0101

complemented:

>BFFA = 1011 1111 1111 1010

plus one:

>BFFB = 1011 1111 1111 1011

A larger negative number is stored like this (the positive form is shown above):

−115,162,923 is stored as -1.15162923×100^4
or
>BB >FF >0F >10 >1D >17 >00 >00

String Variables

String variables are stored in a very simple way. As you can see in Fig. 5-2, a string variable is stored as a one-byte length indicator followed by the characters in the string.

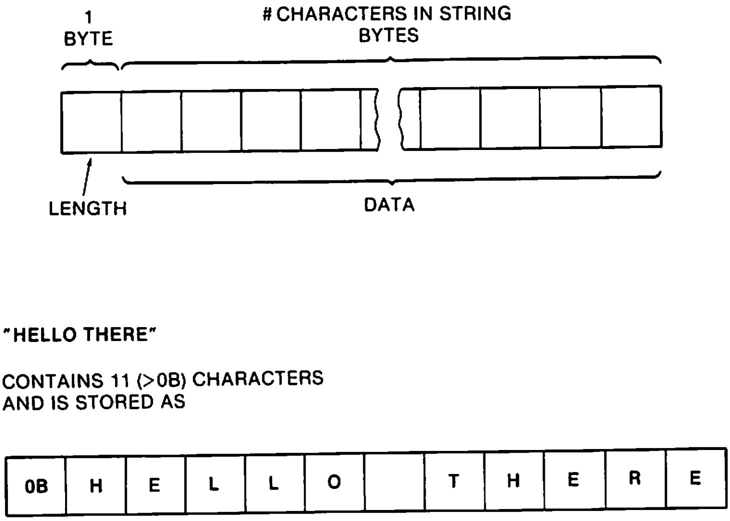


Fig. 5-2. String data format.

The *length byte* contains the current length of the string in characters. Since a single byte can contain values from 0 to 255, a string cannot be longer than 255 characters.

Thus, a simple string variable occupies its length plus one byte of storage.

String data storage is not allocated until you reference the variable. Because string variables vary in size during program execution, they tend to fragment available memory. If TI BASIC needs memory for some reason but finds there is an insufficient amount directly available, it attempts to free some memory by performing a "garbage collect" on the string variables. This action squeezes out the unused memory between the strings, thus freeing it for other uses.

Array Variables

Numeric arrays are stored as groups of simple numeric variables. All elements in a numeric array are stored in a contiguous block of memory—one element immediately following another.

To calculate the memory used by a numeric array, you simply multiply the number of elements in the array by 8.

String arrays are a little more complicated. When you define a string array (usually with a DIM statement), TI BASIC allocates a table of *two-byte pointers*. These pointers are used to record the location of the corresponding string data item when it is used.

The elements in a string array are allocated dynamically. No string array element is assigned storage until you refer to it in your program. Once you do refer to a particular element, it is allocated and a pointer to it is placed into the corresponding location in the string array pointer table.

The allocated string array elements look just like simple string variables: a length byte followed by the characters in the string.

Therefore, each string array in your program uses memory as follows:

- Two bytes for each possible element in the string arrays
- Length + 1 bytes for each string array element actually referenced in your program.

INSIDE A TI BASIC PROGRAM

TI BASIC programs are stored internally in a compact form called *crunched* or *tokenized* code. In this format, all the TI BASIC keywords are reduced to a one-byte code as shown in Table 5-1.

Table 5-1. TI BASIC Keyword Tokens

	Token	
Keyword	Decimal	Hexadecimal
ABS	203	>CB
APPEND	249	>F9
ASC	220	>DC
ATN	204	>CC
BASE	241	>F1
BREAK	142	>8E
CALL	157	>9D
CHR$	214	>D6
CLOSE	160	>A0
COS	205	>CD
DATA	147	>93
DEF	137	>89
DELETE	153	>99
DIM	138	>8A
DISPLAY	162	>A2
ELSE	129	>81
END	139	>8B
EOF	202	>CA
EXP	206	>CE
FIXED	250	>FA
FOR	140	>8C
GO	133	>85
GOSUB	135	>87
GOTO	134	>86
IF	132	>84
INPUT	146	>92
INT	207	>CF
INTERNAL	245	>F5
LEN	213	>D5
LET	141	>8D
LOG	208	>D0
NEXT	150	>96
ON	155	>9B
OPEN	159	>9F

Table 5-1. *(continued)*

Keyword	Token	
	Decimal	Hexadecimal
OPTION	158	>9E
OUTPUT	247	>F7
PERMANENT	251	>FB
POS	217	>D9
PRINT	156	>9C
RANDOMIZE	149	>95
READ	151	>97
REC	222	>DE
RELATIVE	244	>F4
REM	154	>9A
RESTORE	148	>94
RETURN	136	>88
RND	215	>D7
SEG$	216	>D8
SEQUENTIAL	246	>F6
SGN	209	>D1
SIN	210	>D2
SQR	211	>D3
STEP	178	>B2
STOP	152	>98
STR$	219	>DB
SUB	161	>A1
TAB	252	>FC
TAN	212	>D4
TEMPORARY	242	>F2
THEN	176	>B0
TO	177	>B1
TRACE	144	>90
UNBREAK	143	>8F
UNTRACE	145	>91
UPDATE	248	>F8
VAL	218	>DA
VARIABLE	243	>F3

TI BASIC delimiters and operators are also encoded. Table 5-2 shows you the delimiter/operator and its coded one-byte value.

A very simple statement like:

```
PRINT
```

is stored in the program as a single byte:

>9C or 156

This reduces the length of this statement from 5 characters to one.

The line numbers attached to your program statements are arranged in a table of 4-byte entries, separate from the program lines. Two bytes record the line number and the other two bytes store a pointer to the line.

Constants in Your Program

There are three types of constants in a TI BASIC program:

Table 5-2. TI BASIC Delimiter and Operator Tokens

Delimiter or Operator	Token	
	Decimal	Hexadecimal
,	179	>B3
;	180	>B4
:	181	>B5
)	182	>BB
(	183	>B7
&	184	>B8
=	190	>BE
<	191	>BF
>	192	>C0
+	193	>C1
−	194	>C2
*	195	>C3
/	196	>C4
^	197	>C5
#	253	>FD

- *string* constants ("GLOBAL THERMONUCLEAR WAR")
- *numeric* constants (35.69)
- *line number* constants (GOTO 500)

String constants within a statement are preceded by a two-byte header composed of:

- A string constant indicator which is always >C7 (199 decimal)
- A length byte giving the length of the string data which immediately follows

The string constant data is stored as standard ASCII codes. A simple statement like:

PRINT "HI THERE!"

is stored as:

>9C >C7 >09 >48 >49 >20 >54 >48 >45 >52 >45 >21

Where:

>9C = the token for PRINT
>C7 = start of string constant indicator
>09 = length of string constant that follows
>48 = "H"
>49 = "I"
>20 = (space)
>54 = "T"
>48 = "H"
>45 = "E"

>52 = "R"
>45 = "E"
>21 = "!"

Numeric constants are stored in a similar manner, except the numeric constant field indicator is a >C8 (200 decimal). Other than that, the format is exactly the same as for a string constant.

The statement:

PRINT 23.6

is stored as:

>9C >C8 >04 >32 >33 >2E >36

Where:

>9C = the token for the PRINT
>C8 = start of numeric constant indicator
>04 = length of the numeric constant that follows
>32 = "2"
>33 = "3"
>2E = "."
>36 = "6"

Because they are stored as ASCII characters, numeric constants must be converted to internal numeric format before they are used in a calculation.

Line number constants are coded in TI BASIC programs as 16-bit signed binary values. This allows the full range of legal line numbers (1 to 32767) to be stored in a 2-byte field.

Line number constants also have an indicator byte, with a value of >C9 (201 decimal), but no length byte since they are always two bytes long.

Consider, for example, the statement:

GOTO 500

This statement is stored as:

>86 >C9 >01F4

Where:

>86 = token for GOTO
>C9 = start of line number constant indicator
>01F4 = 500 as a two-byte binary value.

Variables in Your Program

Variable names are stored in ASCII code exactly as they appear. For example, the statement:

```
START = 1
```

is "crunched" to:

>53 >54 >41 >52 >54 >BE >C8 >01 >31

Where:

>53 = "S"
>54 = "T"
>41 = "A"
>52 = "R"
>54 = "T"
>BE = token for an equal sign (=)
>C8 = start of numeric constant indicator
>01 = length of the numeric constant that follows
>31 = "1"

Subprogram names are stored the same way as variables. For example, the statement:

CALL CLEAR

is stored as:

>9D >43 >4C >45 >41 >52

Where:

>9D = token for CALL
>43 = "C"
>4C = "L"
>45 = "E"
>41 = "A"
>52 = "R"

APPENDIX A

BASIC Statements, Commands, Functions Summary

This Appendix is a quick reference listing of TI-99/4A BASIC commands and statements. They are listed in alphabetical order for easy reference.

NOTATION

Whenever the format for a statement or command is given, the following rules apply:

1. Words in BOLDFACE AND CAPITALS are *keywords* that you enter exactly as they appear.
2. Words in reversed letters designate keystrokes.
3. *num-exp* means any *numeric expression*, like A + B, 42.34.
4. *num-var* means any *numeric variable*, like X, INTEREST.
5. *str-exp* means any *string expression*, like A$, "XYZ", FIRST$&MIDDLE$&LAST$.
6. *str-var* means any *string variable*, like Y$, NAME$.
7. *variable* means *any* variable, string or numeric, like YES$, PAYMENT.
8. *brackets* ([ ]) mean whatever is between the [ ] is *optional* and you do not have to use it.
9. *ellipsis* (, . . .) means that the preceding item can be repeated as many times as necessary.
10. *device-filename* means the device for cassette files (like CS1). For disk files, it means the name of the file on the disk as well as the device name (like DSK1.MYFILE).

ABS

ABS(*num-exp*)—A function that returns the absolute positive value of *num-exp*.

ASC

ASC(*str-exp*)—A function that returns the ASCII value of the first character of *str-exp*.

ATN

ATN(*num-exp*)—A trigonometric function that returns the arctangent of *num-exp*. The arctangent is the angle whose tangent is *num-exp* radians.

BREAK

BREAK [*line-num-list*]—A command or statement that makes your BASIC program stop until you enter a CONTINUE command. If you use BREAK with a list of line numbers (*line-num-list*), your program stops when it reaches any line in *line-num-list*.

UNBREAK deactivates all BREAK commands.

BYE

BYE—A command that closes all open files and leaves BASIC.

CALL CHAR

CALL CHAR(*ASCII-code,pattern-string*)—A command or statement that redefines the pattern (or image) associated with the character represented by *ASCII-code*. The new pattern is given in the 16 digit hexadecimal *pattern-string*.

CHR$

CHR$(*num-exp*)—A function that returns a one character string representing the character whose ASCII value is *num-exp*.

CALL CLEAR

CALL CLEAR—A command or statement that "clears the screen" to all blank characters (the character with ASCII value 32).

CLOSE #

CLOSE # *file-number* [:DELETE]—A statement or command that closes the file OPENed as *file-number* and, if you say DELETE, removes the file from the device.

You cannot delete files from a cassette tape. If you say DELETE with a cassette file, the file is closed but not removed from the tape.

CALL COLOR

CALL COLOR(*char-set,foreground-color,background-color*)—A statement or command that sets the foreground and background colors for the eight characters in *char-set*.

CONTINUE or CON

CONTINUE or CON—A command that resumes executing a program after the program has executed a BREAK statement/command, had an error occur, or you pressed FCTN CLEAR.

You cannot CONTINUE a program after you have edited it.

COS

COS*(rad-angle)*—A trigonometric function that returns the cosine of the angle *rad-angle* where the angle is expressed in radians.

DATA

DATA *data-list*—A statement that stores numeric or string data in a program.

You use READ statements to put the values in the DATA statement *data-list* into variables in your program. You can select specific DATA with RESTORE statements.

DEF

DEF *fctn-name*[*(parameter)*] = *expression*—A statement that defines a numeric or string function with the name *fctn-name*. You can pass a value to the function with *parameter*. The value returned by the function is defined by *expression*.

You often use a function instead of rewriting *expression* every place you need it. This saves space in your program and makes it easy to change *expression*.

DELETE

DELETE "*device-filename*"—A command or statement that deletes file *filename* from *device*. You cannot use DELETE with a cassette.

DIM

DIM *array-name(dim1*[*,dim2*[*,dim3*]]*)* [, . . .]—A statement that allocates (dimensions) space for arrays. Each array (*array-name*) gets the number of elements (*dim1* . . .) allocated. Each array can have up to 3 dimensions in TI BASIC.

DISPLAY

DISPLAY [*list*]—A statement or command that writes the data in *list* to the screen (as a PRINT statement).

EDIT

EDIT *line-num*—A command that lets you change line *line-num*.

END

END—A statement or command that stops your program's execution.

EOF

EOF [*(file-num)*]—A function that tells you if you are at the end of the file *file-num*. If you are not, you get a 0. If you are, you get a 1. If there is no more room on the disk, you get a −1.

The EOF function does not work with cassette files.

EXP

EXP*(num-exp)*—A function that returns the exponential value of *num-exp*. This is $\underline{e}^x$ where $\underline{e}=2.718281828$.

FOR

FOR *control* = *init-val* TO *end-val* [STEP *incr*]—A statement that repeatedly executes the statements between the FOR and its associated NEXT statement. A control variable, *control*, starts at *init-val*. Each time the associated NEXT statement is executed, *control* is incremented by *incr* or by one (if you do not use STEP). If *control* is less than *end-val*, the statements between FOR and NEXT are executed again.

CALL GCHAR

CALL GCHAR (*row,col,num-var*)—A statement or command that puts the ASCII code for the character at position *row* and *col* into variable *numvar.*

GOSUB or GO SUB

GOSUB *line-num*—A statement that transfers control to the subprogram at line *line-num*.

GOTO or GO TO

GOTO *line-num*—A statement that unconditionally transfers control to the statement at line *line-num*.

CALL HCHAR

CALL HCHAR *(row,col,ASCII-code*[*,repetitions*]*)*—A statement or command that writes the character with value *ASCII-code* at row *row* and column *col*. If you use a value for *repetitions*, you will get that many characters written across the screen beginning at *row,col*.

IF

IF *condition* THEN *line-num1* [ELSE *line-num2*]—A statement that determines if *condition* is true or false and transfers control to line number *line-num1* when the expression is true or line number *line-num2* when the expression is false.

INPUT

INPUT [*prompt:*] *variable-list*—A statement that writes a message (*prompt*) to the screen and reads data into the variables in *variable-list*.

INPUT

INPUT# *file-num* : *variable-list*—A statement reads data from file *file-num* into the variables in *variable-list*.

INT

INT*(num-exp)*—A function that returns the largest integer value less than or equal to *num-exp*.

CALL JOYST

CALL JOYST*(key-unit,x-return,y-return)*—A command or statement that reads the position of either joystick (*key-unit* = 1 or 2).

CALL KEY

CALL KEY *(key-unit, return-var,status-var)*—A command or statement that returns the ASCII value of the key pressed in the *return-var.*

LEN

LEN *(str-exp)*—A function returns the number of characters in *str-exp*.

LET

[LET] *variable* = *expression*—A statement or command that assigns the value of *expression* to *variable*. The keyword LET is an optional part of an assignment statement.

LIST

LIST [[*start-line*] [– [*end-line*]]]—A command lists the lines from the BASIC program in memory, beginning with line *start-line* and ending with line *end-line*.

LOG

LOG*(num-exp)*—A function returns the natural logarithm of *num-exp*.

NEW

NEW—A command that clears the computer's memory, clears the screen, and gets ready to accept a new program.

NEXT

NEXT [*control*]—A statement that ends a FOR loop.

NUMBER or NUM

NUMBER or NUM [*start-line*[*,increment*]]—A command that generates sequenced line numbers for entering a BASIC program.

OLD

OLD *device*[*.program-name*]—A command that loads a BASIC program from device *device*. If you are loading a program from a disk, you need to use *program-name*.

ON . . . GOSUB . . .

ON *num-exp* GOSUB *line-num-list*—A statement that transfers control to the subprogram *num-exp* line number in *line-num-list*.

ON . . . GOTO . . .

ON *num-exp* GOTO *line-num-list*—A statement that unconditionally transfers control to the *num-exp* line number in *line-num-list.*

OPEN

OPEN # *file-num: device-file name* [,*file-org*] [,*file-type*] [,*open-mode*] [,*record-type*]—A statement or command that associates the specified file with number *file-num* and enables the program to read/write data from/to the file.

OPTION BASE 0 or 1

OPTION BASE—A statement that sets the lowest subscript for all arrays to zero or one.

POS

POS *(string1,string2,num-exp)*—A function that returns the position of the first occurrence of *string2* in *string1* beginning the search at character *num-exp* in *string1*.

PRINT

PRINT [*#file-num* [,REC *rec-num*] :] [*list*]—A statement or command that writes the data in *list* to the screen or to the file *file-num.*

RANDOMIZE

RANDOMIZE [*num-exp*]—A statement that resets the random number generator.

READ

READ *variable-list*—A statement or command that assigns values from DATA statements to the variables in *variable-list.*

REM

REM *string*—A statement that lets you include remarks (nonexecutable statements) in a BASIC program.

RESEQUENCE or RES

RESEQUENCE or RES [*initial*] [,*increment*]—A command that renumbers the lines in the BASIC program currently in memory.

RESTORE

RESTORE [*line-num*]—A statement or command that resets the line number for DATA statement used in the next READ statement.

RESTORE

RESTORE *#file-num* [,REC *rec-num*]—A statement or command that resets the current record number for file *file-num.* This is the record used in the next INPUT #$ statement for the file.

RETURN

RETURN—A statement that transfers program control from a subprogram to the statement following the GOSUB or ON GOSUB statement that called the subprogram.

RND

RND—A function that returns a random number between 0 and 1.

RUN

RUN—A command that executes the BASIC program currently in memory.

SAVE

SAVE *device-filename*—A command that writes the BASIC program currently in memory to *device-filename*.

CALL SCREEN

CALL SCREEN *(color-code)*—A statement or command that changes the screen color to that given by *color-code*.

SEG$

SEG$ *(str-exp,position,length)*—A function that returns a substring of *str-exp*. The returned string is *length* characters long and begins at character *position* in *str-exp*.

SGN

SGN *(num-exp)*—A function that returns a one if *num-exp* is positive, a zero if *num-exp* is zero, and a minus one if *num-exp* is negative.

SIN

SIN *(rad-angle)*—A trigonometric function that returns the sine of *rad-angle* where *rad-angle* is in radians.

CALL SOUND

CALL SOUND *(duration,freq1,vol1*[, . . . *,freq4,vol4*]*)*—A statement or command that controls the tone and noise generator. You get a tone of frequency *freq1* at *vol1* for *duration* milliseconds. You can get up to four simultaneous tones (all for the same *duration*).

SQR

SQR *(num-exp)*—A function that returns the square root of *num-exp*.

STOP

STOP—A statement or command that stops program execution. You can use STOP statements anywhere in your program.

STR$

STR$ *(num-exp)*—A function that converts *num-exp* into its string form. VAL works the other way, changing a string to a numeric form.

TAB

TAB (*num-exp*)—A function that positions PRINT or DISPLAY statements at column *num-exp*.

TAN

TAN *(rad-angle)*—A trigonometric function returns the tangent of *rad-angle* where *rad-angle* is expressed in radians.

TRACE

TRACE—A command or statement that lists the line numbers of statements before they are executed.

UNBREAK

UNBREAK [*line-num-list*]—A command or statement that removes the breakpoints for the lines in *line-num-list* or all breakpoints (if you do not use *line-num-list*).

You set breakpoints with a BREAK command.

UNTRACE

UNTRACE—A command or statement that cancels a TRACE command. Line numbers are no longer printed before the statements are executed.

VAL

VAL *(str-exp)*—A function that converts *str-exp* to a numeric form.

CALL VCHAR

CALL VCHAR *(row,col,ASCII-code*[*,repetitions*]*)*—A statement or command that writes the character with the ASCII value *ASCII-code* at row *row* and column *col*. If you use a value for *repetitions*, you will get that many characters written down the screen (vertically) beginning at *row,col*.

APPENDIX B

Glossary

ADDRESS—A unique number assigned to each memory location (byte). The TI-99/4A generates addresses between 0 and 65,535—a 64K range.

ARGUMENT—The actual value that is passed to a function. Note that when it is received by the function, it is a parameter.

ARRAY—A group data item all of whose elements are of the same data type and are referenced by the same name. See also bounds and subscripts.

ASCII—The American Standard Code for Information Interchange is a set of one-byte codes used by many computer systems to represent letters, digits, punctuation marks, and special control codes.

BACKUP—A secure second copy of important information. You should make backups of all tape and disk resident data and programs that you cannot, or do not want to, recreate. The backup can be another tape or disk, or, less desirable, a paper listing of the program or data.

BASIC—Beginners All Purpose Symbolic Instruction Code. BASIC is the most widely used programming language on microcomputers today. TI BASIC conforms to the American National Standard.

BINARY—The base 2 number system computers use to count. The binary system has only two digits (0 and 1) that correspond to the "on" and "off" bits in a computer memory.

BIT—A single binary digit. A bit can be either "on" (value 1) or "off" (value 0) and is the fundamental unit of computer memory. For most purposes, bits are usually arranged in groups of eight to form a byte.

BOUNDS—The upper and lower limits of the subscripts of an array. In BASIC, the lower limit can be either 0 or 1. The upper limit is set in a DIM statement or, by default, at 10.

BYTE—A unit of memory sufficient to store one character. A byte contains eight bits and is the unit most commonly used to measure memory size. The TI-99/4A console contains 16,384 bytes of user memory.

CHARACTER—A letter, number, space, or punctuation mark. Characters in the TI-99/4A are stored as ASCII codes in one byte of memory.

CLOSE—Break the link between your program and a file on an external device. It is essential to properly close files residing on disk.

COMMAND—The set of BASIC instructions that is usually entered and executed immediately. Commands act on your program or files; they do not operate on data elements in your program. Some commands can also be included in programs. Some BASIC statements can also be entered, without line numbers, and executed immediately as though they were commands.

CPU—The Central Processing Unit. The brain of the computer, the 16-bit 9900 microprocessor, is the CPU in the TI-99/4A.

DATA—Values that are manipulated by programs. There are two major types of data in BASIC, string data and numeric data.

DIRECT MODE—Statements and commands are entered without line numbers and are immediately executed.

DISK—A round, 5¼-inch, magnetic storage medium. A disk (sometimes called a diskette) works in a disk drive to store programs and data. Single sided disks store 90,000 characters of data; double sided disks store 180,000 characters.

DISK DRIVE—A fast mass storage peripheral that provides immediate access to programs and data. A disk drive is more than 30 times faster than a cassette recorder.

GRAPHICS—Drawing charts, graphs, or pictures on the screen.

HARDWARE—The physical components of your TI-99/4A. This includes the console, cassette cables, joysticks, disk drive—anything you can put your hands on.

HERTZ—A frequency measurement equal to one cycle per second.

HEXADECIMAL—A base 16 numbering system commonly used in computer systems. Hexadecimal is convenient to use because two hexadecimal digits (0 to 9 and A to F) can represent any one byte value.

INTERPRETER—A program that interprets and executes the statements in another program. BASIC is an interpreted language: the BASIC interpreter reads a BASIC statement, analyzes it, and does what it says.

MACHINE LANGUAGE—The *negative language* of the TMS9900 microprocessor. Human beings do not generally program in machine language. The closest approximation to machine language that is useful to people is Assembly Language, where each statement corresponds to a single machine language instruction.

MONITOR—A high quality television set. You do not need a monitor to use your TI-99/4A; with a Radio Frequency Modulator you can use an ordinary television set.

NULL STRING—A character string data item which has a zero length. A null string has no data in it.

OPEN—The establishment of a link between your program and a file on an external device. All files except the keyboard and screen (file #0) must be opened before they can be written to or read from.

OVERLAY—The plastic strips that fit into the slot above the number keys on the TI-99/4A console. The words on the overlay indicate the action of FCTN and CTRL shifted number keys.

PARAMETER—The symbol which represents the value received by a function. A parameter has no value until the function is invoked and an argument is passed to the parameter, thus giving it a value.

PERIPHERAL—A piece of hardware external to the TI-99/4A console.

PROGRAM—A set of detailed instructions which make the computer perform a desired task. You can write programs in many languages on the TI-99/4A, but BASIC is the most common language.

RAM—Random Access Memory is the memory that is available for your use. You can write to and read from RAM, but RAM forgets what you put in it after you turn off the TI-99/4A. The TI-99/4A console contains 16K of RAM and the system can expand to accommodate 52K of RAM.

READ—The moving of data from an external source into Random Access Memory. The TI-99/4A reads data from such places as the keyboard, cassette tapes, Wafertapes, the RS232 interface, disk drives, and a telephone modem.

RETURN A VALUE—BASIC functions return a value to their point of reference that is treated as though it were a numeric or string data value.

ROM—Read Only Memory has a program permanently stored in it. You cannot use ROM to store your programs. The TI-99/4A console contains 26K of ROM most of which contains the TI BASIC interpreter.

RS-232—An industry-standard hardware and software communications protocol. This is a set of rules defining the way computers talk to modems, printers, or other computers.

SOFTWARE—The instructions that make the computer do what you want.

STATEMENT—The set of BASIC instructions which normally appears as part of a program.

SUBPROGRAM—A logical program segment usually entered via a GOSUB statement and exited through a RETURN.

SUBSTRING—A string which is a segment (piece of) a larger string. In TI BASIC, the SEG$ function allows you to extract substrings from larger strings.

TMS9900—The 16-bit microprocessor that serves as the central processing unit in the TI-99/4A.

TMS9918A—The video display controller in the TI-99/4A. This sophisticated video chip maintains the image you see on the screen.

TMS9919—The sound generator chip in the TI-99/4A. This chip can create three tones and eight noises.

TRANSFER OF CONTROL—As BASIC executes a program, control is at the statement being executed. Control normally advances from a statement to the statement with the next higher line number. Some statements (GOTO, IF/THEN, GOSUB) cause control to transfer to a statement other than that with the next higher line number.

TRAP—The code you write to detect and report an error, sometimes before it occurs. The BASIC interpreter traps many errors before they cause your program to fail.

WRITE—The moving of data from Random Access Memory to an external device. The TI-99/4A can write data to such places as the screen, a cassette tape, a Wafertape, disks, the RS232 interface, a printer, and the telephone modem.

APPENDIX C

Reserved Words

TI BASIC reserves some words for its own use. You cannot use these reserved words as names for variables in your programs. However, you can use the reserved words as part of a variable name.

For example,
NEW cannot be used as a variable name.
NEWDATA or NEW_VAL can be used as variable names.

Table C-1. TI BASIC Reserved Words

These words cannot be used as variable names.			
ABS	END	OLD	SEG$
APPEND	EOF	ON	SEQUENTIAL
ASC	EXP	OPEN	SGN
ATN	FIXED	OPTION	SIN
BASE	FOR	OUTPUT	SQR
BREAK	GO	PERMANENT	STEP
BYE	GOSUB	POS	STOP
CALL	GOTO	PRINT	STR$
CHR$	IF	RANDOMIZE	SUB
CLOSE	INPUT	READ	TAB
CON	INT	REC	TAN
CONTINUE	INTERNAL	RELATIVE	THEN
COS	LEN	REM	TO
DATA	LET	RES	TRACE
DEF	LIST	RESEQUENCE	UNBREAK
DELETE	LOG	RESTORE	UNTRACE
DIM	NEW	RETURN	UPDATE
DISPLAY	NEXT	RND	VAL
EDIT	NUM	RUN	VARIABLE
ELSE	NUMBER	SAVE	

APPENDIX D

Editing Keys

The tables in this Appendix show you the special keys that you can use to make it easier to enter and edit TI BASIC programs.

Remember to hold down the FCTN *key when you press one of these editing keys.*

Table D-1. Line Editing Commands for Entering BASIC Programs

Key	Function
ENTER	Enter the program line. The line you are typing (line number and statement) is entered into the program currently in your computer's memory.
FCTN D (right-arrow)	Forwardspace one character. Move the cursor one character position to the right. No changes are made to any characters the cursor moves past. You use the FCTN D key to position your cursor when you want to add or delete characters on the line you're currently typing.
FCTN E (up-arrow)	Works just like the ENTER key. The program line you just typed is put into your computer's memory.
FCTN S (left-arrow)	Backspace one character. Move the cursor one character position to the left. No changes are made to any characters the cursor moves past. You use the FCTN S key to position your cursor when you want to add or delete characters on the line you're currently typing.
FCTN X (down-arrow)	Works just like the ENTER key. The program line you just typed is put into your computer's memory.
FCTN 1 (DEL)	Delete one character. Delete the character under the cursor. You usually use the FCTN S or FCTN D key to position the cursor to the character you want to delete.
FCTN 2 (INS)	Insert characters. Insert characters at the cursor position. You can use the FCTN S or FCTN D key to position the cursor to where you want to insert the characters. Unlike the other FCTN keys, INS puts you into *Insert Mode,* allowing you to insert as many characters as you need.
FCTN 3 (ERASE)	Erase the entire line. Does not erase the line number if you are in automatic line numbering mode (**NUMBER** command).

Table D-1. *(continued)*

Key	Function
FCTN 4 (CLEAR)	Clear the current line. Cancels the line you are typing. If you are in automatic line numbering mode, FCTN 4 erases the current line and ends NUM command processing.
FCTN = (QUIT)	Quit. Leave BASIC and return to the main title screen. Memory is erased. If you have files opened, they are not closed. Use a BYE command if you want your files closed. Remember, you lose the program in memory if you haven't saved it.

Table D-2. TI BASIC Editing Function Keys

Key	Function
ENTER	Enter the program line. The line you are editing (line number and statement) is entered into the program currently in your computer's memory.
FCTN D (right-arrow)	Forwardspace one character. Move the cursor one character position to the right. No changes are made to any characters the cursor moves past. You use the FCTN D key to position your cursor when you want to add or delete characters on the line you're currently typing.
FCTN E (up-arrow)	Enter the program line. The line you are editing (line number and statement) is entered into the program currently in your computer's memory. The statement with the next lower line number is then presented for editing.
FCTN S (left-arrow)	Backspace one character. Move the cursor one character position to the left. No changes are made to any characters the cursor moves past. You use the FCTN S key to position your cursor when you want to add or delete characters on the line you're currently typing.
FCTN X (down-arrow)	Enter the program line. The line you are editing (line number and statement) is entered into the program currently in your computer's memory. The statement with the next higher line number is then presented for editing.
FCTN 1 (DEL)	Delete one character. Delete the character under the cursor. You usually use the FCTN S or FCTN D key to position the cursor to the character you want to delete.
FCTN 2 (INS)	Insert characters. Insert characters at the cursor position. You can use the FCTN S or FCTN D key to position the cursor to where you want to insert the characters. Unlike the other FCTN keys, INS puts you into *Insert Mode,* allowing you to insert as many characters as you need.
FCTN 3 (ERASE)	Erase the entire line. Does not erase the line number.
FCTN 4 (CLEAR)	Clear the current line. Erases the current line and stops the editing process.
FCTN = (QUIT)	Quit. Leave BASIC and return to the main title screen. Memory is erased. If you have files opened, they are not closed. Use a **BYE** command if you want your files closed. Remember, you lose the program in memory if you haven't saved it.

APPENDIX E

ASCII Codes

This Appendix lists the ASCII codes for the standard character set.

Table E-1. ASCII Character Codes

ASCII Char	Decimal	ASCII Char	Decimal	ASCII Char	Decimal
(space)	32	@	64	‘	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
–	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	|	124
=	61	]	93	}	125
>	62	^	94	¯	126
?	63	_	95	(DEL)	127

APPENDIX F

Graphics

The tables in this Appendix show you the colors you can use and the way TI BASIC groups ASCII characters into sets for setting the foreground/background colors with COLOR.

Table F-1. COLOR Codes

foreground-color *background-color*	Color
1	Transparent
2	Black
3	Medium Green
4	Light Green
5	Dark Blue
6	Light Blue
7	Dark Red
8	Cyan
9	Medium Red
10	Light Red
11	Dark Yellow
12	Light Yellow
13	Dark Green
14	Magenta
15	Gray
16	White

Table F-2. COLOR Character Sets

char-set	*ASCII-values*
1	32-39
2	40-47
3	48-55
4	56-63
5	64-71
6	72-79
7	80-87
8	88-95
9	96-103
10	104-111
11	112-119
12	120-127
13	128-135
14	136-143
15	144-151
16	152-159

NOTE: Sets 1 through 12 are the characters in the standard character set. Sets 13 through 16 are the special character set.

APPENDIX G

Joysticks

This Appendix lists the values that JOYST and KEY return when you get information about the joysticks from TI BASIC.

JOYST returns the position of the joystick levers, as shown in Fig. G-1.

KEY returns the status of the joystick FIRE buttons, as shown in the tables.

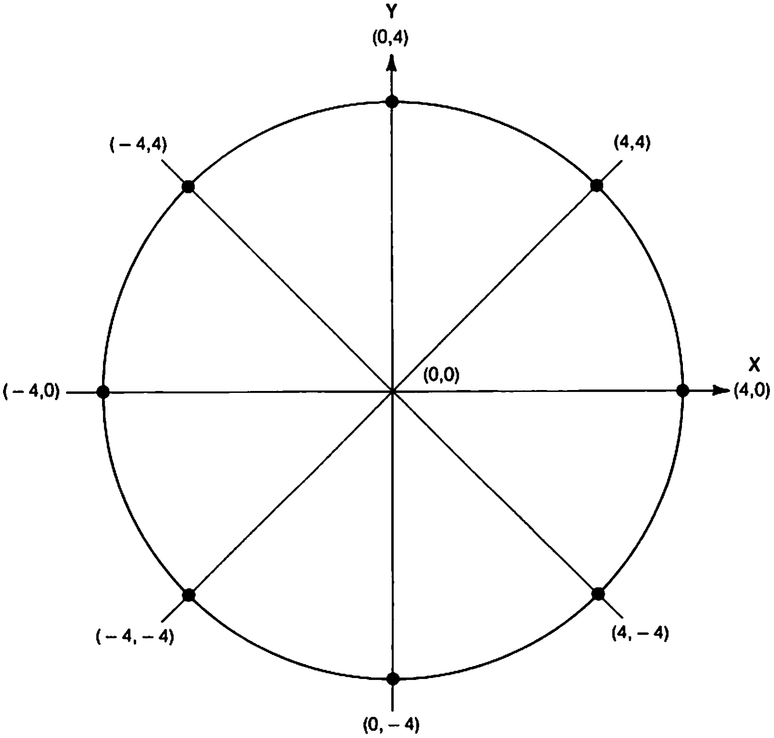


Fig. G-1. JOYST Values.

Table G-1. KEY *key-unit* Values for the Joysticks

key-unit	Meaning
1	Read the FIRE button on joystick 1
2	Read the FIRE button on joystick 2

Table G-2. KEY *return-var* Values for the Joysticks

return-var	Meaning
0	The FIRE button was not pressed
18	The FIRE button was pressed

Table G-3. KEY *status-var* Values for the Joysticks

status-var	Meaning
−1	The FIRE button is pressed and was also pressed the last time you used CALL KEY for this joystick.
0	The FIRE button is not pressed when you CALL KEY for the joystick.
+1	The FIRE button is pressed now but was not pressed the last time you used CALL KEY for this joystick.

APPENDIX H

Sound

This Appendix lists the frequency values that give you true musical notes when you CALL SOUND and those that give you sound effect noises.

Table H-1. Frequencies for Musical Notes

N#	Freq	Note	N#	Freq	Note
0	110	A	32	698	F
1	116	A flat, B sharp	33	739	F sharp, G flat
2	123	B	34	783	G
3	130	C (low C)	35	830	G sharp, A flat
4	138	C sharp, D flat	36	880	A (above high C)
5	146	D	37	923	A sharp, B flat
6	155	D sharp, E flat	38	987	B
7	164	E	39	1046	C
8	174	F	40	1108	C sharp, D flat
9	185	F sharp, G flat	41	1174	D
10	196	G	42	1244	D sharp, E flat
11	207	G sharp, A flat	43	1318	E
12	220	A (below middle C)	44	1396	F
13	233	A sharp, B flat	45	1479	F sharp, G flat
14	246	B	46	1567	G
15	261	C (middle C)	47	1661	G sharp, A flat
16	277	D sharp, D flat	48	1760	A
17	293	D	49	1864	A sharp, B flat
18	311	D sharp, E flat	50	1975	B
19	329	E	51	2093	C
20	349	F	52	2217	C sharp, D flat
21	369	F sharp, G flat	53	2349	D
22	392	G	54	2489	D sharp, E flat
23	415	G sharp, A flat	55	2637	E
24	440	A (above middle C)	56	2793	F
25	466	A sharp, B flat	57	2959	F sharp, G flat
26	493	B	58	3135	G
27	523	C (high C)	59	3322	G sharp, A flat
28	554	C sharp, D flat	60	3520	A
29	587	D	61	3729	A sharp, B flat
30	622	D sharp, E flat	62	3591	B
31	659	E	63	4186	C
			64	4434	C sharp, D flat

NOTE: The value in the N# (note number) column is the factor used to get a true musical note with this calculation:

FREQUENCY = 110 * (2^(1/12)) ^ N#

Table H-2. Frequencies for Noises

Frequency	Noise Description
−1	Periodic noise type 1
−2	Periodic noise type 2
−3	Periodic noise type 3
−4	Periodic noise that varies with the frequency of the third tone in the CALL SOUND
−5	White noise type 1
−6	White noise type 2
−7	White noise type 3
−8	White noise that varies with the frequency of the third tone in the CALL SOUND

APPENDIX I

Derived Trigonometric Functions

This Appendix lists common derived trigonometric functions as user functions (DEF statement). PI = 3.141592653589793 (or as many decimal places as you want to use).

FUNCTIONS

Secant:
`DEF SEC(rad-ang) = 1/COS(rad-ang)`
Cosecant:
`DEF CSC (rad-ang) = 1/SIN(rad-ang)`
Cotangent:
`DEF COT(rad-ang) = 1/TAN(rad-ang)`

INVERSE FUNCTIONS

Inverse Sine:
`DEF ARCSIN(n) = ATN(n/SQR(1 - n*n)`
Inverse Cosine:
`DEF ARCCOS(n) = - ATN(n/SQR(1 - n*n)) + PI/2)`
Inverse Secant:
`DEF ARCSEC(n) = ATN(SQR(n*n - 1)) + (SGN(n) - 1)*PI/2`
Inverse Cosecant:
`DEF ARCCSC(n) = ATN(1/SQR(n*n - 1)) + (SGN(n) - 1)*PI/2`
Inverse Cotangent:
`ARCCOT(n) = ATN( - n) + PI/2`

HYPERBOLIC FUNCTIONS

Hyperbolic Sine:
`DEF SINH(n) = (EXP(n) - EXP( - n))/2`
Hyperbolic Cosine:
`DEF COSH(n) = (EXP(n) + EXP( - n))/2`

Hyperbolic Tangent:
DEF TANH(n)=(EXP(n)-EXP(-n))/(EXP(n)+EXP(-n))
Hyperbolic Secant:
DEF SECH(n)=2/(EXP(n)+EXP(-n))
or
DEF SECH(n)=1/COSH(n)
Hyperbolic Cosecant:
DEF CSCH(n)=2/(EXP(n)-EXP(-n))
or
DEF CSCH(n)=1/SINH(n)
Hyperbolic Cotangent:
DEF COTH(n)=2*EXP(-n)/EXP(n)-EXP(-n))+1
or
DEF COTH(n)=1/TANH(n)

INVERSE HYPERBOLIC FUNCTIONS

Inverse Hyperbolic Sine:
DEF ARCSINH(n)=LOG(n+SQR(n*n+1))
Inverse Hyperbolic Cosine:
DEF ARCCOSH(n)=LOG(n+SQR(n*n-1))
Inverse Hyperbolic Tangent:
DEF ARCTANH(n)=LOG((1+n)/(1-n))/2
Inverse Hyperbolic Secant:
DEF ARCSECH(n)=LOG((1+SQR(1-n*n))/n)
Inverse Hyperbolic Cosecant:
DEF ARCCSCH(n)=LOG((SGN(n)*SQR(n*n+1)+1)/n)
Inverse Hyperbolic Cotangent:
DEF ARCCOTH(n)=LOG(n+1)/(n-1))/2

APPENDIX J

TI BASIC Errors

This Appendix lists the TI BASIC error messages in alphabetical order, along with hints to help you find out why the error occurred.

INTRODUCTION

Errors occur at several different times in your program's life:

1. When you enter a command or statement for immediate execution
2. When you are editing an existing program line
3. When TI BASIC is generating its symbol table before it starts executing your program
4. When a program is actually executing

Errors Can Occur When You Enter or Edit a Line

You get some errors as soon as you press the **ENTER** key after entering or editing a line.

These errors are usually caused by:

- Bad line numbers
- Variable names too long
- Using a command as a statement (with a line number)
- Using a statement as a command (without a line number)

These errors also occur when you enter a line that is too long (more than 112 characters), or when there is not enough memory left to accommodate the statement you entered.

You also get these errors when you enter an incorrect form of a TI BASIC instruction. Perhaps you misspelled the keyword and TI BASIC cannot recognize the instruction. Or, you forgot an equals sign (=) in an assignment statement. TI BASIC needs one keyword in every instruction that you give it.

Errors Can Occur When TI BASIC Is Generating Its Symbol Table

You might notice a slight delay after you enter RUN and before your program actually starts executing. That is because, when you RUN a program, TI BASIC first scans the program to generate a *symbol table*—a list of variables, arrays, functions, and other values used in your program.

TI BASIC also looks for some formatting errors, FOR-NEXT pairing, proper dimension bounds, and name conflicts (using the identical name for an array, function, and simple variable).

You can tell when an error occurs during this phase because your screen remains cyan while TI BASIC generates its symbol table.

Whenever possible, the error message contains the line number of the statement causing the error.

Errors Can Occur When Your Program Is RUNning

By far the most common time for your program to get errors is when it is actually executing. The screen turns light green (color 4) when TI BASIC finishes generating its symbol table and begins running your program.

You sometimes get warning messages while your program is running. TI BASIC lets you know that something is wrong but that it can substitute another value and continue running your program. For example, if TI BASIC gets a number larger than the largest it can handle, TI BASIC substitutes 9.9999999999999E127 for the number and prints the message:

```
NUMBER TOO BIG
```

When an error occurs, TI BASIC prints an error message that contains the line number of the statement causing the error and stops your program.

TI BASIC ERRORS

TI BASIC's errors and warnings are shown in alphabetical order so that you can easily find them. After each message, we list the most common causes for the error and where to look in your program to see what happened.

Bad Argument

This error is most likely to occur when the ASC or VAL functions get a null string as an argument or when the VAL function argument does not represent a valid number.

Bad Line Number

You used a line number that is zero, less than zero, or greater than 32767. This often happens when you RES a program with a large increment causing your new line numbers to get too big.

Or, your program is running and you used a line number in a GOTO, GOSUB, BREAK, or UNBREAK statement that is not really a line number in your program. Perhaps you mistyped the line number, or you removed lines from your program and did not find all references to the removed lines.

RES gives you a line number of 32767 in GOTO and GOSUB statements that reference line numbers that are not in your program. It does not warn you when this happens.

Bad Name

You get this error when you are entering your program and you use more than 15 characters in a variable name. Remember that the ending dollar sign ($) counts as a character in a string variable name. You can also get this error if a variable name contains an illegal character.

You get this error when your program is running and you use one of the TI BASIC subprograms but misspell the subprogram name. TI BASIC subprograms are CALLed, like CALL HCHAR, CALL COLOR, CALL CHAR, etc.

Bad Subscript

You get this error only when TI BASIC is referencing an array with one or more invalid subscripts.

Usually, the current value of one or more subscripts is less than zero or greater than the maximum value you specified in your DIM statement for the array (or greater than 10 if you did not use a DIM statement for the array).

If you use an OPTION BASE 1 statement, you cannot use a subscript of zero for any array in your program.

Bad Value

This error generally occurs when you use an out-of-range value for an argument to one of TI BASIC's functions or subprograms, like CHAR, CHR$, COLOR, GCHAR, HCHAR, JOYST, KEY, POS, SCREEN, SEG$, SOUND, TAB, or VCHAR.

You get this error if you attempt to raise a negative number to a fractional power with the exponent operator (^). Try PRINTing the values in the statement if you get this error in a statement that uses exponentiation.

In ON . . . GOSUB and ON . . . GOTO statements, you get this error if your expression evaluates to zero, less than zero, or more than the number of line numbers in your statement.

This error can also occur when you are referencing a file. The culprit can be a bad file number (greater than 255) in an OPEN, CLOSE, INPUT #, PRINT #, or RESTORE # statement; the number of records (greater than 32767) in an OPEN statement's SEQUENTIAL or RELATIVE option; or, the record length (greater than 32767) in an OPEN statement's FIXED option.

Can't Continue

You tried to CON but you have not used a BREAK to set breakpoints or none of the breakpoints was reached yet. This also occurs when you edit your program after a breakpoint occurs. You tried to CONTINUE after encountering an unrecoverable error.

Can't Do That

This error occurs when you try to get TI BASIC to do something that it cannot do.

Maybe you put too many OPTION BASE statements in your program (you can have only one per program) or you have a RETURN statement without any previous GOSUB or a NEXT statement without encountering a previous FOR statement.

You also get this error when your program is running and the control variable on a NEXT statement that is executed does not match the control variable on the most recently executed FOR statement.

A BREAK command without a line number causes this error since TI BASIC does not know where to set the breakpoint. You can, however, use BREAK as a statement in a program without supplying any line number after the BREAK. In this case, TI BASIC will set the breakpoint at the line number of the BREAK.

Check Program in Memory

You used an OLD command to read a program into memory from a tape or disk and something happened before the program was entirely read into memory.

Perhaps there was a problem with the cassette recorder. Or you used the wrong filename for a program on disk. Or you had the wrong diskette in the disk drive and the program could not be found.

You also get this error when you press FCTN CLEAR (FCTN 4) to stop TI BASIC when it is loading a program into memory. TI BASIC does not clear the current program from its memory before it starts reading in a new program so you may have part of the new program and part of the old program mixed together in your computer. Use LIST to see what is where.

Data Error

This error occurs when TI BASIC detects something wrong with a DATA, READ, or RESTORE statement. PRINT the values in the READ statement's variable list to see if they contain what you expect.

If the error occurs in a RESTORE statement, you used a line number (for the DATA statement to RESTORE) that is larger than the largest line number in your program. Correct the line number.

Perhaps you forgot a comma between two items in a DATA statement. Or, you executed a READ statement and there are no more values to be read from DATA statements.

Or, you assigned a string value to a numeric variable. This happens when the variables in your READ statements and the values in your DATA statements get out of step. Remember that numbers represent valid string data and you will not get an error if you mistakenly assign a numeric value to a string variable.

File Error

You get this error when you try to use a file (with a CLOSE #, INPUT #, PRINT #, or RESTORE #) that has not yet been OPENed or you try to OPEN a file that is already open.

You also get this error when you OPEN a file and then try to do something that cannot be done to the file. Maybe you tried to INPUT # from a file that you OPENed as OUTPUT (write only) or APPEND (write only, starting at the end of the current file contents). Or, you tried to PRINT # to a file that you OPEN as INPUT (read only). If you have to both read and write to the same file without CLOSEing and re-OPENing the file between the reading and writing, OPEN the file as UPDATE (both read and write).

For-Next Error

You do not have the same number of FOR and NEXT statements. TI BASIC counts FOR and NEXT statements during its symbol generation.

Every FOR loop begins with a FOR statement and ends with a NEXT statement. You must have the same number of FORs as you do NEXTs.

Incorrect Statement

TI BASIC generates this error when it cannot understand what you want it to do. Check the format of the statement that gets the error and make sure that you used the correct number and type of operands, that you used correct punctuation (if necessary), and that you spelled everything correctly.

Maybe you forgot the equals sign (=) in an assignment statement. Or, you used invalid separators in a list. Or, there just is not any TI BASIC keyword (command, statement, or function) in the statement.

Check that your statement matches what TI BASIC expects to see. Make sure that the punctuation is correct. For example, are all your parentheses closed? Do you have a string without closing double quotes (")? Are line numbers (in those statements that use a list of line numbers) separated by commas (,)? Do you have a colon (:) after the file number in all your I/O statements? Check your statement format carefully!

Perhaps you used an invalid variable name in a DEF statement or an incorrect number of subscripts in an array reference in a DIM statement (you can have one to three subscripts per array). Or you used the wrong number or type of operands in a subprogram like JOYST, GCHAR, etc. Or you simply misspelled the name of one of TI BASIC's subprograms.

Input Error

This error occurs for INPUT and INPUT # statements and indicates that there is something wrong with the data that you entered from the keyboard or read from a file. If the error occurs for data entered from the keyboard, you get a *warning* and TI BASIC corrects the value or lets you re-enter the data.

Perhaps your program executed an INPUT statement and you tried to enter more characters than can fit into TI BASIC's I/O buffer (about 112 characters). TI BASIC warns you and ignores all characters after the 112th.

Or maybe the number of data items you entered does not match the number of variables in the INPUT statement. TI BASIC warns you that you have made a mistake and lets you re-enter the data items. If this error occurs with an INPUT # statement, TI BASIC calls it an error and stops your program.

Then again, maybe you entered string data for a numeric variable. TI BASIC warns you and lets you re-enter the data from the keyboard. If this happens with an INPUT # statement, TI BASIC stops your program after printing its error message.

If you are entering numeric data from the keyboard and the number generates an overflow when TI BASIC assigns it to the associated variable, TI BASIC prints a warning message and lets you re-enter the data. If this happens when TI BASIC is reading from a file (INPUT #), you get an error message and your program stops.

I/O Error *XY*

I/O errors always have something to do with I/O statements. They occur for many reasons. You can tell which I/O statement caused the error by looking at the *X* value, as shown in Table J-1. Table J-2 lists the values for the *Y* indicator and the most common reasons for errors.

In general, you get an I/O error when you ask TI BASIC to do something that cannot be done with the type of file and device you are using in your I/O statement. Such as delete a file from a printer. Or read from a disk when there is no diskette in the drive. Or write to a tape when the cassette recorder is not connected.

When you get an I/O error and you cannot find any problems with the

Table J-1. I/O Error Statement Indicator

X	Type of I/O Statement
0	OPEN
1	CLOSE
2	INPUT
3	PRINT
4	RESTORE
5	OLD
6	SAVE
7	DELETE

Table J-2. I/O Error Types

Y	Error Type
0	Device name or filename not found. Look for misspelled names. Or you tried to use a device for something that it can't do (like SAVE to an RS-232 printer).
1	You tried to PRINT to a write protected file. Remove the write-protect tape from the diskette or the software file protect through the disk manager.
2	Either you used invalid operands in your OPEN statement or you used operand values which do not match the attributes of the file. (Maybe you used RELATIVE for a cassette file, CS1 or CS2.) Check your software switches if you're using the RS232 interface.
3	You tried an illegal operation for the file. Maybe you tried to PRINT # a file OPENed as INPUT or RESTORE # a file OPENed as OUTPUT.
4	You ran out of space on the device containing the file. This usually happens when you run out of space on a disk. Or your INTERNAL type data can't fit into the I/O buffer. This error also occurs when you try to open too many files.
5	You tried to INPUT # from a file and there's no more data in the file. You are past the end of the file.
6	There's a device error. This can happen when a device is not properly connected or becomes damaged. If you have a disk, you must have the disk turned on before you turn on your computer; also, that you're using an initialized diskette. This also happens if your cassette recorder becomes disconnected after you begin reading from it or writing to it. This can also occur when you press FCTN CLEAR (FCTN 4) when your computer is reading from or writing to a device.
7	There's a file error. The file may not exist or the filename may be misspelled. Or, the file type is wrong. You may be trying to INPUT # from a TI BASIC program file. TI BASIC can INPUT # only data files, not program files.

statement that caused the error, carefully check related I/O statements. Often you get these errors when you OPEN your file as one thing (perhaps INPUT) and then try to use it as something else (maybe PRINT # to an INPUT file).

Sometimes it is not your program that causes the error but it's your device. You get I/O errors if your device becomes inoperative or disconnected while your program is running—most often disconnected.

Perhaps you are using a cassette recorder and the connections are not tight. It is sometimes easy to pull the cassette cable out of the back of your console. Or one of the leads is not in right.

If you are using disk drives, you must turn on your drives before you turn on your computer. The TI-99/4A checks to see what is connected to it (such as your drives or the peripheral expansion system) when you turn it on. You will get I/O errors if your computer does not really "know" that it has disk drives available.

Speaking of disks, you know that you must "initialize" a new diskette before you can write on it. You get an I/O error when you run out of space on a disk.

You know that you cannot read from a printer or delete a file from a printer.

Line Too Long

TI BASIC reads your statements into an I/O buffer that can hold up to 112 characters. You get this error when you try to enter a line that contains more than 112 characters. Remember that TI BASIC counts spaces as characters.

Memory Full

You can get this error when your program is running. TI BASIC uses memory when it allocates an array, variable, or function.

By far the most common reason for running out of memory is allocating large arrays. If your program has a lot of arrays, try to make one or more of the arrays smaller. Or, if you do not use array element zero for any array, use an OPTION BASE 1 statement.

Or, if you did not DIMension your arrays because you used fewer than 10 elements, try using a DIM statement with an exact value for each array and see if you can recover enough space.

This error also occurs when you are entering or editing a program and there is not enough room to put another statement in your computer's memory. Perhaps you have a very large program and you tried to add just one more line. Or, you are editing a line in a very large program and the new line pushes the program past what can fit into memory.

If you get this error when you are entering or editing a program, you have to shorten your program in some way. You can make some messages shorter. Or, maybe you can make your REMarks a bit shorter, if more cryptic. You might be able to shorten some variable names or use fewer variable names.

If you get this error when TI BASIC is allocating a variable or function, you have few choices. You must make your program shorter in some way. Or, use fewer variables or shorter variable or function names.

But there are even more ways to run out of memory. When TI BASIC executes a GOSUB it uses memory to store the line number where it is supposed to RETURN to. You cannot GOSUB to the same line, as in this:

```
200   GOSUB 200 (Memory gets full)
```

Another time this error occurs with subprograms is when you use a lot of GOSUBs and you do not RETURN. If this happens, restructure your program so that you RETURN before you execute so many GOSUBs. This is called a "pending return" situation. Get rid of some of those pendings.

If you use CHAR to redefine standard characters (ASCII code 32 to 127), you do not use more memory space. But when you define characters in the ASCII code range 128 to 159 you use 8 bytes of memory for every character you define. See if you can use some of the standard characters (32 to 127) instead.

Memory gets used for user defined functions (the DEF statement). Each function needs memory to store its definition. If you get this error on a DEF statement, see if you have a DEF that references itself. Or, if you do not, try to make fewer functions or use shorter variable names in the functions, if possible.

Name Conflict

You tried to use the same name for two different purposes. Perhaps you used the identical name for two different arrays, or a variable and an array, or a variable and a function.

This error also occurs when you DIMension an array with one number of dimensions (for example, ARRAY(12,5,15), or three dimensions) and you reference the array with a different number of dimensions (for example, ARRAY(5,1), or two dimensions).

Number Too Big

Your program generated a number that is larger than 1E128. This error can occur when you are READing values from a DATA statement or when you are INPUTing data from the keyboard or INPUT #ing data from a file. If the error occurs when you are INPUTing data from the keyboard, TI BASIC warns you and lets you re-enter the value. At all other times, TI BASIC substitutes its largest number (9.9999999999999E127) for the value and continues.

String-Number Mismatch

You used a string where TI BASIC expects to see a number or you used a number where TI BASIC expects to see a string.

This error occurs when you use any of the built-in functions or subprograms and use the wrong type of operands (strings instead of numbers or numbers instead of strings). It also occurs when you use a user-defined function (defined in a DEF statement) and your function type and expression type disagree or you pass the wrong type of argument to the function.

You also get this error when you try to assign a string to a numeric variable or a number to a string variable.

Index

TO THE READER

Sams Computer books cover Fundamentals — Programming — Interfacing — Technology written to meet the needs of computer engineers, professionals, scientists, technicians, students, educators, business owners, personal computerists and home hobbyists.

Our Tradition is to meet your needs and in so doing we invite you to tell us what your needs and interests are by completing the following:

1. I need books on the following topics:

____
____
____
____
____
____

2. I have the following Sams titles:

____
____
____
____
____
____

3. My occupation is:

______ Scientist, Engineer
______ Personal computerist
______ Technician, Serviceman
______ Educator
______ Student

______ D P Professional
______ Business owner
______ Computer store owner
______ Home hobbyist
Other ____________________
____

Name (print) ______________________________
Address ______________________________
City ________________ State ________ Zip ________

Mail to: **Howard W. Sams & Co., Inc.**
Marketing Dept. #CBS1/80
4300 W. 62nd St., P.O. Box 7092
Indianapolis, Indiana 46206

22246

SAMS

TI-99/4A BASIC Reference Manual

If you want to make the programming of your TI-99/4A in BASIC a more pleasant, rewarding, and productive experience . . . then this book is for you! With over 130 sample programs, it is for TI-99/4A users who want to get the most out of the existing BASIC software.

This reference manual is designed to:

- Give you information . . . not to teach you programming.
- Help both beginning and experienced programmers improve their BASIC skills.
- Make the TI BASIC commands and statements easily accessible.
- Develop your ability to recognize and correct programming errors.
- Provide you with detailed descriptions of all commands, statements, and functions in TI BASIC and Extended BASIC.
- Impart information that is necessary to write anything beyond trivial programs.

You now have compiled into one reference book information about TI BASIC and Extended BASIC that is easy to use and easy to understand . . . a must for present, as well as future, TI-99/4A owners.

Howard W. Sams & Co., Inc.
4300 West 62nd Street, Indianapolis, Indiana 46268 U.S.A.

$17.95/22246

ISBN: 0-672-22246-9