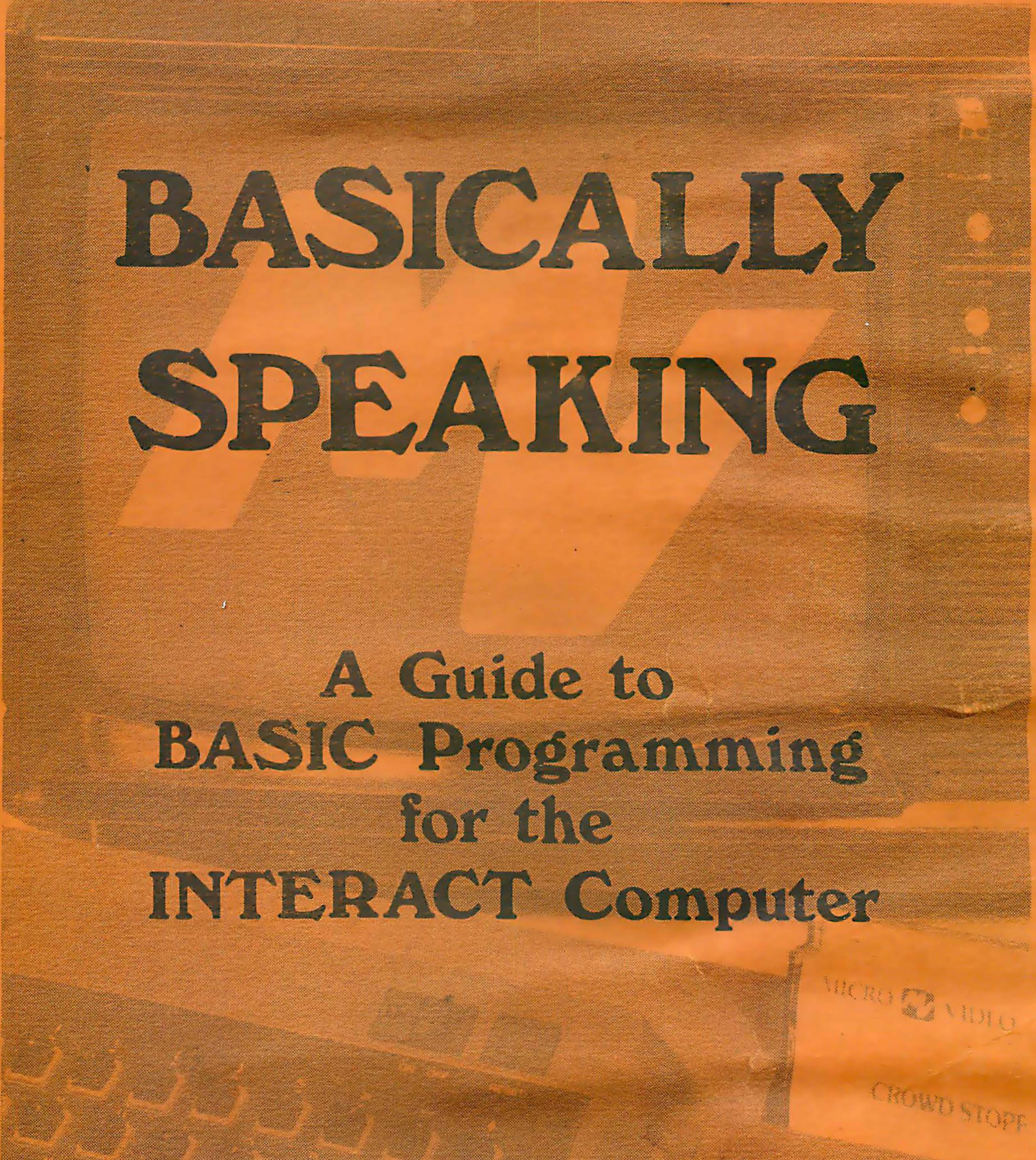


BASICALLY SPEAKING

A Guide to BASIC Programming for the INTERACT Computer

MICRO VIDEO TM
P.O. Box 7357
204 E. Washington St.
Ann Arbor, MI 48107

BASICALLY SPEAKING

A Guide to
BASIC Programming
for the
INTERACT Computer

MICRO VIDEO™
P.O. Box 7357
204 E. Washington St.
Ann Arbor, MI 48107

BASICALLY SPEAKING

is a publication of

Micro Video Corporation

First Printing -- December 1980

Second Printing -- April 1981 (Revisions)

BASICALLY SPEAKING

A Guide to BASIC Programming for the Interact Computer

Table of Contents

AN OVERVIEW OF YOUR NEW MANUAL

Congratulations on your purchase of BASICALLY SPEAKING, the all-new Interact BASIC programming manual! Whether you're just getting started with programming on your Interact or you're an experienced programmer who needs more reference material, we're sure you'll find this book a dramatic improvement over the old Level II BASIC manual.

Because we wanted BASICALLY SPEAKING to address the educational needs of programmers at all levels, we've divided the manual into 12 chapters which explain the fundamentals of BASIC programming (BASIC Basics, Chapter 1) to the technical details of each BASIC statement (BASIC--A TO Z, Chapter 10).

Chapter 1, BASIC Basics, is intended for people with little or no understanding of BASIC programming concepts. It explains the components of the BASIC language and takes a brief look at program logic with a flow chart and annotated listing of a simple game program. Chapter 2, HOW TO SPEAK BASIC, is an introductory walk through BASIC programming in which we experiment with a number of programming concepts under direct mode control and indirect mode program operation. This chapter is loaded with examples you can enter as you read to let you see the structural concepts discussed (keyboard input, looping, conditional relationships, array handling, random number generation, to name a few) in action on your Interact.

Those of you who are intermediate to advanced BASIC programmers may find these first two chapters excruciatingly elementary. Please bear with us--not all Interact owners have attained your level of proficiency with BASIC! Just skip ahead to the more complex material: GRAPHICALLY SPEAKING (Chapter 3), which discusses a variety of methods for producing entertaining visual effects in your program screen display; STRUNG OUT (Chapter 4), which explains string handling; INTERACT GAMESMANSHIP (Chapter 5), which shows you how to use the entertainment controllers to build game programs; READING DATA (Chapter 6), which deals with using DATA and READ statements as an alternate method of data entry; SUBROUTINES (Chapter 7), which discusses the use of subroutines within programs; INTERFACING WITH THE BASIC ENVIRONMENT (Chapter 8), which contains operating and debugging suggestions that can make your programming life easier; MACHINE LANGUAGE INTEGRATION (Chapter 11), which discusses the mechanics of combining BASIC and machine language programming.

In Chapter 9, RS232 BASIC, we discuss how to access a lineprinter under BASIC control and the differences between RS232 BASIC and the other two versions of BASIC (Microsoft 8K and Level II).

Chapter 10, BASIC -- A TO Z, is the Reference Section. An expansion of the BASIC Reference card, this chapter presents all the BASIC statements and functions in alphabetical order, with syntax structure, functional definition, and at least one example of the operator in use. We don't intend for you to read this chapter straight through, but rather use it as you would a dictionary when you need more information about a particular statement or function.

So, load your BASIC interpreter, and let's get started experiencing the fun and satisfaction of making the Interact do what you want it to do!

BASIC Basics

First of all, what is a computer?

A computer is, simply, a machine that processes information. Computers come in several sizes. The largest is a mainframe, or macro. Mainframe computers are used by large businesses for their data processing needs. A mainframe computer has an enormous capacity for processing. In fact, a number of companies have made a business of selling time (called timesharing) on their mainframe computers. The next size of computer is a mini-computer. Mini-computers are commonly used by small businesses for their data processing needs. Mini-computers don't have nearly the processing capacity that mainframes do, but they have greater capacity than the smallest in the computer line--the microcomputer or personal computer.

We define a personal computer as one you can buy with a credit card. Many different microcomputers are available now, but your Interact fits this definition better than some of the others, such as the Apple, which, in reality, can be quite expensive. If you compare your Interact's price and capabilities with other microcomputers, you'll see that, in buying an Interact, you really get a lot more "bang for the buck"!

What is BASIC?

Let's imagine for a moment that you speak French and I speak German and that we are trying to communicate with each other. We won't have much success without a translator--someone to take what I say in German and translate it into French so you can understand it, then take your response and translate it into German so I can understand it. BASIC (an acronym for <u>B</u>eginner's <u>A</u>ll-purpose <u>S</u>ymbolic <u>I</u>nstruction <u>C</u>ode) performs that translation function for you and your Interact.

Computers actually "talk" in machine language, which is a language with only two "letters". These letters are 0 and 1 (or "on" and "off" switches) that are combined into "words" of 8 "letters" (one byte). The "words" are combined to form "sentences" or instructions. Machine code is complicated and difficult to learn, so most people prefer to communicate with their computers using a "higher lever" language that performs the translation service. That's just what BASIC does--it takes what you say in the BASIC language and translates it into machine language. Then it takes the computers response and translates it back into a form that you can understand. In this way you tell, or program, your computer to do what you want it to do.

What, then, is a program?

A <u>program</u> is simply a series of logical "sentences" in the BASIC language that cause your computer to perform certain tasks. Your Interact can be programmed in two ways--<u>direct mode</u> and <u>indirect mode</u>. In direct mode programming, your instructions to the computer are called <u>commands</u>. Your computer performs the task defined in the sentence, or command, as soon as you press the "CR" key to enter the instruction.

Let's clear up one popular misconception. Computers are not smart! In fact, they are extremely dumb. A computer can do only what it is told to do, and it will attempt to follow instructions given it exactly. So, in programming, don't blame your computer if the result is not what you intended. Keep in mind that the computer did just what you told it to do. If that wasn't right, you need to give it different instructions.

When you type a direct mode command or indirect mode program line, your computer has no way of knowing that you are finished typing the line unless you tell it so. (It can't read your mind!) Therefore, you must conclude entry of any command or program line by pressing the "CR" (carriage return) key.

In direct mode, your computer can execute instructions only one line at a time. A single line can, however, contain several commands. You can chain multiple commands together on one line by separating the individual commands with colons. For example,

```
NEW
CLS:COLOR 7,4,0,1:PRINT "HELLO"
```

In indirect mode, each "sentence" is called a statement and is assigned a line number which determines the order in which the defined task(s) will be performed. Statements in indirect, line-numbered programs are executed in the numeric order in which they appear. The statement(s) on the lowest numbered line is always executed first, followed by statements on the next higher numbered line, and so on. Statements on some lines may be skipped altogether if a statement tells the computer to skip over them, however. Indirect mode programs can contain as many lines and statements as memory permits, and the tasks defined in the program statements are not performed until you type the RUN command. If we put the above example into indirect mode, it would look like this:

```
NEW
10 CLS
20 COLOR 7,4,0,1
30 PRINT "HELLO"
```

Whether in direct or indirect mode, program "sentences" can be broken down into the same set of constituent parts. The "words that make up the statements or commands fall into the following categories: **line numbers, keywords, strings, constants, variables, operators, functions, arguments, expressions, and data.** Let's take a closer look at each of these parts.

A **line number** is always the first "word" in an indirect mode program line. The line numbers control the order in which the program lines are executed when the RUN command is given. A line number can be any number between 1 and 65529. Since statements are executed in the order in which they appear on the numbered lines, make sure you number your lines so that the execution of statements follows the logic necessary to accomplish a given task. Line numbers **never** begin direct mode commands.

A **keyword** is one of a specific set of words that tell the BASIC interpreter to perform a task such as PRINTing a string, OUTPUTting a value at a certain location, PLOTting a line, GOing TO a different line, DIMensioning an array, etc. All the keywords recognized by BASIC can be found alphabetically in the Reference Section. Some keywords are primarily used in direct mode, such as LIST and NEW. Some keywords can only be used in indirect mode, such as GOSUB and END. Most of the keywords can be executed in either mode.

A string is a set of alphanumeric characters that is stored in either a string variable (such as A$) or as a string constant enclosed in quotes ("HELLO", for example). Strings are generally used to define something that is to be output on the screen, although they may be used for other purposes, such as testing input from the keyboard.

Constants are numeric or string values that are unchanging. An example of a numeric constant is pi (3.14159), or the number 7. Constants are generally assigned to variables for use in programs. A string constant is defined by enclosing the set of characters in quotes (e.g., "MARY").

Variables are user-defined names that act as storage slots for string or numeric constants or the results of function calls or arithmetic expressions. Variables provide a convenient means of reusing a constant or the result of an operation without having to reprogram that expression every time you want to use it. It's not only easier to reference variables, it also saves program space. For example, if you wanted to compute the area of a series of circles with radii of 2 through 7, you could use a series of statements in indirect mode such as

```
10 PRINT 3.14159 * (2*2)
20 PRINT 3.14159 * (3*3)
30 PRINT 3.14159 * (4*4)
40 PRINT 3.14159 * (5*5)
50 PRINT 3.14159 * (6*6)
60 PRINT 3.14159 * (7*7)
```

You can see, however, that this is a rather cumbersome method of attacking the problem, particularly retyping the value of pi over and over. A much simpler and more space-efficient method of accomplishing the same task is to use variables for the values of pi and the radii.

```
10 P = 3.14159
20 FOR R = 2 TO 7
30 PRINT P * (R*R)
40 NEXT R
```

This small program defines the slot called P as equal to the value of pi. It defines the radius of each circle as R and also tells the computer to perform the operation of finding the area of the circle six times.

There are two types of variables--numeric and string. Numeric variables are used only to store numeric values. Variable names are chosen by the user according to variable naming conventions or rules. These rules are simple: the variable name can be as long as you like, but it must begin with a letter, and the first two characters of the name must be unique. Some examples of numeric variable names are A, X1, IN, A4A, A3ABC. String variables are used to store string information (which can contain numeric characters). String variable names follow the same naming conventions as numeric variables, except that they must end with a dollar sign ($). String variable names are assigned to string constants that are to be reused within a program or to establish storage slots for data to be entered during program execution. For example, the following simple program requests entry of a string, then prints a message that includes that string.

```
10 INPUT "WHAT'S YOUR NAME"; NM$
20 PRINT "HELLO, "; NM$
30 GOTO 10
```

Never, never use any of the BASIC keywords, operators, or functions as variable names!

Operators are like keywords in that they are recognized by BASIC as instructions to perform a specific arithmetic task or evaluation. They are used in the definition of expressions. Valid operators are:

+	addition*
-	subtraction
*	multiplication
/	division (appears on your keyboard as ÷)
^	exponentiation
( )	precedence**
>	greater than
<	less than
=	equal to***
><	not equal to
>=	greater than or equal to
<=	less than or equal to

* + can also be used to concatenate strings, e.g., PRINT A$ + B$.

** precedence is the order in which arithmetic expressions are evaluated. For example, does

A = B + C/6

mean add ~~A~~ C and B, then divide the total by 6, or divide C by 6 and add the result to B? BASIC follows standard mathematical conventions and does multiplication and division operations before addition and subtraction. Therefore, BASIC interprets the above statement as A = B + (C/6). If you want to add ~~A~~ C and B, then divide the result by 6, you would enter the statement as A = (B + C)/6.

*** = acts as the assignment operator as well as a relational operator.

Functions are predefined rules for complex computations of values based on other values. Functions reduce redundancy in programming by eliminating the need to repeat complex operations again and again. Functions fall into two general classifications--those that perform arithmetic operations and those that operate on strings (called string handling functions). There are a number of built-in arithmetic and string handling functions in BASIC. Examples of arithmetic functions are TAN, SIN, COS, SQR, EXP, LOG. Examples of string handling functions are INSTR$, MID$, LEFT$, CHR$, STR$. See the Reference Section for specific information on the various built-in functions. BASIC also allows user-defined functions, in which you name a function and specify what it is to do each time it is called. Again, this is convenient for eliminating needless repetitions within the program (see DEF for more information on user-defined functions).

Arguments are values in parentheses after functions that tell the computer on what value the function is to be performed and a result returned. An argument can be a constant, a variable, an expression, or even another function call. For example,

```
PRINT SQR(A)
```

tells your computer to compute and display the square root of the value of the argument, A.

An expression is a combination of variables, constants, function calls and operators that, when evaluated, has a single value. One or more expressions can appear within a program line. Usually, arithmetic expressions define mathematical processing to be done. String expressions define string handling. The underscored portions of the following statements are valid expressions in BASIC:

```
A = 10

A$ = "LEFT PLAYER NAME"

A$ = CHR$(B$)

P = SQR(A) * 100

N$ = MID$(A$,2,3)

IF C = 10 AND B = 100 THEN F = 256

G = (4 * TAN(X) + 100)/((SQR(V) * (8 + V)) * (ABS(X) + 2)
```

Now, none of the above expressions is particularly meaningful outside the context of a program with other statements, but they do illustrate that expressions can take many forms, from the very simple to the extremely complex. It is through the evaluation of expressions that we define values stored in variables or other operations for reuse within a single program.

Data are values that are entered into the program and then used in subsequent processing. Data take a variety of forms, are used in a variety of ways, and can be entered in several fashions. Data values can be entered from the keyboard during program execution via INPUT statements or the INSTR$ function. Data values can be input from the joystick lever, potentiometer (pot knob), and fire button of either entertainment controller during program execution and used to determine what happens next in a program. The RS232 peripheral interface provides another method of data entry. The tape deck is another data entry "vehicle". Data required to use the BASIC interpreter is entered into your computer every time you load BASIC. The CLOAD command can be used in direct mode or from within a program to enter data from tape. And, data can be embedded in program lines with DATA statements and used as needed by referencing the data values with READ statements.

Just as there are several methods and devices for data entry, there are also several means to output data. The CSAVE command can be used to output data or programs to tape. The OUTPUT, PLOT, and PRINT commands cause data to be output to the most familiar output device, the TV screen. The COLOR statement changes the data that control the colors output on the screen during program and direct mode execution. The LIST command displays the data in the lines of your program on the TV screen. The SOUND and TONE commands convert data values into sounds that are output through your TV's speaker. If your computer is equipped with the Micro Video RS232 peripheral interface and RS232 BASIC, you can use the LLIST and LPRINT commands to output data through the interface port to a lineprinter. With the RS232 interface and the Micro Video COMMUNICATOR program, you can output data through a modem (also called an acoustic coupler) to access a timesharing system.

These "words" in the BASIC language are the building blocks through which we communicate with our computers and control them. Once you learn the "words" and the correct way of putting them together (called **syntax**), all that remains to becoming a "programmer extraordinaire" is mastery of the logic behind programming.

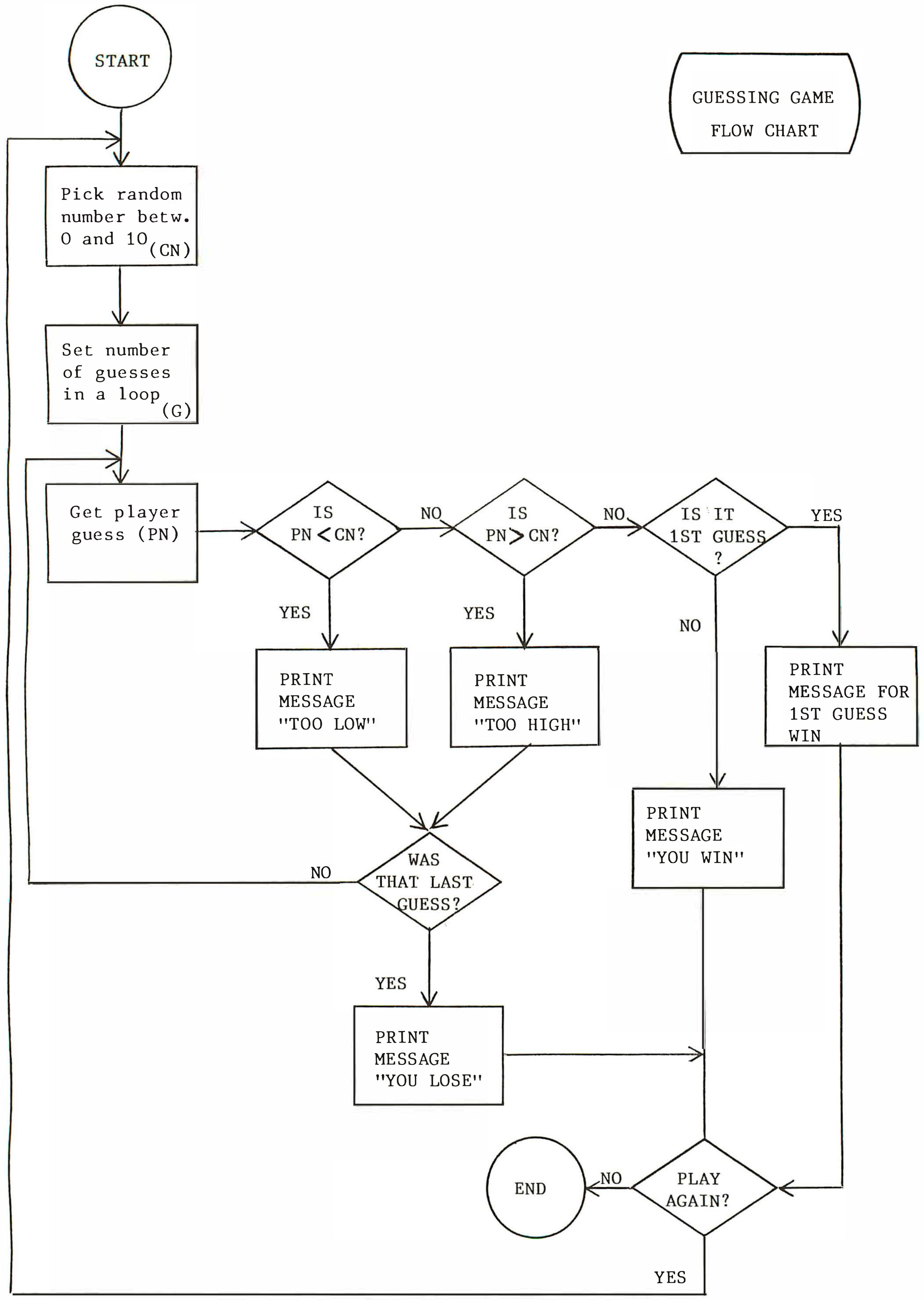
Program logic is the order in which you tell your computer to execute the statements in your programs. As previously stated, the computer will process each line in its sequential order, unless you tell it to execute some other line or set of lines with a GOTO or GOSUB instructions within a program line. You tell your computer to **branch** to other parts of the program based on the evaluation of an IF or ON instruction. You can tell your computer to perform an operation or set of operations more than once by putting those instructions inside a FOR...NEXT **loop**. You can **chain** statements together on a single line and execute them based on the result of a conditional test with the IF...THEN logic instruction. Branching, chaining, and looping are all integral parts of program logic--they provide the means to "move around" within your program.

The logic operations can get rather complicated, so in your initial progamming attempts, you may find it helpful to draw a flow chart of your program logic before you try to enter and run the program. A **flow chart** is a graphic representation of the logic in your program--you might consider it a map through the program roads. By following the lines and arrows, you can trace what the outcome of any flow through the program will be.

Let's take a relatively simple example that illustrates several of these logic operations. Let's say you want to program a guessing game to play with your computer. In this game, the computer will pick a random number between 0 and 10, then ask you to guess it. The computer will let you have three chances to guess the number and will tell you if your guess is too high, too low, or correct. At the end of each game, it will ask if you want to play again. You might draw a flow chart for this program as shown on page 1-8.

On the flow chart on the next page, three symbols are used. Rectangles indicate operations BASIC is to perform. The diamond shaped symbols indicate a decision that is to be made based on the result of a conditional test; branching to different parts of the program occurs depending on whether the answer to the question asked is "yes" or "no". A program loop controls how many guesses the player has to get the correct number. The lines between the symbols chart the flow from operation to operation in the direction indicated by the arrows. Circles indicate starting and ending points of the program.

Follow the paths through this flow chart to see if you understand the program logic. Then, see page 1-9 to see the actual program that plays this game, along with explanation of the various program statements.


START
GUESSING GAME FLOW CHART
Pick random number betw. 0 and 10 (CN)
Set number of guesses in a loop (G)
Get player guess (PN)
IS PN < CN?
NO
IS PN > CN?
NO
IS IT 1ST GUESS ?
YES
YES
YES
NO
PRINT MESSAGE "TOO LOW"
PRINT MESSAGE "TOO HIGH"
PRINT MESSAGE FOR 1ST GUESS WIN
PRINT MESSAGE "YOU WIN"
NO
WAS THAT LAST GUESS?
YES
PRINT MESSAGE "YOU LOSE"
END
NO
PLAY AGAIN?
YES

```
10 CLS
20 CN=INT(RND(1)*11)
30 PRINT"I HAVE PICKED"
40 PRINT "A NUMBER"
50 PRINT"BETWEEN 0-10"
60 PRINT:PRINT"CAN YOU":PRINT"GUESS IT?"
70 FOR G = 1 TO 3
80 PRINT:PRINT"WHAT'S YOUR"
90 INPUT"GUESS";PN
100 PRINT
110 IF PN < CN THEN PRINT "TOO LOW!":PRINT:NEXT
120 IF PN > CN THEN PRINT "TOO HIGH!":PRINT:NEXT
130 IF PN = CN AND G = 1 THEN PRINT"THAT'S RIGHT!":PRINT"DID YOU CHEAT?":GOTO170
140 IF PN=CN AND G>1 THEN PRINT"YOU GUESSED IT!":PRINT"YOU WIN!":GOTO170
150 PRINT "SORRY, YOU LOSE!"
160 PRINT"MY NUMBER":PRINT"WAS";CN
170 FORQ=1TO300:NEXT
180 CLS
190 PRINT"PLAY AGAIN?"
200 A$=INSTR$(1)
210 IF A$="Y" THEN CLS:GOTO 20
220 IF A$="N" THEN END
230 GOTO 200
```

Clears the screen. A nice way to begin all programs.

Chooses a random number between 0 and 10 and calls it the variable "CN".

Outputs message to begin game.

Sets the number of guesses allowed within a loop. The loop will be executed 3 times--so the player has 3 guesses.

Asks player to enter guess and calls it the variable "PN".

Tests to see if PN is less than CN. If it is, prints message, then returns to line 80 for next guess. If it is not less than CN, executes next line.

Tests to see if PN is greater than CN. If it is, prints message and returns for next guess. If it is not greater than CN, assumes number is correct and executes line 130.

Number is correct. Check to see if it was first guess. If yes, print message; if no print different message.

If the number was not guessed correctly after three times through the loop, prints "lose" message and displays the correct number.

Pause loop--holds final message on screen a few seconds.

Asks if you want to play again and waits for answer.

If answer is yes, goes to line 20 and starts program again.

If answer is no, stops program execution.

Returns to keyboard input function (INSTR$) if any key but "Y" or "N" is pressed.

If you would like to try modifying this program to make it work differently, here are some suggestions:

1) Change the number of guesses allowed by changing the 3 in line 70 to a higher number. The number you put in place of 3 will be the number of guesses allowed.

2) Make the game harder by choosing a random number between 0 and 100. To do this, change the 11 in line 20 to 101. You will also want to change line 50 to read "BETWEEN 0 - 100".

3) Add more IF statements to vary the winning message output based on how many guesses it takes the player to get the correct answer. Model additional statements after line 180.

In review, a computer program is a series of logical instructions to the computer. These statements consist of keywords, variables, constants, expressions, function calls, arguments, and other "words" the computer understands, expressed using the correct syntax. The parts of statements and proper syntax is relatively easy to learn--putting the statements into logical order provides the greatest challenge in programming. There's enormous satisfaction in seeing your program do just what you intended it to do when you type RUN. Don't be discouraged if your first efforts end in syntax or other types of errors, however. Even the most advanced programmer rarely writes a program that will run error-free the first time it's executed.

Good luck, and happy programming!

The Three Interact BASIC Languages

There are three different BASIC languages for the Interact. You can load and use one or more of them, depending on the configuration of your computer.

EDU-BASIC	EDU-BASIC is an integer BASIC language. This means that you cannot do any arithmetic operations that require floating-point capability. It has far more limited capabilities than the other two BASICs, but it also consumes less RAM. EDU-BASIC is the only programming language you can use if you have a computer with only 8K of RAM. Programming with EDU-BASIC is <u>not</u> addressed in this manual.
Microsoft 8K FAST Graphics BASIC	Microsoft 8K Fast Graphics BASIC is an upgraded version of Level II BASIC. It replaces Level II BASIC completely. Both Microsoft 8K and Level II BASIC are floating-point BASIC languages--that is, they can perform operations on any real number. To use either of these versions of BASIC, your computer must have at least 16K of RAM. Microsoft 8K and Level II BASIC both consume the same amount of RAM. They take up more RAM than EDU-BASIC, but provide far greater capabilities. The bulk of this manual describes programming with Microsoft 8K (or Level II) BASIC.
RS232 BASIC	RS232 BASIC is an expanded version of Level II BASIC that provides the ability to access a lineprinter to produce program listings or formatted reports from your BASIC programs. You must have at least 16K of RAM and a Micro Video RS232 peripheral interface for RS232 BASIC to load and run properly. RS232 BASIC has two commands that Level II BASIC does not have--LLIST and LPRINT. Because these commands were added, the format of RS232 BASIC is different from that of Level II. You can load and run programs written in Level II BASIC with RS232 BASIC, provided you have the RS232 EZEDIT program editor to translate your programs into the correct format. If you know how to program with Microsoft 8K or Level II BASIC, you will find it easy to learn the two additional commands in RS232 BASIC. Instructions for using RS232 BASIC and discussion of the differences from Level II are included in this manual.

BASIC Dialects

Many people want to know why they can't take TRS-80, APPLE, or other microcomputer programs written in BASIC and load and run them on the Interact as is. The reason you can't do this is that every microcomputer has its own "dialect" of the BASIC language. The syntax of the many BASIC statements can and does vary from microcomputer to microcomputer. Therefore, a statement that a TRS-80 computer understands perfectly may make absolutely no sense to your Interact.

The different BASIC dialects have different commands available. Your Interact, for example, has the SOUND, TONE, and COLOR commands in its BASIC language, and the TRS-80 does not. If you tried to load an Interact BASIC program into a TRS-80 computer, the TRS-80 wouldn't know what to do with those commands.

In addition to the differences in syntax and available commands, BASIC resides in different areas of memory in different computers. All the words that BASIC understands are stored as numbers in a table. To you it looks like PRINT, but to BASIC it's a number, such as 128. In TRS-80 Level II BASIC, PRINT might be stored as number 243, which adds another level of confusion if you try to load and run a TRS-80 BASIC program in the Interact. You computer just flat out doesn't know what to do with the information it's been given!

One final problem is that there are also differences in tape formatting from computer to computer. Each computer can have its own method of reading and writing tapes, and the methods are not always compatible with other computers' methods. A program must be read into memory using the same format in which it was written to tape.

You can, however, convert BASIC programs from other computers to run in Interact BASIC. It's a lot of work, because you will have to type in every line all over again, making adjustments as necessary to conform with Interact syntax conventions and available commands. We suggest that you do not attempt program conversions of this nature until you are completely familiar with the workings of Interact BASIC.

Documentation Conventions

We have used the following general conventions in this manual:

USAGE EXAMPLES

In illustrations of direct mode operation, information you enter is displayed in reverse type. The computer's responses are indicated by **bold face** type. For example,

```
PRINT 3*147
 441
OK
```

In program examples, the program lines are displayed in **bold face** type. If you want to enter the examples as you go along, type each line as it is presented, including the line numbers as shown. For example,

```
10 CLS
20 COLOR 3,1,7,0
30 PRINT 3 * 147
```

Some sample program listings were produced using our RS232-equipped Interact, RS232 BASIC and a COMPRINT 912-S lineprinter. Examples using actual program listings look like this:

```
10 CLS
20 COLOR 3,1,7,0
30 PRINT 3 * 147
```

GENERAL FORMS OF COMMANDS/STATEMENTS

In the Reference Section, the general form of each command or statement is presented. In each general form:

Words that appear in UPPER case are BASIC keywords. You must enter the keywords exactly as shown when using the command or statement.

Words that appear in lower case indicate information that you are supposed to supply when using the command or statement. The type of information you should supply varies from statement to statement, but is generally a variable, argument, expression, etc.

Words that appear in square brackets ([]), whether upper or lower case, indicate that that part of the statement is optional. You can include them or not, as appropriate to your usage of the statement.

HOW TO SPEAK BASIC

This section of the manual teaches you how to converse with your computer in the BASIC language. It's a "walk-through" of BASIC programming that is liberally sprinkled with examples to illustrate the points covered. We suggest you keep your computer handy with BASIC loaded as you read through it so that you can type in the examples as shown and get hands-on experience with your computer and the BASIC interpreter.

After you load the BASIC interpreter, the message "4698 BYTES FREE" and the BASIC "OK" prompt appear on the screen. The first messsage tells you that you have 4,698 bytes ("computer words") available to put your program in. The "OK" prompt is BASIC's way of telling you that it is ready to accept your commands. You will see the "OK" prompt each time BASIC finishes processing a command you type in or a program you run.

The **very first** command you should type when you see the "OK" prompt after loading BASIC for the first time in a processing session is the NEW command.

```
4698 BYTES FREE
OK
NEW
OK
```

You must type NEW because the RAM (Random Access Memory) used for program and data storage is not cleared when BASIC is loaded. If you have loaded BASIC right after turning your computer on or after running a machine language program, the program storage memory locations are filled with random values that are meaningless to BASIC (we call it "garbage"). If you forget to type NEW, an "?OM ERROR" or other type of error is likely to result. NEW sets an internal pointer of BASIC to indicate that no program is in memory.

Why doesn't BASIC do this automatically? Even though it means you must remember to type NEW, there is a major benefit to this organizational scheme. Because memory is not cleared when BASIC is loaded, you can enter a program and run it, then load the EZEDIT program editor to make program corrections or renumber the lines in the program, then load BASIC and run the program again--all without having to save and reload your BASIC program every time. If BASIC cleared that area of RAM, your program would be cleared out every time as well, and you'd have to keep saving and reloading it. Remember that you do not want to type NEW when you reload BASIC after using the program editor!

You should also type NEW any time you want to begin entering an entirely new program.

Once BASIC is loaded and you have typed NEW, you are ready to begin operation in direct mode or program entry in indirect mode.

DIRECT MODE OPERATION

In direct mode, your computer processes your commands as soon as you press the "CR" key to enter them. Your computer can perform mathematical computations in direct mode in much the same way as a calculator does. It can also print the contents of string variables or constants. Nothing is done with the information you type until you hit "CR", so you can correct any errors you make simply by pressing the Backspace key as many times as necessary and retyping the line.

One of the most commonly used commands in direct mode is the PRINT command. PRINT is used to scroll lines of information onto the screen. The information can take a variety of forms. For example, try typing the statement

```
PRINT "HI THERE"
```

and press the "CR" key. The words HI THERE will scroll out on the bottom of your screen. You can scroll anything you want in this way. If you don't put any leading spaces in the string, the first character of your string constant is output as the first character on the line. However, you can add leading spaces to control how far over on the line your string starts. PRINT also has a "wrap-around" feature. If your string has more than the 17 characters that will fit on one line, any extra characters are output at the beginning of the following line. To see how leading blanks affect the screen positioning of strings, type the following statements:

```
PRINT "     HI THERE"  (5 leading blanks)
PRINT "               HI THERE"  (15 leading blanks)
```

Because you use PRINT so often, BASIC will let you abbreviate the command to a question mark (?) in both direct and indirect mode operation. PRINT is the only BASIC command that can be abbreviated in this way. Type

```
? "MARY"
```

You use the PRINT command to perform arithmetic calculations in direct mode almost as you would a calculator. For example,

```
OK
? 11 * 579
 6369
OK
?54/3
 18
OK
```

Note that BASIC responds with the "OK" prompt after it finishes each direct mode command. In subsequent examples we'll leave out that prompt to save space, but if you type in the examples, you will see the prompt each time. Also note that you can put spaces between keywords and operators if you like for easier readability. BASIC ignores those spaces, so operation is the same whether or not you include them.

PRINT can also be used to display the contents of variables and the results of function calls and arithmetic expressions. To illustrate this, let's assign the values 9 and 3 to variables named A and B, respectively.

```
A = 9

B = 3
```

Now that we've defined the variables A and B, we can perform operations using them.

```
? A
 9
? B
 3
? A+B
 12
? A-B
 6
? A*B
 27
? A/B
 3
? SQR(A)
 3
? A+B/SQR(A)
 10
? (A+B)/SQR(A)
 4
? LOG(A)
 2.19722
```

And so on. Experiment with using the PRINT command in this way until you are comfortable working with it.

You can print more than one value on a single line with the PRINT command, using the semi-colon (;) as a separator. You can combine numeric and string constants with numeric and string variables, or even expressions. For example,

B=3, A=9

```
A$ = "BLUE"

? A$;B;"GREEN";A*B/SQR(A)

BLUE 3 GREEN 9

?A$;"GREEN";B;A$

BLUEGREEN 3 BLUE
```

$9 \times 3 / \sqrt{9}$

When you use the semi-colon separator, BASIC outputs a leading and trailing blank around numeric data, but no blanks with string data.

You can also separate items to be printed with commas (,). If you use commas as separators, BASIC puts each item into a field 14 characters long. It will "wrap" string data on the screen, but not numeric data. Try entering one of the above examples using the comma as a separator to see the different result. Because the Interact has only 17 characters per line, the output using commas is not particularly attractive, so the comma separator is not frequently used.

Finally, you can use the PRINT statement alone to produce blank lines in your screen display. Try typing

A$ = Blue   Display's Blue
A = 15   15
B = 5   5
OK

```
?A$:?:?A:?:?:?B
```

to see how the scrolled information looks with blank lines included.

Screen Control

The CLS statement tells BASIC to wipe off all information off the screen. CLS is frequently used as one of the first program statements so that the program starts off with a fresh, clean screen. This is not terribly important operationally, but it gives a better visual effect. Type

```
CLS
```

Note that the BASIC "OK" prompt reappears at the bottom of the screen after it is cleared.

Interact BASIC has 8 programmable colors, each of which has an assigned reference number. These colors and their associated numbers are:

black = 0	red = 1
green = 2	yellow = 3
blue = 4	magenta = 5
cyan = 6	white = 7

You can use only 4 of these available colors at any one time. If you are using Microsoft 8K BASIC, the starting background color of your screen is black (0). With Level II or RS232 BASIC, the starting screen color is blue (4). The COLOR statement controls which of these colors are in use at any one time. Type

0,1,2,3 color position

```
COLOR 7,2,0,1
```

to see a dramatic change in the screen.

You can see that the number placed in the first position of the color set defined in the COLOR statement control the background color of the screen. This first position is called "color 0", regardless what color value you assign to it.

A: Background
B: line number display
C: used by plot on output
D: statement lines, OK prompt, error messages

Color A, B, C, D

The color in the second color set position (called "color 1") is the color in which program line numbers are displayed when you LIST your BASIC program. Type

```
10 ? "HELLO"
```

then type

```
LIST
10 PRINT "HELLO"
```

You will see that the line number 10 is displayed in green, because you have placed the value 2 in the "color 1" position of the color set. Also note that BASIC translates the "?" abbreviation to the keyword PRINT when it lists the program.

The color in the third position in the color set (called "color 2") is not used in screen display unless you reference it from a PLOT or OUTPUT command (discussed later).

The color in the fourth position (called "color 3") is the color in which the line you type in direct mode, information on program lines, the BASIC "OK" prompt, and error messages are displayed. You can see from the previous example that the PRINT "HELLO" part of the line is displayed in red, along with the LIST command and the "OK" prompt.

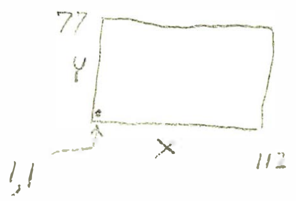
Experiment with different color sets by entering several COLOR commands to see the effects and find those which are most visually attractive. Some color combinations are more aesthetically appealing than others. For example, you'll find that red letters on a blue background (or vice versa) looks horrible and is also hard to read.

The commands we've worked with so far are screen control commands. CLS wipes the screen clear, PRINT scrolls information onto the screen from the bottom of the screen, and COLOR controls what colors appear there. There are others, and we'll look at them shortly. But first, let's talk about the graphic layout of the screen.

Screen Layout

Consider your screen to be a grid or matrix of picture cells (called "pixels"). This grid has 77 horizontal lines and 112 vertical lines. You can control what appears at any given pixel in this matrix by referencing its (x,y) coordinates. In technical terms, the screen is called a "dot-addressable matrix". The x-coordinate tells the computer how many pixels (vertical lines) from the left side of the screen something is to be displayed. The x-coordinate is always a number between 1 and 112. The y-coordinate tells the computer how many pixels (horizontal lines) from the bottom of the screen a value is to be displayed. The y-coordinate is always a number between 1 and 77, where 1 specifies the bottom-most point on the screen and 77 is the top. The Interact's graphics resolution is, therefore, 112x77. This is considered to be medium resolution graphics.

X - 1 and 112
Y - 1 and 77



Now, let's examine the other commands that control screen graphic output.

The WINDOW command specifies how many lines (pixels) up from the bottom of the screen are to be allocated for information scrolling with the PRINT command. The default, full-screen setting is WINDOW 77. WINDOW allows you to "pull a window shade" partway down your screen, so you can "see" only out of the bottom part. Type

```
WINDOW 24
```

Notice that the screen now appears to be cut into two parts. Scrolling "disappears" under the shade about three character lines from the bottom of the screen. Now type

```
CLS:PRINT "HELLO":PRINT "THERE"
```

and watch what happens. See that the scrolling takes place only on the bottom three lines of the screen, while the rest of the screen remains blank. This blank area is useful for graphics development--we'll use it to work with the other screen control (or graphics) commands.

Two final notes about the WINDOW command. If you use WINDOW in a program, remember to reset to WINDOW 77 when you stop execution with a Control-C for listing or other purposes. And, the smallest WINDOW you can set is WINDOW 11. If you try to set a smaller window, a "?FC ERROR" will occur.

Graphics Commands

Interact BASIC has two graphics commands--OUTPUT and PLOT.

The OUTPUT command is commonly used to display numeric or string information on the TV screen. The PRINT command also outputs such information on the screen, but is done in a scrolling fashion, and you can only control the color of display by changing the value of color 3 in the color set (with the COLOR command). The OUTPUT command lets you put that same information anywhere on the screen, in any one of the colors in the currently defined color set. Type

```
      0,1,2,3
COLOR 0,3,4,1
                        X  Y  Color
OUTPUT "HI THERE!",30,40,1
```

Your computer displays the words HI THERE! in yellow (color 1) in the approximate center of your screen. Note that the (x,y) coordinates tell the computer in what pixel to put the top left corner of the first character block in the string. (A character block is 5x5 pixels in a 6x6 pixel area.)

As with PRINT, you can use OUTPUT to display different kinds of information on the screen. You can output numeric or string constants, the results of function calls or expressions, etc. For example, type

```
A = 30
```

```
B = 20

OUTPUT A * A/B,50,50,2
```

You'll see that the value 45 is displayed in the upper middle of the screen.

You can use the OUTPUT command with the CHR$(1) function to display a 5x5 pixel block on the screen. Type

```
OUTPUT CHR$(1),10,70,3
```

and watch a red block appear in the upper left corner of the screen. You can put such an OUTPUT statement in a loop to draw wide lines on the screen with Level II BASIC. In fact, in Level II BASIC, OUTPUT CHR$(1) is the only way to draw wide lines, and you can only draw a line five pixels wide. We'll show you how to do this later on in the Graphics chapter.

The other graphics command is PLOT. In its simplest form, PLOT outputs a single pixel at any (x,y) location on the screen in any one of the colors in the current color set. Type

```
CLS
      X  Y  Color
PLOT 60,50,3
```

and a single red dot will appear on the screen. You can put this PLOT command in a loop to draw a line. Type

```
FOR X = 1 TO 112:PLOT X,25,2:NEXT
```

and watch a blue line go across your screen at the top of your "window".

In Microsoft 8K BASIC, the graphics capabilities of the PLOT command have been extended. Not only can you tell the computer where and in what color to PLOT, you can also tell it how long and wide you want the plot to be. With Microsoft 8K BASIC, you could draw the same line as you did with the three statements chained together above, but with a single command. Try typing

```
        Color      Length
      X  Y
PLOT 1,35,2,112,1
                 width
```

See how much faster that was? This command told your computer to draw a blue line (color 2) that is 112 pixels long and 1 pixel wide, starting at the first pixel on the left (x-coordinate), 35 pixels up from the bottom of the screen (y-coordinate). Try combining these last two examples on a single line to see the difference in speed. Clear the screen first with CLS.

```
FOR X = 1 TO 112:PLOT X,25,2:NEXT:PLOT 1,35,2,112,1
```

To draw a wider, red line, type

```
PLOT 1,40,3,112,6
```

You can also draw vertical lines of any length and width with the PLOT command. Type

```
      x  Y  color  L  W
PLOT 50,35,1,10,30
```

You can specify a length of up to 112 pixels and a width of up to 77 pixels. Type

```
PLOT 1,1,2,112,77
```

This plots a rectangle the same size as the screen. You can use the PLOT command in this way for a different method of clearing the screen in your programs.

The extensions to the PLOT command in 8K BASIC, while reducing the use of loops, do not eliminate their use entirely. The new PLOT command makes 8K BASIC far more powerful for graphics development. See the Graphics chapter for more information on screen graphics development.

Sounds and Music

Your Interact has one feature few other microcomputers have--it can produce sounds through your TV speaker. It can make sounds in three ways. You can enter SOUND commands to make your computer make a wide variety of noises--clicks, buzzes, beeps, trilling rings. You can use TONE commands to play musical notes or tunes. And, you can use the REWIND command to turn the tape motor on to play music, voice, or sound effects on regular audio cassettes.

Reset the window using WINDOW 77. Now type

```
SOUND 6,242
```

and you'll hear a sound like a telephone dial tone. This sound will continue until you press any key to stop it.

The SOUND command is followed by two values separated by commas. The first value must between 0 and ~~7~~, inclusive. The second can be any value between 1 and 32767, inclusive. Type $\longrightarrow$ 254

```
SOUND 0,24844
```

to hear a wailing siren. Type

```
SOUND 2,230
```

and you'll hear what sounds like a motorboat. Or, try

```
SOUND 3,32
```

to create the sound of a passing locomotive.

You can combine SOUND commands in a loop to produce continuously changing sounds. Type

```
FOR X = 1 TO 5000:SOUND 3,X:NEXT
```

and listen to your Interact produce a series of interesting noises. Your Interact can make hundreds of sounds. Experiment with entering different sound commands to hear some of them. (You'll find that not all values within the parameter ranges produce audible sounds.) A list of our favorite sounds and a program to test the various sounds your Interact can make is included with the SOUND entry in the Reference Section. We also discuss sounds further along with FOR...NEXT looping.

The TONE command also produces sound through the TV speaker. It too is followed by two values. The first specifies the frequency of the tone to be generated, the second controls the length of the tone. Type

```
TONE 168,500
```

to hear a "middle C" tone lasting about three seconds. You can combine tones of different pitches and durations to play musical tunes. For instance, try typing the following TONE sequence:

```
TONE 168,200:TONE131,120:TONE110,150:TONE97,200:TONE131,75:TONE97,300
```

Now, this command string will not all fit on one line on your TV screen. Don't press the "CR" key at the end of a screen line--just let the command wrap around to the next line on the screen. Be sure to separate the individual TONE commands with colons.

If you happen to have a regular audio cassette handy, remove your BASIC tape from the cassette drive and insert the music (or other) tape. Depress the READ cassette button, then type the REWIND command. The tape motor turns on, and you'll hear the tape play through your TV speaker. Press any key to stop the REWIND command and turn the tape motor off.

Functions

Before we leave direct mode and move on to indirect mode operation, let's spend a little time with some of the built-in (intrinsic) functions in Interact BASIC. These are more than 25 pre-defined operations that BASIC can do for you. These functions fall into two general categories: arithmetic functions and string handling functions. As the names suggest, arithmetic functions are processes that return numeric values and string functions return string data. The arithmetic functions by far outnumber the string functions.

String handling functions allow you to manipulate strings in a variety of way. LEFT$, RIGHT$, and MID$ let you isolate characters from within a string. The INSTR$ function permits keyboard entry of a string of a specified length and halts all processing until a string that length is entered. STR$ converts a numeric value to a string value for output on the screen in conjunction with other string values. The CHR$ function returns the ASCII character for any given number.

The intrinsic arithmetic functions are commonly used mathematical operations. Because these functions are built into BASIC, you can perform these operations by simply calling the function in your program, rather than having to define the sometimes complex mathematical code whenever you need to use it. This not only saves you headaches in programming, it also saves space in your programs.

Following is a table of the available mathematical functions and what they compute.

FUNCTION	RETURNS 2π Radians = 360°
ABS	the absolute value of the argument
ATN	the arctangent of the argument in radians
COS	the cosine of the argument in radians
EXP	the anti-logarithm to the base e of the argument
INT	the largest integer that is less than or equal to the argument
LOG	the logarithm to the base e of the argument
NOT	the bitwise complement of the argument
SGN	the sign of the argument as 1 or -1
SIN	the sine of the argument in radians
SQR	the square root of the argument
TAN	the tangent of the argument in radians
FRE	the number of bytes of memory available for program or string storage, as specified by the argument

In direct mode, you can use these functions with your computer to simulate a programmable calculator. For example, try typing the following sequence of commands:

```
A = 100
?ATN(A)
 1.5608
? LOG(A)
 4.60517
?COS(A)
 .862315
?SGN(A)
 1
?SQR(A)
 10
```

You can also use a function call as an argument to another function call where appropriate. For example, try typing

```
? EXP(SQR(A))
 22026.5
```

We suggest you try the various functions in this way to become familiar with their operation in BASIC. For more information on any of these arithmetic functions, see the individual functions within the Reference Section.

There are also three numeric functions that perform operations on strings.

FUNCTION	RETURNS
ASC	the decimal code for a string one character long
LEN	the length of the string argument in number of character spaces
VAL	the string argument as a numeric value

The ASC function is commonly used in the conversion of string data to its base numeric form for storage on tape. LEN is frequently used for positioning of string data, as in centering, on the screen. VAL is the converse of the STR$ function. We discuss these functions in further detail later in the String Handling chapter and also individually in the Reference Section.

Another set of arithmetic functions provide control of information positioning on the screen.

FUNCTION	RETURNS
POINT	the color of any given (x,y) screen location
* POS	character position where next character will be displayed
* SPC	the number of spaces specified in the argument
* TAB	a tabulation the number of spaces specified in the argument before printing the next character

* The POS, SPC, and TAB functions are used only as parameters on PRINT statements, and they all affect positioning of printed information on the screen or lineprinter hard copy.

See the individual Reference Section entries for further information on using these functions in programs.

Then, there are several functions that are used for data entry, either from the keyboard or the entertainment controllers. These functions are commonly used in game programming.

FUNCTION	RETURNS
INSTR$	a string of characters entered from the keyboard of length specified by argument
JOY	a value corresponding to the position of either entertainment controller joystick lever
POT	a value corresponding to the setting of the potentiometer (pot knob) of either entertainment controller
FIRE	a value that indicates whether or not the fire button on either entertainment controller is depressed

See the Controller Input chapter and the individual function entries in the Reference Section for details about and examples of using these data entry functions.

One final intrinsic function...RND. RND is used to generate a random number sequence for use within programs. See the Random Number Generation chapter and the RND entry in the Reference Section for details.

User-Defined Functions

The DEF statement adds a powerful capability to BASIC--it lets you define your own functions for use within BASIC programs. With DEF, although a function may not be intrinsic to BASIC, you can still reduce redundancy in your program code by defining an operation for reuse. For example, let's say you have the need to perform a certain calculation numerous times or on numerous values in your program, such as squaring a number and dividing the result by another value. Rather than performing the individual steps of the calculation each time with statements such as

```
A = 135
B = A*5
? A*A/B
 27
```

you can define the operation as a function with the DEF statement and then print the value. If you try to do this in direct mode, you will find that it does not work ("?ID ERROR"). DEF can only be used for function definition and use in indirect mode. So, let's enter a simple program that illustrates performing the operation above from a user-defined function. In this program we'll change the values of A and B after each function call.

```
10 CLS
20 DEF FNC(C) = A*A/B
30 A = 135
40 B = A*5
50 PRINT FNC(C)
60 A = A + 5
70 GOTO 40
```

In using DEF, the variable name you define to contain the function must begin with the characters "FN". Other characters in the function name are up to you. You may find it helpful to assign a variable name that relates to what the function does. See the DEF entry in chapter 10 for more information on user-defined functions. Appendix D contains a chart of the code for other relatively common mathematical operations that are not intrinsically defined in BASIC, but that can be included in programs by using DEF.

INDIRECT MODE OPERATION

Now let's take a look at how the commands we've covered so far fit into the other mode of programming your computer--indirect mode. Indirect mode is the mode in which you write BASIC programs. You enter the commands as line-numbered statements in a program. These statements just sit there in your computer; they aren't executed until you type the RUN command to start the program. When you type RUN, the statements in the program are executed in the order specified by the sequence of line numbers or the internal program logic, beginning with the lowest numbered line.

You can put any of the previous commands or function calls we've discussed into a program by starting each line with a number. (Press the "CR" key to conclude typing of each line.) Type NEW, then enter the following program.

```
10 CLS
20 PRINT "HELLO THERE!"
```

Now, type RUN. There, you have just written a computer program. Pretty easy, huh?

You can add to this program merely by typing some additional lines. For example,

```
30 PRINT
40 PRINT "GREETINGS FROM"
50 PRINT "YOUR COMPUTER"
```

Type RUN again to see the program run with the newly added lines.

What if you forget a command or want to add something else to the program. Simple. You don't have to retype all the lines. Just add a new line on an unused line number in a logical place in the line number sequence. For example, you might want to change the display colors before printing the message. Type a line numbered 15 that contains the COLOR instruction. For example,

```
15 COLOR 1,0,3,7
```

Now, type the LIST command and see that your computer has put line 15 in the correct place in the numerical sequence--between the CLS statement in line 10 and the PRINT statement in line 20. Type RUN to see the new effect.

What if you notice an error in a line you've already completed by pressing "CR", or want to remove a line entirely? That's just as easy as entering a new line. You correct an error in a line by retyping the line with the same line number. For example, let's say we want two output two blank lines between the messages instead of only one. We could do this by adding a line numbered 25 or 35 that contains a PRINT statement, or we can retype line 30 as follows:

```
30 PRINT:PRINT
```

Type LIST and see that the original instruction in line 30 has been changed.

You can also correct errors in program lines with the EZEDIT program editor and its SUBSTITUTE command. SUBSTITUTE is particularly useful when you need to make the same correction in a number of places in your program. For example, if you decided you want to rename a variable such as A$ that appears numerous times in your program to AB$, you could make the change with a single SUBSTITUTE command in EZEDIT rather than retyping all the lines in BASIC.

Program Listings

You'll find in programming that you will use the LIST command more often than any other command, even RUN. As few people can write a program that runs error-free the first time, LIST is important for correcting errors, or "debugging", your programs. When you type LIST, your computer reads back to you all the program lines you've entered, in the correct sequential order, beginning with the lowest numbered line. However, you can control the beginning line of the listing by adding a line number to the LIST command. For example, if in executing a long program, you find there's a syntax error in line 530 ("?SN ERROR IN 530"), you don't really want to see all the preceding, error-free lines. In that case, type LIST 530, and your program listing will begin at line 530.

Since most programs are more than just a few lines long, you'll find that the lines scroll off the screen faster than you can read them during a listing. You can avoid the frustration of having to list the program over and over by using control characters to halt the listing. A control character is the combination of the Control key and another key, pressed simultaneously. Use:

Control-S	to freeze the listing temporarily. Listing continues when you depress any key.
Control-C	to halt program listing completely. Listing is not restarted until another LIST command is given.

Note that when you type either of these control characters, BASIC completes printing the line being listed at the time the control character is entered before halting the listing. Control-C and Control-S can also be used to halt program execution during a RUN command.

Now, as an exercise, we suggest you go back and put some of the previous examples in the Direct Mode section into indirect mode programs and RUN them. You should get the same results as when you executed the commands in direct mode.

Multiple Statements on a Single Line

As we've previously illustrated, you can chain statements together on a single line. This is useful for compacting programs, in particular, sections of a program that are not likely to be changed. Because the line numbers in a program also consume memory, combining statements on a single line can save space. In your initial programming efforts, you will probably find it easier to put statements on separate lines, especially if you are experimenting with different values in timing loops or TONE statements. Although putting statements on separate line numbers uses more RAM, there's less retyping involved to make changes or corrections in the lines.

BASIC does not have a line renumbering capability. The EZEDIT program editor does, however. In its resequencing and appending operations, EZEDIT renumbers the lines in your program in increments of 10, that is, 10, 20, 30,... It also changes the first line reference in any program line to reflect the new line numbers. If you intend to use EZEDIT to resequence or append programs, do not chain more than one statement with a line reference (e.g., GOTO, GOSUB) on a single line. Because EZEDIT only handles the first line reference it encounters in any one program line during renumbering, any additional line references on the line will have to be changed individually to reflect the new line numbers. To avoid the work involved in tracking down and changing line references after resequencing, put line references on separate lines. For example, don't type

```
80 PRINT A;B$:GOSUB 500:GOTO 10
```

Do type

```
80 PRINT A;B$:GOSUB 500
90 GOTO 10
```

The same effect is achieved, and you won't be required to make correcttions when renumbering lines using EZEDIT.

There will be, however, instances in which you will not be able to avoid multiple line references, particularly with IF...THEN or ON conditional constructions. In those cases, make a note of occurrences and change the multiple line references with the EZEDIT SUBSTITUTE command after you resequence your program.

Program Execution

You normally begin execution of a program by typing the RUN command. RUN starts the program at the first program line and resets all variables to zero. However, there are two other methods of executing a program, both of which are used to start execution at a higher line number in the program. This can save time in debugging parts of your program, because you can avoid having to execute other, unaffected, parts of your program.

The first method uses the GOTO statement. In indirect mode, GOTO transfers program control to a specified line number. You can also use GOTO in direct mode to start program execution at a specific point. When GOTO is used in this way, all program variables remain the way they were set during previous program execution (unless you have reset them in direct mode). For example, let's enter the following program. First, type `COLOR 3,7,1,0` to change the color setting.

```
10 CLS
15 COLOR 6,7,4,0
17 A = 97
18 B=50
20 PRINT "HELLO"
30 PRINT" .":TONE A,B
40 PRINT" .":TONE A,B
50 PRINT" .":TONE A,B
55 PRINT:PRINT
60 PRINT"THIS IS"
70 PRINT"YOUR COMPUTER"
80 PRINT "SPEAKING"
```

Now, execute this program by typing RUN. When the program finishes and you see the "OK" prompt, execute it again, only this time type

```
GOTO 20
```

Note that the program runs just as it did before, except that the screen is not cleared because you instructed your computer to skip that step. You also skipped lines 15, 17, and 18, but the COLOR statement and the variables A and B remain initialized from the previous program execution. If you change the value of a variable, execute the program from the beginning with a RUN command to make sure that variable gets reinitialized.

You can also begin execution in mid-program with the RUN command and a line number. With RUN, however, the variables are automatically reset to zero, regardless of whether you start with the first line in the program or some other line. Change the color setting and execute the program again by typing

```
COLOR 1,3,0,7
RUN 20
```

Note that in this execution of the program, the screen color does not change and the screen is not cleared because those lines with the RUN 20 command.

The tones accompanying the printing of each period (".") are also much higher pitched and much longer. This happens because the variables A and B were reset to zero when you executed the program with RUN, and the TONE statements in lines 30-50 are actually processed as TONE 0,0. In general, to avoid problems resulting from reset variables, we recommend using GOTO to begin execution in mid-program rather than RUN with a line number.

Keyboard Input to Programs

As we stated in the BASIC Basics chapter, there are a number of ways to get data into your program--with entertainment controller functions, from tape with the CLOAD command, with DATA/READ statements in the program. These other methods are discussed at length in other places in the manual. We are immediately concerned with input from the keyboard.

Keyboard input can be obtained in two ways--the INPUT statement and the INSTR$ function. With both operations, your computer takes input from the keyboard and stores it in a variable.

The INPUT statement causes your program to stop and wait for data entry from the keyboard. INPUT prints a question mark on the screen to let the user know that input is expected. Further program execution is halted until the computer receives a "CR" to enter the value. With INPUT, you direct your computer to store the value entered in either a numeric or string variable, as appropriate for the data being entered.

The following program takes input from the keyboard, stores it in the named variables, processes it, and then outputs the results of the processing on the screen. In this program, we'll ask for the user's name, height in inches, and weight in pounds, then convert inches to centimeters and pounds to kilograms for the final display.

```
10 CLS
20 COLOR6,7,4,0
30 PRINT"WHAT'S YOUR"
40 INPUT"NAME";N$
50 CLS:PRINT"HELLO, ";N$
60 PRINT:PRINT"HOW TALL ARE"
70 PRINT"YOU IN INCHES"
80 INPUT IN
90 PRINT:PRINT"HOW MUCH DO YOU"
100 INPUT"WEIGH";LB
110 CM=2.54*IN
120 KG=LB/2.2
130 CLS
140 PRINTN$;", YOU ARE"
150 PRINTCM
160 PRINT"CENTIMETERS TALL"
170 PRINT"BUT YOU WEIGH"
180 PRINT"ONLY";KG
190 PRINT"KILOGRAMS!"
195 FORP=1TO2000:NEXT
200 CLS:GOTO30
```

Notice that you can use the INPUT statement with or without a string constant, as shown in lines 80 and 100. If you execute this program, you'll see that an INPUT statement that contains a string constant is visually different from one without a string constant. Without a string, the INPUT question mark prompt appears on a line alone. If you include a string, the ? prompt is on the same line as and immediately follows the string. If you use INPUT and a string, separate the string and the variable for storage with a semi-colon. If you don't use a string, do not put a semi-colon between the INPUT keyword and the variable name. Generally, we suggest you do use a string to reduce confusion for the user and to indicate what kind of data is being requested.

You'll find that BASIC will let you enter numeric data and store it in a string variable. The reverse is not true, however. If you try to enter string data, such as a name, in response to an INPUT query for numeric data, the message

```
?REDO FROM START
```

prints, and the INPUT prompt reappears. Execute this metric conversion program and type a number when it asks for your name and your name when it asks for your height to see this illustrated.

The other method of keyboard data entry is the INSTR$ function. Because it is a string function (ends in $), data entered in response to the INSTR$ function can only be stored in a string variable. With INSTR$, you specify the length (in number of characters) the entered string is to be. For example,

```
200 A$ = INSTR$(2)
```

within a program causes the computer to stop and wait for entry of two characters from the keyboard before continuing program execution. With INSTR$, the screen remains unchanged--no "?" prompt appears to signal that input is required--and the "CR" key is not required to enter the string. Program execution automatically proceeds when the specified number of characters have been typed. Because the screen is undisturbed with this data entry method, it's a good idea to generate a tone or two just before calling the INSTR$ function, to let the user know that input is needed.

Although this function can technically only handle string data, you can use it indirectly to enter numeric data. Because numeric data can be entered as string data, you could accept entry of a number with INSTR$, then convert the numeric string to a numeric value with the VAL function. You could then perform arithmetic operations on the value. See the String Handling chapter for more information on this type of operation.

The INSTR$ function is commonly used as A$ = INSTR$(1) to input "yes" or "no" (true or false) information on which your computer bases a decision to branch to another part of the program. We'll talk more about this in Conditional Relationships later on in this chapter and along with the IF statement in the Reference Section.

Internal Program Documentation

You can document program operations within the program by entering REM (remark) statements. REM statements are not actually executed when you RUN your program. REM lines are included merely to document your program logic. BASIC considers everything following a REM keyword on a line to be documentation, so never chain multiple statements on such a line. If you do, the statements will never be executed.

We also recommend that you don't branch or loop to a REM line with a GOTO statement. REM lines take up considerable RAM, and, as your program grows, they may have to be removed to gain additional programming space. If you GOTO a REM line, then remove that REM later on, you will get a "?UL ERROR" when you execute the program, unless you first correct all the GOTO references to the REM line. So, to avoid needless retyping, pass program control to the first statement on the line immediately following the REM line.

To illustrate the use of REM statements, we have annotated the previous metric conversion program. If you add those lines, then execute the program again, you'll see no difference from the previous operation.

```
10 CLS
20 COLOR6,7,4,0
25 REM ENTER NAME
30 PRINT"WHAT'S YOUR"
40 INPUT"NAME";N$
50 CLS:PRINT"HELLO, ";N$
55 REM ENTER HEIGHT AND WEIGHT DATA
60 PRINT:PRINT"HOW TALL ARE"
70 PRINT"YOU IN INCHES"
80 INPUT IN
90 PRINT:PRINT"HOW MUCH DO YOU"
100 INPUT"WEIGH";LB
105 REM CONVERT HEIGHT AND WEIGHT TO METRIC EQUIVALENTS
110 CM=2.54*IN
120 KG=LB/2.2
130 CLS
135 REM DISPLAY CONVERTED VALUES
140 PRINTN$;", YOU ARE"
150 PRINTCM
160 PRINT"CENTIMETERS TALL"
170 PRINT"BUT YOU WEIGH"
180 PRINT"ONLY";KG
190 PRINT"KILOGRAMS!"
194 REM PAUSE LOOP
195 FORP=1TO2000:NEXT
197 REM START OVER AGAIN
200 CLS:GOTO30
```

Conditional Relationships

Within a program, you can tell your computer to make a decision about program execution based on the result of testing a condition. Any condition always tests as either "true" or "false". A condition can test for equality, non-equality, greater than, or less than, using the relational operators listed in the BASIC Basics chapter. The statements used to test conditional relationships are IF...GOTO, IF...THEN, and ON.

IF...GOTO tells your computer to transfer program control to the first statement on a specified line, if the given condition tests to be true. For example, consider the following program. It asks the user to select a color by pressing a single key. The IF statments test the value of the 1-character string entered with the INSTR$ function. If the value tests true on any of the IF statements, the GOTO statement on that line is executed, and program control tranfers to the specified line. This program also illustrates testing the result of an INSTR$ function to test for a "yes/no" answer, in this example, to decide whether the program is to be executed again or not.

```
10 CLS:COLOR0,3,1,7
20 OUTPUT"COLOR",40,67,1
30 OUTPUT"TEST PROGRAM",25,60,1
35 WINDOW55
40 PRINT"CHOOSE A COLOR:"
50 PRINT"1=RED   2=GREEN"
60 PRINT"0=BLACK 7=WHITE"
70 PRINT"6=CYAN  3=YELLOW"
80 PRINT"4=BLUE  5=PINK"
90 TONE128,50:TONE94,50
95 A$=INSTR$(1)
100 IFA$="1"GOTO300
110 IFA$="2"GOTO440
120 IFA$="3"GOTO400
130 IFA$="4"GOTO320
140 IFA$="5"GOTO420
150 IFA$="6"GOTO380
160 IFA$="7"GOTO360
170 IFA$="0"GOTO340
300 CLS:COLOR1,3,1,7:OUTPUT"RED",48,60,3:GOTO500
320 CLS:COLOR4,3,1,7:OUTPUT"BLUE",46,60,3:GOTO500
340 CLS:COLOR0,3,1,7:OUTPUT"BLACK",42,60,3:GOTO500
360 CLS:COLOR7,3,7,1:OUTPUT"WHITE",42,50,3:GOTO500
380 CLS:COLOR6,3,1,7:OUTPUT"CYAN",48,60,3:GOTO500
400 CLS:COLOR3.3,1,7:OUTPUT"YELLOW",37,60,3:GOTO500
420 CLS:COLOR5,3,1,7:OUTPUT"PINK",48,60,3:GOTO500
440 CLS:COLOR2,3,1,7:OUTPUT"GREEN",42,60,3
500 FORP=1TO1000:NEXT
505 CLS
510 OUTPUT"AGAIN?",35,60,3
520 A$=INSTR$(1)
530 IFA$="Y"GOTO10
540 IFA$="N"GOTO600
550 GOTO520
600 CLS:OUTPUT"THANK YOU",25,60,1
```

There's only one problem you might encounter when you execute this program. If you type a character other than 0 through 7, line 100 is executed. This happens because none of the IF statements tests true, so execution continues with the first statement following the conditional testing. To avoid this problem, we'll tell the computer to return to the INSTR$ function if none of the conditions tests true. Type

```
180 GOTO 95
```

then RUN the program again and try to repeat the problem. You'll see that the computer no longer accepts any key except 0 through 7.

IF...THEN statements also test conditional relationships. If a relation tests true, the computer executes all statements chained following the THEN portion of the statement. If no statement that transfers program control appears following THEN, program control passes to the next higher line number after the chained statements have all been executed.

We could change the previous program to use the IF...THEN construction by changing lines 100 through 180 as shown below, and completely removing lines 300-440. This makes the program considerably shorter, as you can see. If you make these changes, then execute the program again, you won't notice any operational difference. This illustrates the fact that you can use a variety of styles in programming. There is no real "right" way to program, just as there is no "right" way to express a thought in a language, as long as the syntax is correct.

```
10 CLS:COLOR 0,3,1,7
20 OUTPUT "COLOR",40,67,1
30 OUTPUT "TEST PROGRAM",25,60,1
35 WINDOW 55
40 PRINT "CHOOSE A COLOR:"
50 PRINT"1=RED   2=GREEN"
60 PRINT"0=BLACK 7=WHITE"
70 PRINT"6=CYAN  3=YELLOW"
80 PRINT"4=BLUE  5=PINK"
90 TONE 128,50:TONE 94,50
95 A$ = INSTR$(1)
100 IF A$ = "1" THEN CLS:COLOR 1,3,1,7:OUTPUT "RED",48,60,3:GOTO 500
110 IF A$ = "2" THEN CLS:COLOR 2,3,1,7:OUTPUT "GREEN",40,60,3:GOTO 500
120 IF A$ = "3" THEN CLS:COLOR 3,3,1,7:OUTPUT "YELLOW",36,60,3:GOTO 500
130 IF A$ = "4" THEN CLS:COLOR 4,3,1,7:OUTPUT "BLUE",44,60,3:GOTO 500
140 IF A$ = "5" THEN CLS:COLOR 5,3,1,7:OUTPUT "PINK",44,60,3:GOTO 500
150 IF A$ = "6" THEN CLS:COLOR 6,3,1,7:OUTPUT "CYAN",44,60,3:GOTO 500
160 IF A$ = "7" THEN CLS:COLOR 7,3,7,1:OUTPUT "WHITE",40,60,3:GOTO 500
170 IF A$ = "0" THEN CLS:COLOR 0,3,1,7:OUTPUT "BLACK",40,60,3:GOTO 500
180 GOTO 95
500 FOR P = 1 TO 1000:NEXT
                    .
                    .  etc.
                    .
```

So, how do you decide whether to use the IF...GOTO or IF...THEN construction? In many cases, it doesn't make much difference which you use. In general, though, use IF...GOTO if you want program control transferred to another part of the program if a given condition is true. Use IF...THEN if you just want to execute a short series of statements if the condition is true. Note that you can use GOTO in conjunction the IF...THEN construction if you want to transfer program control <u>after</u> executing some other statements.

You can also combine conditions to be tested during execution of an IF statement with either of the BOOLEAN operators--AND and OR.

If you use the AND operator, you tell your computer to execute the GOTO or THEN statements only if all the conditions given are true. If you use the OR operator, at least one of the given conditions must be met for the GOTO or THEN statements to be executed. You can test up to 5 conditions at a time with AND and OR, and you can combine AND and OR tests within a single IF statement.

The following program tests for depression of the fire button on either the left OR right controller (line 30). If either fire button is depressed, program control transfers to line 50.

```
10 CLS:COLOR 4,7,3,7
20 OUTPUT"PUSH FIRE BUTTON",10,60,1
30 IF FIRE(0)=0 OR FIRE(1)=0 GOTO 50
40 GOTO 30
50 CLS:PRINT"FUN ISN'T IT"
60 FOR Q=1 TO 999:NEXT
70 GOTO 10
```

Our next example illustrates conditional testing with the AND operator. It's not a complete program, but it demonstrates how to test to determine if a number is within a specified range, in this case, 1-12.

```
10 CLS:COLOR 7,1,4,2
20 PRINT"ENTER MONTH="
30 INPUT M
35 M=INT(M)
40 IF M>0 AND M<13 GOTO 60
45 PRINT"BAD MONTH CODE"
50 GOTO 20
60 REM ETC., ETC.
```

Look Before You Loop

How do you re-execute a part of your program more than once without retyping all the lines again? We've already seen that we can use the GOTO statement to loop through a set of instructions multiple times. The problem with using GOTO statements for looping is that they can create what are called "infinite loops". That is, the program will execute the statements again and again and again, until you halt execution by typing Control-C. This is fine if you want your program to loop indefinitely, but what if you want to perform an operation five times and no more? In this case, the GOTO statement may be replaced with a different kind of looping--the FOR...NEXT loop.

FOR and NEXT are a pair of instructions that define the beginning and ending points of a loop, respectively. Any statements within the loop will be executed as many times as specified in the beginning FOR statement. In the Direct Mode Operation section of this chapter, we gave a simple example of using a FOR...NEXT loop to draw a line:

```
FOR X = 1 TO 112:PLOT X,50,2:NEXT
```

In this set of instructions, X is the iteration variable. The iteration variable acts as a loop counter--its value is incremented in steps of 1 each time the statements in the loop are executed and the NEXT statement is reached. 1 is defined as the starting value of X and 112 is defined as its highest value. When the loop counter reaches the maximum value, looping ends, and the statement immediately following NEXT is executed. In this example, we also use X as the x-coordinate in the PLOT statement.

The NEXT statement identifies the end of a loop iteration. NEXT increments the value of the iteration variable (loop counter). It also contains an implicit GOTO statement that sends program control back to the originating FOR statement, unless the iteration variable is equal to or exceeds the ending value of X.

The following simple program illustrates looping even more clearly.

```
10 FOR I = 1 TO 10
20 PRINT "HELLO"
30 NEXT I
```

RUN this program and you'll see the word HELLO print on your TV screen ten times. After the tenth printing of HELLO, the program ends.

In an earlier example (Program Execution, page 2-16), we gave three identical PRINT commands on three different lines. You can see now that this was really a waste of space and typing--we could have combined all three statements into a FOR...NEXT loop, as follows:

```
10 CLS:COLOR 6,7,4,0
15 A = 97
17 B = 50
20 PRINT "HELLO"
30 FOR I = 1 TO 3
40 PRINT " .":TONE A,B
50 NEXT I
60 PRINT:PRINT
70 PRINT"THIS IS YOUR"
80 PRINT"COMPUTER SPEAKING"
```

An even simpler form of looping is used to control program timing. It's called the pause loop. In pause loops, you simply give the beginning FOR and ending NEXT statements without any other statements in between:

```
200 FOR P = 1 TO 500:NEXT
```

Essentially, what you are telling your computer is "Go do nothing 500 times, before you do anything else." Pause loops are particularly useful for holding visuals on the screen long enough so that they can be fully appreciated, or instructions long enough to be read.

What goes inside a loop and what stays out? This is very important for the successful execution of a program containing loops. If you don't have everything you need in the loop, or if you have statements included that shouldn't be there, your program will not do what you want it to do. Put only those statements in a loop that you want re-executed on each iteration through the loop. Initialization assignments of variables with values to be incremented during the loop, for example, should not be included in the loop. If you do include them, the variables will be reinitialized each time instead of incremented. As an example, enter the following program.

```
10 CLS
20 COLOR6,4,7,0
30 X=55
40 Y=38
50 XL=5
60 YL=4
70 FORC=1TO3
80 PLOTX,Y,C,XL,YL
90 X=X-2
100 Y=Y-2
110 XL=XL+4
120 YL=YL+4
125 FORQ=1TO100:NEXT
130 NEXT
140 IFY>=6GOTO70
150 GOTO10
```

This program requires Microsoft 8K BASIC. It draws a series of progressively larger squares on the screen by incrementing the (x,y) coordinates and X and Y lengths of each plot within the loop. Note that the initial values of X, Y, XL, and YL are defined outside the loop. If you put them inside the loop, the same size square will be redrawn over and over again. Find out more about using FOR...NEXT loops for graphics development in the Graphics chapter.

The uses for FOR...NEXT loops are virtually infinite. They can be used for graphic image development, putting pauses in programs, controlling data entry, performing calculations, etc. You'll find that almost every program you write will contain at least one FOR...NEXT loop.

Let's look at a data entry example. The following program segment might be used in a game program in which you want to input four players' names, then randomly choose one of them as the first player in the game.

```
10 CLS:COLOR7,4,0,1
20 WINDOW50
30 OUTPUT"PLAYER NAME",18,58,1
35 FORI=0TO3 ← 4 total
40 INPUTN$(I)
50 NEXT
60 F=RND(1)*4
65 WINDOW77:CLS
70 OUTPUTN$(F),56-LEN(N$(F))/2*6,55,1
80 OUTPUT"GOES FIRST",25,45,1
90 A$=INSTR$(1):GOTO10
```

This program takes in four names within the loop and assigns them to the variables N$(0), N$(1), N$(2), and N$(3). (Note that you can store more than one data value in one variable name by using subscripts to the variable name.) The program then tells the computer to choose a number between 0 and 3 to determine which player goes first. Statements on lines 60 through 80 are not executed until after looping completes.

FOR...NEXT can be embedded, or "nested", within other FOR...NEXT loops. For example,

```
10 CLS
20 COLOR1,0,6,7
30 FOR X = 40 TO 70
40 FOR Y = 60 TO 30 STEP -1
50 PLOT X,Y,3
60 NEXT:NEXT
```

Note that the starting variable need not be 1. The starting value merely defines what the initial value of the iteration variable is to be. In this case, we want to start our plot 40 pixels from the left edge of the screen. The starting variable can be less than the ending variable, as illustrated in line 40, for "backwards" looping. For backwards looping, you must use the STEP option with a negative value to specify negative incrementation of the iteration variable. Change line 40 to read

```
40 FOR Y = 60 TO 30 STEP -2
```

and run the program again. This time, a set of vertical stripes will be drawn on the screen.

The STEP option specifies the incrementation of the iteration variable. If STEP is not included on the FOR statement, your computer assumes you mean STEP 1. If you add a different incrementation value in line 30,

```
30 FOR X = 40 TO 70 STEP 2
```

and run the program another time, you'll see a pattern of dots appear on the screen.

In the previous example, both FOR...NEXT loops concluded at the same point. This need not be the case, however. You can and will insert other statements between the end of a nested (or inner) loop and the end of an outer loop. The following program illustrates a FOR...NEXT loop that contains two nested loops. The second nested loop is executed between the end of the first nested loop and the end of the outer loop.

```
10 CLS:COLOR0,2,1,6
20 FORN=1TO3
30 INPUT"NAME";N$
40 CLS:OUTPUTN$,56-LEN(N$)/2*6,45,3
50 FORC=1TO10
60 COLOR0,2,6,1
70 TONE76,25
80 COLOR0,2,1,6
90 TONE76,25
100 NEXTC
110 FORP=1TO150:NEXTP
120 CLS:NEXTN
```

The outer loop is the data entry loop (N). When a string is entered, then the color loop (C) is executed, followed by a pause loop (P), then the next iteration of the data entry loop.

Inclusion of the iteration variable on the NEXT statement is completely optional. As your programs increase in size, you'll find that elimination of this extra character gains you a couple more bytes of RAM and helps compact the program. We've included them in the previous examples for the sake of logic clarity.

You can also use loops to combine sounds or tones in your programs. The next two examples illustrate how FOR...NEXT loops can be used to add amusing sound effects to a program execution.

The first program plots a dotted, diagonal line down one side of the screen and up the other. As it plots, the computer produces tones that slide up and down the scale in accompaniment.

```
10 CLS
20 X=10
30 FORY=60TO20STEP-1
40 PLOTX,Y,2
50 TONEX,Y
55 X=X+1
60 NEXT
65 Z=X
70 FORY=20TO60
80 PLOTX,Y,1
90 TONEZ,Y
100 X=X+1
105 Z=Z-1
110 NEXT
120 GOTO10
```

Our second example creates a series of continuously changing sounds from within two loops, one of which is nested in the other.

```
10 CLS
20 PRINT"LISTEN TO ME..."
30 FOR Y= 1 TO 4000 STEP 5
35 FOR X=0 TO 7
40 SOUND X,Y
50 NEXT:NEXT
```

Try increasing or decreasing the STEP parameter in line 30 for different sets of sound effects.

The value of the iteration variable can be sensed within a loop and its value used as a condition on which to base a program logic decision. However, NEVER change the value of the iteration variable within a loop, or your computer can get hopelessly confused.

As an example of sensing the value of the iteration variable, let's say we want part of a program to allow entry of up to six lines of text for display elsewhere in the program. However, we want the user to be able to enter fewer than six lines as well. You might set up such a program as follows:

```
10 CLS:COLOR2,3,7,0
20 OUTPUT"ENTER UP TO 6",15,65,2
30 OUTPUT"LINES OF TEXT",18,57,2
40 OUTPUT"STOP BY TYPING",15,45,2
50 OUTPUT"END",47,37,3
60 FORI=1TO6
70 INPUT T$(I)
80 IF T$(I)="END" OR I=6 THEN N=I-1:GOTO200
90 NEXTI
200 CLS
205 FORP=1TON
210 PRINTT$(P)
220 NEXT
```

At the end of each loop iteration, your computer tests to see if it was the last iteration through the loop or if the string entered during that iteration is "END". If either of those conditions is met, looping ends, and execution of the rest of the program proceeds.

In several of the previous examples, we've stored values in subscripted variables (e.g, N$(0), N$(1), T$(I)). BASIC has ten automatically defined slots within any variable that can be used to store data associated with that variable. These slots form what's called an <u>array</u>. Arrays provide a convenient method of storing data that is related in some way. Let's leave FOR...NEXT loops now and move on to a discussion of arrays and their dimensions. If you're still confused about using FOR...NEXT loops in programs, review this section again. But, because looping is so widely used in program logic, you'll see plenty more examples that illustrate looping in conjunction with other operations as we go along.

Arrays -- Putting Data in Its Place

So far, our walk through BASIC has focused on the manipulation of information which is made of a single part. We've seen how to manipulate numbers by using them as constants or giving them variable names. We've seen how we can manipulate a group of letters by using it in a string constant or storing it in a string variable name.

However, data often consists of information that is made up of several parts. In everyday terms, we call this information "lists". We have lists of phone numbers, part numbers, names and addresses of friends, etc. In BASIC we call these lists <u>dimensioned arrays</u>. We refer to a particular data element in an array as a <u>subscripted variable</u>.

As an example, let's suppose you want to write a program to keep track of your total family expenses over a twelve-month period. One approach would be to name 12 variables (TE1, TE2, TE3,...TE12 -- one for each month), and work with them individually. However, you would find that your program would grow large and cumbersome very quickly. Let's say we want to keep track of the total expense for each month and determine what percentage

each month is of the grand total. First, we'd define 12 variables to store the monthly totals:

```
                 10 TE1 = 849
                 20 TE2 = 569
                 30 TE3 = 723
                      .
                      .
                      .
                120 TE12 = 1248
```

Then, we'd have to calculate the yearly total (YE).

```
                 130 YE = TE1 + TE2 + TE3 + ... + TE12
```

And, to calculate the monthly percentage of total, we'd have to do a separate instruction for each calculation:

```
                 140 P1 = YE/TE1
                 150 P2 = YE/TE2
                 160 P3 = YE/TE3
                       .
                       .
                       .
                 260 P12 = YE/TE12
```

You can see that this is already starting to get complicated. We're generating lots of variables, using lots of lines, and we've only done one major operation. In lines 140 through 260, the same operation is being performed on each month. Since many operations in a program like this would conceivably be the same for any given month, there must be a better approach.

And, there is. Fortunately, in BASIC we have the concept of a dimensioned array. A dimensioned array lets you work with this type of data in compact BASIC statements. This simplifies the programming task and also reduces the amount of RAM used.

Let's consider instead a total expense variable, called TE, that has 12 associated values--one for each month. When working with a list or array like this in a program, you first need to tell BASIC how many values will be associated with that variable name. You do this with the DIM (dimension) statement. For example, enter

```
                 20 DIM TE(12)
```

This tells the computer that 12 storage slots are to be reserved for use by the variable named TE. It says nothing about the contents of the slots. That information is specified later on in the program.

The DIM statement for an array is executed only once in a program, usually at the beginning. Numeric and string arrays are dimensioned in the same way. You can dimension several variables in one DIM statement. For example,

```
                 25 DIM TE(12),EC$(6),A(50)
```

You can, of course, use separate DIM statements if you prefer.

If you do not dimension a variable before you reference it, you'll find that BASIC automatically reserves 10 slots for the variable. Two examples of using automatically dimensioned arrays were given in the previous FOR...NEXT looping discussion. In practice, it's a good idea to dimension all variables that you'll be using as lists by entering a DIM statement. If you forget to dimension an array and try to reference above the 10th subscript, you'll get a "?BS ERROR" (bad subscript).

In dimensioning arrays, be careful not to specify more storage than you actually need, because your computer can quickly run out of memory. The following program, for example, will return a "?OM ERROR" if you try to execute it. It's simply too large for your Interact to handle.

```
10 DIM A(1500)
20 PRINT "HELLO"
```

To reference a value stored in an array (an item on our list), we use the variable name followed by a subscript in parentheses. Going back to our family expense problem, we've dimensioned the total expense variable into 12 slots, one for each month, with the statement

```
20 DIM TE(12)
```

If we wanted to see what was stored in the slot for April, we'd call the TE variable with the 4th subscript:

```
PRINT TE(4)
```

Subscripts can also be variables. In particular, they are often the same as the iteration variable in a FOR...NEXT loop used to index through an array. The following example computes the yearly total by incrementing the variable YE (yearly expense) by the value of each month. YE is initialized as 0.

```
290 YE=0
300 FOR M=1 TO 12
310 YE=YE+TE(M)
320 NEXT
330 PRINT"YEARLY TOTAL";YE
```

If we wanted to perform the percent of total calculations we did earlier with separate variables, this process could also be done within a loop. An array to contain the percentage figures must also be dimensioned.

```
30 DIM P(12)
```

See how ~~that~~ the calculation from a loop is much simpler than using individual statements for calculations.

```
500 FOR M=1 TO 12
510 P(M) = YE/TE(M)
520 NEXT
```

Two-Dimensional Arrays

Arrays can have more than one dimension. In familiar terms, a two-dimensional array is a table with rows and columns. Each position in the table can have an associated data value. Extending our previous example, let's assume we have five expense categories--food, housing, auto, computers, and total expense--for each month in a twelve-month period. You can store all this information under a single variable name in a two-dimensional array. You could dimension this array as follows:

```
20 DIM TE(5,12)
```

We have now defined TE as a table that has 5 rows (one for each of the categories) and 12 columns (one for each month). Our choice of rows (the first subscript position) to represent the categories and columns (the second subscript position) to represent the months is arbitrary. The statement

```
20 DIM TE(12,5)
```

would have worked equally well. However, all subsequent referencing of an array must be consistent with how it is defined in the DIM statement.

The number specified in the DIM statement is the highest subscript that can be used in referencing that dimension of the array. The DIM statement actually allocated one more slot than is indicated in the DIM argument. DIM A(12) really allocated 13 slots, A(0) through A(12). Similarly, in two-dimensional arrays there is a zeroth row and a zeroth column. You can use these zeroth slots for storing other data related to the array, such as row and/or column totals.

You can compute values associated with slots in an array by using other information stored in the array. In our family expense problem, we could compute the total expense for each month by adding the values in the other four categories. We'll store the value obtained in the 5th row, using FOR...NEXT loops.

```
10 DIM TE(5,12)
600 FOR M = 1 TO 12
610 FOR R = 1 TO 4
620 TE(5,M) = TE(5,M) + TE(R,M)
630 NEXT R
640 NEXT M
```

The subscripts in a variable reference select which data value from the array is to be referenced. BASIC always interprets subscripts as integers. Subscripts cannot be negative, and they must be less than or equal to the maximum subscript declared for that dimension in the associated DIM statement.

The following "Temperature Tracker" program illustrates using a two-dimensional array to collect daily low and high temperatures for seven days, then computes the average low and average high temperature for that week.

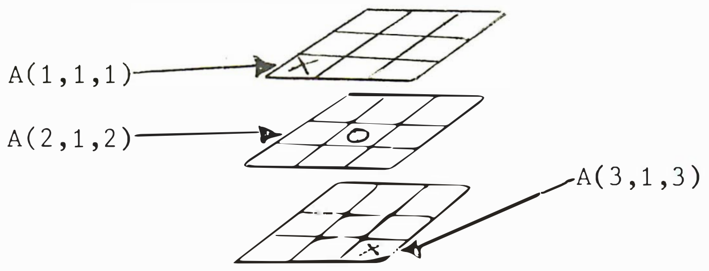
```
10 DIM T(7,2)
20 CLS:COLOR 4,3,0,7
30 OUTPUT" DAILY TEMP.",10,66,1
40 OUTPUT"DAY LOW,HIGH",10,58,2
50 WINDOW 50
55 REM- INPUT DAILY LOW AND HIGH
60 FOR I=1 TO 7
70 PRINT I;
80 INPUT T(I,1),T(I,2)
90 NEXT
95 REM- SUM THE LOWS AND HIGHS
100 FOR I=1 TO 7
115 FOR J=1 TO 2
120 T(0,J)=T(0,J)+T(I,J)
130 NEXT:NEXT
135 REM- COMPUTE AVERAGES
140 LOW=T(0,1)/7
150 HIGH=T(0,2)/7
155 REM- REPORT
160 CLS
170 PRINT"AVG. LOW":PRINTLOW
180 PRINT"AVG. HI ":PRINTHIGH
190 PRINT
```

Higher Dimensional Arrays

In BASIC, an array can have as many as five dimensions or subscripts. When might you use a three-dimensional array? Consider the implementation of a 3-D Tic-Tac-Toe game. You might want to use an array dimensioned as follows:

```
10 DIM A(3,3,3)
```

to represent all twenty-seven cells in the game. The value associated with any particular cell in the game cube would indicate if an "X" or "O" (or neither) had been placed in that cell.



A four-dimensional array might be used for keeping track of dollar sales by product, by store, by department, over time.

A five-dimensional array might be used to track sales by product, by district, by office, by salesperson, over time. However, an application that size cannot realistically be visualized for the Interact with its current memory limitations.

Entering Data into Arrays

Most programs that process numeric data in arrays contain some mechanism for entering the data. In our previous family expense problem, if you used the PRINT command to disply values stored in the array, all values returned would be zero. That's because we never entered any data values into the array for the program to work with.

The most straight-forward approach to data entry is to accept data the user enters from the keyboard in response to INPUT statements. Here are some guidelines as to how this can be accomplished efficiently and painlessly.

1) Prompt the user as to what data is being requested. You can do this by interleaving PRINT and INPUT messages, or by supplying the prompt information as a string within the INPUT statement.

2) Ask for the data in a logical order.

3) Immediately check to verify that the data is reasonable. Unreasonable data might be a negative value or an excessively large value. Prompt for the data item again if there is any question as to its validity. Allow the user to reenter part of the data if he finds an error.

4) Ask only for data that the program can't assume. For example, if you have an array with 300 slots defined for product sales information, but you only have 35 products with non-zero sales information, prompting for all 300 values consumes needless time and effort. Allow entry of the non-zero information and product code on the same line, separated by a comma, e.g., PC,595.

On the next page is a portion of a program that accepts family asset information for processing. Each asset category is stored in a position of the "A" array. On examining this program, you'll find that a WINDOW statement keeps the "ENTER ASSETS" instructions visible at all times. Each asset category is identified. The INPUT statement in line 320 takes each value and stores it in the next (Qth) position of the array A. When the last asset has been input, the total is computed (lines 120-130), and the user is given a chance to verify that it is correct. If "YES", the program would begin to process the information, starting with line 500. If "NO", then a repeat variable "R" is set, and the questions are repeated. However, in the repeat mode of this program, the value previously entered for each asset is displayed on the screen. If the value is correct, striking the "CR" key will retain that value and move to the next item. If a correction is required, the user can supply the corrected value and press "CR".

In other words, the INPUT statement will leave the variable unaltered if nothing is typed in at the keyboard to change it.

```
10 DIM A(5)
20 CLS:COLOR 0,3,7,7
25 Q=0
27 T=0
30 OUTPUT"ENTER ASSETS",10,66,1
40 OUTPUT"IN $000",10,60,1
50 WINDOW 54
60 PRINT"HOME VALUE"
70 GOSUB 300
80 PRINT"AUTOS"
90 GOSUB 300
100 PRINT"STOCKS AND BONDS"
110 GOSUB 300
120 FOR J=1 TO 3
130 T=A(J)+T:NEXT
140 PRINT:PRINT:PRINT"TOTAL";T
150 PRINT:PRINT"CORRECT?"
160 A$=INSTR$(1)
170 IF A$="Y" GOTO 500
180 IF A$="N" GOTO 200
190 GOTO 160
200 R=1
210 GOTO 20
299 REM-INPUT AND STORE IN ARRAY
300 Q=Q+1
310 IF R=1 THEN PRINT"(WAS $";A(Q);")"
320 INPUT A(Q)
330 PRINT:RETURN
500 END
```

Entering string data with the INPUT statement is essentially the same operation as entering numeric data, except that the data is stored in a string variable. There is, however, a potential conflict. Since the comma (,) can be used as a delimiter to allow entry of multiple values with a single INPUT statement, a string that contains a comma, such as "KENNEDY, JOHN", must be flagged as a string containing a comma, rather than two separate values. You do this by entering the string in quotes in response to the INPUT prompt. If the string data contains no commas, no quotes are required to enter it. Consult the Reference Section for more details on INPUT data entry.

You can also load data into a program with a CLOAD*A statement, where A represents a previously dimensioned array name. Your program should provide a guide through loading and positioning the data tape prior to issuing the CLOAD*A statement. The programs in Financial Library I illustrate how data is passed from one program to another in this fashion. Also see the CLOAD and POKE statements in the Reference Section for more information on passing data from program to program.

One final note about arrays. Both the RUN command and CLEAR statement set all numeric variables to zero and all string variables to null or empty. Therefore, there's no need to initialize arrays to zero at the beginning of a program. Save that RAM for other, more important, processing steps.

Pick a Number Between 1 and 100...

Chance is an element of many computer games--the card hand that you draw in Black Jack, whether or not you fumble on any given play in SUPERBOWL, or the roll of the dice in Knockdown. To add this element of chance to your own BASIC programs, you will need to become familiar with random number generation. The RND function in the BASIC language provides the facility for adding this chance factor to your games and displays.

What's a random number generator, anyway? You can think of it as a little machine that sits in your computer and spits out a random number each time it's called. The number it returns always has a value between 0.0 and 1.0. The likelihood that a returned number will be within a specified interval (say, between .30 and .50) is proportional to the length of the interval. That is, about 20% of the numbers returned would be within the .30 - .50 interval. If you take the entire possible interval of 0.0 to 1.0, half the numbers returned will be less than .50, 25% of the numbers will be less than .25 or greater than .75, 10% will be between .90 and 1.0. Of course, for these ratios to be true, large quantities of random numbers must be "drawn from a hat." To see a list of random numbers displayed on your TV screen, RUN the following program:

```
10 FOR I=1 TO 8
20 PRINT RND(1)
30 NEXT
```

In using the random number generator, you can't predict what particular value you'll draw next. You can only assign a probability or likelihood that the number drawn will lie within a certain interval. To test the random number generator, let's draw 1,000 random numbers and count how many are larger than .50 and how many are smaller than .50. If the numbers are uniformly distributed over the interval, then you should see approximately 500 numbers in each category. If you run the following program several times, you'll see different counts, but all will hover about the 500 mark.

```
10 CLS
20 PRINT"RANDOM NUMBER"
30 PRINT"GENERATOR AT
40 PRINT"WORK..."
50 PRINT
60 FOR N=1 TO 1000
70 R=RND(1)
80 IF R>.50 THEN H=H+1
90 IF R<.50 THEN L=L+1
100 NEXT
110 PRINT"ABOVE .5 =";H
120 PRINT"BELOW .5 =";L
130 PRINT
```

The more numbers you draw, the better the 50% rule will hold.

Now, let's consider a subroutine that will return a random number, R, that has a value between 1 and 6, with equal probability of any number in that range being drawn, as would be the case with the throw of a single die. The subroutine will use the RND function and then apply an arithmetic expression to spread the number in an interval of 0.0 to 6.999, then extract the integer part of the number computed. This can be done with the single line

```
50 R = INT(6 * RND(1)) + 1:RETURN
```

You might put this subroutine in a program that simulates 8 throws of the die, such as

```
10 CLS
30 FOR I=1 TO 8
40 GOSUB 50
42 PRINTR
45 NEXT
46 END
50 R=INT(6*RND(1))+1:RETURN
```

To draw a number between 0 and 3, change line 50 to read

```
50 R = INT(3 * RND(1)) + 1:RETURN
```

You have to add 1 to the random number returned in the statement above, or the number 3 will rarely be returned. In drawing a random number in the range 0.0 to 1.0, there is only a very small probability that the number 1.0 will be returned. Therefore, you must either add 1 to the random value returned, or increase the range specified. You could also change line 50 to read

```
50 R = INT(4 * RND(1)):RETURN
```

to draw a number between 1 and 3. The likelihood that a value of 4.0 will be returned is miniscule.

The argument (1) following the RND call simply tells the computer to draw the "next" random number. Actually, any positive value can be used as the argument in this call without affecting the sequence of random numbers that will be returned.

How random are the random numbers? Well, actually the Interact has what's called a "pseudo-random number generator". The random values are computed, and the number generator produces the same sequence from the time the computer is turned on, each time it's turned on. You can, however, randomize the generator starting point (or "seed") in the series by calling the RND function with a negative argument to set the seed. While there are several ways to initialize this seed randomly, the easiest way is to pass the negative value of the computer's clock to the subroutine. To do this, use a statement such as

```
30 J = RND(-PEEK(24559))
```

This statement sets the initial seed for random number generation to the point in the number sequence that is equal to the value of the clock. This value will be different each time the program in executed. A statement of this type should only be executed once at the beginning of your program, and the returned variable, J, has no particular meaning outside the initialization call, nor would it normally be used elsewhere in the program.

Want to draw a hand from a deck of cards? If you want to allow repeated draws from the "deck" during game play, set up an array in memory that has 52 locations. The position of any element in the array is associated with the suit and value of the card. The value in the array will be 0 if the card has been drawn before. As more of the deck is played, you note that the program runs more slowly. It has to perform the card selection sequence in lines 100-120 numerous times to find a card that has not already been drawn.

```
10 REM-DRAW A CARD
20 CLS:COLOR 4,3,0,7
30 DIM C(52)
35 FOR I=1 TO 52:C(I)=1:NEXT
40 OUTPUT"HIT ANY KEY",10,66,1
50 OUTPUT"TO GET NEXT",10,60,1
60 OUTPUT"CARD...",10,54,1
70 WINDOW 48
80 A$=INSTR$(1):NC=NC+1
90 IFNC=53THEN PRINT"OUT OF CARDS":PRINT:END
100 R=INT(RND(1)*52)+1
110 IF C(R)=0 GOTO 100
120 C(R)=0
130 S$="CLUBS"
140 IF R>13 AND R<27 THEN S$="SPADES":GOTO 170
150 IF R>26 AND R<40 THEN S$="DIAMONDS":GOTO 170
160 IF R>39 THEN S$="HEARTS"
170 FOR I=1 TO 14
180 IF R<14 GOTO 200
190 R=R-13:NEXT
200 V$=STR$(R)
210 IF R=1 THEN V$="ACE":GOTO 250
220 IF R=11 THEN V$="JACK":GOTO 250
230 IF R=12 THEN V$="QUEEN":GOTO 250
240 IF R=13 THEN V$="KING"
250 PRINTV$;" ";S$
260 GOTO 80
```

To conclude our discussion of random number generation, let's look at a classical problem in probability theory called the "random walk". Imagine that a drunk has been placed in the exact center of a square field that is bounded by bars on all sides. In the irrationality of his drunken stupor, our drunk thinks that he needs yet another drink, so he attempts to make his way out of the field and back to one of the bars. He does this by taking random, stumbling steps in each of the four directions. That is, on any given step, he is just as likely to walk forward as backward, to the right as to the left. The question is, if the size of the field is known in number of steps, what's the average number of steps it will take for the drunk to reach one of the bars?

We can solve this problem by simulating the walk on the Interact and displaying the number of steps the drunk takes each time we run the program. In this example, we've placed the drunk in the middle of a 40x40 square field. The listing of the program that simulates this walk is given below. It illustrates not only the use of random numbers, but also the POINT and ON...GOSUB elements of the BASIC language.

```
10 REM-RANDOM WALK
20 CLS:COLOR 0,7,4,3
30 PLOT 20,20.1.40,40
40 C=0:X=40:Y=40
50 GOSUB 90
60 GOSUB 150
70 GOTO 50
80 REM-PLOTS POINT
90 IF POINT(X,Y)=0 GOTO 110
100 PLOT X,Y,2:PLOTX,Y,3:C=C+1:RETURN
110 OUTPUT C,70,50,3
120 OUTPUT"STEPS",70,44,3
130 FOR Q=1 TO 999:NEXT:TONE 50,50:RUN
140 REM-TAKE A STEP
150 R=INT(4*RND(1))+1
160 ON R GOSUB 180,190,200,210
170 RETURN
180 X=X+1:RETURN
190 X=X-1:RETURN
200 Y=Y+1:RETURN
210 Y=Y-1:RETURN
```

As an exercise, try adding lines of code that will execute this program repeatedly and store the number of steps taken on each execution in a running, moving average.

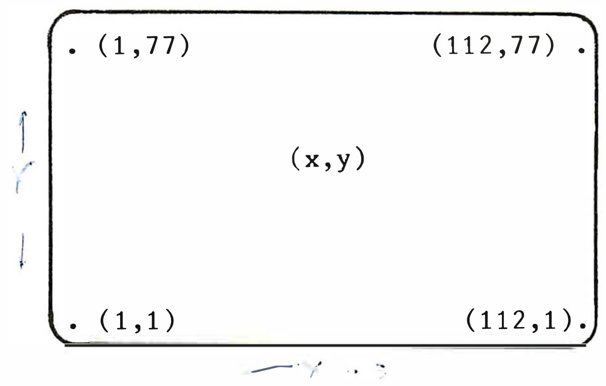
See the RND function in the Reference Section for more information on random number generation. There are also numerous other examples in this manual that use the RND function to generate random values for a variety of purposes.

GRAPHICALLY SPEAKING

The old adage "A picture is worth a thousand words" rings particularly true for the Interact. Despite its screen resolution, your Interact is capable of producing graphic displays that are visually dazzling. You'll find that graphics add another dimension to your BASIC programs, making them amusing and entertaining to execute. Virtually all the effects in the various game programs can be adapted for use in your own programs and games written in BASIC. While some effects are naturally more difficult to implement than others, you'll find that the graphics capabilities of your Interact provide one of the best ways to sharpen your programming and artistic skills. Effective use of the graphics display makes any program you write more interesting. Let's take a look at the features available and how you can use them.

The Screen

As we previously discussed, your TV screen can be addressed as a matrix of picture cells ("pixels"). Your TV screen, whether it is a 13-inch or 4-foot large screen, has 77 rows and 112 columns. Each point on the screen can be addressed as an (x,y) coordinate pair of this matrix, where x varies from 1 to 112 and y varies from 1 to 77.



The origin of the screen is at the lower left corner, and can be thought of as location (0,0), although that point can't be seen on most TV screens. The scanning, or placement, of the matrix on the screen can vary from TV to TV. Therefore, we recommend that you avoid putting graphic images or text too close to the screen edges. Images too close to any edge can appear to be "chopped off" on some TV screens.

Each pixel always displays a single color. The pixels are approximately square, but they are slightly taller than they are wide. Pixel colors are controlled by the color set chosen with the current setting of the COLOR command and references to the color set from various PLOT, OUTPUT, and some POKE commands.

Screen Colors

We also know from our previous discussions, that eight colors are available for graphic display. Only four colors can be in use on the screen at any one time. You define the color set you want to use with the four positions on the COLOR command. See the Screen Control section of the previous chapter for a list of the available colors and more definition of the color set.

The statement

```
COLOR 0,7,1,4
```

defines a color set in which black is the background color, and red, white, and blue are the colors that can be used for display against the black background. This color set might be used for drawing the American Flag, for example.

The COLOR statement changes the color set and therefore the colors associated with any pattern or message that is on the screen at the time the COLOR statement is executed. While this can be used to heighten the visual impact of a display with blinking, flashing, or color rolls, it's not always the most visually effective. For that reason, you frequently clear the screen with the CLS statement before changing the color set.

To use colors effectively in a program, you should first make certain the color controls on your TV set are accurately set. You can use the Color Test on the 16K Diagnostic Tape to adjust the colors, or, better yet, use this simple BASIC program. It displays four bands of color on the screen: blue, red, green, yellow (from top to bottom). This program uses Microsoft 8K BASIC extended PLOT parameters to draw the color bands quickly. If you have only Level II BASIC, you can adapt this program by putting the statements in lines 40, 60, and 80 into FOR...NEXT loops to draw the color bands.

```
10 REM-TV COLOR ADJUST PROGRAM
20 CLS:COLOR 4,1,2,3
30 OUTPUT"BLUE",10,60,3
40 PLOT 1,35,1,112,18
50 OUTPUT"RED",10,45,3
60 PLOT 1,18,2,112,18
70 OUTPUT"GREEN",10,27,3
80 PLOT 1,1,3,112,18
90 OUTPUT"YELLOW",10,14,0
100 A$=INSTR$(1)
```

When the COLOR statement is executed, a potentially unfortunate side effect occurs--the tape motor is always turned off. While this won't affect most of your programming, it may be limiting when you wish to run a program with COLOR changes that also keeps the tape motor on to permit a music tape to be played through the TV speaker. Technically, this happens because a color register and a tape motor control bit share the same byte in memory. You can, however, control the color and tape motor simultaneously by using POKE commands to set the color registers with the desired color and tape motor control information.

Let's assume that we set the four COLOR parameters (CO, C1, C2, C3) as in the previous program and define a variable, M, that is set to a value of 1 if the tape motor is to be turned on, 0 if it is to be turned off. The following subroutine will set the colors and tape motor control as desired:

```
100 T1=64*M+8*C2+C0
110 T2=8*C3+C1
120 POKE 4096,T1
130 POKE 6144,T2
140 RETURN
```

This subroutine combines the variables in two arithmetic expressions and stores the results in two variables, T1 and T2. Those values are then stored in the two color register locations at 4096 and 6144 with the POKE statements.

Using this subroutine in a program with the tape motor turned on (M=1) is straightforward. If you execute the following program, you'll see the word HELLO continuously change color on the screen with the tape motor turned on. You can put in a music cassette during execution of this program and play music through the TV speaker.

```
5 POKE 19215,25
10 CLS:M=1
20 C0=0:C1=1:C2=3:C3=7
30 GOSUB 100
40 OUTPUT "HELLO", 40,50,1
50 FOR C1=1 TO 7
60 GOSUB 100
70 FOR Q=1 TO 500:NEXT
80 NEXT
90 GOTO 50
100 T1=64*M+8*C2+C0
110 T2=8*C3+C1
120 POKE 4096,T1
130 POKE 6144,T2
140 RETURN
```

Good Color Combinations

The choice of the colors you use in your program is, of course, strictly up to you. However, there are some combinations of colors which are much better than others. Some of our recommendations for color selection are as follows:

1) Black backgrounds (CO=0) are very effective and allow maximum flexibility in the choice of the foreground colors. In addition, black produces less "static noise" on some TV screens than other colors. It also reduces the visibility of "herringbone" or "clock lines" that may appear on the screen if your Interact needs some timing adjustments.

2) Avoid red lettering on blue backgrounds or vice-verse. Red and blue adjacent pixels tend to bleed into each other, making the result difficult to read. This is also true of red/green combinations.

3) Black or white lettering on any background color is effective.

When developing your graphics programs, experiment with different color combinations. To get a feel for the color combination effects with your TV and Interact, enter and RUN the following Microsoft 8K BASIC program.

```
10 CLS:COLOR 0,1,3,7
20 PLOT 20,20,1,90,30
30 PLOT 40,30,2,50,12
40 PLOT 1,62,3,112,13
50 OUTPUT"HELLO",50,68,1
60 OUTPUT"MICRO",50,52,2
70 OUTPUT"VIDEO",50,32,3
80 WINDOW 18
```

When you see the "OK" prompt, enter some different COLOR commands in direct mode to see the effect of different combinations of lettering on various colored backgrounds. Some examples you might try are:

```
COLOR 7,3,4,2
COLOR 5,6,0,7
COLOR 1,2,4,6
```

Learn to choose effective color combinations--that's an important part of using your Interact's graphics capabilities! A poor color combination can destroy the visual effect of even the most spectacular graphic.

Split Screen Techniques

The WINDOW command can be used to divide the screen into an upper and lower portion, as shown in the previous example. The dividing line can be moved up or down by changing the value supplied with the WINDOW command. All scrolling from PRINT commands in a program takes place only on the lower portion of the screen. Thus, two sequences of seemingly independent visual activity can take place on the same screen quite easily. With a split screen effect, the lower portion of the screen usually contains textual material, while the upper part is generally graphic. RUN the following example to see a "READY, AIM, FIRE" sequence that's quite effective.

```
10 CLS:COLOR 0,1,7,3
20 W=18
30 WINDOW W
40 PLOT 1,35,1,112,1
50 SOUND 3,522
60 PRINT"       READY"
70 FOR X=1 TO 50 STEP 2
80 IF X=27 THEN PRINT"        AIM"
90 OUTPUT"<>",X,40,2
100 OUTPUT"*",X+3,40,1:OUTPUT"<>",X,40,0
110 OUTPUT"*",X+3,40,0:NEXT
120 OUTPUT"*",X+3,40,1:OUTPUT"<>",X,40,2
130 SOUND 1,255
140 PRINT"       FIRE!"
150 FOR N=41 TO 77
160 PLOT 56,N,2:PLOT 56,N,0
170 NEXT
180 GOTO 10
```

This program illustrates several ideas in graphic display, but let's take a closer look at the concept of "windowing" a message in the lower portion of the screen. You can create a different effect in the program above by raising the window. Change line 20 to read

```
20 W = 30
```

and RUN the program again to see the difference in effect. For another variation, decrease the WINDOW setting by six pixels every time a PRINT statement is performed. This has the effect of graphically "adding-on" subpoints to the upper part of the screen display.

Want to use a two-color split screen display? Simply PLOT the upper part of the screen in color 1, 2, or 3. The bottom part of the screen remains in the background color defined by the color 0 position in the color set. Try this example:

```
10 CLS:COLOR 7.6.0,1
20 WINDOW 36
30 PLOT 1,37.1,112,40
40 OUTPUT"SPLIT SCREEN",22,60,2
50 J=J+1
60 PRINT TAB(7);J
70 GOTO 50
```

A horizontal stripe just above the window boundary adds even more graphic appeal to either the single or two-color split screen. Add this line to the example above to see the additional graphic effect.

```
35 PLOT 1,37,3,112,1
```

To use scrolling with windowing in effect, we recommend that the WINDOW value be an even multiple of 6 (e.g., WINDOW 18, WINDOW 36, etc.), so the tops of letters are not "chopped off" during the scrolling process. Restore full screen scrolling by typing a WINDOW 77 command or the RESET button for a RESET-R program restart.

For a more elaborate example of scrolling messages that includes a timepiece with rotating hour, minute, and second hands in a split-screen display, see the MY GRANDFATHER'S CLOCK program.

Simple Point Plotting

The PLOT command can be used to draw individual points and lines on the screen. Try each of the following examples in direct mode to see the results achieved. Set your computer for this exercise by entering the following set of commands:

```
WINDOW 24
CLS
COLOR 0,1,3,4
```

TO DRAW	ENTER THIS COMMAND
A single point	PLOT 50,50,2
Horizontal line	FOR X = 1 TO 112:PLOT X,60,1:NEXT
Dotted horizontal line	FOR X = 1 TO 112 STEP 2:PLOT X,55,2:NEXT
Vertical line	FOR Y = 40 TO 70:PLOT 10,Y,3:NEXT
Dotted vertical line	FOR Y = 30 TO 60 STEP 2:PLOT 15,Y,2:NEXT
45^{o} upward sloping line	FOR Y = 30 TO 60:PLOT Y,Y,1:NEXT
Double horizontal stripe	FOR X = 1 TO 112:PLOT X,64,1:PLOT X,65,2:NEXT
Horizontal line (right to left)	FOR X = 80 TO 20 STEP -1:PLOT X,65,2:NEXT
Filled box (slow speed)	FOR X=30 TO 80:FOR Y=50 TO 57:PLOT X,Y,2:NEXT:NEXT

More complicated figures can be drawn by applying geometric equations within the program. For example, the following program draws three concentric circles in three different colors, centered at (50,50) with radii of 25, 20, and 15 pixels. The variable T is incremented between 0 and 2π (approx. 6.28). T represents the angle in radians.

```
10 CLS:COLOR 7,1,2,4
20 FOR R=25 TO 15 STEP -5
30 C=C+1
40 FOR T=0 TO 6.28 STEP .05
50 PLOT 50+R*COS(T),50+R*SIN(T),C
60 Y1=10*SIN(X)
70 NEXT:NEXT
80 A$=INSTR$(1)
```

Multi-point Plotting

The PLOT command in Microsoft 8K Fast Graphics BASIC has been extended to allow two optional parameters for graphics up to 30 times faster than are available in Level II BASIC. These parameters allow you to specify the horizontal and vertical lengths of a plot, in addition to the origin and color:

```
COLOR x,y,c,xlen,ylen
```

The first three parameters of the PLOT command are the same as in Level II BASIC. The first two specify the (x,y) coordinate pair that is the origin of the plotted area. The third is the color from the color set, specified by position in the set (0-3). xlen specifies the horizontal length of the plot, and ylen specifies the vertical length. The (x,y) coordinates in the extended PLOT statement determine the placement of the lower left corner of the plotted area. Both xlen and ylen must be greater than 0, and x+xlen must be less than or equal to 113 and y+ylen must be less than or equal to 78. If either xlen or ylen has a value of 1, a line one pixel wide is drawn on the screen.

Thus, a horizontal or vertical line or a rectangle of any size, position, and color may be drawn with a single PLOT command. The FOR...NEXT loops used to draw lines or fill in areas in Level II BASIC can be eliminated from programs written with Microsoft 8K BASIC, so programming is simpler, and programs require less RAM for the same operations. An additional benefit is that the speed of the graphics can be up to 30 times faster than the older Level II plot methods, because BASIC uses the system ROM routine RFILL to perform the graphics display. While this ROM routine can be invoked through POKE statements and a USR call (see the Micro Video MONITOR and the Guide to ROM Subroutines), it's far more straightforward to use the extended PLOT command in 8K BASIC.

For example, to clear the screen in direct mode, use this PLOT statement:

```
PLOT 1,1,1,112,77
```

See for yourself how much faster the graphics are in 8K BASIC by entering the following commands:

TO DRAW	ENTER THIS COMMAND
Horizontal line	PLOT 1,60,1,112,1
Vertical line	PLOT 40,50,2,1,20
Wider horizontal line	PLOT 1,60,3,112,3
Rectangle	PLOT 20,20,2,80,50

Game Grids

A game grid is nothing more than a set of equally-spaced horizontal and vertical lines. These can easily and quickly be established on the screen with Microsoft 8K BASIC as illustrated in the following simple program.

```
10 REM-GAME GRID
20 CLS:COLOR 0,7,1,3
30 FOR X=10 TO 70 STEP 10
40 PLOT X,10,1,1,60:NEXT
50 FOR Y=10 TO 70 STEP 10
60 PLOT 10,Y,1,61,1:NEXT
70 A$=INSTR$(1)
```

This grid could also be developed in Level II BASIC using FOR...NEXT statements, but the display would be markedly slower.

Checker Board

A checker board is a set of alternating color squares. In the next example, a single PLOT statement fills each square in the appropriate color. The subroutine in line 40 alternates the color variable, C, between values of 1 and 2 to change the colors in the squares. The second GOSUB 140 in line 110 causes a skip of the color sequence each time a new column of squares is started, so that the square colors alternate horizontally as well as vertically.

```
10 REM:CHECKER BOARD
20 CLS:COLOR 0,1,7,3
30 GOSUB 60
40 A$=INSTR$(1)
50 REM-CHECKER BOARD ROUTINE
60 FOR X=20 TO 82 STEP 8
70 FOR Y=6 TO 68 STEP 8
80 GOSUB 140
90 PLOT X,Y,C,8,8
100 NEXT
110 GOSUB 140
120 NEXT
130 RETURN
140 C=C+1:IF C=3 THEN C=1
150 RETURN
```

Graphic Design

You might want to add graphic images to your programs that are appropriate to a particular season or occasion. An American flag would fit well into an Independence Day display. You might want to use your Interact as a "birth-day card" display at a birthday party and develop images of a birthday cake and flickering candles. Or, at a Christmas party, you might want to include the following code in a display program. This small program draws a Christmas tree on the screen by defining a set of horizontal rectangles that narrow as they approach the top of the tree (using Microsoft 8K BASIC).

```
10 REM-CHRISTMAS TREE
20 CLS:COLOR 0.2.1,3
30 X=26:XL=60
40 FOR Y=12 TO 65 STEP 2
50 PLOT X,Y,1,XL,2
60 X=X+1
70 XL=XL-2
80 NEXT
90 FOR Q=1 TO 1000:NEXT
100 GOTO 10
```

As an exercise, add some PLOT or OUTPUT statements to draw a trunk, decorate the tree, and place packages beneath it.

Note that you could draw the Christmas tree above in Level II BASIC. However, the logic to do so would be more complex because a FOR...NEXT loop is required to draw each rectangle, rather than a single PLOT statement. The graphic development on the screen would also be significantly slower, as you'll see if you enter and RUN the version of this program below.

```
5 CLS:COLOR 0,1.2,3
10 A = 10:B = 15
20 C = 30:D = 80
30 FOR L = 1 TO 9
40 FOR Y = A TO B
50 FOR X = C TO D
60 PLOT X,Y,2
70 NEXT X:NEXT Y
80 A= A+6: B=B+6
90 C = C+3:D = D-3
100 NEXT L
```

See the differences in speed and visual effect?

Extra-Large Lettering

If you think the character size on your Interact is too small, here's a way to increase the size of the letters! Seriously, extra-large lettering can be a useful mechanism to attract viewer attention to your display program. If you enter and RUN the following short program, your Interact will give you a bold greeting.

```
10 REM-LARGE LETTERING
20 CLS:COLOR 0,2,1,3
30 X=20:Y=20
40 PLOT X,Y,2,8,50
50 PLOT X,Y+22,2,30,8
60 PLOT X+30,Y,2,8,50
70 PLOT X+55.Y,2,8,50
80 FOR Q=1 TO 1000:NEXT
```

You'll find it useful to set all (x,y) coordinates as variables rather than constants, so that the complete word can be properly positioned on the screen with minimal retyping effort. Obviously, only short words will fit on the screen if you use large lettering. However, you can size the lettering as required to fit in many cases.

"Three-dimensional" lettering can also be effective in combination with large letters. We can easily adapt our large lettering program to illustrate this concept. For a three-dimensional effect on any graphic image, draw the image three times, in three different colors. Plot the second copy of the image one pixel up and one pixel to the right of the original image. Then, plot the third one up and one over from the second. In our program, we'll accomplish this by putting the PLOT statements that produce the lettering in a FOR...NEXT loop that will draw the image three times, each time in a different color from the color set (C). At the end of each loop, we'll increment the values of X and Y by one.

```
20 CLS:COLOR 0,7,4,1
30 X = 20:Y=20
35 FOR C = 1 TO 3
40 PLOT X,Y,C,8,50
50 PLOT X,Y+22,C,30,8
60 PLOT X+30,Y,C,8,50
70 PLOT X+55,Y,C,8,50
75 X = X+1:Y = Y+1
77 NEXTC
80 FOR Q = 1 TO 1000:NEXT
90 CLS:GOTO 30
```

Bar Graphs

A bar graph is the most effective way of displaying trends in a series of data points with the Interact. With Microsoft 8K BASIC, each bar can be produced by a single PLOT command with appropriate scaling and axis positioning. Excellent examples of bar plotting can be found in the BASIC Examples Booklet and in the Trends program in Financial Library II. Color changes in the bars can be used to signify negative values or to differentiate actual from forecasted data.

Character Patterns

The OUTPUT statement lets you display any character, string, or value at any position on the screen, in one of the colors specified in the color set. The (x,y) position specified is the position of the pixel in the upper left corner of the character. Your computer displays each character as a 5x5 pixel grid within a 6x6 pixel field. Since characters can be positioned so as to overlap their neighbors with OUTPUT statements, interesting displays can be created by choosing appropriate colors, positioning, and characters. For example, try the following program. It uses the OUTPUT statement to overlap the characters "C" and ";" to produce a colorful pattern on your TV screen.

```
10 REM-CHARACTER PATTERNS
30 CLS:COLOR 7,1,3,2
35 FOR X=1 TO 112 STEP 5
40 FOR Y=6 TO 77 STEP 5
50 OUTPUT"C",X,Y,3*RND(1)+1
55 OUTPUT";",X+1,Y+1,3*RND(1)+1
60 NEXT:NEXT
70 GOTO 35
```

Try experimenting with the effects you can achieve by varying the characters used in the OUTPUT statement and the colors in the color set.

Interact Slang -- Non-Standard Characters

Your Interact has a number of non-standard or "slang" characters in its character table. Your Interact understands these characters in that it knows that they exist, but you can do little with then except use them for display in your programs. These patterns are different depending on which BASIC you have, but all the BASICs have them, and you can turn them into controllable entities within game programs. You access these characters by indexing beyond the limits of the standard ASCII character table your Interact uses for communication. The following sample program will let you look at the non-standard characters. You might want to keep a piece of paper handy and make note of your favorites as the program runs, for use in future programs.

```
10 REM  NON-ASCII SPECIAL CHARACTERS
20 CLS:COLOR 4,0,7,3
25 WINDOW 18
30 PRINT CHR$(8)
35 FOR C=1 TO 255
36 PRINT"CODE=";C
40 FOR X=10 TO 100 STEP 2
50 OUTPUT CHR$(C),X,50,2
60 OUTPUT CHR$(C),X,50,0
70 NEXT:NEXT
80 WINDOW 77
```

5x5 Pixel Blocks

If you do not have Microsoft 8K BASIC, you can use the CHR$(1) function to color-fill an area with a 5x5 pixel block, instead of using the PLOT statement. You'll find that it provides a faster graphics capability, although not nearly as fast as is possible with 8K BASIC. You can OUTPUT CHR$(1) within a FOR...NEXT loop to draw 5 pixel wide lines for developing images.

As this character is outside the normal ASCII character set, we first have to ensure that the pointer to access the character table is properly positioned. We do this by printing the backspace character (CHR$(8)), as shown in line 12 of the following example. This example shows how the OUTPUT CHR$(1) can be used to draw the set of alternating red and white stripes in the American flag.

```
10 REM-5 PIXEL STRIPES
12 PRINTCHR$(8)
20 CLS:COLOR 0,1,7,1
25 FOR Y=10 TO 70 STEP 5
30 C=C+1
40 IF C=3 THEN C=1
50 FOR X=5 TO 110 STEP 5
60 OUTPUT CHR$(1),X,Y,C
70 NEXT:NEXT
100 RUN
```

Of course, the extended PLOT statement in Microsoft 8K BASIC is still much faster and offers more flexibility than this method, but CHR$(1) offers significant improvement over single point plotting for those still using the old Level II BASIC interpreter.

Creative Motion -- Stick Figures

Stick figures can be made to move around on the screen to provide an entertaining visual in your program. You can make these figures act independently or in groups. Stick figures are created by drawing a set of characters with OUTPUT statements at a specified location on the screen from a subroutine in the program. After a stick figure is output, it is typically redrawn in the background color to "erase" it before returning to the calling program. The program then issues another call to the subroutine to output the set of characters again. This makes the stick figure appear to "walk" across the screen. In the following example, the stick figure is created by combining the set of OUTPUT characters in lines 110 through 140.

```
               10 REM-STICK FIGURES
  Draws line   20 CLS:COLOR 6,0,2,7
               25 Y=50
               28 FOR X=1TO112:PLOT X,41,2:NEXT
               30 FOR X=10 TO 100 STEP 2
               40 GOSUB 100
               50 NEXT
               60 GOTO 20
Erases figure  99 REM-DRAW AND ERASE FIGURE AT (X,Y)
               100 FOR C=1 TO 0 STEP -1
               110 OUTPUT"0",X,Y+5,C
  Draws figure 120 OUTPUT">",X+1,Y,C
               125 OUTPUT"<",X-1,Y,C
               130 OUTPUT"/",X-2,Y-4,C
               140 OUTPUT CHR$(92),X+2,Y-4,C
               200 NEXT C
               300 RETURN
```

Word Smears

You can achieve the visual effect of "spraying" a word on the screen to attract attention in your programs. The following example shows how this can be done. You can control the direction and amount of the smearing by the parameters in the FOR...NEXT loops. You can also add tones to the visual to increase its impact.

```
10 REM-WORD SMEARS
20 CLS:COLOR 0,1,7,3
30 FOR Y=10 TO 50 STEP 2
35 TONE 100-Y,10
40 OUTPUT"WHEE",50,Y,1
50 NEXT
60 OUTPUT"WHEE",50,Y-1,2
70 FOR Q=1 TO 500:NEXT
100 RUN
```

3-D Lettering

We've already considered the concept of three-dimensional lettering with extra-large characters. You can also apply this idea to normal-sized lettering by repeated OUTPUT statements in different colors and increased horizontal and vertical displacement of the message. The choice of colors in three-dimensional lettering is important. The last color used to display the message should have the greatest contrast with the background color for the best effect.

```
10 REM- 3D LETTERING
20 CLS:COLOR 0,4,6,7
25 S$="BON JOUR"
30 OUTPUT S$,34,50,1
40 OUTPUT S$,35,51,2
50 OUTPUT S$,36,52,3
60 GOSUB 100
70 GOTO 20
100 FOR Q=1 TO 300:NEXT
110 RETURN
```

Graphic Color Control

You can achieve a number of different graphic effects just by manipulating the colors in the current color set, such as blinking, flashing, color rolls, shimmering, and instantaneous writing. We'll take a look at these visual effects to show you how they can increase the drama and impact of your program display.

Blinking Objects

You can make a word or object "blink" on and off by changing the color in which it is displayed to the background color. You can use timing loops to control the speed of the blinking--the relative "visible" and "invisible" portions of the blinking cycle.

```
10 REM-BLINKING
20 CLS:COLOR 3,7,0,1
30 OUTPUT"SAVE",51,60,1
40 FOR X=51 TO 73:PLOT X,54,1:NEXT
50 GOSUB 100
60 COLOR 3,3,0,1
70 GOSUB 100
80 COLOR 3,7,0,1
90 GOTO 50
100 FOR Q=1 TO 100:NEXT:RETURN
```

To make the word blink faster or slower in the preceding program, raise or lower the highest value of Q in the FOR...NEXT pause loop (line 100).

Flashing the Screen

You can produce the effect of a screen flash by switching the values in the color set of the background color and the color in which an object or word is displayed. Very dramatic effects can be achieved with high contrast colors such as black and white and large objects such as oversized letters. You control the speed and duration of the flashing by the corresponding FOR...NEXT pause loops.

```
10 REM-FLASHING
20 CLS:COLOR 0,7,1,3
25 OUTPUT"TODAY",46,30,2
30 OUTPUT"SAVE",51,60,1
40 FOR X=51 TO 73:PLOT X,54,1:NEXT
50 GOSUB 100
60 COLOR 7,0,1,3
70 GOSUB 100
80 COLOR 0,7,1,3
90 GOTO 50
100 FOR Q=1 TO 50:NEXT:RETURN
```

Color Rolls

You can produce a color roll in your programs by rotating the colors associated with a displayed object or message through the available eight colors. We've done a color roll in the following program by changing the color in the second position of the color set (color 1) within a FOR...NEXT loop. Each time through the loop, the value of the variable, T, is incremented by one to change the color setting of color postion 1.

```
10 REM-COLOR ROLLS
20 CLS:COLOR 0,7,1,4
30 FOR X=42 TO 82
32 FOR Y=38 TO 40
33 PLOT X,Y,1:NEXT:NEXT
40 OUTPUT"ROLL ON",42,48,2
50 FOR T=0 TO 7
60 COLOR 0,T,1,4
70 FOR Q=1 TO 100*T+50:NEXT
80 NEXT
90 GOTO 20
```

Shimmering

In the example on page 3-10, we showed you how to create three-dimensional, oversized letters within your programs. You can add another effect to make the large letter display even more powerful--making the word HI shimmer. In the following program, we've created this shimmering effect by inserting a FOR...NEXT loop between lines 80 and 90 that rapidly switches the colors in which the background dimensions are displayed. Type in this program and run it to see the amazing effect--the main HI lettering also appears to vibrate!

```
10 REM SHIMMERING
20 CLS:COLOR 0,7,4,1
30 X = 20:Y = 20
35 FOR C = 1 TO 3
40 PLOT X,Y,C,8,50          ← HI
50 PLOT X,Y+22,C,30,8
60 PLOT X+30,Y,C,8,50
70 PLOT X+55,Y,C,8,50
75 X = X+1:Y = Y+1
77 NEXT C
80 FOR Q = 1 TO 1000:NEXT   Pause
85 FOR I = 1 TO 25
87 COLOR 0,4.7.1            Changes color
88 COLOR 0,7,4,1            25 times
89 NEXT I
90 CLS:GOTO 30
```

Instantaneous Writing

We'll discuss one other effect that can be obtained through manipulating the color set. We call it "instantaneous writing," because the words seem to pop onto the screen from the background color. You can use this type of effect for displaying messages in an interesting fashion or for making objects suddenly appear on the screen.

You achieve the effect of instantaneous writing by initially setting all colors in the color set to the background color, then drawing the visuals referencing the three foreground colors (colors 1, 2, and 3 in the color set). When the code establishing the visuals is done, just change the COLOR statement to place the parts of the image or message in the desired colors. The image will "pop" onto the screen instantly. Keep in mind when using this technique that the more complex the image is to develop, the longer the time the screen will remain the background color.

```
10 REM-INSTANTANEOUS DISPLAY
20 CLS:COLOR 0,0,0,0
30 OUTPUT"GOOD MORNING,",20,60,1
40 OUTPUT"MR. PHELPS...",20,54,1
50 OUTPUT"YOUR MISSION,",20,40,2
60 OUTPUT"SHOULD YOU",20,34,2
70 OUTPUT"DECIDE TO",20,28,2
80 OUTPUT"ACCEPT IT...",20,22,2
90 COLOR 0,3,7,1
100 FOR Q=1 TO 5000:NEXT
110 GOTO 10
```

Advanced Graphics -- POKEing the Screen

The graphic display on the Interact is memory-mapped beginning at RAM location 16384 and occupies the next 2,463 bytes of contiguous RAM (ending at location to 18943). Each full line on the screen actually contains 128 pixels, although only 112 of them are displayed on the TV screen. Each byte addresses four pixels, so each full line on the screen consumes 32 bytes of memory (128/4). (The last 16 pixels (4 bytes) of each line are "invisible.") The leftmost four pixels in the top row on the screen are controlled by the value of memory location 16384. Therefore, the pixels on the screen are controlled by the memory locations as follows:

415
−384
32 bytes

16384 - 16415	First row of 128 (112 visible) pixels (32 bytes)
16416 - 16447	Second row of pixels
16448 - 16479	Third row of pixels
.	
.	
.	
18912 - 18943	Seventy-seventh row of pixels

If you are using Level II BASIC, before you can draw on the screen with the POKE statement, you must enable the POKE command with the statement

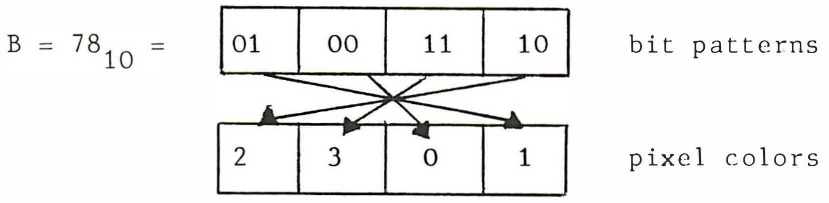
```
POKE 19215,25
```

If you are using Microsoft 8K BASIC, you do not have to enter this statement because the POKE statement is automatically enabled when BASIC loads.

To use the POKE command to control colors on the screen, you specify two values on the statement, separated by a comma. The first value, which we'll call A, defines the memory location that controls the group of four pixels you want to address. The second value, which we'll call B, is a data value from 0 to 255, which is interpreted as a bit pattern to determine the color of each pixel in the group. These four, two-bit bit patterns call the colors in the color set as follows:

00	– color 0 (background color)
01	– color 1
10	– color 2
11	– color 3

The pixels controlled by the two-bit bit patterns in each byte are addressed in reverse order:



In other words, the first two-bit bit pattern in the byte determines the color of the fourth pixel controlled by that byte. The second bit pattern determines the color of the third pixel, the third bit pattern the color of the second pixel, and the last bit pattern the color of the first pixel.

To display a string of four pixels in a single color, use the data values from the following table:

B=	Color
0	0
85	1
170	2
255	3

To display pixels of different colors within a single byte, you will have to convert the binary representation of the colors into a data value.

Enter the following commands in direct mode:

```
COLOR 0,1,2,3
```

```
POKE 17580,170
```

This command outputs a string of four green pixels in the center of the screen. Now type

```
POKE 17581,236
```

and you'll see another string of four pixels, alternating yellow and red, on the right end of the green bar from the previous POKE command.

The following simple program illustrates how POKE commands can control the screen display from within a program. By poking the memory locations of the screen with various values, you can display different color patterns. Our program below produces colorful vertical striping across the entire screen. You can POKE other values to the screen to display variable patterns. While this technique is not particularly useful for animation, it can provide an interesting graphic display.

```
10 REM-SCREEN POKES
20 CLS:COLOR 0,7,3,1
30 POKE 19215,25
40 FOR L=16384 TO 18624 STEP 32
50 FOR X=1 TO 32
60 POKE L+X,X
70 NEXT:NEXT
80 A$=INSTR$(1)
```

A couple of final notes about POKE and screen control:

1) When CLS is used to clear the screen, all bits are reset to the background color.

2) Remember to save any program that contains POKE statements on tape before running the program, as it is perfectly possible to POKE your program right out of existence and lose all your work. To avoid this frustrating experience, make a habit of CSAVEing your programs during program development.

For further information and examples on fast graphics, refer to the image development of the moving airplane and other visuals employed in the BOMBS AWAY! Programming Tutorial. The program combines BASIC and machine language programming and is suitable for study and exploration of yet another facet of graphics development.

The Vector Graphics Subroutines package provides the BASIC user with callable subroutines for very fast vector and triangle plotting. The package is useful for programs that use spokes, windshield wiper wipes, triangles, large, filled circles, rotating clock hands (as in the "MY GRANDFATHER'S CLOCK" BASIC program). Consult the Micro Video Product Catalog for information on pricing and how to order these items.

Graphic Guidelines

For effective visual presentations, we recommend the following guidelines:

1) Choose color combinations carefully. They should be attractive, readable, and related to the program's graphics.

2) Avoid tedious visual development. Use routines or techniques which captivate the viewers' imaginations.

3) Add surprise to your graphic image development wherever possible-- by unexpected endings, tones and sounds, or varying speeds.

The January 1981 issue of Creative Computing features an article on the "CROWD STOPPER" which contains many more suggestions for improving graphic screen development.

STRUNG OUT

(String Handling)

In BASIC terms, a string is a list of alphanumeric characters. JOHN JONES, YOU WIN, AND WOW! are all strings. You can use strings in BASIC as either string constants or string variables. However, you must have a way to tell BASIC that you are working with string information rather than numeric values. You do this by adding a dollar sign ($) at the end of a variable name or by enclosing a string constant in quotation marks.

STRING VARIABLES	STRING CONSTANTS
S$	"YOU WIN"
NM$(4)	"NEW YORK, NY"
M$(3,I)	"$324.45"

A string can vary in length from zero characters (we call these "null" strings) to a maximum of 255 characters. A string may contain alphabetic and numeric characters, punctuation marks, special characters, and blanks.

When you RUN a program, all strings are initialized as null strings.

As strings contain a different type of information from numeric variables, there is a different set of functions and operators for working with them. The LEFT$, MID$, and RIGHT$ functions isolate a specified number of characters from the left, middle, and right of a designated string and place the result in a new string. There are no analogous functions for working with numeric variables.

You'll notice that functions which return a string value always have a "$" at the end of their names--INSTR$, STR$, INSTR$, and CHR$. Each of the string handling functions is documented fully, along with examples of their usage, in the Reference Section.

The "+" operator also takes on a different meaning when you use it to work with strings. It becomes a concatenation operator that says: "Tack the second string onto the end of the first string." Let's examine how a few of these string functions and operators work. For the time being, we'll work in direct mode so that we can see the returned results quickly.

We'll start by initializing two strings with the following assignment statements:

```
F$ = "MADAME"
L$ = "BOVARY"
```

Now, let's verify that the strings were stored correctly by displaying them.

```
PRINT F$
MADAME
PRINT L$
BOVARY
```

Next, we'll look at how we can manipulate these strings with the LEFT$, RIGHT$, and MID$ functions.

```
? LEFT$(F$,3)
MAD

? RIGHT$(F$,4)
DAME

? MID$(F$,2,3)
ADA
```

We can use the "+" operator to concatenate the two strings:

```
NM$ = F$ + L$

? NM$
MADAMEBOVARY
```

You see that the concatenation is complete--there are no blank spaces between the strings. We can insert a separating blank into the concatenated full name by adding a blank string to the concatenation operation.

```
NM$ + F$ + " " + L$

? NM$
MADAME BOVARY
```

One attribute of a string is its length. Length is a numeric value that is equal to the number of characters contained in the string. The LEN function returns that value.

```
? LEN(NM$)
13
```

If LEN returns a value of zero, it means the string is null, or empty. If you want to set a string to empty during the course of a program, simply assign the string variable to a null string, delimited by two immediately adjacent quotation marks.

```
NM$ = ""
?LEN(NM$)
0
```

A primary use of the LEN function is in the positioning of string information on the screen. To center information on the screen, for example, you must know the length of the string so that you can compute the value of the X variable in the OUTPUT statement. Since the width of the screen is 112 pixels, the mid-point on the horizontal axis is at an X-value of 56. Characters are six pixels wide, so you have to backspace three pixels from the center line for each letter in the string to be displayed. If you want to center the string stored in F$, then type the following statement to compute the value of X:

```
X = 56-3*LEN(F$)
```

To continue with our direct mode example, let's output the two strings "MADAME" and "BOVARY" on two centered lines.

```
WINDOW 36
CLS
OUTPUT F$,X,66,1
OUTPUT L$,56-3*LEN(L$),60,1
WINDOW 77
```

Note that we can use the variable X in the command to display F$ ("MADAME"), because we defined X for F$ previously in direct mode. We must compute the X-coordinate for L$, however, because the string may be a different length. (In this case they happen to be the same.)

Data Mode Conversions

In Interact BASIC, a variable exists in one of two modes--numeric or string. While other languages may have additional modes (integers, double precision, etc.), we need to be concerned with only these two modes.

You may occasionally have the need to convert data from one mode to the other for processing. The VAL and STR$ functions facilitate this conversion. The VAL function takes numeric data that is stored in a string and converts it into a numeric variable that may be manipulated in the same way as any other numeric variable. For example, let's say we have a program that normally accepts numeric information in response to INPUT statements in the program. We may, however, want to be able to enter non-numeric information such as HELP or END to communicate with and control the program flow in a straightforward manner. If an INPUT statement requests numeric data and you type HELP, the error message "?REDO FROM START" is displayed and the INPUT prompt repeated. This can baffle the first-time user. That error message is built into BASIC--there's no way we can suppress its printing.

An alternate method of handling this problem would be to input all the data in the program as string data, then check each string to see if it begins with one of the acceptable keywords. If it did, then the requested task would be performed, such as printing a helpful message. If it did not, the program would assume that the string is really numeric data and use the VAL function to place the "string" data value into a numeric variable. The sample program on the following page illustrates the concept of converting data modes. The program accepts numeric student numbers or the keyword "END".

```
300 CLS:COLOR 0,1,7,3
310 CLEAR (300)
320 OUTPUT"ENTER STUDENT",10,66,1
330 OUTPUT"NUMBER OR TYPE",10,60,1
340 OUTPUT"'END' ",10,54,1
350 WINDOW 48
360 INPUT P$
370 IF P$="END" GOTO 1000
380 P=VAL(P$)
399 REM-CODE GOES HERE TO WORK WITH
400 REM-STUDENT,  P.
410 REM
412 PRINT"GOOD STUDENT !"
420 GOTO 300
999 REM-WRAPUP PROGRAM
1000 CLS
1500 PRINT"DONE"
1510 END
```

The reverse process uses the STR$ function to make a string value out of any given number. For example,

B$ = STR$(N)

takes the numeric value of N and stores it in string format in B$. This can be useful in formatting columnar data in printed reports. See the RS232 Loan Evaluator program to examine an application that uses this concept extensively.

String Input from the Keyboard

There are three ways in which you can accept string data from the keyboard in programs:

INPUT Statement	INPUT B$
INSTR$(n)	C = INSTR$(1)
Keyboard Peeks	IF PEEK(24529) = "N" THEN ...

Let's examine each of these approaches in more detail.

The INPUT statement is by far the most common approach. INPUT is used in several examples throughout this manual. You can type a string any length in response to the INPUT "?" prompt. With INPUT data entry, all other processing stops until the input operation is complete. You must press the "CR" key to enter the data before the next statement in the program can be executed. In addition, if the information being entered contains embedded commas that are to be considered part of the string as opposed to separators between strings, then such strings must be enclosed in quotation marks when they are entered.

```
?"CHICAGO, ILL"
```

is accepted as a single string in response to the input query, while

```
?"CHICAGO","IL"

       or

?CHICAGO,IL
```

is accepted as two separate strings which will be read into two string variables. As a number of typing errors can easily be made when attempting to place quotation marks around string input, we suggest that your programming techniques work toward reducing or eliminating entirely the cases in which quotes are required.

The INSTR$(n) function is the most useful in accepting short strings of information from the keyboard. With INSTR$, you are freed from having to press "CR" to enter the string. The most common use of the *INSTR$ function* is as a "pause control". The statement

```
A$ = INSTR$(1)
```

halts program execution until you strike a key, any key, on the keyboard. The value stored in A$ is not important, although it can be used for display, conditional testing, or other purposes in the program if desired.

You can also use the INSTR$ construct to ask for YES/NO (Y or N) input from the keyboard or in menu selection. In this case, the value of the string variable is tested to determine what operation the computer will subsequently perform. The following program illustrates using INSTR$(1) to select a difficulty level in a game by typing the first character of the level's description. The character typed determines the value of the numeric variable, D, for program difficulty. Although it is not illustrated here, D would then presumably be referenced later in the program code during game play.

```
20 GOSUB 100
22 PRINTD
99 END
100 CLS:COLOR 7,1,2,4
110 OUTPUT"LEVEL OF PLAY ?",15,66,1
120 OUTPUT"E = EASY",27,54,2
130 OUTPUT"I = INTERMED.",27,48,2
140 OUTPUT"D = DIFFICULT",27,42,2
150 OUTPUT"S = SUICIDE",27,36,2
160 A$=INSTR$(1)
170 IF A$="E" THEN D=1:RETURN
180 IF A$="I" THEN D=2:RETURN
190 IF A$="D" THEN D=3:RETURN
200 IF A$="S" THEN D=4:RETURN
210 GOTO 160
```

Peeking the Keyboard

The last method we'll consider for accepting character data from the keyboard is the "keyboard peek". This technique employs the **PEEK** function to examine a specific memory location which is known to contain the ASCII value of the last character depressed on the keyboard. Because other audio and visual effects aren't inhibited by the wait for input, as they are with both INPUT and INSTR$, keyboard peeking can provide a method for keeping the screen active while awaiting input.

The following program illustrates a game that does not start until the space bar is depressed. You might use such a technique for unattended game operation in which you want the program to "idle" if no one is playing the game.

```
5 CLS:COLOR 0,7,1,3
10 POKE 19215,25
20 GOSUB 100
22 CLS:COLOR 0,7,1,3
25 OUTPUT"SHORT GAME",25,60,1
30 OUTPUT"HUH ?",42,54,1
40 FOR Q=1 TO 1000:NEXT
45 CLS
50 GOTO 20
100 POKE 24529,0
110 OUTPUT"PLAY WITH ME !",18,60,1
115 OUTPUT"HIT SPACE-BAR",15,18,3
116 OUTPUT"TO START",32,12,3
120 IF PEEK(24529)=32 THEN RETURN
130 COLOR 0,1,3,7
140 FOR Q=1 TO 30:NEXT
150 COLOR 1,0,3,7
160 FOR Q=1 TO 30:NEXT
170 GOTO 120
```

The memory location where the ASCII code for the last character entered is stored is 24529. To use this in a program, you must first store a zero in this location to destroy all record of past characters. You can then establish a program loop that looks for a value of 32 in that location. If it finds a value of 32 there, then program execution restarts. 32 is the ASCII value of the space bar, as can be seen in the ASCII character table included with the CHR$ entry in the Reference Section.

Storing Strings on Tape

The CSAVE and CLOAD commands let you write and read dimensioned numeric arrays to cassette tape. For example, the statement

```
CSAVE*Z
```

copies all of the data in the numeric array named Z to cassette tape, providing that a tape has been mounted in the tape deck and that the READ and WRITE cassette buttons have been depressed.

How do you copy the contents of a string or string array to tape?

```
CSAVE*A$
```

would seem the natural approach, but this won't work. Instead, you must convert the string characters into their ASCII equivalent codes and store them in a numeric array. Then, if you save that numeric array, you are essentially saving the string data to tape.

When you want to read the data back in, you'll CLOAD the data back into a numeric array, then convert the numeric data back to its string form with the CHR$ function.

Two subroutines that can be included in your programs for data conversion are listed below. Note that the length of the string is written to tape in the first word of the array. If you do a GOSUB 300 within a program containing these subroutines, you can write the string S$ to tape. To read the data back into the string S$, perform a GOSUB 400. The numeric array, B, (for Buffer) is an array for working storage which must be dimensioned sufficiently long in the calling program to contain the longest string. For example, you might begin this program with the statements

```
10 DIM B(50)
20 S$ = "MICRO VIDEO"
30 GOSUB 300
```

If you want to run this program to test the subroutines, be sure to position the tape correctly for the reading and writing operations. In an actual program, you would want to include tape positioning instructions and the REWIND command in your program logic to facilitate the process.

```
299 REM-TO TAPE SUBROUTINE
300 B(1)=LEN(S$)+1
310 IF B(1)=1 THEN CSAVE*B:RETURN
320 FOR J=2 TO B(1)
330 L$=MID$(S$,J-1,J)
340 B(J)=ASC(L$)
350 NEXT
360 CSAVE*B:RETURN
380 REM
399 REM-FROM TAPE SUBROUTINE
400 S$="":CLOAD*B
410 IF B(1)=1 THEN RETURN
420 FOR J=2 TO B(1)
430 S$=S$+CHR$(B(J))
440 NEXT:RETURN
```

A slightly more sophisticated approach can be used in which three characters can be "packed" into each position of the B array. Consult the DATALOG program listing for more information on how to use this storage technique.

INTERACT GAMESMANSHIP

(Controller Input)

You say you want to program an action game? You want it to be a one-person game using the entertainment controller? You want some visual excitement, sound effects, and a little strategy required to get a high score? *Have* we got a game for you! In this chapter, we'll see how we can take an idea, develop it a portion at a time, and end up with an entertaining action game in BASIC. Our game will illustrate the use of the FIRE and JOY functions to control the action in the game.

Our idea for this game is a variation of the popular EARTH OUTPOST I machine language game, adapted to the BASIC environment. In the program, we'll have our own Earth Station that will be able to move laterally across the screen under joystick control. When the program is completed, we'll be able to shoot down fixed targets in a black sky and score points based on how many targets we hit. We don't want the game to run endlessly, so we'll use an internal clock to control the length of the game in a timing loop. The object of the game will be to determine the order in which the targets should be attacked to get the highest possible score within the game period. We'll do all this in only one page of BASIC code!

We'll start, of course, by typing the NEW command in direct mode to prepare the computer for entry of a new program. Our first program line will be a REM line to identify the program for future generations of admirers. Then, we'll clear the screen and select the color set--a black background with red, white, and yellow foreground colors. We'll draw the ground area and a line across the top of the screen to define the playing area *in red*, using two single Microsoft 8K BASIC PLOT statements. That's enough to start with...let's look at how we'd put this into our initial program lines.

```
10 REM*** MICRO ARCADE ***
20 CLS:COLOR 0,1,3,7
30 PLOT 1,1,1,112,10
40 PLOT 1.70,1,112,1
50 A$=INSTR$(1)
```

Note that we've used the INSTR$ function in line 50 to hold the screen image after it's drawn, so the BASIC "OK" prompt and scrolling won't appear immediately. We'll take this line out, of course, as we proceed with development of the program.

Next, we'll add the character to represent our Earth Station and develop a subroutine to move it back and forth across the screen without exceeding the screen limits in either direction. In looking through the non-standard characters, as described in the Graphics chapter (3-11), we find that a CHR$(6) seems like a reasonable character to use for the Earth Station, so we'll store it in a string variable named GUN$. We'll choose a random starting (X) position for the Station, then OUTPUT the Station at that random point. (These steps are done on lines 50-60 of the listing on the following page.)

To establish the duration of the game, we'll create a loop that checks repeatedly for movement of the joystick lever via a subroutine. If the joystick lever is moved to the left or right, we'll pass program control to a subroutine that moves the Station to the left or right in increments of 2 pixels. At the end of this duration, or timing, loop, we'll terminate the program with an END statement.

```
10 REM*** MICRO ARCADE ***
20 CLS:COLOR 0,1,3,7
30 PLOT 1,1,1,112,10
40 PLOT 1,70,1,112,1
50 X=RND(1)*100+5
52 GUN$=CHR$(6)
60 OUTPUT GUN$,X,16,2
70 FOR T=1 TO 500
100 GOSUB 300
280 NEXT
295 END
300 ON JOY(0) GOTO 360,380
310 RETURN
360 IF X<6 THEN RETURN
362 X=X-2
364 OUTPUT GUN$,X+2,16,0
365 GOTO 385
380 IF X>112 THEN RETURN
382 X=X+2
384 OUTPUT GUN$,X-2,16,0
385 OUTPUT GUN$,X,16,2
390 RETURN
```

The subroutine that starts at line 300 checks the joystick position on the left controller via the JOY(0) function call. Since our Station can move only laterally, we're only interested in two values (Left=1, Right=2). We can therefore use the ON...GOTO construction to transfer program control to line 360 or 380 if a value of 1 or 2 is returned, respectively. All other values of JOY(0) simply pass control to the next statement, a RETURN statement that returns control to the main program duration loop.

Before we increment X (line 382) or decrement it (line 362), we must first check to see that we have not exceeded the limits of the screen. If we have, we RETURN to the next iteration of the main program loop. If we can move, we'll modify the value of X, erase the old Station by redrawing it in the background color in either line 364 or 384. Finally, we OUTPUT the new position of the Station at position X in line 385, then return to the next iteration of the main program loop.

We should check this program at this stage of development to be sure it's operating correctly before the "plot thickens" (if you'll pardon the pun). Oops! We forgot something. In order to display the character in CHR$(6), we have to set BASIC's pointer to the character table by using the instruction PRINT CHR$(8) (the backspace character). We'll do this in line 51.

Now that we've developed our basic screen image and output the Station, we need to place some random "targets" in the nighttime sky. We'll do this in a subroutine that indexes in the X-direction two pixels at a time and randomly plots single pixels in the sky (lines 700-760). Fifty percent of the time we'll plot a point somewhere along the Y vertical line, so that a bullet will encounter at most one target along any given vertical trajectory. Once a target is hit, we'll stop the bullet's upward movement. Within this subroutine, we'll also set the game score variable, Z, to zero, foreshadowing the concept of automatic game restart. Since we're placing the targets two pixels apart at the closest, and the Station moves in two-pixel increments, it's important that the ship always start on an even pixel location. Therefore, we'll rethink our original random positioning of the station and decide to set the initial position of the ship at 50 (line 50) instead. Our program now looks like this:

```
10 REM*** MICRO ARCADE ***
20 CLS:COLOR 0,1,3,7
30 PLOT 1,1,1,112,10
40 PLOT 1,70,1,112,1
50 X=50
51 PRINTCHR$(8)
52 GUN$=CHR$(6)
60 OUTPUT GUN$,X,16,2
65 GOSUB 700:REM-SPRINKLE TARGETS
70 FOR T=1 TO 1000
100 GOSUB 300:REM-MOVE GUNSHIP
280 NEXT
295 END
300 ON JOY(0) GOTO 360,380
310 RETURN
360 IF X<6 THEN RETURN
362 X=X-2
364 OUTPUT GUN$,X+2,16,0
365 GOTO 385
380 IF X>112 THEN RETURN
382 X=X+2
384 OUTPUT GUN$.X-2.16.0
385 OUTPUT GUN$,X,16,2
390 RETURN
700 FOR A=10 TO 106 STEP 2
715 IF RND(1)>.50 GOTO 740
720 B=20+45*RND(1)
730 PLOT A,B,2
740 NEXT
750 Z=0
760 RETURN
```

Now that we've got our space station and target displays in order, we need to add the capability of firing at the targets. Again, we'll use a subroutine to perform the operation. We'll call the subroutine from within the main program loop by adding two more lines. The first, at line 80, tests to see if the fire button on the left controller is depressed. If it is not depressed (FIRE(0)=1), we'll go back to line 100 and execute the subroutine for Station movement again. If the fire button is depressed (line 80 tests false), then we'll execute a subroutine to fire a bullet. In this subroutine, the first three lines (500-520) will generate an explosive sound. Then, in a loop (530-560), we'll propel the bullet upwards with the variable Y. But, before each upward movement, we'll check to see if the target has been hit with the POINT function (line 532). If it has been hit, we'll flash the screen, remove the target, and increment the score stored in the variable Z before returning to the main program loop for the next gunship movement. If the target is not hit, we'll keep incrementing the upward movement with Y until the loop is exhausted and the bullet is at the top of the screen. Then we'll make a "dud" sound (580) before returning to the main program loop. Now, our program looks like this:

```
10 REM*** MICRO ARCADE ***
20 CLS:COLOR 0,1,3,7
30 PLOT 1,1,1,112,10
40 PLOT 1,70,1,112,1
50 X=50
51 PRINTCHR$(8)
52 GUN$=CHR$(6)
60 OUTPUT GUN$,X,16,2
65 GOSUB 700:REM-SPRINKLE TARGETS
70 FOR T=1 TO 1000
80 IF FIRE(0)=1 GOTO 100
82 GOSUB500:REM-FIRE BULLET
100 GOSUB 300:REM-MOVE GUNSHIP
280 NEXT
295 END
300 ON JOY(0) GOTO 360,380
310 RETURN
360 IF X<6 THEN RETURN
362 X=X-2
364 OUTPUT GUN$,X+2,16,0
365 GOTO 385
380 IF X>112 THEN RETURN
382 X=X+2
384 OUTPUT GUN$,X-2,16,0
385 OUTPUT GUN$,X,16,2
390 RETURN
```

```
500 SOUND 1,512
510 FOR Q=1 TO 40:NEXT
520 SOUND 1,513
522 XP=X+2
530 FOR Y=19 TO 69
532 IF POINT(XP,Y)<>2 GOTO 540
533 Z=Z+1
534 PLOT XP,Y,0:PLOT XP,Y-1,0
535 SOUND 1,514:COLOR 7,3,1,0
536 FOR Q=1 TO 10:NEXT
537 COLOR 0,1,3,7:SOUND 1,515
538 RETURN
540 PLOT XP,Y,3
550 PLOT XP,Y-1,0
560 NEXT
570 PLOT XP,69,0
580 TONE 600,8
600 RETURN
700 FOR A=10 TO 106 STEP 2
715 IF RND(1)>.50 GOTO 740
720 B=20+45*RND(1)
730 PLOT A,B,2
740 NEXT
750 Z=0
760 RETURN
```

In this program, our main controlling loop in lines 70-280 contains only five statements. The rest of the program is performed through subroutines. In this main loop, we increment a time counter, T, to 1000 before the game ends. Finally, we'll add lines 286-288 to display the score when the time counter has expired. We'll also embed a RUN command in line 289 to start the game over again automatically.

```
286 OUTPUT"YOUR SCORE IS",15,60,3
287 OUTPUT Z,46,50,3
288 FOR Q=1 TO 1000:NEXT
289 RUN
```

The final program listing for this game is presented on the following page. In retrospect, we find that a few lines are really unnecessary. This is frequently the case in developing a complex program. We can remove lines 295 and 750 with no resulting difference in program operation.

Game Extensions

No game is ever really "done." Modifications and improvements can always be made, and the Micro Arcade is no exception. Here are some ideas you might want to implement with this program as an exercise in game programming.

a) Find an alternate character for the Earth Station. Examine other non-standard characters, or use two or more "overlaid" characters with the OUTPUT statement.

b) Add sound to the movement of the Earth Station.

c) Add difficulty levels that will change the number of targets and game duration. Accept input from the keyboard to determine what the difficulty level will be.

d) Maintain the name and high score of the "current winner" and display it at the end of each game.

e) Use multiple color targets and award points for their destruction that are inversely proportional to the color's frequency of occurrence.

```
10 REM*** MICRO ARCADE ***
20 CLS:COLOR 0,1,3,7
30 PLOT 1,1,1,112,10
40 PLOT 1,70,1,112,1
50 X=50
51 PRINTCHR$(8)
52 GUN$=CHR$(6)
60 OUTPUT GUN$,X,16,2
65 GOSUB 700:REM-SPRINKLE TARGETS
70 FOR T=1 TO 1000
80 IF FIRE(0)=1 GOTO 100
82 GOSUB500:REM-FIRE BULLET
100 GOSUB 300:REM-MOVE GUNSHIP
280 NEXT
286 OUTPUT"YOUR SCORE IS",15,60,3
287 OUTPUT Z,46,50,3
288 FOR Q=1 TO 1000:NEXT
289 RUN
295 END
300 ON JOY(0) GOTO 360,380
310 RETURN
360 IF X<6 THEN RETURN
362 X=X-2
364 OUTPUT GUN$,X+2,16,0
365 GOTO 385
380 IF X>112 THEN RETURN
382 X=X+2
384 OUTPUT GUN$,X-2,16,0
385 OUTPUT GUN$,X,16,2
390 RETURN
500 SOUND 1,512
510 FOR Q=1 TO 40:NEXT
520 SOUND 1,513
522 XP=X+2
530 FOR Y=19 TO 69
532 IF POINT(XP,Y)<>2 GOTO 540
533 Z=Z+1
534 PLOT XP,Y,0:PLOT XP,Y-1,0
535 SOUND 1,514:COLOR 7,3,1,0
536 FOR Q=1 TO 10:NEXT
537 COLOR 0,1,3,7:SOUND 1,515
538 RETURN
540 PLOT XP,Y,3
550 PLOT XP,Y-1,0
560 NEXT
570 PLOT XP,69,0
580 TONE 600,8
600 RETURN
700 FOR A=10 TO 106 STEP 2
715 IF RND(1)>.50 GOTO 740
720 B=20+45*RND(1)
730 PLOT A,B.2
740 NEXT
750 Z=0
760 RETURN
```

READING DATA

So far, we've examined several ways in which data can be entered for use into a BASIC program:

Keyboard Input	- Using INPUT, INSTR$, or keyboard PEEK
Controller Input	- Using POT, JOY, and FIRE functions
Tape Input	- Using CLOAD* to load arrays of numeric data or a numeric representation of string data

In this chapter we'll examine a fourth general method of entering data into a program--use of the DATA, READ, and RESTORE statements. This approach can be effective for entering large numbers of constants or table data into a program--to supply parameters for TONE sequences, (x,y) coordinate pairs for graphic image development, mathematical tables for calculations. In this programming approach, the data values are stored as constants in line-numbered DATA statements. The READ statement pulls the requested number of data values from the list in a sequential pass through the data stream. The RESTORE statement resets READ's internal pointer back to the first item in the first DATA statement, which lets you READ data items in a list more than one time in a single program.

There are several advantages to keeping data in this form:

1) The data values are stored as a list in the program itself instead of being typed in during program execution. The list can be easily referenced, stored, listed, and changed just like you would change any portion of the program's logic.

2) The data list can be read repeatedly during the execution of a program, selectively referenced, and passed on to the program's processing logic.

3) Surprisingly large amounts of tabular data can be entered with little storage overhead, even with the Interact BASIC's 4698 bytes of memory.

4) Data lists can be stored on tape in separate files and used in programs by adding the file to a program with the EZEDIT APPEND command. Thus, the concept of a "data library" can be achieved. For example, you could write an extensive musical repertoire which consists of a series of files with DATA statements that contain TONE parameters, the number of tones, and the title of the work. A BASIC music program could then be developed to play, transpose, or display any tune, similar to the Music Maestro program.

However, there are two drawbacks to this approach to data entry:

1) The amount of data that can be stored and accessed is limited by the amount of available RAM and the program logic size.

2) The stored data values cannot be changed by program logic. Changes to the data can only be made by retyping the DATA statements and using CSAVE to store the changed program.

DATA statements provide an "invisible stream" of data that flows through normal program execution. DATA statements are never actually executed; they are only "seen" and accessed by the READ statement, which is the only statement that knows that they exist. READ has an internal pointer that keeps track of how many values have been read from the list during program execution.

You can use DATA statements for multiple purposes within a single program. For example, you might have DATA statements containing TONE parameters, and also DATA statements containing string values to be displayed during program execution. These two operations might be completely unrelated in the flow of program execution. Technically, you could combine data values for both these operations onto the same DATA statement. However, that can lead to more complicated program logic to access the values and more work if you want to add or change a data value. Generally, we advise you to store sets of related data values on one or more adjacent DATA statements and place data values for unrelated operations on separate DATA statements.

The following Sounds Library program illustrates the use of DATA statements to catalog information. In this program, there is a DATA statement for each sound in the library. Each sound has four related items of information: two numeric sound parameters and two strings of descriptive text. In the program, these four items are called S1, S2, T1$, and T2$.

```
10 REM-SOUNDS LIBRARY
20 CLS:COLOR 0,3,1,7
30 MAX=5
40 FOR N=1 TO MAX
50 CLS
60 READ S1,S2,T1$,T2$
70 OUTPUT T1$,20,60,1
80 OUTPUT T2$,20,54,1
82 OUTPUT S1,40,40,2
84 OUTPUT S2,52,40,2
90 SOUND S1,S2
100 FOR Q=1 TO 2000:NEXT
105 NEXT
110 SOUND 7,4096
111 CLS:PRINT"DONE":END
120 REM-SOUNDS LIBRARY
130 DATA 3,182,"PT 109",""
140 DATA 5,392,LOCUST, ATTACK
150 DATA 3,66,1938 PLYMOUTH,STUCK HORN
160 DATA 6,460,PHASER,""
170 DATA 6,170,TELEPHONE,RING
```

Note that we entered a null string as the fourth parameter in lines 130 and 160. Because the READ statement accesses the data sequentially, four items at a time in this program, we must use null strings to keep our data consistent. If we just omit that fourth parameter in lines 130 and 160, the READ statement would take the first data item in the next line as the fourth parameter. This would not only yield an incorrect display on the first sound processed, it would also create an error when the computer reads the second set of four data items. Therefore, we include the null string to tell the computer that there is no fourth parameter in those cases--READ can't make any assumptions about our data as it references it.

You can expand this program to catalog and play back your own favorite Interact sounds. All you have to do is enter additional DATA statements following line 180 and change the value of MAX in line 30 to reflect the current count of your sounds. No other change to the program logic is required to handle the increased amount of data.

Reading Complex Data Tables

The items in DATA statements can be read more than once during the execution of a program. This is convenient when you want to access table data repeatedly in your programs. The following program illustrates how to compute the Federal Estate Tax for estates of a wide range of sizes. The DATA statements in the program contain the values in the table below.

FEDERAL ESTATE TAX

Net Taxable Estate	Estate Tax	% Tax in Next Bracket
Up to $10,000	18% of amount	
$ 10,000	$ 1,800	20%
20,000	3,800	22
40,000	8,200	24
60,000	13,000	26
80,000	18,200	28
100,000	23,800	30
150,000	38,800	32
250,000	70,800	34
500,000	155,800	37
750,000	248,300	39
1,000,000	345,800	41
1,250,000	448,300	43
1,500,000	555,800	45
2,000,000	780,800	49
2,500,000	1,025,800	53
3,000,000	1,290,800	57
3,500,000	1,575,800	61
4,000,000	1,880,800	65
4,500,000	2,205,800	69
5,000,000	2,550,800	70

```
10 REM: FEDERAL ESTATE TAX
20 REM:      CALCULATOR
30 CLS:COLOR 4,3,0,7
40 PRINT:PRINT"NET TAXABLE"
50 PRINT"ESTATE IN"
60 PRINT"THOUSANDS (000)"
70 INPUT NT
80 IF NT<0 GOTO 70
90 GOSUB 300
100 PRINT"TAX= ";ET
110 GOTO 70
300 IF NT<10 THEN ET=.18*NT:RETURN
309 RESTORE
310 DD=-1
320 READ T1,T2,T3
330 DD=DD+1
340 IF NT>T1 GOTO 320
360 REM:HAVE PASSED APPROPRIATE BRACKET
370 REM:RESTORE AND READ DOWN AGAIN
375 RESTORE
380 FOR A=1 TO DD
390 READ T1,T2,T3
400 NEXT
410 ET=T2+T3*(NT-T1)
420 RETURN
480 REM- FEDERAL ESTATE
482 REM - TAX TABLE
500 DATA 10,1.8,.2
510 DATA 20,3.8,.22
520 DATA 40,8.2,.24
530 DATA 60,13,.26
540 DATA 80.18.2,.28
550 DATA 100.23.8,.3
560 DATA 150,38.8,.32
570 DATA 250,70.8,.34
580 DATA 500,155.8,.37
590 DATA 750.248.3,.39
600 DATA 1000.345.8,.41
610 DATA 1250,448.3,.43
620 DATA 1500,555.8,.45
630 DATA 2000,780.8,.49
640 DATA 2500,1025.8,.53
650 DATA 3000,1290.8,.57
660 DATA 3500,2575.8,.61
670 DATA 4000,1880.8,.65
680 DATA 4500,2205.8,.69
690 DATA 5000,2550.8,.70
800 DATA 99999,0,0
```

Given the Net Taxable Estate, NT, the program searches the table to find the corresponding tax bracket. Once it finds the correct bracket, it performs tax computations using information in the previous line of the table. Therefore, the tax table must be read twice for each tax computation in this program.

How does this work? Well, let's say the estate was valued at $200,000. If you look at the table on page 6-3, you'll see that this bracket is between $150,000 and $250,000 categories. To compute the tax the program uses the values associated with the $150,000 line in the table.

ET = 38,800 + .32*(200,000-150,000) = 54,800.00

The first READ of the table in line 320 looks for the data corresponding to the Net Taxable Estate amount entered in reponse to the INPUT statement in line 70. However, since the program calculates the tax based on the values on the previous line in the table, it has already passed by the values it needs to use for the calculations. Therefore, it must go back and read through them again. The RESTORE statement in line 375 returns READ's internal pointer to the first data item in the first DATA statement (line 500) so that all values in the table can be referenced again.

To conserve RAM space, we could combine two lines of the table (or even three) into a single DATA statement. Each DATA statement would then have six values associated with it, e.g.,

```
500 DATA 10,1.8,.2,20,3.8,.22
```

With this construction, fewer lines are used to store the same amount of data; hence, less RAM is consumed. There would be no change in the operation of the program.

Consult the DATA, READ, and RESTORE statements in the Reference Section for further explanation and examples of this mode of data entry.

SUBROUTINES

A BASIC program is, as we have seen, a set of statements that are performed in line-numbered order, unless the statements themselves direct BASIC to begin executing a statement at another line. Statements which change the order of program execution include the GOTO and GOSUB statements.

The GOTO statement simply transfers control to a specified line in the program. Essentially, it's a "one-way ticket" to a destination in the program. GOSUB, on the other hand, is analogous to a "round trip ticket". GOSUB tells the computer to transfer program execution to a subroutine that begins on a specified line, but to remember the point at which the GOSUB was made so that program control can be returned to the statement following the originating GOSUB when the subroutine completes. In the subroutine, you tell BASIC you want to make the "return trip" by concluding it with a RETURN statement.

A subroutine, then, is a set of instructions that is executed from a GOSUB statement elsewhere in the program. Subroutines are generally positioned outside the general flow of the program, and their statements are executed only when a GOSUB to the starting line number is encountered during program execution. Subroutines are virtually always terminated with RETURN statements. Subroutines may be nested--that is, a subroutine may in turn call another subroutine which in turn may call another subroutine, etc. But, as GOSUB and RETURN statements come in pairs, BASIC is able to find its way back to the main program logic.

Using subroutines is good programming practice. The benefits of using them are:

1) Subroutines conserve memory. The logic for a single subroutine can be called from numerous points in the program by GOSUB statements, rather than repeating the statements several times within the program logic. Thus, programs with subroutines generally consume less RAM, or, conversely, you can build larger programs in the same amount of RAM.

2) Subroutines make programming easier. A subroutine's logic needs to be debugged only once. After it is operational, you can usually take for granted that it will do what it is supposed to do when it's called.

3) Subroutines mean less typing. You only have to type the statements once.

4) Subroutines allow better coding. You can afford to develop better subroutines that include error checking, consistent performance, etc., if you know you only have to develop them once.

5) Subroutines also make program modification easier. You can make changes that affect the operation of the entire program all at once within the subroutine, rather than having to modify statements at several spots within the program.

Take, for example, a simple subroutine that accepts a "Y" or "N" from the keyboard. It might be used several times in a program that requires the user to respond to a series of yes/no questions. The subroutine below sets the variable S to a value of 1 if the answer to the question was "Y" or to 0 if the answer was "N". No other keys can satisfy the subroutine; it will not perform any further program execution until a "Y" or "N" is typed.

```
500 A$=INSTR$(1)
510 IF A$="Y" THEN S=1:RETURN
520 IF A$="N" THEN S=0:RETURN
530 GOTO 500
```

Pause loops can frequently be placed within subroutines and called to allow the user to read a screen of instructions, to vary visual displays, etc. A pause loop that takes approximately one second to perform is

```
FOR K = 1 TO 480:NEXT
```

By placing this statement in a subroutine which is given the number of seconds to pause with the variable S, we can build a generalized subroutine.

```
100 SM=480*S
110 FOR Z=1 TO SM:NEXT
120 RETURN
```

Then, if we place this subroutine within a program that requires pauses of different durations to accompany different operations, we can compute and perform the pause loops from this generalized timing loop. The following program calls this timing loop three times, from lines 50, 78, and 90, with pause durations as specified by the variable S.

```
10 REM - DELAYING WITH A SUBROUTINE
20 CLS:COLOR 4,7,1,2
30 PRINT"HOLD THIS MESSAGE"
40 PRINT"FOR 10 SECONDS"
50 S=10:GOSUB 100
60 CLS
70 REM-FLASH EVERY .5 SECONDS, 10 TIMES
72 OUTPUT"HI",50,50,1
75 S=.5:FOR N=1 TO 10
77 COLOR 7,4,1,1
78 GOSUB 100
80 COLOR 4,7,1,1
90 GOSUB 100
95 NEXT
97 END
99 REM-DELAY LOOP SUBROUTINE
100 SM=480*S
110 FOR Z=1 TO SM:NEXT
120 RETURN
```

Subroutines are commonly used in Interact BASIC to:

- Read and play TONE information from DATA statements
- Plot a complex graphic entity whose position on the screen is specified in variables external to the subroutine
- Accept yes/no information from the keyboard
- Initialize a game grid or display
- Perform a set of complex arithmetic operations that are common to several portions of the overall program logic
- Sort data
- Read or write string data to cassette tape by converting the string information into equivalent numerical arrays

INTERFACING WITH THE BASIC ENVIRONMENT

This section contains some general operating hints for interfacing with BASIC. These suggestions on how best to carry on a meaningful dialogue with your computer will help make your programming life easier.

Storing Programs on Tape

Always store programs on tape during program development, particularly if the program is long and involved. As soon as you get part of it completed and working to your satisfaction, use the CSAVE command to store it on tape. Then, if operation of BASIC is inadvertantly interrupted, you won't lose your entire program and have to start all over again. This is especially important if your program uses POKE statements, as it is completely possible to POKE your program out of existence through a faulty value or mistyped POKE location.

In addition to providing you with "back-up" copies of your program in the event of a BASIC "disaster", saving your program at various stages of development lets you back up easily if you decide to alter the working of the program radically.

Take advantage of the file naming option when you use CSAVE to store a partial or completed program on tape, especially if you're saving multiple programs on the same data tape. If you save files with names, they are easier to CLOAD later when you want to back up or restart a program.

Tape Positioning

Before you issue the CSAVE command to save a program, or CLOAD to read one in, you may need to position the tape for reading or writing. You can do this with the REWIND command. REWIND turns the tape motor on. You can then use any of the cassette drive buttons to control tape positioning. Use the REWIND or F-FWD buttons to move the tape backward or forward quickly. You can also use the READ button for slower forward tape positioning. READ lets you position the tape accurately, because you can hear the data sounds and tell where one program begins and another ends. Although you will hear "tape loading" sounds if you depress the READ button during REWIND command execution, no data is being read into your computer. You'll avoid overwriting and destroying other programs on the tape if you use this operating control in BASIC.

* Note that if you depress both the READ and WRITE cassette buttons during REWIND execution, any information stored on the tape <u>will be erased</u>.

Control Characters

Your Interact has several control characters that can be used to affect program execution or listing. Control characters are issued by pressing the Control and another key simultaneously.

<u>Control-C</u> acts as a "break" key during program execution or listing. If you type this Control character, BASIC finishes printing the

line it's listing or executing, then stops program execution or listing.

Control-S lets you halt program execution or listing temporarily, to examine a program line or screen detail. Execution of the program or the listing continues when any key is depressed.

Control-U acts as a "cancel" key. It stops the line you were typing at the time you issued the Control-U from being entered into memory. You can use this to abort a line during program entry.

Control-O is used during program execution to suppress output from PRINT statements. This can be useful if you are printing out a long list of data and don't particularly want to see it. The Control-O is cancelled when an OUTPUT statement is processed. Control-O suppresses output from the PRINT command only and has no effect on other program statements.

What Happens When You Press RESET?

If you press the RESET button, the current execution of BASIC is stopped, including program execution and listing. Parameters of some commands, such as COLOR, are returned to the default state when you press the "R" key to restart BASIC. The setting of WINDOW is also returned to 77. Variables are not reset, however, nor is the value of the CLEAR statement affected.

We recommend that you not use RESET to stop a program. Use Control-C instead, especially if the program contains OUTPUT statements. If you depress RESET at the same time that BASIC happens to be processing an OUTPUT statement, that OUTPUT statement is likely to be "clobbered". If you restart BASIC and run the program again, you may get a syntax error ("?SN ERROR") or other error when BASIC encounters the damaged OUTPUT statement.

Program Debugging

There are a number of means for debugging programs. You can use the GOTO line number and RUN line number commands in program debugging, as discussed in Program Execution (page 2-16). You can also use the EZEDIT program editor to make global corrections in your program, resequence it, etc.

You can also use the STOP statement within a program to act as a breakpoint. A program containing a STOP instruction will terminate execution when the STOP instruction is encountered. This can be useful in program debugging. Let's say, for example, that you've got an error in part of your program. You suspect it is the setting of a variable, but you aren't certain. You can put a STOP statement in that part of the program, and, when the program stops, you can PRINT the variable to check its current value, change the value if necessary, then type

```
CONT
```

to resume execution where the program left off, using the new variable value.

Space Saving Hints

There's a programmer's maxim that states: "A program always grows in size to fill the memory available, regardless of the size of the computer's memory." Because the Interact has limited memory capacity, you'll want to write programs efficiently to utilize RAM to the maximum. Consider the following list of hints on how to save space before you write even the first line of a program that might eventually encounter memory limitations. If you can incorporate these suggestions into your own programming style, you'll be surprised at the complexity of programs that can be developed in the 4,698 bytes of memory your Interact has for BASIC programming.

1) Use subroutines extensively. If certain operations are similar throughout a program, then put them in subroutines or in a set of nested subroutines to conserve space. Using GOSUBs to invoke processing whereever possible, rather than repeating statements, consumes considerably less RAM.

2) Combine statements on a single line. Many sequences of statements can logically and easily be placed on a single line by separating the individual statements with colons (:). Compacting programs in this manner reduces the program size.

3) Reuse variable names. Try to use simple variable names--I, J, P, Q, etc., for non-related operations such as pause loops, rather than defining new variable names. Each variable you name requires another entry in the symbol table.

4) Use short variable names. Single-character variable names consume less RAM than longer names.

5) Use CLEAR(O). If your program does not use any string variables, place a CLEAR(O) statement early in your program to eliminate the space automatically allocated for string variables by BASIC. This increases the space available for numeric variables or statements by 50 bytes (to 4748 bytes). Do not use CLEAR(O) if you use even one string variable. In this case, you could use CLEAR to reduce the string variable space to all but a few bytes.

6) Omit the END statement. END is not required to terminate a program and can be omitted without affecting program operation.

7) Use REM statements sparingly, if at all. Remark statements, while they do document a program's operation, require precious RAM for storage. Use them only when absolutely necessary and be prepared to delete them from programs to gain memory as programs increase in size.

8) Use Microsoft 8K Fast Graphics BASIC. This BASIC has the same capabilities as the older Level II BASIC, but its extended PLOT capabilities let you eliminate many FOR...NEXT loops associated with screen graphics entirely. Microsoft 8K BASIC lets you develop larger programs using the same amount of RAM.

9) Omit embedded blanks. Blanks are not necessary in BASIC statements. In fact, BASIC ignores them. You can omit blanks without affecting program operation, except as needed in string constants.

10) Place frequently called subroutines on low line numbers. Line numbers also consume RAM, and, by placing these subroutines on lines with single or two-digit line numbers, the line number references in subsequent GOSUB statements will be shorter and consume less RAM.

11) Omit the iteration variable name in NEXT statements. The variable name is optional and can be omitted to gain a couple bytes of storage from each FOR...NEXT loop.

12) Use AND and OR in IF statements to test multiple conditions, rather than using multiple IF statements. The logic for multiple tests can usually be combined into a single IF statement.

13) Use the IF...THEN construction and chain multiple statements to be performed if the condition tests "true" following THEN. This method generally requires less RAM than the IF...GOTO construction. (See Conditional Relationships in Chapter 2 for an example of reducing program size in this way.)

14) Use the "wrap-around" feature of the OUTPUT statement. When using OUTPUT to display text on the screen, multiple lines can be directed to the screen from a single OUTPUT statement by embedding the necessary number of blanks in the string.

15) Store frequently used string constants or numeric values in variables. The subsequent reference to the variable name will generally require less RAM than repeating the string or the computation, particularly if the strings or computations are lengthy.

16) Use the zeroth element of arrays. Note that a DIM A(10) statement actually allocated eleven locations (A(0) - A(10)). Use the A(0) cell for data storage.

17) Use variables to dimension multiply-dimensioned arrays. The statement DIM A(N,M) allows dimensioning to occur during program execution to fit the size required by the program, rather than over-dimensioning to fit a maximum-sized problem.

18) Consider a program's intent when converting programs from another computer. Many programs have long PRINT statements that require large amounts of RAM. Converting such programs into a graphic orientation may make them more interesting as well as more space-efficient.

19) Use user-defined functions. The DEF statement can help you eliminate repeated calculations of sub-expressions. Such redundancy will usually consume more space for computational processes than is actually needed.

20) Consider program segmentation. If your program is a large, sequential operation, you may be able to divide it into two or more separated programs that are called with CLOAD statements at the end of each program.

21) Use DATA statements for data reference in programs. DATA statements are more efficient for working with preset tables of numeric or string information than is assigning the table values to dimensioned arrays. The extra time required to RESTORE and READ the DATA statements is usually insignificant, while the space savings are substantial. You save space because only one "copy" of the information resides in RAM, rather than two. We also recommend that you put as many data values into a single DATA statement as possible.

RS232 BASIC

RS232 BASIC is a version of Level II BASIC that has been expanded to provide the capability of accessing a lineprinter to produce program listings or formatted reports. It has all the same language capabilities and features as Level II BASIC, but with two additional commands--LLIST and LPRINT.

Your Interact must be equipped with an RS232 peripheral interface in order to load and run RS232 BASIC. If you attempt to load RS232 BASIC into an Interact that does not have the peripheral interface, the interpreter will appear to load correctly, then return to the DEPRESS L TO LOAD TAPE screen. This happens because RS232 BASIC attempts to initialize the port when it loads. If it does not find a port to initialize, the interpreter cannot function properly.

Lineprinter Access

Two commands are available for accessing a lineprinter in RS232 BASIC.

The LLIST command produces a lineprinter listing of the BASIC program currently in memory. LLIST can be used in direct mode only. To produce a listing, type

```
LLIST
```

The LLIST command will appear on your TV screen. The program, however, will list out on the lineprinter.

Like the LIST command, LLIST will let you specify a starting line number for the program listing. If you have a program that's 500 lines long, and you only want to see the last 200 lines, you might type

```
LLIST 300
```

The program listing will begin with line 300. Listed lines can be up to 72 characters long.

The LPRINT command also outputs information to a lineprinter. LPRINT can be used in direct mode, but it's more commonly embedded in a program to produce formatted reports, analyses, summaries, etc. In this case, the LPRINT command is not executed until the program is RUN.

Like the PRINT command, LPRINT can be used to display various types of string or numeric information. It operates in the same way that the PRINT command does, allowing multiple items to be included on a single statement. You'll find the arithmetic display control functions (discussed in chapter 2, page 2-11) to be useful for formatting output to the lineprinter with LPRINT.

What kind of printer should you use with your interface and RS232 BASIC? The only restriction on the kind of printer that can be used to produce formatted reports and program listings with your Interact is that it must be RS232-compatible to run directly off the interface port.

If you go shopping for a printer to keep your Interact company, you'll find that there are many different RS232-compatible printers over a wide price range. The price you'll pay for a printer, of course, is directly related to the printer's features and capabilities.

Personally, we like the COMPRINT 912-S. It provides consistent quality hard copy that photocopies extremely well. Virtually all the examples in this manual were produced on our RS232-equipped Interact with RS232 BASIC and our COMPRINT 912-S printer.

I/O Parameter Control

Although you can use any RS232-compatible lineprinter with the Micro Video peripheral interface and RS232 BASIC, some printers are not set up to operate under the default input/output parameters in RS232 BASIC. The default I/O parameters in RS232 BASIC are:

1200 BAUD

8-BIT WORD LENGTH, 1 STOP BIT

ODD PARITY

You can control the settings of the I/O parameters as appropriate to your particular printer by using the POKE command to store non-default parameter values in several different memory locations.

Baud Rate

RS232 BASIC and the Micro Video interface can be used to drive an RS232-compatible lineprinter at standard and non-standard baud rates between 110 and 19200 bps. The default, 1200 baud, is the most commonly used baud rate.

To set the baud rate between 600 and 19200, use the following POKE initialization:

```
POKE 25098,A          ; BAUD LATCH AT .COOOH
```

where A = 111860/desired baud rate. To make things a little simpler, the table below identifies the value of A for standard baud rates.

A =	BAUD RATE
186	600
93	1200
47	2400
23	4800
12	9600
6	19200

To set a baud rate lower than 600 bps, a two-byte divisor must be initialized with two separate POKE statements.

```
POKE 25098,A

POKE 25099,B
```

The following table defines the values of A and B for standard baud rates slower than 600 bps.

A =	B =	BAUD RATE
116	1	300
232	2	150
246	3	110

Note: Some devices do not operate at exactly standard (600, 1200, etc.) baud rates. If you have difficulty using a device with your Interact and RS232 BASIC, it is far more likely that the device's required baud rate is slightly off standard than that there is a problem with your computer or interface. Try decreasing or increasing the baud rate divisor slightly for correct operation of the printer.

Data Format

The default data format parameters in RS232 BASIC are 8-bit word length with 1 Stop bit, odd parity. Although this is a standard format used by most printers, some devices require a different data format. To set the individual bits for a non-default format, select as appropriate from the following statements.

Start with:

```
A = 0
```

To change word length:

```
A = A OR 0        ; FOR 5-BIT WORD LENGTH
A = A OR 1        ; FOR 6-BIT WORD LENGTH
A = A OR 2        ; FOR 7-BIT WORD LENGTH
A = A OR 3        ; FOR 8-BIT WORD LENGTH (DEFAULT)
```

To set number of Stop bits:

```
A = A AND 251     ; FOR 1 STOP BIT (DEFAULT)
A = A OR 4        ; FOR 2 STOP BITS
```

To set parity on or off:

```
A = A AND 247     ; FOR NO PARITY CHECKING
A = A OR 8        ; FOR PARITY CHECKING (DEFAULT)
```

To set even or odd parity:

```
                A = A AND 239          ; FOR ODD PARITY IF PARITY SET ON (DEFAULT)
                A = A AND 16           ; FOR EVEN PARITY IF PARITY SET ON
```

Then, to initialize the non-default format, POKE the value of A into memory location 25100.

```
                POKE 25100,A           ; LINE CONTROL REGISTER AT .C004H
```

You can reset to the default data format with the statement

```
                POKE 25100,11
```

This is, of course, not the most efficient way to define format parameters. This explanation in intended to explain the construction of the value to be POKEd into location 25100. To determine the value of A for your particular device, you can enter these statements in direct mode and then use the statement

```
                PRINT A
```

to obtain the value of A. Then, for future use, you can poke that value directly into location 25100.

If you have a printer that uses non-default I/O parameters, you must initialize the parameters in BASIC before trying to access the printer. The best way to do this is to build a small initialization program and store it on tape. You'll enter and run this initialization program before you try to produce hard copy with your printer. Note that the interface is not initialized with the latest POKE information until BASIC reinitializes itself, which is signalled by the reappearance of the "OK" prompt.

Line Feed Control

You can also control automatic line feeding on your printer through RS232 BASIC. The line feed flag is initialized a 0 (no line feed). To set automatic line feed upon encountering a carriage return, use the statement

```
                POKE 25097,10
```

Program Listings

You can get listings of your Level II BASIC programs or of programs written in RS232 BASIC. (You can also get listings of Microsoft 8K BASIC programs, although you will not be able to execute any extended PLOT commands under RS232 BASIC control.) To list a program written in RS232 BASIC, just type

```
                LLIST
```

If you want a listing of a Level II or Microsoft 8K BASIC program, that too is possible. However, one other step is required. You must use the

TRANSLATE command in RS232 EZEDIT to convert the programs into RS232 format. RS232 BASIC has internal formatting different from Level II or Microsoft 8K BASIC because it has the two additional commands in its keyword table; therefore, non-RS232 BASIC programs must be converted into the different format.

There's another method you can use to get listings of your programs, even if you're not ready to invest in an interface and printer. Micro Video offers a printer service to those who do not have the printer capability on their Interacts.

Machine Language Integration -- the USR Function

With the Micro Video MONITOR, you can write your own machine language subroutines, then call them from a BASIC program with the USR function. Before entering the USR function, you must define the starting address (LSB and MSB) of the machine language routine you want to call. You do this by entering two POKE statements. The value put into the first POKE location (30499) defines the least significant bit (LSB) of the starting address. The value put into the second POKE location (30500) identifies the most significant bit (MSB). Since machine language addresses are in hexadecimal, but BASIC requires that you enter a decimal value with the POKE statement, you must convert the starting address into two decimal values. See the Hexadecimal/Decimal Conversion Table in chapter 11 to convert your subroutine's starting address to decimal quickly and effortlessly.

```
POKE 30499,L

POKE 30500,M

USR
```

Note that the USR call in RS232 BASIC is different from the other BASIC interpreters. USR calls in Microsoft 8K and Level II BASIC require that the argument, (0), be included on the call. In RS232 BASIC, the USR keyword is entered alone, as shown above.

In concluding our discussion of RS232 BASIC, following are a lineprinter listing of an RS232 BASIC program that accesses a lineprinter and the "hard copy" result of running that program.

```
10 LPRINT "      TEST PROGRAM"
20 CLEAR(200)
30 A$ = ""
40 FOR CH = 32 TO 96
50 A$ = A$ + CHR$(CH)
60 LPRINT A$
70 NEXT
```

```
     TEST PROGRAM

!
!"
!"#
!"#$
!"#$%
!"#$%&
!"#$%&'
!"#$%&'(
!"#$%&'()
!"#$%&'()*
!"#$%&'()*+
!"#$%&'()*+,
!"#$%&'()*+,-
!"#$%&'()*+,-.
!"#$%&'()*+,-./
!"#$%&'()*+,-./0
!"#$%&'()*+,-./01
!"#$%&'()*+,-./012
!"#$%&'()*+,-./0123
!"#$%&'()*+,-./01234
!"#$%&'()*+,-./012345
!"#$%&'()*+,-./0123456
!"#$%&'()*+,-./01234567
!"#$%&'()*+,-./012345678
!"#$%&'()*+,-./0123456789
!"#$%&'()*+,-./0123456789:
!"#$%&'()*+,-./0123456789:;
!"#$%&'()*+,-./0123456789:;<
!"#$%&'()*+,-./0123456789:;<=
!"#$%&'()*+,-./0123456789:;<=>
!"#$%&'()*+,-./0123456789:;<=>?
!"#$%&'()*+,-./0123456789:;<=>?@
!"#$%&'()*+,-./0123456789:;<=>?@A
!"#$%&'()*+,-./0123456789:;<=>?@AB
!"#$%&'()*+,-./0123456789:;<=>?@ABC
!"#$%&'()*+,-./0123456789:;<=>?@ABCD
!"#$%&'()*+,-./0123456789:;<=>?@ABCDE
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEF
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFG
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGH
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHI
                .
                .
                .
```

BASIC

A TO Z

Reference Section

ABS

ABS is an arithmetic function that returns the positive numeric value of the given argument, disregarding the negative sign if it exists. The absolute value of a number, G, is defined as G if G is greater than or equal to zero and -G if G is less than zero. This function has the form:

```
ABS(n)
```

where:

n	is a numeric value. n may be a constant, variable, function call, or arithmetic expression.

You could compute the absolute value of a number within a BASIC program statement such as

```
IF G < 0 THEN G = -G
```

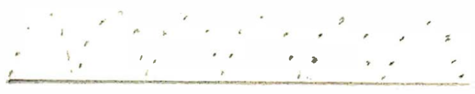
However, it is easier and more space-efficient to use the ABS function to perform the task, in a statement such as

```
G = ABS(G)
```

The value returned by the ABS function will never be negative. ABS is useful for finding the actual difference between two numbers without regard to whether they are positive or negative. You might also use it to convert negative values returned by other function calls into a positive value that could be used in other mathematical calculations or screen display.

EXAMPLE

```
10 CLS
15 FORX=1TO112:PLOT X,20,3:NEXT
20 FOR X=1 TO 112
30 Y=20+30*ABS(SIN(X/6))
40 PLOT X,Y,2
50 NEXT
60 GOTO 10
```



AND

AND is a relational (BOOLEAN) operator that performs a logical, bitwise ANDing operation on two or more relations. Generally used in conjunction with IF statements, AND tests to determine that both of relations adjacent to the AND keyword are true before performing the subsequent part of the IF statement. The results of any ANDing operation is either "true" or "false". Both of the relations must test true for the rest of the conditional operation to proceed. If one or both test to be false, then program control passes to the next higher line-numbered statement.

EXAMPLE

```
10 PRINT"ENTER SEX,AGE"
20 PRINT"TO ? PROMPT"
30 INPUT SX$,AG
40 IF SX$="M" AND AG>20 THEN PRINT "MAN"
50 IF SX$="F" AND AG> 20 THEN PRINT "WOMAN"
60 IF SX$="M" AND AG<21 THEN PRINT"BOY"
70 IF SX$="F" AND AG<21 THEN PRINT "GIRL"
80 GOTO 10
```

NOTES

AND may also be used in IF statements along with the other logical operator, OR. See the IF statement for further details on using AND for conditional testing.

ASC

ASC is a function that converts alphanumeric characters into the code in which they are stored in memory--their ASCII equivalents. ASCII is one of two universal codes used for character handling in all computers. This function has the form

ASC(a$) ASC("A")

where:

a$	is any alphanumeric character stored in a string variable or as a string constant for which the ASCII equivalent (decimal code) is to be returned. If the ASCII equivalent for a string constant is to be returned, the string must be enclosed in quotes within the parentheses.

ASC is commonly used to convert string data to numeric representation so that it can be saved on tape for future use in programs. The following example illustrates a subroutine that performs a character-by-character conversion of a string to numeric data.

EXAMPLE

In this subroutine that converts string characters to their ASCII equivalents, the string is treated as a dimensioned array. The conversion is performed by indexing through the elements of the string array with a loop and converting the Ith character of the array to numeric form. To use this subroutine within a program, you would also want to include a CSAVE* command to store the converted array on tape. See Chapter 4 for more information on this type of string handling and a complete example of the conversion subroutine.

```
300 WA(1)=LEN(N$(I))+1
310 IF WA(1)=1 THEN RETURN
320 FOR J=2 TO WA(1)
330 L$=MID$(N$(I),J-1,J)
340 WA(J)=ASC(L$)
350 NEXT J
360 RETURN
```

NOTES

ASC is a "cousin" of the CHR$ function, which can be used to convert numeric data back to string data. See the CHR$ function for more information on that function and a table of ASCII equivalents for characters on the Interact keyboard.

ATN

ATN (arctangent) is a trigonometric function that computes the angle, in radians, that has the tangent specified in the function argument. It has the general form

```
ATN(n)
```

where:

n	is a numeric argument to the function. n may be a numeric constant, variable, or arithmetic expression that represents a tangent.

The result of the ATN function, if printed, is expressed in radians. For example,

```
PRINT ATN(.707)
```

returns the value .615408, which is equal to 45 degrees.

EXAMPLE

There are 2π radians in a circle, and 57.2958 degrees in a radian. The following example illustrates how to convert an angle's tangent to number of degrees.

```
5 PRINT
10 INPUT"TANGENT";T
20 A=ATN(T)
30 D=57.2958*A
40 PRINT"ANGLE IN DEGREES"
50 PRINT"IS";D
60 GOTO 5
```

NOTES

ATN is one of several trigonometric functions that can be used in BASIC. Others are TAN, COS, and SIN.

ATN is used extensively in the Aircraft Lander BASIC program to spread the runway visually as the aircraft approaches the landing field.

CHR$

CHR$ is a string handling function that is a "cousin" of the ASC function. While ASC converts characters into their decimal equivalent codes, CHR$ does the reverse. Given a numeric value, CHR$ returns the character that is stored in memory as that number. CHR$ has the form

CHR$(n)

where:

n	is a numeric constant, variable, or expression for which the associated character is to be returned.

CHR$ is frequently used to convert the numeric representation of string data back into string form. Because only numeric data can be stored on tape, string arrays must be converted to numeric values with the ASC function for storage with CSAVE*. They are later read back into a program with CLOAD*, then converted back to string form with CHR$. See Chapter 4 for details on this process.

EXAMPLE

We ran this small program in RS232 BASIC to produce the data for the ASCII Equivalents Table on the following page. If you do not have the RS232 interface and a lineprinter, change the LPRINT statements in this program to PRINT statements to display the values on your TV screen.

```
10 LPRINT TAB(20);"ASCII EQUIVALENTS TABLE"
15 LPRINT:LPRINT:LPRINT
20 FOR I = 33 TO 126
30 LPRINT I;TAB(6);CHR$(I)
40 NEXT
```

NOTES

See the ASCII Equivalents Table on the next page to view the Interact character set.

ASCII EQUIVALENTS TABLE

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

CLEAR

CLEAR is a statement that can be used in either direct or indirect mode to set all variables equal to zero or to allocate memory for string handling. It has the form

```
CLEAR[n]
```

where:

n	is an optional numeric constant which, if included, defines the number of bytes of memory to be reserved for strings. If CLEAR is used without the argument n, all variables are reset to zero and the amount of space allocated for string handling returns to the default, 50 bytes.

CLEAR is normally placed at the beginning of a program that needs more string space than the default of 50 bytes. If used, it should be placed in the program before any DIM statements, because, in addition to setting all variables to zero, CLEAR also "undimensions" all arrays. Note that as you allocate more memory for string handling, the amount of memory available for programming decreases.

CLEAR stays at the value set until it is respecified by another CLEAR statement, or until BASIC is reinitialized by reloading it. The NEW statement does not reset CLEAR to the default, nor does the RESET-R restart sequence. Therefore, if you use a CLEAR statement in one of your programs, remember to reset it in direct mode before running other programs, as they may not run unless they also contain a CLEAR statement.

If you try to save more string data than for which space is allocated, an "?OS ERROR" will result. This happens because you are trying to save more text than the program allows. Correct the error by using the CLEAR statement to allocate more bytes for string storage.

EXAMPLE

Type the following statements in direct mode to see the result of the CLEAR statement:

```
?FRE(0)
4698
CLEAR(2500)
?FRE(0)
2198
?FRE("A")
2500
CLEAR
?FRE(0)
4698
?FRE("A")
50
```

NOTES

Use the ?FRE("A") statement to find out how many bytes of memory are allocated for string handling.

You cannot CLEAR more memory than is available.

There are actually more than 4698 bytes available for programming, because BASIC by default allocates 50 bytes for strings. If you type CLEAR(0), then type the ?FRE(0) statement, you will see that BASIC returns available memory of 4748 bytes. Use this with caution, however, as it requires that the program have absolutely <u>no</u> string handling.

CLOAD

CLOAD is a direct mode command or indirect mode program statement that allows you to load a program or data array into memory from tape. CLOAD can take three forms:

```
CLOAD

CLOAD "name"

CLOAD*a
```

where:

name	is the name, enclosed in quotes, you assigned to the program when you saved it on tape with the CSAVE command. "name" is optional on CLOAD, even if you saved the program with a name. However, if you have more than one program stored on a tape, using "name" tells BASIC to look for and load only the program with that name. If a program with that name cannot be found, the tape will continue to run, and you will have to use the RESET-R sequence to resume normal BASIC operation.
a	is a dimensioned array name into which an array that has been stored on tape will be read. We recommend that the array you read into be dimensioned identically to the specifications in which it was written to tape, particularly if it is multi-dimensioned, so that BASIC will handle it properly. You must specify "a" if you use CLOAD* to enter array data. BASIC always loads the full array stored on tape when you use the CLOAD* command.

EXAMPLE

To write a numeric array to tape, you could use a sequence as follows:

```
10 DIM A(25)
20 FOR I=1 TO 25:A(I)=I:NEXT
30 CSAVE*A
```

Then, to read the data back in, your program might be:

```
10 DIM A(25)
20 CLOAD*A
30 FOR I=1 TO 25:PRINT A(I):NEXT
```

NOTES

CLOAD turns the tape motor on. For reading in numeric arrays, the tape should be properly positioned before CLOAD is issued. You can use the REWIND command for tape positioning. The READ cassette button should be depressed before a carriage return to execute CLOAD is given.

CLOAD -- NOTES

CLOAD cannot be used to load string data from tape with the form CLOAD*A$. BASIC stores information, whether textual or numeric, on tape as numeric data. You must therefore have a routine in your program that converts string data to numeric data to CSAVE* it on tape, and another routine to convert the numeric data back to string form before it can be reused by the program. See Chapter 4 and the ASC function for an example of such a routine.

CLS

CLS can be used as a direct mode command or a program statement. In both cases, CLS clears the TV screen of any information currently there. In direct mode, the BASIC "OK" prompt appears at the bottom of the screen immediately after the screen is cleared. When CLS is used within a program, the screen remains blank until another program statement causes something to be output.

In direct mode, CLS may be followed by a WINDOW command to aid in graphics development. CLS is also frequently used as one of the first instructions in a program so that the program begins execution with a fresh screen.

EXAMPLE

```
10 CLS
20 PRINT"WHAT'S YOUR NAME"
30 INPUT N$
35 CLS
40 FOR I=1 TO 100
50 C=RND(1)*8
60 OUTPUT N$,35,40,C
70 NEXT
80 GOTO 10
```

NOTES

Alternate methods for clearing the screen with more visual variety include:

- PLOT 1,1,0,112,77 to wipe the screen clear from left to right
- PLOT 1,1,1,112,77:PLOT 5,5,0,102,68 to wipe from left to right in two different colors
- **Vector Graphics Subroutines** for various other wipe effects, such as windshield wiper wipes, circular wipes, diagonal wipes, triangle wipes

COLOR

Your Interact has eight colors available for information display and animation. However, only four of the eight colors can be in use at any one time. The COLOR statement lets you select the color set you want to use in direct mode or your programs. It has the form

```
COLOR color0,color1,color2,color3
```

where:

color0	is one of the eight available colors, referenced by color number (see below). The color defined in this position of the color set determines the background color of the screen.
color1	is one of the eight colors, as above. The color in this position in the color set is the color in which program line numbers appear when a program is listed. color1 can also be referenced from a PLOT or OUTPUT statement.
color2	is one of the eight available colors, as above. The color in this position in the color set is not used unless referenced by a PLOT or OUTPUT statement.
color3	is one of the eight available colors, as above. The color in this position of the color set is the color in which your program lines appear when the program is listed. color3 is also the color in which the BASIC "OK" prompt and error messages appear. color3 can be referenced with the PLOT or OUTPUT statement.

You can select from the following colors to specify the color set:

0 - BLACK
1 - RED
2 - GREEN
3 - YELLOW/ORANGE
4 - BLUE
5 - MAGENTA
6 - CYAN (LIGHT BLUE)
7 - WHITE

Any of the colors defined in the color set can be used for display with an OUTPUT or PLOT statement by referencing its position in the color set, as shown in the example below.

EXAMPLES

```
10 CLS:COLOR 0,6,3,1
20 OUTPUT"GREETINGS",30,60,1
30 OUTPUT"FROM",42,50,3
40 OUTPUT"MICRO VIDEO",25,40,2
```

COLOR -- EXAMPLES

```
10 CLS:COLOR 0,6,3,1
20 OUTPUT"GREETINGS",30,60,1
30 OUTPUT"FROM",42,50,3
40 OUTPUT"MICRO VIDEO",25,40,2
50 FOR C=1 TO 20
60 COLOR 7,0,2,4
65 FOR P=1 TO 50:NEXT
70 COLOR 0,7,2,4
75 FOR P=1 TO 50:NEXT
80 NEXT
```

NOTES

The COLOR statement sets the color registers, The color register shares a bit with the tape motor control function. Therefore, when you use a COLOR statement, the tape motor is automatically shut off. See chapter 3 (page 3-2) for information on colors and tape motor control.

The color set established with the COLOR statement is reset to the default when the RESET-R sequence is used to restart BASIC. The default color sets are:

COLOR 4,3,0,7	- Level II and RS232 BASIC
COLOR 0,3,4,7	- Microsoft 8K BASIC

We suggest you establish your program colors early in the program and to change them within program execution with regard to aesthetic effects.

COS

COS is a trigonometric function that computes the cosine of an angle, given the angle in radians. It has the form

COS(n)

where:

n	is an angle, expressed in number of radians, for which the cosine is to be returned. n can be a numeric constant, variable, or expression.

EXAMPLE

```
10 PRINT:PRINT"ANGLE IN DEGREES"
20 INPUT DE
30 R=DE/57.2958
40 PRINT COS(R)
50 GOTO 10
```

NOTES

COS is one of several trigonometric functions intrinsic to BASIC. Others are SIN, ATN, and TAN.

CSAVE

CSAVE is a command that can be used in direct mode or from within a program to save a program or numeric data array onto tape. CSAVE can take three forms:

```
CSAVE

CSAVE "name"

CSAVE*a
```

where:

"name"	is an optional name, entered in quotation marks, that you can assign to a program when you store it on tape. Only the first five characters of "name" are written to tape, although the name can be as long as you wish. If a program is saved on tape with an assigned name, it can be reloaded later with or without specifying the name.
a	is a dimensioned array from which data is to be stored on tape. BASIC always saves the full array named 'a' on tape with the CSAVE command.

EXAMPLE

Examples of using CSAVE to store data on tape, then read it back into memory can be found in chapter 4 and with the CLOAD entry in this chapter.

NOTES

CSAVE <u>cannot</u> be used to save string data with the form CSAVE*A$. BASIC stores information, whether textual or numeric, on tape as numeric data. You must therefore have a routine in your program that converts string data to a numeric representation to CSAVE the data on tape. You must also have another routine to convert the numeric data back to string form to reuse the data after reading it back into memory. See the ASC function and chapter 4 for examples of this operation.

CSAVE turns the tape motor on. For writing programs or numeric arrays to tape, the tape should be properly positioned before the CSAVE command is given. Use the REWIND command for tape positioning. The READ and WRITE cassette buttons should be depressed before executing CSAVE.

Always use the REWIND command to position the tape forward slightly before issuing a CSAVE. This will ensure that all the leader tone required to read the program back in successfully will be recorded. If you are saving a program or array data on a tape that already has other information stored on it, use the REWIND command and the READ cassette button for proper positioning of the tape.

DATA

DATA statements provide a convenient, space-efficient method of entering constant or specific (table or plotting) data into a program. DATA statements may be used anywhere within the program because they are never executed. The READ statement is used to reference the data items in DATA statements. DATA statements have the form

```
DATA item[,item,item,...]
```

where:

item	is a string or numeric data value that is to be entered into the program. More than one data value can be included in a single DATA statement. The data values are entered separated by commas. String data may or may not be enclosed in quotation marks. You must use quotes, however, if the string contains an embedded comma, for example, "Ann Arbor, Michigan", since the comma is the delimiter in DATA statements.

DATA statements are particularly useful when you have a lot of constant data to enter. It is less time-consuming and more space-efficient to enter the data with DATA statements and the READ statement than to store the data points in dimensioned arrays.

Data values in the DATA statements are not accessible to the program until they have been read into variables with the READ statement. The first READ statement executed starts with the first data value on the first DATA statement in the program and reads subsequent data values sequentially. READ has an internal pointer that "remembers" where it left off. Subsequent READ statements begin reading data with the value following the last data value entered with the previous READ statement. After a data value has been READ in, the value will not be reused unless a RESTORE statement is given to reset the READ internal pointer.

EXAMPLES

In the following example, the number of data points entered with the READ statement is controlled by the iterations in the FOR...NEXT loop.

```
10 CLS
20 Y=60
30 FOR NL=1 TO 4
40 READ XL,XH
50 Y=Y-2
60 FOR X=XL TO XH
70 PLOT X,Y,2
80 NEXT X
90 NEXT NL
100 DATA 10,90
110 DATA 20,80,30,70,40,50
```

DATA -- EXAMPLES

Note that one or more data points can be entered with a single READ statement.

The following example requires Microsoft 8K BASIC:

```
10 CLS
20 Y=60
30 FOR NL=1 TO 4
40 READ XL,XH
50 Y=Y-2
60 PLOT XL,Y,2,XH-XL+1,1
70 NEXT NL
80 DATA 10,90
90 DATA 20,80,30,70,40,50
```

Numeric data can be entered as string data, as shown in the example below. String data cannot, however, be entered in place of numeric data.

```
10 DATA"ANN ARBOR, MI",48104
20 DATA"CORONADO, CA","92118","MINNEAPOLIS, MN",55409
30 FOR I=1 TO 3
40 READ A$,B$
50 PRINT A$;" ";B$
60 NEXT
```

NOTES

If you try to READ in more data values than appear on the DATA list, a "?OD ERROR" will result.

Other examples of using the DATA and READ statements to enter data are included throughout this chapter. See READ, RESTORE, LOG, and POS for other examples that use this data entry method. Also, consult chapter 6 for further information.

DEF

DEF statements let you define your own functions within a program for operations used multiple times. DEF might be considered a one-line subroutine with a built-in RETURN statement. DEF statements take the form

```
DEF FNabc(n) = expression
```

where:

abc	is a user-defined function name that must begin with the characters 'FN'. You can choose the rest of the function name as appropriate for what the function does.
n	is a numeric variable which acts as the argument on which the function is to be performed.
expression	is the operation that is to be performed whenever the function FNabc is called within the program.

You must be able to summarize the function you define with a DEF statement in a single statement. Complex operations cannot be defined as functions; use subroutines instead.

DEF statements can be used in indirect mode only.

EXAMPLES

The following classic example uses a user-defined function to round fractions of cents to the nearest penny.

```
10 DEF FNPT(ZQ)=INT(ZQ*100 +.5)/100
20 PRINT
30 INPUT"AMOUNT";ZQ
40 PRINTFNPT(ZQ)
50 GOTO 20
```

The next example shows how you might use the DEF function to calculate the sales tax for any given amount. In this example, sales tax is 4%.

```
10 CLS
20 DEF FNTX(A)=A*.04
30 INPUT"SALE AMT";A
40 TX=FNTX(A)
50 PRINT"TAX IS:";TX
60 PRINT
70 GOTO 30
```

DIM

The DIM statement allocated storage for arrays (lists or tables) of information to be referenced and used later in the program. DIM statements typically appear near the beginning of the program. A DIM statement must be given for any array of information that has more than 10 data elements; arrays with fewer than 10 elements may be automatically dimensioned by BASIC. Any single array should be dimensioned only once in the flow of any program. An array must be dimensioned before information can be stored in it or referenced from it. The DIM statement has the form

```
DIM a(n[,q,r,s,t]
```

where:

a	is the variable name under which items in the array will be stored and later referenced by subscript. 'a' can be a string or numeric variable name, to dimension a string or numeric array.
n[,q,r,s,t]	identify's the maximum subscript for each dimension in the array. An array can have up to 5 dimensions. The dimensions are generally used as follows:

n - rows
q - columns
r - layers
s - undefined, possibly time
t - undefined

'n' (q,r,s,t) can be a numeric constant, variable, or expression. Its value specifies the highest subscript that will be used to reference that particular dimension of the array. As the first subscript location of any array can be 0, there is actually one more location per dimension than specified by the maximum subscript value. For example, DIM A(20) allocates 21 locations for array A, which are referenced as A(0) through A(20) to return the data elements stored in the array locations.

You can dimension more than one array with a single DIM statement. For example,

```
DIM A(20),BD$(12,I),CN(14,X+5,3)
```

allocates storage for three arrays--two numeric (A and CN) and one string (BD$). Array A has one dimension that can be subscripted to a maximum value of 2$\cancel{0}$! BD$ has two dimensions--twelve rows and I columns. Note that variables can be used as the maximum subscript specifier in a DIM statement. This allows maximum flexibility in sizing the array to fit the problem during program execution. The value of the variable used as a subscript can be supplied in a number of different ways, but the value of the variable must be defined before the DIM statement is executed.

DIM

DIM

You can dimension arrays up to the space available. If you try to dimension an array larger than the available RAM or more arrays than for which RAM is available, an "?OM ERROR" will result.

EXAMPLES

```
10 DIM A(15)
20 FOR I=1 TO 15
30 INPUT A(I)
40 NEXT
45 PRINT"ARRAY CONTAINS:"
50 FOR I=1 TO 15
60 PRINT A(I)
70 NEXT
```

The next example illustrates the use of a variable in place of a numeric constant in the DIM statement. This feature lets you dimension as appropriate to the size of the problem, rather than setting up a huge array which may contain unused subscripts. In the following example, the program user would be requested to enter a value that would be used as the maximum subscript for the array(s).

```
10 PRINT"NUMBER OF"
20 INPUT "STORES";S
30 DIM SALES(S),EMP(S)
        .
        .
        .
```

Using variables instead of constants can also be useful for performing mathematical operations on a matrix of variable size. You can set the size and dimensions of the program during program execution, as well as index the data values in the arrays. For example:

```
10 PRINT"MATRIX SIZE"
20 INPUT N,M
30 DIM A(N,M)
        .
        .
        .
```

Data items stored in an array are referenced according to their dimension and subscript position within that dimension. In the following example, the program sets up a 3 by 4 (12 slot) matrix as shown below on the right, computing the values of each subscripted position by indexing through the array with FOR...NEXT loops. In running this program, to find out the value in any cell in the matrix, enter the subscripts for the cell's position (I,J).

DIM

```
10 DIM A(3,4)
20 FOR I=1 TO 3
30 FOR J=1 TO 4
40 A(I,J)=10*I+J
50 NEXT:NEXT
60 INPUT"WHAT CELL";I,J
70 PRINTA(I,J)
80 PRINT:GOTO 60
```

	A(*,1)	A(*,2)	A(*,3)	A(*,4)
A(1,*)	11	12	13	14
A(2,*)	21	22	23	24
A(3,*)	31	32	33	34

The DIM statement is also used to dimension arrays for string data. Strings stored in an array are, like numeric values, referenced according to their position in the array by subscript.

```
10 DIM NM$(15)
20 DATA JONES,SMITH,BROWN,SIMPSON,PETERS
30 DATA HENRY,WATSON,MARTIN,SANDBURG,WATERS,FRANK
40 DATA ROSE,DAVIDSON,CHURCH,FORD
50 FOR I=1 TO 15
60 READ NM$(I)
70 NEXT
80 PRINT:PRINT"STUDENT NUMBER"
90 INPUT S
100 IF S<1 OR S>15 THEN 90
110 PRINT"NAME: ";NM$(S)
120 GOTO 80
```

NOTES

For more information on arrays and their dimensions, see the discussion on arrays in chapter 2 (page 2-28).

Largest found 1-29-83

Single array - DIM Q(1343)

Double array - Dim Q(31,31) = 951

Dim Q(51,19) = 969

END

The END statement marks the final line of a program. The use of END is optional if the highest numbered line in a program is also the last executable statement. This allows subroutines to be placed on lines with higher numbers than the last executable line in the main program logic. It has the form

```
END
```

The END statement is used in indirect mode only; it has no function in direct mode operation.

When BASIC encounters an END statement within a program, it halts program execution and responds "OK". The program variable status is not affected by program termination with END, that is, the variables are not reset to zero.

EXAMPLE

```
10 INPUT A
20 GOSUB 100
30 PRINT A
40 END
100 A=A*30/5
110 RETURN
```

EXP

EXP is an arithmetic function that computes the anti-logarithm to the base e (2.71828) of the argument. It has the form

EXP(n)

where:

n	is a numeric constant, variable, or expression that indicates the exponentiation to be performed. It specifies the power to which e (2.71828) is to be raised to produce a value. To obtain the value of V raised to the Xth power, you could use the statement
	
	V = 2.71828↑X
	
	However, it is simpler and more RAM-conscious to use the EXP function:
	
	V = EXP(X)

EXAMPLES

The following two programs perform the same function, computing V (as above), given the value of X. In the first example, the exponentiation value is entered from the keyboard. In the second example, DATA statements are used to input the values.

```
10 FOR I=1 TO 5
20 INPUT"EXPONENT";X
30 PRINT X;EXP(X):PRINT
40 NEXT
```

```
10 FOR I=1 TO 5
20 READ X
30 PRINT X; EXP(X)
40 NEXT
50 DATA -1,0,1
60 DATA 10,20
```

NOTES

To solve for X, given a value of V, use the LOG function.

FIRE

FIRE is an arithmetic function that returns the status of the fire button on either the left or right entertainment controller, as specified by the argument. It has the form

FIRE(n)

where:

n	is either 0 or 1. FIRE(0) checks the status of the fire button on the left controller. FIRE(1) checks the fire button on the right controller. For both controllers, FIRE returns a value of 0 is the fire button is depressed, and a value of 1 if it is not.

Because the FIRE function by nature requires a test, it is used within IF conditional testing statements.

EXAMPLES

```
10 IF FIRE(0)=0 THEN PRINT"LEFT PRESSED"
20 IF FIRE(1)=0 THEN PRINT "RIGHT PRESSED"
30 GOTO 10
```

In the following example, if the fire button on the left controller is depressed, a colored rectangle appears on the screen in a random (x,y) position. This example was written in Microsoft 8K BASIC. To run a similar program using Level II BASIC, change line 120 to read: 120 PLOT X,Y,2

```
5 CLS
10 IF FIRE(0)=0 THEN GOSUB 100:GOTO 10
20 GOTO 10
100 X=RND(9)*110
110 Y=RND(1)*74
120 PLOT X,Y,2,5,3
130 FOR Q=1 TO 30:NEXT
140 RETURN
```

NOTES

A more detailed example of using the FIRE button to control a game program, see chapter 5. FIRE is also used in the example program in the JOY entry in this section.

FOR

The FOR statement establishes a program looping sequence in which statements on line numbers between the originating FOR and ending NEXT statements are executed as many times as indicated in the FOR statement. It has the form

```
FOR a = exp1 TO exp2 [STEP exp3]
```

where:

a	is the iteration variable, which control the number of times the statements in the FOR...NEXT loop are executed. Its starting value is equal to exp1, and its maximum value is equal to exp2. 'a' is incremented by 1 each time the end of the loop is reached, unless the optional STEP exp3 option is used to specify alternate incrementation. If the STEP option is included, 'a' is incremented by the value of exp3 at the end of each loop iteration.
exp1	is the starting value of the iteration variable. exp1 may be a numeric constant, variable, or expression.
exp2	is the maximum value to which the iteration variable can be incremented. At the end of each loop iteration, signalled by the NEXT statement, the value of the iteration variable is incremented by the STEP size and compared against exp2. If the incremented value of 'a' is less than or equal to exp2, all statements within the loop are executed again. If the incremented value of 'a' is greater than exp2, looping ends, and the statement immediately following the NEXT statement is executed. (Note that if a negative STEP value is used, the reverse occurs--looping ends if the incremented variable is less than exp2.) exp2 may be a numeric constant, variable, or expression.
exp3	is an optiona value that specifies non-default incrementation of the iteration (looping) variable. If exp3 is used, it must be preceded by the STEP keyword. STEP exp3 causes the iteration variable to be incremented by the value of exp3 each time the loop is executed, rather than the default, STEP 1. exp3 may be a positive or negative numeric constant, variable, or expression. If a negative STEP value is used, the program decrements the value of the iteration variable each time the loop is executed. In this case, exp1 must be greater than exp2.

EXAMPLES

In its simplest form, the FOR...NEXT loop performs a single operation a specified number of times.

FOR -- EXAMPLES

```
10 FOR J=1 TO 5
20 PRINT "HI"
30 NEXT
```

In Level II BASIC, FOR...NEXT loops are used to draw lines on the screen. Note that although this example will work in Microsoft 8K BASIC, lines 10 through 30 would usually be replaced with the single statement PLOT 1,60,2,117,1

```
5 CLS
10 FOR X=1 TO 117
20 PLOT X,60,2
30 NEXT
```

With the STEP option, the FOR...NEXT loop can draw a dotted line:

```
10 CLS
20 FOR X=1 TO 117 STEP 2
30 PLOT X,60,2
40 NEXT
```

If you have Microsoft 8K BASIC, change line 30 to read PLOT X,1,2,1,77 to fill your screen with striped lines.

You can use the STEP option with a negative value to draw from right to left. Note that in this case, the starting value of the iteration variable is greater than the ending value.

```
10 CLS
20 FOR X=117 TO 1 STEP -1
30 PLOT X,60,2
40 NEXT
```

FOR...NEXT loops can be "nested" or embedded within other FOR...NEXT loops. Statements can also be executed between the end of a nested loop and the next iteration of the outer loop, as illustrated below.

```
10 CLS
20 FOR X=1 TO 50
30 FOR Y=60 TO 50 STEP -2
40 PLOT X,Y,2
50 TONE X,Y
60 NEXT Y
70 TONE 100,100
80 NEXT X
```

FOR

NOTES

The value of the iteration variable may be sensed within a loop, but should not be changed within the loop, or an "?FC ERROR" may result.

Do not GOTO a line within a FOR...NEXT loop, as a "?NF ERROR" will result. If you want to enter a FOR...NEXT loop from another part of your program, enter it at the originating FOR statement.

FRE

FRE is a numeric function that indicates the number of bytes of RAM still available for programming and data storage or the amount of unused RAM remaining for string handling. It is generally used with the PRINT statement during program development to determine the amount of RAM the program has consumed. FRE serves no other purpose. It has the form

```
FRE(x)
```

where:

x	can be a numeric constant or variable or a string constant or variable enclosed in quotes. If 'x' is a numeric constant or variable, FRE returns the number of bytes of RAM remaining for program development, less the space allocated for string handling. If 'x' is a string constant or variable enclosed in quotes, the number of bytes remaining for string handling is returned. 'x' is a dummy variable--the value returned will be the same regardless of whether you specify FRE(0) or FRE(A), FRE("HELLO") or FRE("A").

EXAMPLE

If you enter the following commands in direct mode immediately after loading the BASIC interpreter, you should see the following:

```
?FRE(0)
4698
?FRE("A")
50
```

The default of 50 bytes for string handling can be changed with the CLEAR statement.

```
CLEAR(500)
?FRE("Q")
500
?FRE(6)
4248
```

GOSUB

The GOSUB statement transfers program logic control to a subroutine beginning at a specified line number. When the RETURN statement at the end of the subroutine is encountered, program control returns to the statement immediately following the initializing GOSUB. It has the form

```
GOSUB line
```

where:

line	is the line number at which the subroutine begins.

Subroutines called by a GOSUB statement are generally used when you want to perform an operation of a number of times from different places in the program code. Setting up a subroutine to perform these types of operations helps you avoid redundant code that consumes unnecessary RAM. It also makes program changes simpler. Subroutines <u>must</u> be concluded with a RETURN statement.

EXAMPLE

```
10 CLS
20 FOR N=1 TO 100
30 X=RND(1)*111+1
40 Y=RND(1)*76+1
50 GOSUB 110
60 NEXT N
70 OUTPUT"HELLO EARTHLINGS",10,60,1
80 A$=INSTR$(1)
90 RUN
100 REM PLOT X,Y IN RANDOM COLOR
110 C=1+RND(1)*3
120 PLOT X,Y,C
130 TONE X,Y
140 RETURN
```

NOTES

Although you might begin a subroutine with a REM statement, you should generally GOSUB to the line number at which the statements in the subroutine actually begin. Then, if space limitations require that you remove REM statements, you can avoid having to make extensive corrections in the GOSUB lines of your program to correct program logic.

Other examples of using subroutines appear throughout this section. See IF, JOY, ON, POT, and RESTORE. Also consult chapter 7, which is devoted entirely to a discussion of subroutines.

GOTO

The GOTO statement causes program control to be transferred completely to the first statement on the specified line number. It has the form

```
GOTO line
```

where:

line	is the line number to which program control transfers. The first statement on the specified line is executed and logical execution of subsequent lines continues.

EXAMPLE

The following example illustrates GOTO in its simplest form. This small program is an infinite loop--execution will continue forever, or until you type a Control-C to terminate program execution.

```
10 PRINT"   MICRO VIDEO"
20 PRINT
30 GOTO 10
```

NOTES

GOTO can be used in direct mode to start program execution at a line other than the first line in the program. If program execution is started with a GOTO statement, variables are not initialized to zero as with the RUN command. You can therefore restart program execution at some point in the program with variables set as they were by the previous program execution. This feature can be extremely useful for program debugging. See chapter 2 (page 2-16) for an example of this.

Examples of programs which contain GOTO statements are used extensively throughout this manual.

IF

The IF statement is used to determine whether a given condition is true or false. If the test shows that the condition is true, then subsequent statements on the IF statement line are executed. If the condition tests false, other statements on the IF statement line are not executed, and program control passes to the first statement on the next higher line number. The IF statement has two forms:

```
IF condition GOTO line

IF condition THEN statement1[:statement2:...]
```

where:

condition	is one or more expressions that provide the basis for a relational test. You can test for equality, non-equality, greater than, etc., using the various relational and BOOLEAN operators (AND and OR). Operands can be numeric or string variables, constants, and arithmetic expressions.
line	is the line number of the program to which program control will be transferred if the given condition tests true.
statement1,...	are program statements that will be executed sequentially if the given condition tests to be true. In order for statements to be executed as a result of the given condition, the statements must be separated by colons and be placed on the same line as the conditional IF statement.

EXAMPLES

The following example illustrates the IF...GOTO construction.

```
10 CLS
20 OUTPUT"PLAY AGAIN?",20,60,1
30 A$=INSTR$(1)
40 IF A$="Y" GOTO 100
50 IF A$="N" GOTO 300
60 GOTO 30
100 PRINT"I WIN !"
110 GOSUB 1000
120 GOTO 10
300 PRINT"WHATSA MATTER?"
310 PRINT"CHICKEN?"
320 GOSUB 1000
330 GOTO 10
1000 REM -- PAUSE LOOP
1010 FOR Q=1 TO 500
1020 NEXT Q
1030 RETURN
```

IF -- EXAMPLE

The next example tests two numeric expressions with the AND operator and illustrates the IF...THEN construction of conditional testing.

```
10 CLS
20 COLOR 0,1,3,7
30 OUTPUT"HELLO",40,50,1
40 WINDOW 18
50 INPUT "COLOR";C
60 IF C>-1 AND C<8 THEN COLOR 0,C,3,7:GOTO 50
70 PRINT"BAD CHOICE"
80 GOTO 50
```

NOTES

Multiple conditions can be tested with the IF statement. See AND and OR in this chapter for examples. Other examples illustrating the use of IF can be found in a variety of other places in this chapter as well. Also, consult chapter 2 (page 2-20) for an introductory discussion of conditional relationships.

INPUT

The INPUT statement is used to request user input from the keyboard. An INPUT statement produces a question mark (?) prompt on the screen and waits for user input of string or numeric data. To terminate an INPUT statement, the user must press the carriage return (CR) key to enter the value. INPUT can be used for entry of either string or numeric data. It has two forms:

```
INPUT a[,a,...,a]

INPUT "string";a
```

where:

a	is a numeric or string variable name into which the user input is to be stored for future use. If a numeric variable is specified, the value entered must be numeric, or the message "?REDO FROM START" appears, and the INPUT '?' prompt is repeated. Numeric values can, however, be entered into a string variable.
"string"	is an optional string constant or string variable that is to be output on the screen on the same line as the '?' prompt. This allows you to specify what is to be entered by the user, rather than having the question mark prompt appear alone, which can be confusing, especially to the novice user. If "string" is included, it must be entered in quotes and followed by a semi-colon to separate it from the input variable.

EXAMPLE

```
5 CLS
10 INPUT "NAME";NM$
20 CLS
30 OUTPUT NM$,10,50,1
40 FOR C=1 TO 7
50 COLOR 0,C,3,7
60 FOR Q=1 TO 100:NEXT
70 NEXT C
80 GOTO 5
```

NOTES

If an INPUT statement requests string data and the string data to be entered contains an embedded comma (e.g., Ann Arbor, Michigan), the string entered in response to the INPUT query should be placed in quotation marks. Otherwise, BASIC ignores everything to the right of the comma and prints the message "?EXTRA IGNORED".

See chapters 2 and 4 for further discussion and examples of data entry via INPUT statements.

INSTR$

INSTR$ is a string handling function that accepts user input from the keyboard. Unlike the INPUT statement, INSTR$ does not require that the user enter a carriage return to enter the information into memory. It also allows you to specify the length of the string to be entered. The INSTR$ function halts the program and waits for user input before executing any subsequent program statements. The screen remains unchanged when the INSTR$ function is executed--no prompt of any kind automatically appears. This function has the form

INSTR$(n)

where:

n	is an argument that specifies the number of characters the INSTR$ function is to read from the keyboard before executing subsequent program lines. 'n' can be a numeric constant, variable, or expression.

EXAMPLE

In its simplest form, INSTR$ waits for depression of a single key before continuing program execution.

```
5 CLS
10 FOR X=1 TO 100
20 PRINT X
30 GOSUB 100
40 NEXT
50 STOP
100 TONE X,100
110 A$=INSTR$(1)
120 RETURN
```

NOTES

As no prompt normally appears on the screen to accompany the INSTR$ function, we suggest you output a tone or two just before the INSTR$ function is executed to let the user know that response is expected.

Use of the INSTR$ function to enter data is discussed in several other places in this manual. In particular, see chapters 2 and 4 (pp. 2-17 and 4-4).

INT

INT is an arithmetic function that returns the largest whole number (integer) that is less than or equal to the value given in the argument. It has the form

```
INT(n)
```

where:

n	is the real number for which is to be "integerized". 'n' may be a numeric constant, variable, or expression. Function calls, such as RND, can also act as arguments to the INT function.

With positive arguments, INT rounds down (truncates) the number to its integer value. With negative numbers, INT returns the next smallest negative integer. For example,

```
?INT(314.5)
314
?INT(-314.5)
-315
```

EXAMPLE

```
10 CLS
20 COLOR 0,1,3,7
25 FOR I =1 TO 50
30 R=RND(1)*3+1
35 C=INT(R)
40 OUTPUT C,56,12,3
45 PLOT 20,20,C,80,30
50 OUTPUT C,56,12,0
55 NEXT
```

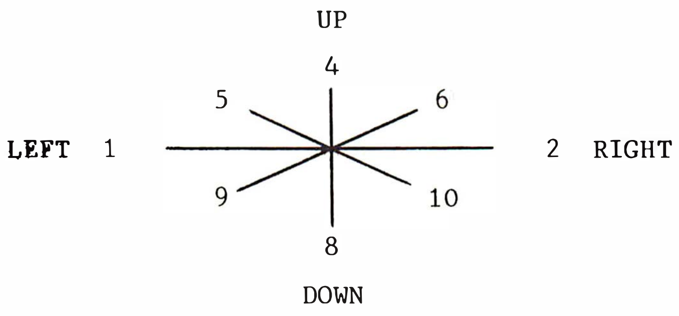
JOY

JOY is an arithmetic function that reads input from the left or right joystick lever. It has the form

JOY(n)

where:

n	can be either 0 or 1. If 'n' is 0, JOY reads the joystick on the left entertainment controller. If 'n' is 1, JOY reads the joystick lever on the right controller. For either joystick, JOY returns values for the various positions as shown below:



A value of zero is returned if the joystick lever is not moved.

EXAMPLE

The following example lets you draw on the TV screen with the left joystick. You can depress the fire button to move the "pen" to a different point on the screen without drawing a line.

```
10 CLS
20 X=55:Y=35
30 PLOT X,Y,1
40 J=JOY(0)
50 IF J=0GOTO30
60 GOSUB200
70 IF FIRE(0)=1 GOTO 30
80 PLOT X,Y,0
90 GOTO 40
200 IFJ>=4ANDJ<=6THENY=Y+1
210 IFJ>=8ANDJ<=10THENY=Y-1
220 IFJ=2ORJ=6ORJ=10THENX=X+1
230 IFJ=1ORJ=5ORJ=9THENX=X-1
240 RETURN
```

NOTES

See chapter 5 for a detailed example of using joystick input to control a game program.

LEFT$

LEFT$ is a string handling function that isolates a specified number of characters in a string, beginning with the leftmost character in the string. It has the form

LEFT$(a$,n)

where:

a$	is the string variable name containing the string from which characters are to be isolated. (a$ could also be a string constant, but using LEFT$ in that case seems rather silly.)
n	is the number of characters to be isolated from the string, starting with the leftmost character. 'n' must be greater than or equal to zero. If the string is shorter than 'n' characters, all characters in the string are isolated.

Use of the LEFT$ function concept is used within the Troll Hole Adventure, a machine language program. In that program your commands (LIGHT LAMP, PICKUP SHOVEL, etc.) are read as only the first three characters. You can also enter them in that abbreviated form. In fact, if you specify additional characters with the command, they are simply ignored.

You can use this concept within your own BASIC programs to achieve similar effects. Use a construct such as

C$ = LEFT$(I$,3)

to limit C$ to the first three characters of the command word.

EXAMPLE

```
10 CLS
20 DATA JONES,SMITH,REYNOLDS,PARKER,FARMER
30 DATAPETERSON,CARTER,WILLIAMS,BLACK,MCNARY
40 FOR I=1 TO 10
50 READ A$
60 B$= LEFT$(A$,5)
70 PRINT  B$
80 PRINT:NEXT
```

LEN

LEN is an arithmetic function that counts the length of a string in characters (or bytes). It is most commonly used to position information on the screen, for centering, for example. It has the form

LEN(a$)

where:

a$	is the string for which length in characters is to be returned.

EXAMPLE

```
10 CLS
20 WINDOW 24
30 Y=70
40 PRINT:INPUT"NAME";A$
50 J=LEN(A$)
55 IF J>17 GOTO 40
60 OUTPUT A$,56-3*J,Y,1
70 Y= Y-6
80 GOTO 40
```

Remember to reset the WINDOW to 77 after running this example.

LET

LET is a statement that defines an arithmetic expression or numeric constant as equal to a numeric variable or a string expression or constant as equal to a string variable. The expression is evaluated, and the results stored in the named variable. It has the form

```
LET a = exp
```

or, more commonly,

```
a = exp
```

where:

a	is the string or numeric variable in which the result of exp is to be stored.
exp	is an expression which, when evaluated, produces a data value to be stored in the named variable, 'a'. exp can be a string or numeric constant, variable, or expression. If exp is an arithmetic expression, the necessary calculations are performed according to the given syntax, and the result stored in the named variable.

LET is probably the most commonly and least commonly used of all statements in BASIC. It's most commonly used because it's used to assign the variables that are the basis of any program. It's least commonly used because the LET keyword is optional. Although a program will generally contain a number of assignment statements in which LET is implied, the keyword LET is not actually included in the statement in most cases. LET was included in BASIC for compatibility with other BASIC "dialects" and is usually omitted from assignment statements. The following two statements appear identical to BASIC:

```
LET A = 12/5.5

A = 12/5.5
```

Since you're likely to run into memory limitations as your programs increase in size, why use an extra word in your code that's redundant? Save those extra couple bytes each time you do an assignment by using the shorter form.

EXAMPLE

```
10 CLS
20 A=5
30 B= 7
40 C = 12
50 PRINT A+B+C
```

LOG

LOG is an arithmetic function that calculates the logarithm to the base e (2.71828) of the given argument. Given any value, LOG computes the exponent to which e must be raised to produce that value. It has the form

LOG(n)

where:

n	is the value for which the logarithm is to be calculated. 'n' can be a numeric constant, variable, or expression. It must be greater than 0 or an "?FC ERROR" will result.

EXAMPLE

The two programs below both illustrate performing the same operation, that is, computing the logarithm of a given value. The examples differ in the mode of data entry. The first example reads three values in from DATA statements within the program. The second example is an infinite loop in which data values are supplied to the program from the keyboard.

```
10 FOR I=1 TO 3
20 READ V
30 PRINTV;LOG(V)
40 NEXT
50 DATA 2.71828,10,20
```

```
10 INPUT V
20 PRINTV;LOG(V)
30 GOTO 30
```

NOTES

LOG is the reverse of the EXP function. Other arithmetic functions usable in BASIC are COS, SIN, TAN, and ATN.

LPRINT

The LPRINT statement is an RS232 BASIC instruction that outputs values to a lineprinter that is attached to the Interact through the Micro Video RS232 peripheral interface. This statement is available in RS232 BASIC only. LPRINT has the form

```
LPRINT a [;a;...]

LPRINT a [,a,...]
```

where:

a	can be

- a string or numeric constant (e.g., "MARY" or 3.14159)
- a numeric or string variable (e.g., A or C$)
- a function call (e.g., SQR(A))
- an arithmetic expression (e.g., 3*B or COS(C)/A)
- a string expression (e.g., A$+B$)

More than one variable can be placed on a single LPRINT statement. You can separate variables with either commas or semi-colons. While commas are not terribly useful for displaying information on the TV screen with the PRINT command, due to the limited number of characters per line, they are more commonly used as a separator with the LPRINT command, to output data items through the lineprinter in fields 14 characters wide. The semi-colon separator adds a leading and trailing blank to numeric data values output; it concatenates string data.

You can use the TAB, POS, and SPC functions in conjunction with the LPRINT command to produce effective, formatted reports or analyses on your lineprinter. Printed lines can be up to 80 characters in length.

LPRINT cannot be abbreviated to L? in the way that PRINT can be abbreviated to ?. You must type the entire keyword.

EXAMPLE

The program on the following page illustrates the use of LPRINT to produce a formatted, printed report. In this program we have calculated sales summary information for a three-year period and defined the format of the report (also shown). This listing and the resultant report (also shown) were produced with our RS232-equipped Interact, RS232 BASIC, and a COMPRINT 912-S lineprinter.

LPRINT -- EXAMPLE

```
10 REM-LPRINT EXAMPLE
12 Y8=11.2
13 Y9=12.5
14 Y0=42.8
16 LPRINT TAB(8);"SALES SUMMARY"
20 LPRINT SPC(4);"1978  1979  1980  1981"
40 LPRINT
50 LPRINT"QTR"
60 FOR Q=1 TO 4
70 LPRINT Q;Y8;Y9;Y0
75 Y8=2*Y8:Y9=Y9+4.2:Y0=Y0-4
80 NEXT
90 LPRINT
100 LPRINT"*** NOT FOR DISTRIBUTION ***"
110 FOR J=1 TO 5:LPRINT:NEXT
```

```
        SALES SUMMARY
    1978  1979  1980  1981

QTR
 1  11.2  12.5  42.8
 2  22.4  16.7  38.8
 3  44.8  20.9  34.8
 4  89.6  25.1  30.8

*** NOT FOR DISTRIBUTION ***
```

MID$

MID$ is a string handling function that isolates characters from the middle of a given string. It has the form

MID$(a$,n,m)

where:

a$	is the string variable containing the string from which characters are to be isolated.
n	indicates the position in the string at which character isolation is to begin, the nth character in the string.
m	is the number of characters to be isolated, starting with the nth character.

MID$ isolates the specified number of characters from the string and stores them in another string. If a string is shorter than the specified starting character (n), MID$ returns a null string. If the string is shorter than the number of characters to be isolated, all characters from the starting character on are returned.

EXAMPLE

```
10 CLS
20 DIM A$(15)
30 DATA JAMES,SIMON,PAUL,HENRIETTA,MATTHEW,DAVID
40 DATAELIZABETH,LILLIAN,PAULINE,MARIAN,JOHN,STEPHEN,RALPH
50 DATA MICHAEL,SAM
60 FOR I=1 TO 15
65 READ A$(I)
70 B$=MID$(A$(I),2,4)
90 PRINT B$
100 NEXT
```

NOTES

MID$ is one of the functions used in converting string data to its numeric representation for storage on tape. See chapter 4 and the ASC function in this chapter for details.

MID$ is also useful when you want to put several strings within a single string, then isolate the strings individually for output or other processing. The Perpetual Calendar BASIC program provides an excellent example of using MID$ for this type of operation.

NEXT

The NEXT statement defines the end of an iteration of a program loop initiated by a preceding FOR statement. The NEXT statement is required to end any FOR...NEXT loop. If omitted, a "?NF ERROR" will result. NEXT has the form

```
NEXT [a]
```

where:

a	is the iteration variable specified in the originating FOR statement. NEXT can be used with or without the iteration variable, but its inclusion makes loop completion and program logic in complex, nested FOR...NEXT constructions easier to follow.

EXAMPLE

Following is a very simple example of the NEXT statement. FOR...NEXT loops are used in examples throughout this manual. For more information about looping, see the FOR statement in this chapter or the looping discussion in chapter 2 (page 2-23)

```
10 FOR J=1 TO 10
20 PRINTJ
30 NEXT
```

NOT

NOT is an arithmetic function that returns the bitwise complement of the given argument. The bitwise complement is determined by setting each bit in the byte of the argument to its reverse position. For example, if the byte in machine code is

1	0	1	0	1	1	0	1

its bitwise complement would be

0	1	0	1	0	0	1	0

NOT has the form

NOT(n)

where:

n	is a numeric constant, variable or expression for which the bitwise complement is to be returned. 'n' can also be a function call, e.g., PRINT NOT(SQR(81)).

The bitwise complement of a number is always the negative of that number, minus 1. For example, the bitwise complement of 9 is -10; of 456 is -457; of -34 is 33; etc.

EXAMPLE

```
5 CLS
10 FOR I=1 TO 10
20 INPUT "NUMBER";A
30 PRINTNOT(A)
40 PRINT
50 NEXT
```

In the following example, NOT is used as a logical operator in the sense that the term is used in symbolic logic notation: A OR B ≡ NOT(A AND B).

```
10 CLS
20 INPUT "NUMBER";A
30 IF NOT(A>10) THEN PRINT"NOT MORE THAN 10":PRINT:GOTO 20
40 PRINT"MORE THAN 10"
50 GOTO 20
```

ON

ON is a multiple branching statement that transfers program control to a line number specified in a list of line numbers within the statement, based on the value of the ON variable. ON has two forms:

```
ON a GOTO line1,line2,...,linen

ON a GOSUB line1, line2,...,linen
```

where:

a	is a numeric variable, the value of which determines the line number in the list following GOTO or GOSUB to which program control is passed. 'a' should have a value of 1 through n, where 'n' is the number of line numbers in the GOTO or GOSUB list. 'a' must have a positive value; if 'a' is negative, an "?FC ERROR" results. If 'a' is zero or greater than the number of line numbers in the list, no transfer of program control to a line in the list occurs. Program execution continues with the next higher numbered program line in that case.
line1...linen	is the list of line numbers following the GOTO or GOSUB statement. Program control transfers to one of the line numbers on the list, based on the value of 'a'. With the GOSUB construction, the subroutine beginning on the a-th line in the list is executed, then program control is returned to the first statement on the program line immediately following the ON line. With GOTO, complete control passes to the specifed program line, and no return occurs unless specified by a later GOTO.

ON is frequently used in programs that use the "menu selection" technique to determine what part of the program is executed next. It is functionally equivalent to using a series of IF statements, but ON provides an easier and more space-efficient means of testing for transfer of program control. For example,

```
IF J = 1 GOTO 120
IF J = 2 GOTO 360
IF J = 3 GOTO 80
IF J = 4 GOTO 790
```

could be replaced with

```
ON J GOTO 120,360,80,790
```

ON - EXAMPLE

EXAMPLE

```
10 CLS
20 INPUT J
30 ON J GOTO 100,200,300
40 PRINT"BAD NUMBER":GOTO 10
100 PRINT"AT LINE 100":GOTO 10
200 PRINT"AT LINE 200":GOTO10
300 PRINT"AT LINE 300":GOTO 10
```

NOTES

The ON construction can only be used with a numeric variable. If you want to allow menu selection by letter rather than number, use the VAL statement to convert the letter depressed into numeric representation, then test the value with the ON statement.

OR

OR is a relational (BOOLEAN) operator that performs a logical, bitwise ORing operation on two or more relations. Generally used in conjunction with IF statements, OR test the given conditions to see if one of them satisfies the condition before performing the subsequent part of the IF statement. The result of any ORing operation is always "true" or "false". One of the conditions tested with OR must be true for the subsequent part of the IF statement to be performed.

EXAMPLE

```
10 CLS
20 OUTPUT"DEPRESS",30,60,3
30 OUTPUT"LEFT OR RIGHT",10,50,3
40 OUTPUT"FIRE BUTTON",15,40,3
50 OUTPUT"TO START",25,30,3
60 IF FIRE(0)=0 OR FIRE(1)=0 GOTO 100
70 GOTO20
100 CLS
110 PLOT50,60,1,8,6
120 FORY=1 TO 60 STEP2
130 PLOT54,Y,3
135 FOR Q = 1 TO 20:NEXTQ
140 PLOT54,Y,0
150 TONE 54,Y
160 IFY=59 OR Y=60 GOTO 300
170 NEXTY
300 FORC=1TO7
310 PLOT50,60,C,8,6
320 FOR Q = 1 TO 50
330 NEXTQ
340 NEXT
345 FOR Q = 1 TO 500:NEXTQ
350 GOTO 10
```

NOTES

OR may also be used in IF statements along with the other logical operator, AND. See the IF and JOY sections for other examples of using OR. Or, consult chapter 2 (page 2-20) to find out more about using AND and OR to test conditional relations.

OUTPUT

The OUTPUT statement lets you display information at a given (x,y) location on the screen in one of the colors in the current color set. OUTPUT has the form

OUTPUT exp,x,y,c

where:

exp	is the value to be displayed at the specified (x,y) coordinates. exp can be a numeric or string constant, variable, or expression.
x	is the horizontal screen coordinate at which the upper left corner of exp will be placed. x can be a numeric constant, variable, or expression.
y	is the vertical screen coordinate at which the upper left corner of exp will be placed. y can be a numeric constant, variable, or expression.
c	references the color in one of the positions of the current color set. 'c' can be 0, 1, 2, or 3 and determines the color in which exp is displayed.

EXAMPLES

```
10 CLS
20 COLOR 0,3,1,7
30 OUTPUT"HELLO",42,60,1
40 GOTO 10
```

```
10 CLS
20 COLOR 0,3,1,7
25 FOR Y=72 TO 10 STEP -6
30 OUTPUT"HELLO",42,Y,1
40 NEXT Y
50 GOTO 10
```

NOTES

Never use the RESET-R sequence to terminate program execution in programs that contain OUTPUT statements. If you hit the RESET button at a time when the computer is processing an OUTPUT statement, the statement can be mutilated beyond recognition, causing the program to fail when it tries to execute that line during a subsequent RUN. Use Control-C to stop execution instead.

PEEK

The PEEK function lets you examine the contents of specific memory addresses in the Interact. PEEK returns the contents of the specified byte address as a value between 0 and 255. PEEK has the form

PEEK(loc)

where:

loc	is a numeric variable that specifies the location for which contents are to be returned. loc must be an integer value within the range of locations that can be examined. This range will depend on which version of BASIC you are using:

	LOW	HIGH
Microsoft 8K BASIC and RS232 BASIC	0	32767
Level II BASIC	2049	25127

The values returned by the PEEK function are expressed as decimal numbers. To determine the equivalent hexadecimal value (2 hex digits), use the table in chapter 11 (page 11-3).

For a list of addresses that are worthwhile to PEEK and POKE, consult the table under the POKE command in this chapter.

EXAMPLE

The following RS232 BASIC program display's your computer's system ROM. The value stored in each location is given in both decimal and hexadecimal notation. Change all occurrences of LPRINT in the program to PRINT to run this program under Microsoft 8K BASIC control and display the values on your TV screen.

By knowing the instruction equivalents and instruction lengths, it is possible to extend this program to disassemble the ROM into 8080 mnemonics. However, the entire disassembly process can be easily done with the Micro Video Disassembler, which does not require as much RAM as BASIC and an extension to this program would.

```
10 REM-DISPLAY SYSTEM ROM
20 REM-ON LINEPRINTER
30 REM-IN DEC AND HEX NOTATION
32 DIM H$(2),D(2)
50 LPRINT
60 LPRINT"LOC      DEC    HEX"
70 FOR L=1 TO 2048
80 V=PEEK(L)
90 D(1)=V AND 240
92 D(1)=D(1)/16
100 D(2)=V AND 15
110 REM-DO FOR EACH HEXDIGIT IN BYTE
120 FOR I=1 TO 2
130 IF D(I)<=9 THEN H$(I)=STR$(D(I)):GOTO 150
132 RESTORE
134 J=D(I)-9
135 FOR K=1 TO J
136 READ H$(I)
137 NEXT K
138 GOTO 160
150 H$(I)=RIGHT$(H$(I),1)
160 NEXT I
170 DATA A,B.C,D,E,F
172 A$=H$(1)+H$(2)
180 LPRINT L;TAB(8);V;TAB(16);A$
182 IF INT(L/10)*10=L THEN LPRINT
190 NEXT L
```

```
LOC      DEC     HEX
 1        64      40
 2        50      32
 3        2       02
 4        40      28
 5        195     C3
 6        12      0C
 7        0       00
 8        243     F3
 9        195     C3
 10       4       04

 11       8       08
 12       62      3E
 13       56      38
 14       50      32
 15       0       00
          .
          .
          .
```

NOTES

Not only is Level II BASIC more restrictive in what locations can be accessed, an initializing or "enabling" POKE instruction (POKE 19215,25) must be executed each time the Level II BASIC interpreter is loaded or the RESET-R sequence is used to restart BASIC, or PEEKs will not be permitted. Failure to execute this POKE instruction results in a "?SN ERROR". In Microsoft 8K and RS232 BASIC, PEEK and POKE are automatically enabled when BASIC is initialized.

PLOT

The *PLOT statement* outputs one or more pixels on the screen, as specified by the parameters on the statement. PLOT is generally used for production of graphic images on the screen. The PLOT command has two forms; the form you can use depends on which version of BASIC you have. If you have Level II or RS232 BASIC, the form of PLOT is:

```
PLOT x,y,c
```

The form above can also be used with Microsoft 8K BASIC. However, in this new version of BASIC, the PLOT statement has been extended to allow two additional parameters for faster graphics productions. With 8K BASIC, the PLOT statement has the form

```
PLOT x,y,c,xl,yl
```

where:

x	is the horizontal screen plotting coordinate. It specifies how many pixels from the left edge of the screen plotting is to start. x can be a numeric constant, variable, or expression.
y	is the vertical screen plotting coordinate. It specifies how many pixels from the bottom of the screen the plot is to start. y can be a numeric constant, variable, or expression.
c	references one of the positions in the color set (0-3) to determine the color for plotting. The color stored in that position in the current color set is used for plotting.
xl	is the number of pixels to be output horizontally, starting at the x-coordinate. In other words, xl is how long you want the plot to be. The xl parameter can only be used in Microsoft 8K BASIC.
yl	is the number of pixels to be output vertically, starting at the y-coordinate. In other words, yl is the height (or width) you want the plot to be. The yl parameter can only be used in Microsoft 8K BASIC.

EXAMPLES

The following examples illustrate the use of the PLOT statement for several types of graphic effects. Some of the examples can be executed with either Microsoft 8K or Level II BASIC; others require Microsoft 8K BASIC because they use the extended PLOT parameters. All examples that can be entered in Level II BASIC can also be entered and RUN with Microsoft 8K BASIC.

The first example draws a single line across the screen. Note that Level II BASIC requires three statements to accomplish the same effect as can be done with a single statement in Microsoft 8K BASIC. The PLOT is also much faster in 8K BASIC.

Level II BASIC	Microsoft 8K BASIC
10 FOR X=1 TO 112 20 PLOT X,60,1 30 NEXT	10 PLOT 1,60,1,112,1

Our next example combines the SIN function and the PLOT statement to produce a SIN curve similar to that produced in the Biorhythm program. This program can be entered and RUN in either version of BASIC.

```
10 CLS
20 FOR X=1 TO 112
30 Y= 40+15*SIN(X/4)
40 PLOT X,Y,2
50 NEXT
```

The following program requires Microsoft 8K BASIC. It illustrates the production of a series of "telescoping" rectangles on the screen in different colors.

```
10 CLS
20 X=1:Y=1
30 XL=110:YL=75
40 C=1
50 FOR I=1 TO 12
60 PLOT X,Y,C,XL,YL
70 X=X+3:Y=Y+3
80 XL=XL-6
90 YL=YL-6
100 C=C+1
110 NEXT
120 OUTPUT "BOO!",45,40,3
130 FOR P=1TO 500:NEXT
140 GOTO 20
```

Our final example is a more lengthy sample program that illustrates various effects that can be achieved easily and in relatively little programming area with Microsoft 8K BASIC.

```
5 REM-DEMO PROGRAM FOR THE
6 REM-MICROSOFT 8K FAST GRAPHICS
7 REM-BASIC BY
8 REM-MICRO VIDEO CORP.
9 REM-ANN ARBOR, MI
10 CLS
20 COLOR 0,1,7,3
30 REM-TWO LARGE SQARES
35 PLOT 7,7,3,96,66
40 PLOT 10,10,1,90,60
45 GOSUB 1000
50 REM-A GRID
60 FOR Y=10 TO 70 STEP 10
70 PLOT 10,Y,2,90,1
80 NEXT
90 FOR X=10 TO 100 STEP 10
100 PLOT X,10,2,1,60
110 NEXT
120 GOSUB 1000
130 REM-A TRIANGE
140 L=1
150 FOR Y=70 TO 10 STEP -1
160 PLOT 10,Y,3,L,1
170 L=L+1:NEXT
180 GOSUB 1000
240 REM-SOME STRIPES
245 C=0
250 CLS
260 FOR Y=5 TO 72
262 C=C+1
270 PLOT 1,Y,C,117,1
280 NEXT
290 GOSUB 1000
300 REM-A STAR IS BORN
310 CLS:COLOR 0,4,3,2
320 FOR C=1 TO 2
325 XS=59
326 XL=1
330 FOR Y=70 TO 30
340 XS=XS-1
350 XL=XL+2
360 PLOT XS,Y,C,XL,1
370 NEXT
380 XS=18
390 XL=83
400 FOR Y=60 TO 20 STEP -1
410 PLOT XS.Y,C.XL,1
420 XS=XS+1
430 XL=XL-2
440 NEXT
500 NEXT
510 GOSUB 1000
600 REM-RANDOM COLO SQUARES
610 CLS
612 COLOR 0.5,6,7
615 FOR Q=1 TO 150
620 Y=5+INT(55*RND(1))
630 X=5+INT(80*RND(1))
640 C=1+INT(3*RND(1))
650 XL=2+INT(15*RND(1))
670 YL=1+INT(15*RND(1))
680 PLOT X,Y,C,XL,YL
690 NEXT
700 GOSUB 1000
999 GOTO 10
1000 REM-A PAUSE LOOP
1010 FOR Q=1 TO 500
1020 NEXT
1030 RETURN
```

POINT

POINT is an arithmetic function that returns a value of 0, 1, 2, or 3. This value indicates the position in the color set in which any (x,y) location on the screen is displayed. POINT has the form

```
POINT(x,y)
```

where:

x	is the horizontal coordinate of the pixel for which the color value is to be returned. x can be a numeric constant, variable, or expression.
y	is the vertical screen coordinate of the pixel for which the color value is to be returned. y can be a numeric constant, variable, or expression.

POINT can be used to allow movement on the screen up to but not beyond a certain point by testing for the color of the pixel in a certain screen position. This can be useful in game programs in which you want to have a ball bounce off a wall or set up barriers for game pieces to maneuver around.

EXAMPLE

The following example illustrates how to create a bouncing ball between two walls. You could create your own Breakthrough-type game using this and the example shown for the POT function. This program requires Microsoft 8K BASIC.

```
10 CLS        0,1,2,3
20 COLOR 0,1,3,7
30 PLOT 1,50,1,112,1
40 PLOT 1,40,1,112,1
50 Y=41:IN=1
60 FOR X=10 TO 110
70 J=POINT(X,Y)
80 IF J<>0 GOTO 200
90 PLOT X,Y,2
100 FOR P=1 TO 10:NEXT
110 PLOT X,Y.0
120 Y=Y+IN
130 NEXT
140 GOTO 50
200 IN=-IN
210 GOTO 120
```

plot

POKE

The POKE statement allows you to set the contents of specific memory addresses to specific values to create advanced effects in your programs. The statement has the form

```
POKE loc,n
```

where:

loc	is a numeric variable or constant that specifies the location for which stored contents are to be changed. loc must be the integer value of the variable or constant.
n	is the decimal value to be stored in the specified memory location. 'n' must be an integer between 0 and 255; it may be either a variable or a constant.

You can poke any locations between 16384 (the top of the screen) through 32767 (the highest RAM address in the 16K machine). For a more complete description of the memory organization in the Interact, refer to the documentation available with the Micro Video MONITOR.

Use POKE statements with caution. If you make an error in logic, typing, or POKEd values, you can destroy your program. Therefore, always CSAVE a program containing POKE instructions before executing it. That way, you'll have a back-up copy in the event that something terrible does happen to the program when you RUN it.

Level II BASIC requires an initial enabling POKE instruction to allow subsequent POKE statements to be successful. This instruction is

```
POKE 19215,25
```

In Microsoft 8K and RS232 BASIC, POKE is enabled when the interpreter loads, so there is no need to enter this instruction. If you forget to enter it in Level II BASIC and attempt to POKE a memory location, you'll get a "?SN ERROR".

On the next page is a table of POKE locations, values, and commentary that you'll find useful in exploring the effects you can achieve by "POKEing around". We've given you the POKE addresses for both Microsoft 8K/Level II BASIC and RS232 BASIC. In many cases, the addresses are the same for all BASICs.

POKE Location			
Microsoft 8K or Level II BASIC	RS232 BASIC	Default Contents	Description
19215	same	0	Enables POKE for Level II BASIC
24888	24881	32	Controls scrolling direction: 32=up, 1=sidewise, others = diagonal
24559	same	0-255 (variable)	Clock/counter increments each 1/60th of second. See Digital Clock program in BASIC Examples Booklet
24529	same	variable	ASCII code of last character depressed on keyboard. See chapter 4.
16384-18943	same	bit pattern	Color TV screen control. See chapter 3.
4096	same	8*C2+C0	Tape motor control and color register for colors C0 and C2.
6144	same	8*C3+C1	Color registers for colors C1 and C3. See chapter 3.
24864	24857	6	Number of pixels per PRINT scroll.
24545	same	none	RCHRAD. Address of user-defined character table. Format: Byte 0 Character height (bits) 0 Character width " 2 Start of character images
24558	same	none	Striped letters = 102 Other values give different color mixes depending on color set. See example (10-58)
19474 19473	same	none	Address for USR function. See chapter 11.
N/A	25098 25099	93	Baud rate specification for RS232 interface
N/A	25100	11	Port control--bits, parity, length

POKE

N/A	25097	0	Port control--no line feed default, 10 to set on automatic line feed
24624	24619	none	Change sound with keyboard stroke: 1=splatter; 3=honk; 4=hiss+honk; 7=typewriter
24626 24627	24621 24622	none	Change tonal sound with keyboard stroke. Analogous to second parameter of SOUND command. For sample run set 24626=8, 24627=1

The POKE locations and parameters above are just our suggestions. Experiment with POKEing different values into the different locations to see what the result will be. Again, make sure you CSAVE a copy of any program containing POKE statements before you execute the program, so you'll be safe in case you've made an error.

EXAMPLE

The following example program uses a POKE location given in the previous table to perform scrolling with colorful, striped letters. When RUN, this program produces a dazzling "light show" on your TV screen. You can modify the program to print your own special message simply by changing the PRINT statement in line 52.

```
10 REM-COLOR STRIPES WITH POKES
15 CLS
20 POKE 19215,25
30 POKE 24558,102
52 PRINT"  MICRO VIDEO"
53 PRINT
54 FOR Z=1 TO 10
55 COLOR 0,3,1,7
58 GOSUB 100
60 COLOR 0,1,7,3
70 GOSUB 100
72 COLOR 0,7,3,1
74 GOSUB 100
80 NEXT
92 GOTO 30
100 TONE 60,10
110 RETURN
```

POS

POS is an arithmetic function that returns character position of the last character printed on the current line using the LPRINT or PRINT statement. POS has the form

POS(n)

where:

n	is a dummy variable that defines the function. The value returned in the same no matter what variable is included within the parentheses.

The value returned indicates the position of the last character output on the line. If a value of 0 is returned, no space remains on the current line; any subsequent information will be printed on a new line.

POS is frequently used with the SPC function to produce columnar output of data on the screen or a lineprinter. It is generally used with PRINT or LPRINT statements ending in a semi-colon. With POS, you can use multiple PRINT or LPRINT statements to output information on a single line.

EXAMPLE

Enter and RUN this program to see the effect of the POS function on the information placement on the screen.

```
5 CLS
10 DATA LU, RICHARD
20 DATA WILSON. DAVE
30 DATA DRISCOLL, SUE
40 DATA WILLIAMS, JOHN
50 DATA WOLF. THOMAS
60 FOR I=1 TO 5
70 READ L$,F$
80 PRINT L$;
90 J=POS(0)
95 K=9-J
100 PRINT SPC(K); F$
110 NEXT
```

NOTES

If you have the Micro Video RS232 peripheral interface and RS232 BASIC, you may find POS helpful in combination with the LPRINT statement to produce columnar reports. In general, for output to the screen, the OUTPUT statement is more convenient and easier to use.

POT

POT is an arithmetic function that reads the potentiometer (pot knob) on on either the left or right entertainment controller and returns a value proportionate to the pot knob setting. POT has the form

```
POT(n)
```

0 = Left
1 = Right

where:

n	is either 0 or 1, depending on which controller is to be read. POT(0) returns the value of the pot knob on the left controller; POT(1) returns the value of the pot knob on the right controller.

In reading either controller, the lowest values are obtained when the pot knob is at the farthest counter-clockwise position. The value increases as the knob is turned in a clockwise direction. The highest value is obtained when the pot knob is at the farthest possible clockwise position.

The POT function is used to read within a range of values, not a specific value, as in game paddle movement.

EXAMPLE

The following simple program lets you examine the POT values on each controller as you turn the knob.

```
10 PRINT POT(0);POT(1)
20 GOTO 10
```

Our next example illustrates how to produce and move a game paddle on the left side of the screen. You can use this as the basis for paddle movement in your own BASIC game programs. This example requires Microsoft 8K BASIC.

```
10 CLS:COLOR 0,7,1,3
20 PLOT 5,10,1,2,6
30 YY=10
40 GOSUB 500
50 GOTO 40
500 Y=POT(0)
510 Y=Y/2
520 IF Y>71 OR Y=YY THEN RETURN
530 PLOT 5,YY,0,2,6
540 PLOT 5,Y,1,2,6
550 YY=Y
560 RETURN
```

NOTES

The value returned by the POT function can vary from computer to computer and controller to controller. The reading will generally range from 3 to 154 when controllers are plugged in. In controllers are not plugged in, POT returns a value greater than 200.

PRINT

PRINT can be used as a program statement or in direct mode to print information on the TV screen in a scrolling fashion. Scrolling with PRINT takes place from the bottom-most line on the screen. PRINT has the forms

```
PRINT a [;a;...;a]

PRINT a [,a,...,a]
```

where:

a	can be
	o a string or numeric constant (e.g., "MARY" or 3.14159)
	o a numeric or string variable (e.g., A$ or C)
	o a function call (e.g., SQR(A))
	o an arithmetic expression (e.g., 3*B or COS(C)*A)
	o a string expression (e.g., A$ + B$)

More than one constant, variable, expression, or function call can be placed on a single PRINT statement. You can combine items to be printed from different modes, so long as you do not try to combine data items from different modes into an expression (such as PRINT A$ + B). The items to be printed can be separated with either semi-colons or commas. If you use the comma as a separator, the data items are output in fields 14 characters wide. While this is useful for printing data on a lineprinter with LPRINT, its utility is somewhat questionable for screen printing, due to the Interact's 17-character length lines. The semi-colon separator is far more frequently used with PRINT. If you separate items with semi-colons, numeric data are output with both a leading and trailing blank, while string data values are concatenated. You can use the SPC function or a string of blanks to add spaces between strings in PRINT statements. For example,

```
PRINT "MICRO";"VIDEO"
MICROVIDEO
PRINT "MICRO";SPC(2);"VIDEO"
MICRO  VIDEO
```

Note that if the comma separator is used, string data will wrap around the edge of the screen, while numeric data will not.

PRINT used alone scrolls a blank line on the screen.

EXAMPLES

Because the PRINT command is so widely used throughout this manual, we have elected to present only the above simple example to illustrate its use. See chapter 2 in particular for a relatively lengthy discussion of the PRINT command's uses.

READ

The READ statement references data values stored in DATA statements in a BASIC program and stores them for use in variables identified in the READ statement. READ has the form

READ a[,b,...,z]

where:

a	is a string or numeric variable name into which a data value from a READ statement is to be stored. More than one variable name can be included on the READ list. The variables must all have different names and must be separated by commas.

READ is the most convenient and efficient manner of entering large numbers of data constants into a program for subsequent use. When the READ statement is used for the first time in a program, it always reads data values from the first DATA statement in the program. Subsequent access to the data in the DATA statements is sequential. READ has an internal pointer that keeps track of which data points in the DATA statements have already been used. Once data points have been entered as a result of a READ statement, they are not reused unless the RESTORE statement is used to reset READ's internal pointer back to the first data item on the first DATA statement. See RESTORE for more information on this feature of BASIC.

READ statements are frequently placed within FOR...NEXT loops when more than one data value or set of data points are to be used in a single operation. This concept is illustrated in the following example.

EXAMPLE

```
5 REM-"CHARGE" NOTES
10 DATA 168,33,124,45
20 DATA 97,58,80,40
30 DATA 97,58,80,250
35 REM-GIVE MY REGARDS NOTES
40 DATA 148, 74
50 DATA 130,140,117,70
60 DATA 110,170,97,150,110,280,117,235
70 CLS:OUTPUT"CHARGE!",35,60,1
80 C=6:GOSUB 600
90 CLS:C=7
100 OUTPUT"GIVE MY REGARDS",10,60,1:GOSUB 600
110 RESTORE
120 GOTO 70
590 REM-SUBROUTINE TO PLAY "C" TONES FROM CURRENT
595 REM-POSITION IN THE 'DATA' STATEMENTS
600 FOR I=1 TO C
610 READ A,B
620 TONE A,B
630 NEXT:RETURN
```

NOTES

See DATA and RESTORE in this chapter for more information on this mode of data entry. Also consult chapter 6.

REM

With REM (remark) statements, you can document your program internally. REM is a "do-nothing" statement that exists only for documentation purposes-- REM statements are never executed. This statement has the form

```
REM text
```

where:

text	is a string of up to 72 characters (including the line number and REM keyword) that documents part of your program, usually program logic.

REM statements are most often used to document program logic, although you can use them in any way you wish. They add clarity to the program, but at the expense of RAM. Because REM statements consume considerable amounts of your program storage space, we recommend you use them sparingly.

You can transfer program control to a REM line with a GOTO or GOSUB statement, but we recommend avoiding this. Because they take up large amounts of RAM, as your programs grow larger you may have to remove REM lines to gain additional memory. If you GOTO a REM line from another program line, then have to remove the REM line to conserve space, you will have to retype the line containing the GOTO or GOSUB statement to correct the program logic.

Never put any other statements on the same line as a REM statement. BASIC considers everything after the REM keyword to be documentation, and any statement on the same line will be ignored.

EXAMPLE

Is an example really needed? If you feel you need more information or an example of a program with REM lines, see chapter 2 (2-19).

RESTORE

The RESTORE statement resets the READ statement's internal pointer that keeps track of which data items in the available DATA statements have been referenced. Data items on DATA statements are not reused within a program unless the RESTORE statement is given. When RESTORE is executed, the internal pointer is set back to the first data item in the first DATA statement in the program. This permits multiple READs through DATA statements in a program. The RESTORE statement has the form

RESTORE

No variables or other information are used with RESTORE.

With RESTORE and "dummy" READ statements in a program loop, you can position the internal pointer to any point within any DATA statement in the program. The following example illustrates this concept.

EXAMPLE

```
5 REM-"CHARGE" NOTES
10 DATA 168.33.124,45
20 DATA 97,58,80,40
30 DATA 97,58,80,250
35 REM-GIVE MY REGARDS NOTES
40 DATA 148. 74
50 DATA 130,140,117,70
60 DATA 110,170,97,150,110,280,117,235
70 CLS:RESTORE
80 OUTPUT"C=CHARGE",10,60,1
85 OUTPUT"OR",22,54,2
90 OUTPUT"G=GIVE MY ...",10,48,1
100 A$=INSTR$(1)
110 IF A$="C" THEN C=6:GOSUB 600:GOTO 70
120 IF A$<>"G" GOTO 100
130 REM-FLUSH FIRST 6*2=12 TONE PAIRS
140 FOR I=1TO 12:READ A:NEXT
150 C=7:GOSUB 600
160 GOTO 70
590 REM-SUBROUTINE TO PLAY "C" TONES FROM CURRENT
595 REM-POSITION IN THE 'DATA' STATEMENTS
600 FOR I=1 TO C
610 READ A,B
620 TONE A,B
630 NEXT:RETURN
```

NOTES

See the DATA and READ entries in this chapter and also chapter 6 for more information on entering data into programs in this way.

RETURN

RETURN is a statement that is required to terminate a subroutine initiated with a GOSUB statement. When RESTORE is executed in a program, program control is returned to the statement following the originating GOSUB statement. RETURN has the form

RETURN

No other parameters are included with a RETURN statement.

RETURN must be the last statement in each subroutine in order for there to be successful program execution and return to the calling program logic. In any program using subroutines, the number of executed RETURN statements should equal the number of executed GOSUB statements.

EXAMPLE

The following program lets you explore your SOUND chip via a subroutine. The program produces a series of sounds within the range you specify on the INPUT statement. Try entering the values 0,5000 at the INPUT prompt to get an idea of how this program operates.

```
10 CLS
20 OUTPUT"THE NEW WAVE",22,60,1
30 WINDOW 24
35 INPUT"LOW, HIGH";L,H
40 GOSUB 100
45 PRINT:PRINT
50 GOTO 35
99 REM-SUBROUTINE TO QUICKLY GO THROUGH SOUNDS
100 FOR X=L TO H
110 SOUND 3,X
120 NEXT
130 SOUND 7,4096
140 RETURN
```

NOTES

See RESTORE, GOSUB, INSTR$, JOY and others, as well as chapter 7, which deals exclusively with subroutines, for more information.

REWIND

REWIND can be used as a direct mode command or program statement to turn the tape motor on for tape positioning or to play a music tape during program execution. It has the form

```
REWIND
```

No other parameters are given on the REWIND statement.

Once you've issued the REWIND command in direct mode, you can depress any of the cassette buttons for tape positioning. Depression of the cassette buttons in conjunction with REWIND have the following effects:

REWIND	rewinds tape quickly
F-FWD	fast-forward positioning of the tape
READ	moves the tape forward slowly. If you use the READ cassette button, you will hear sounds as if a program were loading. Ignore this sound; no data is being read into your computer when the READ cassette button is depressed in conjunction with the REWIND command. Since you can hear the sounds without overwriting your existing program, this provides an effective method of accurately positioning the tape for a subsequent reading or writing operation.
READ + WRITE	erases any information stored on the tape. You can use REWIND in this way to erase old tapes for reuse.

Always use the REWIND command to position the tape before using CSAVE to store a program on tape. If you rewind the tape completely, be sure to depress the READ cassette button for a few seconds before terminating the REWIND command and issuing CSAVE. This ensures that all the "leader" tone, required for proper program loading, is saved along with the program data.

Terminate the REWIND command by pressing any key.

You can also use REWIND to turn on the tape motor for positioning and saving data on tape from within a program or to play audio cassettes in conjunction with program display. The former concept is illustrated in the following example.

REWIND

EXAMPLE

```
10 CLS
15 COLOR 0,3.2.7
20 DIM A(20)
30 FOR Q=1 TO 20
40 A(Q)=5*SQR(Q)-4.5
50 GOSUB 500
55 CLS
57 OUTPUT"SAVING DATA",24,50,3
60 CSAVE*A
70 CLS
80 PRINT"DATA SAVED":PRINT
90 END
494 REM
495 REM-POSITION TAPE ROUTINE
496 REM
500 CLS:OUTPUT"INSERT DATA TAPE",10,60,1
510 OUTPUT"AND DEPRESS THE",10,54,1
520 OUTPUT"'REWIND' BUTTON",10,48,1
530 OUTPUT"HIT ANY KEY",10,24,2
540 OUTPUT"WHEN DONE",10,18,2
550 REWIND
560 RETURN
```

RIGHT$

RIGHT$ is a string handling function that isolates a specified number of characters from a string, beginning with the rightmost character in the string. It has the form

RIGHT$(a$,n)

where:

a$	is the string variable name containing the string from which characters are to be isolated.
n	is the number of characters to be isolated from the string, starting with the rightmost character. 'n' must be greater than or equal to zero. If the string is shorter than 'n' characters, all characters in the string are isolated.

RIGHT$ takes the specified number of characters from the string and uses them in subsequent program operation.

EXAMPLE

In the following example, a "City, State" portion of an address is presumed to be in the string CS$. Assuming that all addresses are in the U.S. and have proper Postal Service two-character state abbreviations, the RIGHT$ function effectively isolates the state abbreviation, regardless of the length of the city name. You might use RIGHT$ in an extension of this program to isolate the state portion of the data for accumulation and sorting for a demographic survey, for example.

```
10 CLS
20 FOR I=1 TO 5
30 READ CS$
40 PRINTRIGHT$(CS$,2)
50 PRINT:NEXT
60 DATA"CHICAGO, IL"
70 DATA"ADA, MI"
80 DATA"NEW YORK, NY"
90 DATA"PITTSBURGH, PA"
100 DATA"FARGO, ND"
```

NOTES

See chapter 4 for more information on string handling with RIGHT$ and the other string functions.

RND

RND is a numeric function that returns a uniformly distributed random number between 0.0 and 1.0. RND is the call to the random number generator that is frequently used for adding elements of chance to game play, production of random graphics and tones, etc. RND has the form

RND(n)

where:

n	is a value that determines how the random number generator is seeded and accessed. 'n' can be a negative, positive, or zero value.
	If 'n' is a negative number, then a new seed is used for all subsequent calls to the RND function. If you use the same negative argument on subsequent calls to RND, the same series of numbers will be returned.
	If 'n' is zero, then the same numbers returned on the previous call is returned again.
	If 'n' is positive, then the "next" random number in the series is returned. The value of 'n' in this case does not affect the series of random numbers that are returned. The value of positive 1 is commonly used.

EXAMPLE

The following program returns a uniformly distributed set of random integers having values of 0, 1, 2, and 3. These values could then be used for random color selection of one of the four colors in the current color set, although this program does not demonstrate that use.

```
5 POKE 19215,25
6 J=RND(-PEEK(24553))
10 R=INT(RND(1)*4)
20 PRINT R
30 GOTO 10
```

Several other examples of using the RND function for random number generation within programs are presented in chapter 2 (2-35). The game program in chapter 5 also uses the RND function in developing its random graphic skyline.

NOTES

The Interact's random number generation scheme is actually a pseudo-random number generator. That is, there is a large series of numbers from which the function draws. Since these numbers are in series, they can, from time to time, be repeated. You can generate a more "random" starting seed in the series by using the PEEK function to determine the negative value of the internal clock and using that as the seed. This will lessen the frequency of repetitions in values drawn from the series, although the probability of repetitions is actually quite low. To set the starting seed for random number generation with respect to the clock, use the following statement:

```
J = RND(-PEEK(24559))
```

The value returned by the function call with a negative argument is generally not used in other places in the program.

SGN

SGN is an arithmetic function that returns a value of -1, 0, or 1, depending on the sign of the argument. It has the form

SGN(n)

where:

n	is a numeric variable or expression for which the algebraic sign is to be determined. If the result of 'n' is negative, SGN returns a value of -1. If the result of 'n' is positive, SGN returns a value of 1. If 'n' is zero, SGN returns a value of 0.

The statement G = SGN(V/F), for example, is equivalent to the longer, three-statement sequence:

```
G=0:IF V/F > 0 THEN G = 1
IF V/F < 0 THEN G = -1
```

EXAMPLE

```
5 PRINT
10 PRINT"ENTER ANY"
20 INPUT "NUMBER";N
30 B=SGN(N)+2
40 ON B GOSUB 100,200,300
45 GOTO 5
100 PRINT"IT'S NEGATIVE":RETURN
200 PRINT"IT'S ZERO":RETURN
300 PRINT"IT'S POSITIVE":RETURN
```

SIN

SIN is a trigonometric function that computes the sine of an angle given in radians. It has the form

SIN(n)

where:

n	is an angle, expressed in radians, for which the tangent is to be computed. 'n' can be a numeric constant, variable, or expression.

EXAMPLE

Test the use of this function with an angle of 0 degrees. The sine should be 0. The sine of 90 degrees is 1.0, while the sine of 45 degrees is .707.

```
10 PRINT:PRINT"[illegible]LE IN DEGREES"
20 INPUT [illegible]
30 R=DE/57.2[illegible]
40 PRINT SIN([illegible]
50 GOTO [illegible]
```

SOUND

The SOUND statement produces various sounds through the TV speaker, according to the values specified in its two arguments. The SOUND statement has the somewhat unique capability of letting other commands execute while the sound specified by the parameters is being produced. The sounds can be turned off by a SOUND 7,4096 statement within a program or by hitting any key on the keyboard. SOUND has the form

```
SOUND n,m
```

where:

n	is a value between 0 and 7 which may be specified as a constant, variable, or expression.
m	is a value between 1 and 32767. 'm' also may be specified as a constant, variable, or expression.

While there are literally hundreds of addressable sounds in your Interact, not all parameter combinations produce sounds. To get you started, here are some of our favorite sound combinations.

SOUND	DESCRIPTION	SOUND	DESCRIPTION
0,24844	Siren	3,182	PT-109
1,28	Pouring rain	3,258	Tugboat horn
1,8770	Niagara Falls	3,262	Airplane, twin-cycle
2,10	Medium flutter	3,264	Laser
2,12	Intermittent white noise	3,268	Phaser
2,22	Tractor	3,276	B-26 bombers
2,140	Fast clock	3,282	Deep space assault
2,264	High speed steam engine	3,284	Heavy phaser
2,422	High speed motor	3,328	Machine guns
2,600	Helicopter	3,340	Machinery vibration
3,14	Satellite signal	4,40	Low pulse
3,16	Diesel horn	5,392	Locust attack
3,30	Warning alarm	5,398	French ambulance
3,62	Low warning tone	5,422	Outboard motor
3,66	1938 Plymouth--stuck horn	6,170	Telephone
3,80	Factory lunch signal	6,456	Mad organist
3,84	Annoying buzzing sound	6,460	Phaser
3,109	Busy circuit	6,3500	Pulsing drone

EXAMPLE

If you'd like to search through all the sounds your Interact can make, try this little program, the Interactive Noisemaker. This program indexes through all possible SOUND values under the control of the left joystick. As the program runs, SOUND values you select by moving the joystick to the left or right are displayed on the screen and produced through the TV speaker. Move the joystick to the right to increment the value of the second SOUND parameter; push it to the right to increase the value of the first parameter. If you don't move the joystick at all, the last SOUND selected will play continuously.

```
5 REM - INTERACTIVE NOISEMAKER
10 CLS:COLOR 7,1,2,4
70 FOR J = 0 TO 7
80 FOR K = 0 TO 32767
90 SOUND J,K
100 PRINT"SOUND";J;SPC(1);K
110 L = JOY(0)+1
120 ON L GOTO 110,160,150
140 GOTO110
150 NEXT K
160 NEXT J
```

NOTES

You can find out more about the sounds your Interact can make, as well as musical tones, by consulting the looping section of chapter 2 (2-23) and the TONE statement in this chapter.

SPC

The SPC (space) function is used in PRINT and LPRINT statements to insert a specified number of blank spaces between printed data items. SPC is analogous in function to depressing the space bar on a typewriter a specified number of times while forming a printed line of material. SPC has the form

SPC(n)

where:

n	is a numeric value that indicates the number of spaces to be inserted between other printed values. 'n' must be greater than or equal to zero and not so large as to produce a printer line that is longer than 80 characters or a wrapped screen line that is longer than 72 characters.

EXAMPLE

The following program and printer listing illustrate the use of the SPC function to insert a steadily incrementing number of spaces between two words in printed output. This program listing and hard copy from program execution were produced on an Interact equipped with an RS232 interface, RS232 BASIC, and an attached COMPRINT 912-S lineprinter.

```
10 FOR S=0 TO 20
20 LPRINT "HI";SPC(S);"THERE"
30 NEXT
```

```
HITHERE
HI THERE
HI  THERE
HI   THERE
HI    THERE
HI     THERE
HI      THERE
HI       THERE
HI        THERE
HI         THERE
HI          THERE
HI           THERE
HI            THERE
HI             THERE
HI              THERE
HI               THERE
HI                THERE
HI                 THERE
HI                  THERE
HI                   THERE
HI                    THERE
```

SQR

SQR is an arithmetic function that returns the square root of a given argument. SQR has the form

SQR(n)

where:

n	is the value for which the square root is to be computed. 'n' must be greater than or equal to zero, and may be in the form of a numeric constant, variable, or arithmetic expression.

EXAMPLE

```
10 PRINT:PRINT"YOUR NUMBER,"
20 INPUT"PLEASE";N
30 IF N<0 THEN PRINT"WILL NOT COMPUTE":GOTO 10
40 PRINT"SQUARE ROOT IS";SQR(N)
50 GOTO 10
```

STOP

STOP is an indirect mode statement that instructs BASIC to interrupt processing of program statements and return to direct mode, "OK" status. It has the form:

```
STOP
```

When a stop is executed, the message "BREAK IN nnnn" prints to identify the line in which the STOP statement was encountered. Using direct mode commands you can then examine and change various values in the program. When your changes are made, you can restart program execution where it broke ff by issuing the CONT (continue) command.

STOP and CONT are used during the program debugging process at places where you need to "get into" the program and look at the values of variables to determine logic errors.

EXAMPLE

```
10 CLS
20 A=3
30 B=4
40 C=A↑4-1
50 STOP
60 D=C-(A+B)
70 PRINT D
```

When you run this program, it will stop in mid-execution with the message

```
BREAK IN 50
OK
```

You may then use the PRINT command to display the values of the variables A, B, C, and D and to see the progress of the computations. Then, when you type the CONT command, the program picks up where it left off, continues the computations, and prints the computed result of the variable D.

STR$

STR$ is a string handling function that returns the value of a numeric argument in string format. STR$ is thus a function that converts numeric values into string representations. It is complementary to VAL, which takes a string value and converts it to numeric representation. STR$ has the form

STR$(n)

where:

n	is the numeric constant, variable, or expression that is to be converted into the string.

EXAMPLE

In their normal modes, string and numeric data cannot be displayed on the screen with a single OUTPUT command--that type of mode mixing is not allowed in BASIC. But what if you have a game in which you keep score, and at the end of the game you'd like to display the names of the winner and the loser and their scores. Naturally, you could do this with several OUTPUT statements, or put the string data and numeric data on separate lines of the screen. However, the STR$ function provides you with a way to combine the two data modes for output with a single command. You just convert the numeric data into string form with the STR$ function. Then, all the data is of the same mode and can be concatenated on the OUTPUT statement with the '+' operator, as shown below.

```
5 CLS
10 S1=49
20 S2=14
30 OUTPUT"DAVE"+STR$(S1)+" CORI"+STR$(S2),10,60,1
40 A$=INSTR$(1)
```

NOTES

See chapter 4 for further information about string handling.

TAB

TAB is an arithmetic function that is used with PRINT and LPRINT statements to position items subsequently printed at a specified printing column or screen position. It is analogous in function to the TAB key on a typewriter, except that you specify the printing position as an argument to the function. TAB has the form

TAB(n) ; TAB(N) →

where:

n	is a numeric constant, variable, or expression that specifies the column in which the next information is to be printed. If 'n' specifies a column that is less than the current position of the print head or screen cursor, then the TAB call is ignored. 'n' must between 0 and 80 in value, so as not to exceed the 80 character column width allowed. Printer

The TAB function is usually inserted in the PRINT list and separated from adjacent items with semi-colons (;).

EXAMPLE

The following program listing and lineprinter output show how a variable can be graphed on a lineprinter using RS232 BASIC and the Micro Video interface.

```
10 FOR X=0 TO 6.28 STEP .4
20 S=15 + 12*SIN(X)
25 LPRINT "I";TAB(S);"*" ,TAB(0);"%"
30 NEXT
```

```
I              *
I                  *
I                      *
I                         *
I                         *
I                        *
I                      *
I                  *
I             *
I        *
I    *
I  *
I  *
I   *
I      *
I          *
```

TAN

TAN is a trigonometric function that computes the tangent of an angle, given in radians. TAN has the form

```
TAN(n)
```

where:

n	is an angle, expressed in radians, for which the tangent is to be computed. 'n' can be a numeric constant, variable, or expression.

EXAMPLE

In this short program, you are asked to supply the size of an angle in degrees. The program converts that value into number of radians, then computes the tangent. Test the use of this function by entering an angle of 45 degrees. The tangent should be 1.0. The tangent of 0 degrees is zero. The tangent of 135 degrees is -1.0.

```
10 PRINT:PRINT"ANGLE IN DEGREES"
20 INPUT DE
30 R=DE/57.2958
40 PRINT TAN(R)
50 GOTO 10
```

TONE

The TONE statement produces a musical tone through the TV speaker. The frequency and duration of the tone are determined by the values of the two parameters included with the TONE statement. TONE has the form

```
TONE fi,d
```

where:

fi	is an integer expression, the value of which is inversely proportional to the frequency of the tone. That is, the smaller the value of fi, the higher the pitch of the resulting tone.
d	is an integer expression, the value of which is proportional to the tone. The larger the value of d, the longer the resulting tone. Large or negative values of d should be avoided, as they produce extremely long-lasting tones.

TONE statements can be used to:

- o Play simple melodic tunes.
- o Complement visual activity in screen display, such as falling objects, with rapid tonal sequences.

Musical sequences can be produced by combining a series of TONE statements that draw from the following list to determine the first TONE parameter.

	NOTE	1ST TONE PARAMETER		NOTE	1ST TONE PARAMETER
LOW	G	224	MIDDLE	G	110
	G#	212		G#	104
	A	200		A	97
	A#	189		A#	91
	B	179		B	85
	C	168		C	80
	C#	158		C#	75
	D	148		D	71
	D#	139		D#	67
	E	131		E	63
	F	124		F	59
	F#	117		F#	55
			HIGH	G	51

For example, to play the C-E-G-C note sequence, RUN the following program:

```
10 REM C-E-G-C TONAL SEQUENCE
20 TONE 168,150
30 TONE 131,192
40 TONE 110,229
50 TONE 80,315
```

The values of the second parameters in each TONE statement were chosen so that the notes had equal duration. To explore the length of tones further, RUN the next sample program. In this program, the notes stay the same, but the duration of each note becomes shorter as the value of the second parameter gets smaller.

```
300 PRINT"LONG TONE"
310 TONE 200,100
320 PRINT"MEDIUM TONE"
330 TONE 200,50
340 PRINT"SHORT TONE"
350 TONE 200,10
360 PRINT"SHORTEST TONE"
370 TONE 200.1
380 GOTO 300
```

In order for a tonal sequence to have each note held an equal length of time, the product of (fi*d) must be a constant. For example, in the C-E-G-C note sequence above, you'll note that for each TONE statement the product of the two parameters equals approximately 25,200. To double the speed of the note sequence, reduce the value of the second parameter by approximately half by multiplying the second parameter of each tone by .5. To speed it up still further, to four times as fast, multiply by .25.

Of course, individual melodies will not have constant tone durations--they'll contain quarter, half, and eighth notes. The durations of the second TONE parameters are multiplied to achieve the correct rhythm of the music. Here's a familiar favorite which illustrates this concept. You could combine this musical sequence in a program with appropriate graphics and messages and use it for family celebrations. RUN this program to hear the melody:

```
1800 REM-A MELODY USED AT LEAST ONCE A YEAR
1970 GOSUB 3000
1975 TONE 124,124
1976 TONE 131,200
1990 GOSUB 3000
2000 FOR N=1 TO 15
2010 READ A,B
2020 TONE A,B
2030 NEXT
2040 END
2050 DATA 110,110,124,200
2060 DATA 168,84,168,84
2070 DATA 80,160,97,200
2080 DATA 124,150,131,131
2090 DATA 148,148,91,91
2092 DATA 91,91.97,97
2100 DATA 124,124,110,110
2200 DATA 124,150
3000 TONE 168,42
3010 TONE 168,42
3020 TONE 148,99
3030 TONE 168,84
3040 RETURN
```

This next example simulates some rather amazing "keyboard runs". Play it through at high volume on your TV set to get the full effect.

```
10 REM- UP THE SCALE
20 FOR FI=400 TO 1 STEP -1
30 TONE FI,600/FI
40 NEXT
45 REM-DOWN THE SCALE
50 FOR FI=1 TO 400
60 TONE FI,800/FI
70 NEXT
```

And finally, tones can be used to highlight visual motion on the screen as shown in the following sample program.

```
10 CLS:COLOR 0.1,3,7
20 FOR X=1 TO 112:PLOT X,11,1:NEXT
30 FOR Y=76 TO 16 STEP -2
32 SOUND 7,4096
40 OUTPUT"*",56,Y,3
45 TONE 100-Y,20
50 OUTPUT"*",56,Y,0
60 NEXT
70 SOUND 1,514
72 FOR Q=1 TO 30:NEXT
75 SOUND 1,515
76 FOR Q=1 TO 1000:NEXT
80 GOTO 30
```

USR

The USR function instructs BASIC to begin executing machine language instructions that begin at a prespecified address. The machine language instructions are processed until a "C9" instruction is encountered. Then, control of execution is returned to BASIC, to the statement immediately following the USR call. USR has two forms. The one you will use depends on which version of BASIC you are using:

```
J = USR(O)        in Microsoft 8K and Level II BASIC

USR               in RS232 BASIC
```

In the function call format in Microsoft 8K or Level II BASIC, the value of the argument and the returned value, stored in J, are unimportant. Only the USR keyword is required to call a machine language routine from RS232 BASIC.

The starting address of the machine language routine is set by two POKE statements that initialize locations 19474 and 19473 with that address. See chapter 11--Machine Language Integration--for a more complete discussion of how to use USR to transfer program control to machine language subroutines.

NOTES

Machine language programming is more difficult than programming with higher level languages, such as BASIC. The USR function involves programming techniques for the intermediate to advanced programmer.

To get started with the concepts of combined BASIC and machine language programming, we suggest the BOMBS AWAY! Programming Tutorial. It's an entertaining game with clever animation that uses machine language subroutines extensively. The program is heavily documented and provides a good training ground for those who are interested in learning to combine the two programming modes.

VAL

VAL is a function that returns the numeric equivalent of a number that is stored in string format. VAL is thus a conversion function complementary to STR$. It has the form

VAL(s$)

where:

s$	is a string that contains numeric information. If the first character of the string is not a digit (0-9), a plus or a minus sign, or a decimal point, then VAL returns a value of zero for that string.

The VAL function can be useful in accepting data which may include numeric values or special keywords, such as "HELP" or "END", from the keyboard. The keyboard input is always read into a string variable, then converted to its numeric equivalent if it is not a special keyword. The following addition quiz program illustrates this concept. It also provides a nice drill capability for young children.

EXAMPLE

```
5 REM-ADDITION QUIZ
10 CLS
20 OUTPUT"HOW MUCH IS ?",20,60,1
30 OUTPUT STR$(R1)+"+"+STR$(R2),30,40,0
35 R1=INT(9*RND(1))
36 R2=INT(9*RND(1))
40 OUTPUT STR$(R1)+"+"+STR$(R2),30,40,2
50 WINDOW 24
55 INPUT S$
58 IF S$<>"HELP" GOTO 70
59 PRINT
60 PRINT"IT'S ";R1+R2
65 A$=INSTR$(1)
67 PRINT:PRINT
68 GOTO 30
70 A=VAL(S$)
80 IF A=R1+R2 THEN PRINT"RIGHT!":PRINTCHR$(7):PRINT:PRINT:GOTO 30
90 PRINT"NO WAY. TRY AGAIN"
95 PRINT"OR TYPE:  HELP"
100 GOTO 50
```

WINDOW

The WINDOW statement establishes the number of lines on the lower portion of the TV screen that will be used for the scrolling result of the PRINT or LIST statements. WINDOW has the form

WINDOW n

where:

n	is the y-coordinate below which scrolling can occur. 'n' must be a value between 11 and 77.

By default, the WINDOW is set at the top of the screen so that all material on the screen is scrolled. This is equivalent to the setting obtained with the statement **WINDOW 77.** We suggest that you set WINDOW as an integral multiple of 6 so that partial lines at the window boundary are not "chopped off" at the top. The RESET-R restart sequence automatically sets windowing back to the default, WINDOW 77.

WINDOW VALUE	# TEXT LINES IN SCROLLING AREA
12	1
18	2
24	3
.	.
.	.
.	.
72	11

The WINDOW statement is useful for:

1. Displaying title lines on the screen and scrolling sub-point information beneath them.

2. Displaying a question on the top of the screen and having answers appear at the bottom. Incorrect answers won't cause the question to be scrolled out of sight (see VAL example).

3. Developing graphics with direct mode statements.

WINDOW

EXAMPLE

```
300 CLS
310 WINDOW 24
320 OUTPUT"THE VALUE OF PI",10,60,1
330 OUTPUT"TO 4 PLACES IS",10,54,1
340 INPUT ANS
350 IF ANS=3.1416 GOTO 380
360 PRINT"NO. TRY AGAIN"
370 GOTO 330
380 CLS:PRINT"RIGHT !"
390 WINDOW 77
400 LIST
```

MACHINE LANGUAGE INTEGRATION

When you've become completely acquainted and familiar with how BASIC operates, you may wish to move to a more advanced type of programming that combines both BASIC language statements and subroutines written in machine language. BASIC's USR function lets you transfer program control to the starting address of a machine language subroutine. The last instruction in a machine language routine, C9, transfers control back to BASIC--to the statement immediately following the USR call.

Examples of machine language code you may wish to invoke from your BASIC programs include:

SOURCE	EXAMPLE	FOR MORE INFORMATION
ROM Subroutines	RPLOT Graphics	Guide to ROM Subroutines
Your own code	Specialized arithmetic functions	BOMBS AWAY Programming Tutorial Micro Video MONITOR
Subroutines supplied by others	Vector Graphics Subroutines	Vector Graphics documentation

To transfer control to a machine language routine you must:

1) POKE the starting address of the machine language routine into locations 19474 and 19473.

2) POKE any required calling parameters into specified locations where it can be read by the machine language routine.

3) Execute one of the following statements, depending on which version of BASIC you are using:

A = USR(0)	Microsoft 8K or Level II BASIC
USR	RS232 BASIC

Note that BASIC works with decimal values, while machine language instructions and addresses are in hexadecimal (base 16). The locations 19474 and 19473 can contain four hexadecimal digits which identify the starting address of a machine language routine. Since these two bytes must be set in BASIC to two individual POKE instructions with decimal arguments, you must first convert the hexadecimal address into two decimal values to be used in the POKE commands.

For example, let's assume that we have developed a machine language routine that begins at hex address 5E6B. Because machine language reads addresses in "reverse" order, a decimal value equal to 5E must be poked into location 19474, and a value equal to 6B must be poked into 19473. By looking at the following bit patterns for each byte, we can compute the decimal values for the POKE instructions.

Hex Value	5	E	6	B
Bit Values	0 1 0 1	1 1 1 0	0 1 1 0	1 0 1 1
	19474		19473	
Decimal Value	94		107	

Therefore, the POKE statements required are

POKE 19474,94

POKE 19473,107

While a discussion of 8080 machine language programming is beyond the scope of this manual, we have provided a hexadecimal to decimal conversion chart on the following page for ease in machine language address conversion. If you want to learn to program in machine language, you will need the Micro Video MONITOR. The MONITOR documentation contains a number of suggested references for learning 8080 programming. We've also listed some other sources for information in this chapter. Consult the Programming Aids section of the Micro Video Product Catalog to find out what products are available to help you get started in machine language integration. You'll probably find that, in addition to the MONITOR, the BOMBS AWAY! Programming Tutorial and the Guide to ROM Subroutines will be helpful in getting you started in this more complex type of programming.

HEXADECIMAL/DECIMAL CONVERSION CHART

(Hex)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
(Dec.)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
	90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF
	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF
	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

APPENDIX A

ERROR MESSAGE HANDLING

During program development or running your BASIC programs, you are likely to encounter several error messages. Error messages indicate that something has gone wrong and that corrective action on your part is needed. Error messages are BASIC's way of telling you it does not understand what you are telling it to do. You can get an error message because you have not followed the proper syntax for a statement, you have tried to call a function with an illegal argument (such as trying to call an arithmetic function with a string variable), you have forgotten to dimension an array you're trying to reference with a subscript, and many other conditions. Any error message you get will have one of two forms:

```
?XX ERROR

   or

?XX ERROR IN YYYYY
```

where:

XX	is one of the error codes defined below.
YYYYY	if included, indicates the line number on which the error occurred.

In general, if a line number is given in the error message, you should start corrective action by examining the line to determine what's wrong. Unfortunately, the BASIC error messages do not impart much information about any error. They merely define the code for the program failure. To help you in learning to understand and deal with errors in your programs, we've provided further information about the possible causes and suggested corrective steps for all the BASIC error messages.

?BS ERROR -- Subscript out of range

A subscript value on one of your arrays is either less than zero or greater than the maximum subscript declared with the DIM statement or allowed by default (10). Check the values of the subscripts referenced in the incorrect line. Your subscript calculation may have been done incorrectly. Or, you may have dimensioned the array improperly or not dimensioned it at all. Or, you may have used the wrong number of dimensions in the subscript reference.

?CN ERROR -- Can't continue

The CONT (continue) command you just entered in direct mode cannot be performed. Correct any errors encountered with previous error messages, then RUN the program again.

?DD ERROR -- Redimensioned array

An array variable name has been defined more than once. You either have the array variable name appearing in more than one DIM statement, or you have attempted re-execution of a DIM statement that has already been performed. The latter condition can be caused by improper branching back to the beginning of a program. This error will also occur if you put the initializing DIM statement inside a FOR...NEXT loop. DIM statements should only be entered once in the flow of a program, and an array should be dimensioned before the first reference to the array is made.

?FC ERROR -- Illegal function call

The value of the argument (calling parameter) for a function is incorrect. For example, you might get this message if you try to reference the -5th character in a string. This error message will also be output if the values used as parameters in various BASIC statements, such as PLOT, OUTPUT, COLOR, are inadmissable. For example, if you try to address screen coordinates that are outside the allowable range. Print and check the values of all variables referenced in the flagged line.

?ID ERROR -- Illegal direct

The BASIC command you have just typed cannot be performed in direct mode. It can only be performed within a program in a line-numbered statement. Perhaps you forgot to type the line number when entering the statement. You'll get this error, for example, if you try to use the DEF statement to define a function in direct mode.

?LS ERROR -- String too long

The string variable just referenced is too long (exceeds the maximum length of 255 characters). Check to make sure that a string concatenation operation is not being done within an infinite loop.

?MO ERROR -- Missing operand

The statement is missing an operand. For example, this error will occur in the statement 30 C = A + where 30 C = A+B was intended. Check the identified statement for accuracy.

?NF ERROR -- NEXT without FOR

A NEXT statement was encountered for which there was no originating FOR statement. Check to make sure the associated FOR statement has not been deleted accidentally or that improper branching into the inside of the FOR...NEXT loop has not occurred. You'll also get this error message if the variable specified in the NEXT statement is not the same as the iteration variable in the originating FOR statement.

?OD ERROR -- Out of data

A READ statement is being attempted for which no unused values in DATA statements are available. Check that the READ statement is not being done more times than desired and that enough data has been supplied in the DATA statements. If whole program looping is being done and you want to restart at the beginning of the DATA list, a RESTORE statement may have been inadvertantly bypassed or omitted.

?OM ERROR -- Out of memory

There is insufficient memory in your Interact to contain the desired program, arrays, or data. If this error occurred immediately after you loaded BASIC, you may have forgotten to type the NEW command to clear memory for entry of a new program. If you did type NEW, you may have to compact your program by removing REM lines, grouping multiple statements on single lines, reducing array sizes, using subroutines to reduce repetitive statements. See Space Saving Hints in chapter 8 for more information on how to compress your program. To check the amount of available memory, type **PRINT FRE(O).**

?OS ERROR -- Out of string space

You have attempted to create a string variable which is too long to be stored in the memory allocated for string storage. Check to make sure no unnecessary strings are being formed, increase the available string space with the CLEAR(size) statement, or reuse string variable names where possible. To check the amount of available string space, type **PRINT FRE("A").**

?OV ERROR -- Overflow

The result of a calculation was too large to be represented in BASIC's internal number format. Check the calculations being performed to ensure that they are being done correctly.

?RG ERROR -- RETURN without a GOSUB

A RETURN statement was encountered before a previous GOSUB was executed. Check that your program flow is correct and that a GOTO was not inadvertantly used in place of a GOSUB.

?SN ERROR -- Syntax error

A statement has been improperly formed due to a typing error, missing parenthesis in an expression, misspelled keyword, missing part of a statement, invalid parameter on a statement, etc. Retype the line correctly.

?ST ERROR -- String formula too complex

An expression involving strings or string functions is too long or too complex. Break the expression into two or more shorter statements.

?TM ERROR -- Type mismatch

The type of variables within a statement is inconsistent. You may have made an attempt to store a numeric result into a string variable or vice versa. Or, you may have omitted the $ from a string function name. Check that numeric and string variables are not being improperly mixed in the statement.

?UF ERROR -- Undefined user function

Reference was made to a user-defined function that has not been defined with the DEF statement. Check for a spelling error or a missing or unexecuted DEF statement.

?UL ERROR -- Undefined line number

The line number referenced in a GOTO, GOSUB, ON, or IF...THEN statement does not exist. Check to make sure the line does exist in the program with the LIST command, or check for a typing error in the line number of the flagged line. The line may have been deleted accidentally. Or, the line number reference may not have been changed by a renumbering operation in EZEDIT if the reference was not the first line reference in a program line. If this is the case, retype the command in BASIC or use the EZEDIT SUBSTITUTE command to make the required change(s).

?/0 ERROR -- Division by zero

You have attempted either to divide by zero or to raise zero to a negative power. Check to make sure the calculations used to form the divisor are correct.

Of course, we can't possibly identify all the conditions that could potentially cause an error message. However, the information in this Appendix should make it easier for you to determine why an error occurred in your program and how to correct it.

APPENDIX B

BASIC RESERVED WORDS

The keywords in the following list are considered to be "reserved". They are defined in BASIC as having a special meaning. Never use a BASIC reserved word as a variable in your programs, or your program will not execute successfully.

ABS	FN	NOT	SGN
AND	FOR	NULL	SIN
ASC	FRE	ON	SOUND
ATN	GOSUB	OR	SPC
CHR$	GOTO	OUTPUT	SQR
CLEAR	IF	PEEK	STEP
CLOAD	INP	PLOT	STOP
CLS	INPUT	POKE	STR$
COLOR	INSTR$	POS	TAB
CONT	INT	POT	TAN
COS	JOY	PRINT	THEN
CSAVE	LEFT$	READ	TO
DATA	LEN	REM	TONE
DEF	LET	RESTORE	USR
DIM	LIST	RETURN	VAL
DUMMY	LOG	REWIND	WAIT
END	MID$	RIGHT$	WINDOW
EXP	NEW	RND	LLIST*
FIRE	NEXT	RUN	LPRINT*

and the arithmetic operators:

+, -, *, /, >, <, ∧, =

Note that not all the keywords listed above are actually used by or usable in programming with BASIC. For example, the WAIT, INP, and DUMMY keywords have no function within BASIC. Attempting to use DUMMY, for example, can cause serious damage to your program execution. However, these words are still considered "reserved" by BASIC and should not be used as variable names.

* RS232 BASIC reserved words, not applicable to Microsoft 8K or Level II BASIC.

APPENDIX C

SUGGESTED PROGRAMMING REFERENCES

There are literally dozens of books, manuals, and learning packages designed to teach you about BASIC. If you need more help than this manual provides, we suggest you visit your local computer book store and review the selection. Try to find reference materials that are appropriate to your level of programming expertise. Also look for those which are directed toward the implementation of applications that are of interest to you.

Here are several references we think are worthwhile for learning to talk BASIC to your Interact:

Forsyth, Richard, The BASIC Idea -- An Introduction to Computer Programming. John Wiley & Sons, New York, 1978. A systemized overview of the BASIC language with numerous applications and extensions.

Micro Video Corporation, BOMBS AWAY! Programming Tutorial. Ann Arbor, MI, 1980. An excellent example of combined BASIC and machine language programming. Well-documented, with exercises for the advanced BASIC programmer.

Micro Video Corporation, Guide to ROM Subroutines. Ann Arbor, MI, 1980. Documents more than 18 subroutines in the Interact system ROM that intermediate to advanced programmers can invoke in BASIC through USR calls.

Ross, David L., "The Crowd Stopper", Creative Computing, January, 1981. A description of visual animation concepts developed and marketed by Micro Video on the Interact computer.

Texas Instruments, Calculator Analysis for Business and Finance. Dallas, TX, 1977. An excellent handbook for formulas and methods for solving business problems that can be implemented in BASIC programs for the Interact.

Waite and Pardee, BASIC Primer. Howard W. Sams & Co., Indianapolis, IN, An excellent, light-hearted introduction to BASIC programming for the beginner. The authors were formerly with Microsoft and the descriptions of several features in the book match exactly with Microsoft 8K Fast Graphics BASIC features on your Interact.

APPENDIX D

MATHEMATICAL FUNCTIONS

The following functions are outside the range of BASIC's intrinsic functions. However, you can use the DEF statement and existing built-in functions to use these calculations in your programs.

FUNCTION	TO DEFINE IN BASIC USING DEF
SECANT	FNSEC(X) = 1/COS(X)
COSECANT	FNCSC(X) = 1/SIN(X)
COTANGENT	FNCOT(X) = 1/TAN(X)
INVERSE SINE	FNARCSIN(X) = ATN(X/SQR(-X*X+1))
INVERSE COSINE	FNARCCOS(X) = -ATN(X/SQR(-X*X+1)+1.5708
INVERSE SECANT	FNARCSEC(X)=ATN(SQR(X*X-1))+(SGN(X)-1)*1.5708
INVERSE COSECANT	FNARCCSC(X)=ATN(1/SQR(X*X-1))+(SGN(X)-1) *1.5708
INVERSE COTANGENT	FNARCCOT(X) = ATN(X) + 1.5708
HYPERBOLIC SINE	FNSINH(X) = (EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	FNCOSH(X) = (EXP(X) + EXP(-X)/2
HYPERBOLIC TANGENT	FNTANH(X) = EXP(-X)/(EXP(X)+EXP(-X)*2+1
HYPERBOLIC SECANT	FNSECH(X) = 2/(EXP(X)+EXP(-X))
HYPERBOLIC COTANGENT	FNCOTH(X) = EXP(-X)/(EXP(X)-EXP(-X))*2+1
INVERSE HYPERBOLIC SINE	FNARCSINH(X) = LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	FNARCCOSH(X) = LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	FNARCTANH(X) = LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	FNARCSECH(X) = LOG ((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	FNARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X
INVERSE HYPERBOLIC COTANGENT	FNARCCOTH(X) = LOG((X+1)/(X-1))/2
A MOD B	FNMOD(X) = A-B*INT(A/B)