INTERACT™

USERS MANUAL

Level II BASIC Language

USERS MANUAL

Level II BASIC Language

INTERACT™ ELECTRONICS INC.
P.O. Box 8140 • Ann Arbor, Michigan 48106 • (313) 973-0120

PERSONAL COMPUTER

RETURN AUTHORIZATION FORM

CUSTOMER NAME _____ SERIAL NUMBER _____

ADDRESS _____ DATE OF PURCHASE _____

CITY, STATE, ZIP _____ PURCHASED FROM _____

REASON FOR RETURN: (PLEASE BE SPECIFIC)

_____

_____

_____

THE FOLLOWING QUESTIONS MUST BE ANSWERED:

1. Did you receive all the items you expected? If not, what was missing?

_____

2. Is there any apparent physical damage to any item? If yes, please explain.

_____

_____

3. Did unit function properly immediately after removal from its packing carton? If not, what doesn't work?

_____

_____

4. How long was unit running before it failed? _____

5. Have you tried to load all tapes supplied with unit? _____

6. List all tapes that failed to load or run properly.

_____

_____

_____

Signature _____ Date _____

NOTE: THIS FORM MUST BE FILLED IN COMPLETELY AND RETURNED WITH UNIT BEFORE ANY REPAIR WORK WILL BE DONE. PLEASE RETURN ALL ITEMS RECEIVED.

NOTE

When LEVEL II BASIC is first loaded, the RAM not occupied by BASIC is not cleared. This allows you to switch back and forth between BASIC and Interact's program editor without reloading your program. If you have no program statements in memory when you load BASIC, you must type NEW to clear the RAM or you will get OM (Out of Memory) errors when you try to input commands or statements.

CONTENTS

0. TUTORIAL

This section serves as a brief tutorial for those who are unfamiliar with computer programming. Important terms are defined and illustrated, and some short LEVEL II BASIC exercises and programs are presented. The reader is encouraged to load the LEVEL II BASIC language tape and experiment with the examples, exercises and sample programs in this section before continuing with the rest of this manual.

0.1 Computer and Programming Concepts

Your Interact Model One is a true computer. Although offering somewhat smaller capacity than many used in business and science, your Model One possesses the same computational and logic capabilities as these larger machines. By giving your Model One properly-stated instructions you can use it to perform an almost limitless number of tasks to help you and your family learn new skills, manage your home or your own business and have fun!

It will be easier for you to program your Model One if you have a basic understanding of how a computer works. Inside your Model One are two components of primary interest--the central processing unit (CPU) and memory. The CPU carries out your instructions when you give your Model One a command or use it to run a program. Memory is simply a place for the CPU to store your instructions and data. Let's look more closely at each of these components and their importance to you.

The CPU works on the principle that an electrical current may be on or off, just as a light switch may be on or off. We can represent "on" with a one and "off" with a zero. The CPU circuitry itself is designed to produce certain results such as addition and subtraction when fed combinations of on/off (0/1) pulses. One way to program a computer, then would be to give it the right series of 0's and 1's and get back your answer in 0's and 1's. How tedious and cumbersome that would be! Fortunately, no one has to program a computer that way. On your Model One computer you use the BASIC programming language instead of 0's and 1's.

Your LEVEL II BASIC language tape is really an interpreter. When loaded into your Model One it allows you to enter words and phrases that are easy for you to read and understand. The interpreter translates your words and phrases into the right sequence of 0's and 1's to give your instructions to the CPU in a language it can understand. It also translates the computer's answers from the computer's language--0's and 1's--into ours.

When you use BASIC on your Model One you give the interpreter a list of instruction which describe how the computer is to accomplish your desired task. Usually your instructions will describe what the CPU should do with information that you provide. The information may be numbers or text and is called data. The CPU needs a place to store your instructions and your data just as you need a file folder, drawer, cabinet or whatever in which to keep your records. This storage place

is called memory. Certain instructions and data that your Model One always requires regardless of the task are stored in read-only memory (ROM). This section of memory is protected such that it cannot be changed or cleared. Your Model One also has random-access memory (RAM). This is the space available for the LEVEL II BASIC interpreter instructions and your programs and data.

The RAM is like a large shelf with many empty, unlabelled boxes. When you place data into RAM you need a way to explain where to put it and where to get it when you need it again. To accomplish this you give the data a name called a variable name. The interpreter uses the name to label a "memory box" called a storage location in which to put your data. The interpreter also makes a note to itself about where the box can be found. When you want to put something else into the storage location--or take something out of it--you use the variable name and the interpreter tells the CPU where to find it.

In LEVEL II BASIC you may choose variable names of any length, as long as each begins with a letter. Many people like to choose names which are indicative of the data to which the names refer. However the interpreter only records the first two characters of the name. Therefore if you use longer names, make sure that each name begins with a different combination of two characters to avoid confusion about which storage location you want.

0.2 Direct Mode Tutorial

Now that you have a basic understanding of how your Model One works, let's look more closely at the BASIC vocabulary that your LEVEL II interpreter can translate. The interpreter can work in two modes of operation. You may give it an instruction to be carried out, or executed, immediately. These instructions are called direct mode commands. Or you may give it one or more instructions to be filed away and performed later at your command. These instructions are called indirect mode statements. If you type in a line number and then an instruction, it is stored as an indirect statement. If you type in the instruction without a line number, it is executed immediately as a direct mode command. We will begin in direct mode. You might want to load your LEVEL II BASIC language tape now, and try the commands presented below.

After you load your tape and the computer displays the "OK" message, clear your TV screen by typing

```
CLS
```

followed by the 'CR' key. 'CR' stands for "carriage return". The interpreter does not do its translation until you type a CR. Everything you type into your Model One should end with a CR. If you type an incorrect letter but have not yet typed a CR, simply backspace over the error and it will disappear. Then type in the correct letter and continue with the line from the point of error. If you want the whole line to be ignored so you can start it over and if you have not yet typed a CR, hold down the Control key and at the same time type a U. The line you were typing will be ignored by the computer but will stay on the screen.

Now let's input some data to work with. Type

```
LET A=5

LET B=3
```

A and B are variable names which identify storage locations in memory. As a result of these two instructions, the location labelled "A" now contains a 5; the one labelled "B" contains a 3. The word "LET" is optional--you could also type "A=5" and "B=3" and get the same results.

Now let's do some arithmetic with these data. The PRINT keyword tells the computer to display things on the TV screen. What you type after the PRINT tells your Model One what to display. Type

```
PRINT A+B

PRINT A-B

PRINT A*B        ("*" means "multiply")

PRINT A/B        (displayed as A/B)

PRINT A^ B       (" " means "to the power". Here this means 5^3.
                  To produce this symbol on the screen, use the
                  up-arrow above the + sign.)
```

Your session should appear on your screen like the listing below. In the listing, what you type has been underlined; what the computer prints is not. No underlines actaully appear on your screen.

```
 OK

 PRINT A+B

 8

 OK

 PRINT A-B

 2

 OK
```

```
PRINT A*B
15
OK
PRINT A/B
1.66667
OK
PRINT A ^B
125
OK
```

You may specify formulas for calculation which are quite complex. However, care must be taken because the Model One does arithmetic operations in a certain order:

1. First it performs all exponentiation (A ^B, C ^2, and so on).
2. Next it does all multiplication and division from left to right across your formula.
3. Finally it does all addition and subtraction from left to right across your formula.

Examples:

a. 3+5*2

First multiply 5*2 to get 10. Then add 3 to get 13.

b. 2-3 ^2

First square three to get 9. Then subtract 9 from 2 to get -7.

Now calculate and fill in your answers to the problems below using the above rules in your calculations. Then check your answers on your Model One using the direct mode commands listed in the table.

Problem	Your Answer	BASIC Direct Mode Command
A=3 B=4 C=2		
3*4-2^3 A*B-C^A	_____	PRINT 3*4-2 ^ 3
2:4*5	_____	PRINT 2:4*5
4*5:2	_____	PRINT 4*5:2

You can change the computer's standard order of operation by using parentheses. What appears inside the parentheses is calculated first.

Examples:

c. (3+5)^2

First add 3 and 5 to get 8. Then multiply by 2 to get 16.

Compare this with example a., above.

d. (2-3)^2

First subtract 3 from 2 to get -1. Square -1 to get +1.

Compare this with example b., above.

Now calculate and fill in your answers to the problems below using the above rules in your calculations. Then check your answers on your Model One using the direct mode commands given in the table. Please note that some of the direct mode commands may only show one line on the screen. A command or statement may use several lines in the screen as long as the entire command or statement is less than 255 characters long. Do not type a CR until you reach the end of the command. The computer automatically goes to a new line when necessary, but does not process your program until you type a CR.

Problem	Your Answer	BASIC Direct Mode Command
(2+3)*4÷2	_____	PRINT (2+3)*4÷2
(2+3)*(4-2)	_____	PRINT (2+3)*(4-2)
$((2+3)*4)^2$	_____[1]	PRINT ((2+3)*4)^2

In the exercises you have done so far, you asked the Model One to calculate an answer and print it right away. To do this you used the PRINT command. You could just as easily have asked it to record a new storage location name and store the answer to the calculation. To do this, you would use the LET command:

```
LET C=A+B
LET D=A-B
```

and so on. The Model One would compute the answers and store them for later use in printing and making other calculations, using the names you give to label the storage locations which contain the results.

There are several ways in addition to the LET command to put data into memory. These other methods are not fully discussed here. You are encouraged to read about them in section 2-3 of this manual after you are comfortable with the material in this tutorial section. Some other methods for inputting data are summarized below:

1. Data may be loaded into memory from DATA statements using the READ command. The primary advantage to the READ. . .DATA combination is that all your data values appear in one list that you can easily locate, verify and change.

---

1 When more than one set of parentheses appears, the inside of the innermost is evaluated first, then the next innermost, and so on.

2. Data may be accepted from the keyboard while an indirect mode program is running using the INPUT statement or the INSTR$ function. This allows you to interact with your program as it runs, influencing the order in which instructions are executed and/or providing different data values to use in calculations.
3. Data may be loaded into memory from cassette tape using CLOAD*.
4. Data may be accepted into memory from the joystick controls using FIRE, JOY and POT. You can write your own games using these and other statements such as COLOR and SOUND.

0.3 Introduction to Indirect Mode

Up to this point you have been using your Model One in direct mode. You typed in commands which were executed immediately. Direct mode is useful for quick calculations, experimentation and for developing indirect mode programs. This section explores the Model One's indirect mode capabilities.

Indirect mode is used to define and execute a series of instructions called statements. These statements form what is typically called a program. The direct mode commands and operations you used in the previous section also work in indirect mode. To create and run an indirect mode program you:

1. Type in the NEW command to tell the Model One to clear its memory for new entries.

2. Type in your instructions preceeded by line numbers. A line number may be any whole number (integer) between 0 and 65529. It is important to remember that unless you tell the computer otherwise, statements are executed in line number order. It is a good idea to separate successive line numbers by 5 or 10 to leave room in case you want to add lines later on.

3. Use the LIST command to display your instructions in line number order. Using the LEVEL II BASIC Reference Card or this manual verify that your instructions are correctly entered. If not, correct the errors as follows:

   a. To delete an entire line, type the line number to be deleted followed by a CR. Although the line may remain displayed on your screen by a prior listing, it has been removed from your program and will not appear in any later listings.

   b. To change a line, type the line number to be changed followed by the entire corrected statement for that line. Although the line may remain incorrectly displayed in prior listing on your screen, it has been corrected in your program and will appear correctly in any later listings.

   c. To add a new line, type in an appropriate unused line number followed by the new statement for that line. Remember when you pick the new line number that statements are executed in line number order. The new line will automatically appear in its proper place in any later listing of the program.

4. Type in a RUN command to execute the program.

0.4 Indirect Mode Example

Let's use the commands from the direct mode section and a few new ones to create a program. Suppose you want to analyze your electric bills. Specifically, you want to find your average bill over the past year. Admittedly there may be easier ways to compute your average bill than to write a BASIC program for your Model One. However, this example makes a good exercise for learning to program in BASIC.

We know we will need 12 storage locations in which to put the amounts from each electric bill. We could name these locations JAN, FEB, MAR and so on. However the program will require fewer, simpler statements if we use a special variable called an array variable. An array variable gives one name to a series of storage locations. When you want one of the locations you use a subscript, as in the example below. You use a DIM statement to tell the LEVEL II interpreter how many locations are in the series. "a subscript picks out one of the locations in the DIM statement."

Example

For our utility analysis, we need 12 locations for the electric bill data, which we will call B. To reserve 12 locs we use the statement

```
DIM B(12)
```

This created twelve BD slots which we may reference individually using the subscripts 1,2,...,11,12 to make the names BD(1), BD(2),...BD(11), BD(12).

Now get out your checkbook or other records of your electric bills. Or use the numbers provided in the example in your program. First clear the TV screen with the CLS command. Then type a NEW command to tell the Model One that you will begin typing in a new program. Type in the first part of your program as follows:

Statement	Explanation
10DIM BD(12)	Reserves 12 storage locations for electric bill data
20BD(1)= 75.86 30BD(2)= 79.13 40BD(3)= 60.32 50BD(4)= 35.58 60BD(5)= 20.16 70BD(6)= 11.59 80BD(7)= 10.02 90BD(8)= 11.43 100BD(9)= 18.29 110BD(10)=29.74 120BD(11)=46.66 130BD(12)=57.92	Statements 20-130 puts your electric bill data into the 12 BD storage locations. If you want to use your own data, substitute the amount of one of your bills for each of the example amounts.

Now to compute the average bill, we'll need to add up BD(1) through BD(12). Here's where using an array variable helps us save time and effort telling the computer to add the twelve months of electric bills. An easy way to state these instructions is to use a FOR...NEXT loop. A loop defines a series of instructions which are to be performed several times. A control variable is used to specify how many times a loop should be performed.

Examples

1. FOR I=1 TO 12

The control variable in this loop is named I. It is common programming practice to use I, J, K, L, M or N as names for loop control variables. This FOR statement defines the beginning of a loop which will be performed 12 times--once for I=1, once for I=2 and so on up to and including I=12.

2. This example shows an entire loop:

```
10DIM A(10)
20FOR I=1 TO 10
30PRINT A(I)
40NEXT I
```

Statement 10 reserves ten storage locations for the array variable named A. These are referenced individually as A(1), A(2)...A(9), A(10). Statement 20 defines the beginning of a loop which is performed ten times--once for I=1, once for I=2 and so on up to and including I=10. Statement 30 is "inside the loop" so it will be executed ten times. The first time through the loop, statement 30 prints the data stored in location A(1). The second time through the loop, it prints the data in A(2). The loop continues through I=10, printing the value from one location in the A array each time through the loop. Statement 40 defines the end of the loop.

In our electric bill analysis, we can use a loop which is performed twelve times to add up the twelve months of electric bills. Each time through the loop, we add in another month's data. Type in the following statements, adding them to your program:

Statement	Explanation
140FOR I=1 TO 12 150SUM=SUM+BD(I)	Initiates loop to be performed 12 times. Each time through the loop another month of electric bill data is added into the running total which is kept in a location called SUM.
160NEXT I	Defines the end of the loop.

Now all that remains is to divide by 12 and print the resulting average.

Type in

```
170?"AVERAGE=";SUM:12
```

The question mark stands for PRINT. Anywhere that you want to use PRINT, you may type a ? instead. The interpreter substitutes the word PRINT for your question mark. "AVERAGE=" is called a string. A string is printed exactly as it appears inside the quotes. The semi-colon tells the Model One to display the results of the calculation immediately after the string, for example:

```
AVERAGE=56.72
```

Let's review the whole program. To see a list of the statements you have typed in, use the LIST command. Your program statements will begin to appear on the screen in line number order. To freeze the listing temporarily so you can inspect lines more carefully, press the Control key and at the same time press an S. The Model One will finish listing the current line and then will pause. To continue the listing from where it left off, ~~hold down the Control key and at the same time type a Q~~ Press S. The listing will continue from the line at which it was stopped by the Control/S. Type a LIST command now and compare your program with the listing below:

```
10DIM BD(12)
20BD(1)= 75.86
30BD(2)= 79.13
40BD(3)= 60.32
50BD(4)= 35.58
60BD(5)= 30.16
70BD(6)= 11.59
80BD(7)= 10.02
90BD(8)= 11.43
100BD(9)= 18.29
110BD(10)=29.74
120BD(11)=46.66
130BD(12)=57.92
140FOR I=1 TO 12
150SUM=SUM+BD(I)
160NEXT I
170PRINT "AVERAGE=";SUM/12
```

(Note: If you supplied your own data, your electric bills should appear in lines 20-130 substituted for the numbers shown in this listing.)

(The Model One automatically substitutes PRINT for ? in your listing.)

If there are errors in your listing correct them using the methods described in the previous section. When the listing appears to be correct, execute your program by typing

```
RUN
```

What was your average electric bill?

38.05

0.5 Review

Before going on, let's pause a minute and summarize the things we have covered so far. If you are unsure about the meaning of any of the words, symbols or rules below you are encouraged to experiment some more with the exercises and examples in sections 0.1-0.4. If you are comfortable with the lists, go on to section 0.6

A. Computer Concepts and Terms

array variable	loop
central processing unit (CPU)	memory
control variable	modes of operation
data	program
direct mode commands	RAM
execute	ROM
indirect mode statements	storage location
interpreter	string
line number	variable name

B. LEVEL II BASIC Words

CLS	LET	NEXT	PRINT or ?
DIM	LIST	NEW	RUN
FOR			

C. Editting and Control

backspace clears last figure
Control/Q starts list
Control/S stops list
Control/U ignores whole line

deleting a line - Type line # then CR
adding a line - Type line # then new statement (new line #)
changing a line - Type line # then new statement

D. Arithmetic Operators and Rules

= addition
- negation, subtraction
* multiplication
/ division (prints as a slash mark)
^ exponentiation

Rules for order of operation, use of parentheses.

anything in ( ) is done first.

0.6 Interactive Input

In your first computer program, you specified your data in program statements like

```
BD(1)=75.86
```

When the computer executes this statement, it stores one month of electric bill data in storage location BD(1). As was mentioned, another way to put data into memory is to enter it while the program is running. To do this, the INPUT statement is used.

Let's start writing a new version of the electric bill program. The first change is to make the input interactive. That is, use the INPUT statement so that you can enter data when the computer asks for it while the program is running. INPUT is not used as a direct mode command but only as an indirect mode statement.

The form of the INPUT statement is

```
INPUT ["string";]<variable name>
```

This way of showing the INPUT statement is called a general form because it describes many ways to use INPUT. The square brackets are used to show options. That is, you may use INPUT without using a string in quotes and a semi-colon. The angle brackets denote a parameter. When you use a statement, you substitute a value for the parameter that is appropriate for your specific task. In the case of INPUT you would supply the name of the variable you want to enter in place of <variable name>. .

Examples

a. 10 INPUT A

When this statement is executed the Model One prints a question mark on the TV screen. Then it waits for you to type a value followed by a CR. It stores the value you type in a storage location labelled A.

b. 10 INPUT "NUMBER";A

The Model One prints

NUMBER?

and waits until you type in a value and a CR, then stores the value in a location named A.

If you supply an optional string and a semi-colon in an INPUT statement, the Model One prints the string then a question mark then waits for your input. If you don't use a string it prints a question mark only, then waits for your input. In either case, it stores the data you enter in the storage location you name in the INPUT statement.

Now clear your TV screen, type in a NEW command and start your new home budget analysis program by typing in the statements listed below.

Statement	Explanation
10DIM D(12)	Reserves 12 storage locations
20FOR I=1 TO 12	Initiates a loop to be performed 12 times
30INPUT D(I)	Prints a question mark and accepts data for one location in the D array each time the loop is performed.
40NEXT I	Closes the loop initiated by line 20

When using interactive input to enter a lot of data it is usually a good idea to give yourself a chance to fix any mistakes you may have made. Let's do that as follows. First, we'll print out all twelve numbers along with a sequence number to identify each. Then we'll ask if there are any changes. If there are, we'll make them. If not, we'll compute the average bill. Before we enter program statements to do this, we'll need to introduce two new concepts--conditional clauses and subroutines.

0.7 Conditional Clauses

When writing computer programs it is frequently necessary to test to see if a particular condition holds true or not. For example, in our editing example we will want to ask if there are any changes. If the answer is "yes," we'll edit them. If the answer is "no," we can go on and compute our average. That is, we need to test the answer to a question to see if it's a "yes" or a "no." Other kinds of tests that you may need to make are:

a. Is this data value equal to that data value?

b. Is it less than a certain value?

c. Greater than?

To make these kinds of tests you use symbols called relational operators. That's a fancy name for something familiar:

Relational Operator	Meaning
=	equal to
<>	not equal to
<	less than
>	greater than
<=or=<	less than or equal to
>=or=>	greater than or equal to

Relational operators are used to form conditional clauses. The form of a conditional clause is

IF<expression1><relational operator><expression2>

Examples

a. IF SUM<0

This clause tests to see if the value stored in the SUM location is negative.

b. IF A>=B

This clause tests to see whether or not the value in location A is greater than or equal to that in B.

c. IF A^2>10*B

This clause tests to see if the square of the value in A is greater than 10 times the value in B.

A conditional clause is either true--has a value equal to -1--or it is false--has a value equal to zero. If the clause is true, you want to do one thing. If it is false, you want to do something else. There are several ways to use a conditional clause to control what gets done next. The one we will use is the IF...THEN statement:

IF<condition>THEN<statements>

If the conditional clause you substitute for <condition> is true, the Model One performs the statements following THEN. If the clause is false it goes to the next line, skipping the statements after THEN.

Example

```
10IF A<0 THEN PRINT "ERROR":STOP
20C=A/B
```

If A is less than zero the Model One will display the word "ERROR" on the screen and then stop. If A is greater than or equal to zero, it will compute A divided by B and store the result in C.

Note the colon in line 10. You may use a colon to put more than one BASIC statement on any line. You may put as many statements on a line as you want as long as they are separated by colons and the total length of the line is not more than 72 characters.

Let's expand our program now to include displaying the input data, asking if there are changes, testing the answer, and making a decision based on the answer. Type in the following statements, adding them to your program:

Statement	Explanation
50FOR I=1 TO 12	Initiates a loop
60PRINT I;SPC(2);D(I)	Prints current value of I (our sequence number) then 2 spaces then the value in D(I) each time through the loop.
70NEXT I	Closes the loop
80INPUT "CHANGES";A$	Prints "CHANGES?" and waits for an answer to store in A$
90IF A$="YES" THEN GOSUB 200	Continues at line 200 if answer was YES

In line 80 you requested that the data input be stored in a location labelled A$. Variable names that end with a dollar sign are called string variables because they label locations for text (letter) data instead of numbers. The answers "YES" and "NO" are text, so we need a location with a label ending in a dollar sign.

In line 90, you stated that if the answer is "YES" the Model One should go execute a subroutine beginning at line 200. The next section explains subroutines and develops the one we need for our program.

Now enter these statements:

Statement	Explanation
100FOR I=1 TO 12	Initiates loop to total 12 months of data.
110LET SUM=SUM+D(I)	Accumulates total.
120NEXT I	Closes loop.
130PRINT "AVERAGE=";SUM/12	Prints average bill.
140END	Marks the end of the program.

These statements are performed only after the data are verified to be correct. That is, only after you type NO when asked "CHANGES?"

0.8 Subroutines

A subroutine is a series of program statements contained within the rest of the program. The subroutine statements usually perform a task that is:

a. Not really central to the task at hand. For example editing entries to INPUT statements, although sometimes very necessary, is not central to computing an average. After all, if we were perfect typists we wouldn't need to edit our input at all!

b. Performed at several times at different points in the program. Rather than type in the same series of statements at each point they are needed you type them in once instead. Then you refer to them as a subroutine when you need them.

To initiate execution of a subroutine, you use the statement

GOSUB <linenumber>

where <linenumber> gives the first line of the subroutine series. The Model One automatically skips to that line and continues from that point, processing line by line until it finds a RETURN statement. The RETURN causes the Model One to go back to the statement immediately after the GOSUB.

Example

```
50GOSUB 60:PRINT "DONE":STOP
60FOR I=1 TO 10
70PRINT I
80NEXT I
90RETURN
```

When the Model One executes line 20, it skips automatically to the subroutine beginning at line 60. The subroutine displays the numbers 1 through 10 on the screen one by one. Then the Model One goes back to line 50, prints the word DONE and then stops.

Let's write the editing subroutine for our program now. In your program you already have the statement

```
90IF A$="YES" THEN GOSUB 200
```

That means the subroutine must start at line 200. The strategy for the subroutine is as follows:

1. In the main program we printed a list of the input data items with the I value which tells where each item is stored in the D array. So we'll use that same I value to identify which month of data needs to be changed. Type in the following statement, which gives you a chance to type in an I value when the subroutine is run:

   200INPUT "SEQ.NO.";I

2. The following statement allows you to enter the new data value for D(I):

   210INPUT "NEW VALUE";D(I)

3. Now we need to know if there are any other changes. If there are we want to go back to line 200 and get another value for I:

   220INPUT "MORE CHANGES";B$

   230IF B$="YES" GOTO 200

   240RETURN

   Line 230 uses the IF...GOTO statement. The IF...GOTO is very similar to the IF...THEN. The IF...THEN gives statements to perform if the stated condition is true. IF...GOTO gives a line number at which to continue if the stated condition is true. If the condition is false, the Model One continues with the next line after the IF...GOTO. In this case, unless you enter YES to the question "MORE CHANGES?" the Model One returns to the body of your program at line 100, which follows the GOSUB. If you answer YES you are asked for another I value to specify another data value to change.

Let's look at the entire home budget program. Type a LIST command and use Control/S and Control/Q as needed to stop and start the listing. Check your statements with the listing below:

```
 10DIM D(12)
 20FOR I=1 TO 12
 30INPUT D(I)
 40NEXT I
 50FOR I=1 TO 12
 60PRINT I;SPC(2);D(I)
 70NEXT I
 80INPUT "CHANGES";A$
 90IF A$="YES" THEN GOSUB 200
100FOR I=1 TO 12
110LET SUM=SUM+D(I)
120NEXT I
130PRINT "AVERAGE=";SUM/12
140END
200INPUT "SEQ.NO.";I
210INPUT "NEW VALUE";D(I)
220INPUT "MORE CHANGES";B$
230IF B$="YES" GOTO 200
240RETURN
```

Your program performs the following tasks:

1. Accepts 12 months of electric bills from you while the program is running, typing a question mark for each bill amount. It puts the data into the array labelled D, which has 12 slots.
2. Next it displays the 12 numbers you entered on the screen along with the position of the number in the D array.
3. Next it asks you if you want to make any changes. If you answer YES, it proceeds to the editing subroutine. If you answer NO (or anything except YES) it totals the 12 numbers and prints the average then stops.
4. The editing subroutine asks which item you want to change by typing SEQ.NO.? then

   NEW VALUE?

   to get the correct entry for D(I). Then it asks if there are more changes. If you answer YES it asks for a new sequence number and a

value. Any other answer causes the Model One to return to the main program, compute the sum, print the average and stop.

If your program statements appear correct type in a RUN command and try out the program. If there are errors use the methods described in section 0.3 to fix them, then type the RUN command. While running the program make sure to make at least one mistake so you can try out your editing subroutine. After you are done experimenting go on to the next section to see how to save your program on a cassette tape. Be sure not to turn your Model One off until you have saved your program on tape. When you turn the Model One off, whatever data values and program statements are in the memory are lost.

0.10 Saving Programs on Cassette Tape

You may save your program on a blank Data Tape by following the steps listed below:

1. Insert a blank Data Tape into the cassette drive. Depress the REWIND button on the tape unit.
2. Type in a REWIND command. When the tape finishes rewinding press any key on the keyboard to indicate that rewinding is complete.
3. Depress the READ and WRITE buttons simultaneously on the tape unit. Pick a name up to five characters long for your program. Type in the word CSAVE--for Cassette Save--followed by your program name in quotes, for example

   CSAVE "BDGET"

   The tape should begin to turn as your program is written onto the tape. When the Model One types "OK" your program is saved.

To load the program from tape later on, follow the steps listed below. Please note that when you load a program from tape it automatically erases any program statements and data values currently in memory.

1. Load the LEVEL II BASIC language tape if you have not already done so.
2. Insert the Data Tape containing your program. Depress the REWIND cassette button and type a REWIND command. Type any key on the keyboard when rewinding is complete.
3. Depress the READ cassette button. Type in the word CLOAD--for Cassette Load--followed by your program name in quotes, for example

   CLOAD "BDGET"

4. You should hear the beeping sounds of a loading program. To make certain your program loaded correctly type in a LIST command after the Model One displays the "OK" message. Check the listing to make sure it appears complete and correct. If so, type a RUN command to execute your program.

0.11 Review

Let's review what we have covered since section 0.5 when we did the last review. If you are unsure about the meaning of any of the words, statements or symbols listed below you are encouraged to experiment some more with the exercises and examples in sections 0.6-0.10. If you are comfortable with the lists, try some of the sample programs in Appendix F or continue with the rest of this manual.

a. Computer Concepts and Terms

conditional clause	parameter
general form	relational operators
interactive	string variables
options	subroutines

b. LEVEL II BASIC words

CLOAD	IF	RETURN
CSAVE	IF...GOTO	SPC
END	IF...THEN	STOP
GOSUB	INPUT	

c. Relational Operators

=	equal to	<=or=<	less than or equal to
<>	not equal to	>=or=>	greater than or equal to
<	less than		
>	greater than		

d. Other

-Use of colon(:) to separate multiples statements on one line
-Saving programs on tape, loading programs from tape
-Using "?" as an abbreviation for "PRINT"

1. GENERAL GUIDELINES

1-1 Introduction to this Manual.

a. Conventions. For the sake of simplicity, some conventions will be followed in discussing the features of the BASIC language.

1. Words printed in capital letters must be written exactly as shown. These are mostly names of instructions and commands.

2. Items enclosed in angle brackets (<>) must be supplied as explained in the text. Items in square brackets ([]) are optional. Items in both kinds of brackets, [<W>], for example, are to be supplied if the optional feature is used. Items followed by dots (...) may be repeated or deleted as necessary.

3. Shift/ or Control/ followed by a letter means the character is typed by holding down the Shift or Control key and typing the indicated letter.

4. All indicated punctuation must be supplied.

b. Definitions. Some terms which will become important are as follows:

Alphanumeric character: all letters and numerals taken together are called alphanumeric characters.

Carriage Return: Refers to the key labeled 'CR' on the terminal which causes commands, statements, or data to be entered into memory, and printing to begin on a new line on the screen.

Command Level: After BASIC prints OK, it is at the command level. This means it is ready to accept commands.

Commands and Statements: Instructions in BASIC are loosely divided into two classes, Commands and Statements. Commands are instructions normally used only in direct mode (See Modes of Operation, section 1-2). Some commands, such as CONT, may only be used in direct mode since they have no meaning as program statements.

But most commands will find occasional use as program statements. Statements are instructions that are normally used in indirect mode. Some statements, such as DEF, may only be used in indirect mode, but most may also be issued as direct mode commands.

Edit: The process of deleting, adding and substituting lines in a program.

Integer Expression: An expression whose value is truncated to an integer. The components of the expression need not be of integer type.

Pixel: A pixel is the unit of measure for the TV screen. The screen is approximately 112 pixels wide, and 77 pixels tall.

Reserved Words: Some words are reserved by BASIC for use as statements and commands. These are called reserved words because they may not be used in variable or function names. See section 5-6.

String Literal: A string of characters enclosed by quotation marks (") which is to be input or output exactly as it appears. The quotation marks are not part of the string literal, nor may a string literal contain quotation marks. (""HI, THERE"" is not legal.) Blanks within the quotation marks are significant.

Type: The word "type" to refers to the process of entering information into the computer using the keyboard. The user types, the computer prints. "Data type" refers to the classification of data as numbers or strings.

1-2 Modes of Operation.

BASIC provides for operation of the computer in two different modes. In the direct mode, the statements or commands are executed as they are entered into the computer. Results of arithmetic and logical operations are displayed and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC in a "calculator" mode for quick computations which do not justify the design and coding of complete programs.

In the indirect mode, the computer executes instructions from a program stored in memory. Program lines are entered into memory if they are preceded by a line number. Execution of the program is initiated by the RUN command. Lines are always executed in numerical order, regardless of the order in which they are input.

1-3. Formats.

a. Lines. The line is the fundamental unit of a BASIC program. The format for a BASIC line is as follows:

nnnnn <BASIC statement>[:<BASIC statement>...]

Each BASIC line begins with a line number. The line number indicates the order in which the statements are executed in the program. It also provides for branching linkages and for editing. Line numbers must be in the range 0 to 63529. A good programming practice is to use an increment of 5 or 10 between successive line numbers to allow for insertions.

Following the line number, one or more BASIC statements are written. The first word of a statement indentifies the operations to be performed. The list of arguments which follows the identifying word serves several purposes. It can contain the data or variables which are to be operated upon by the statement. In some important instructions, the operation to be performed depends upon conditions or options specified in the list. Each type of statement will be considered in detail in sections 2, 3 and 4.

Several statements can be written after one line number if they are separated by colons (:). Any number of statements can be joined this way provided that the line is no more than 72 characters long.

b. REMarks. In many cases, a program can be more easily understood if it contains remarks and explanations as well as the statements of the program proper. In BASIC, the REM statement allows such comments to be included without affecting execution of the program. The format of the REM statement is as follows:

REM <remarks>

A REM statement is not executed by BASIC, but branching statements may link into it. REM statements are terminated by the carriage return or the end of the line but not by a colon. Example:

```
10Ø REM DO THIS LOOP:FOR I=1T01Ø          -the FOR statement will not
                                           be executed
101 FOR I=1 to 1Ø: REM DO THIS LOOP       -this FOR statement will
                                           be executed.
```

c. Errors. When the BASIC interpreter detects an error that will cause the program to be terminated, it prints an error message. The error message formats in BASIC are as follows:

Direct statement ?XX ERROR

Indirect statement ?XX ERROR IN nnnnn

XX is the error code or message (see section 5-5 for a list of error codes and messages) and nnnnn is the line number where the error occurred. Each statement has its own particular possible errors in addition to the general errors in syntax. These errors will be discussed in the description of the individual statements.

1-4 Editing - elementary provisions.

Editing features are provided in BASIC so that mistakes can be corrected and features can be added and deleted without affecting the remainder of the program. If necessary, the whole program may be deleted.

a. Correcting lines. A line being typed may be deleted by typing Control/U instead of typing a carriage return. To delete an entire line that has already been entered, type the line number followed by a carriage return. To correct a line that is already typed in, type the line number followed by the correct information. To add a new line, pick an appropriate line number, and enter it along with the new information. Remember that statements are always executed in line number order.

b. Correcting whole programs. The NEW command causes the entire current program and all variables to be deleted. NEW is generally used to clear memory space prior to entering a new program.

2. STATEMENTS AND EXPRESSIONS.

2-1 Expressions.

The simplest BASIC expressions are single constants, variables and function calls.

a. Constants. BASIC accepts integers or floating point real numbers as constants. It accepts string constants as well. See section 4-1. Some examples of acceptable numeric constants follow:

```
123
3.141
Ø.Ø436
1.25E+Ø5
```

Data input from the terminal or numeric constants in a program may have any number of digits up to the length of a line (see section 1-3a). However, only the first 7 characters of a number (including the decimal point) are significant and the seventh digit is rounded up. Therefore, the command

```
PRINT 1.2345678ØØ123
```

produces the following output:

```
1.23457
OK
```

The format of a number displayed using PRINT or OUTPUT is determined by the following rules:

1. If the number is negative, a minus sign (-) is printed to the left of the number. If the number is positive, a space is printed.
2. If the absolute value of the number is an integer in the range Ø to 999999, it is printed as an integer.
3. If the absolute value of the number is real, and greater than or equal to .Ø1 and less than or equal to 999999, it is printed in fixed point notation with no exponent.
4. If the number does not fall into categories 2 or 3, scientific notation is used.

The format for input and output of constants in scientific notation is:

MX.XXXXXESTT

Where M is the sign of the mantissa and the X's are the digits of the mantissa. The E indicates the start of the exponent, the S the sign of the exponent, and the T's the digits of the exponent. The exponent must be between -38 and +38. The largest number that may be represented in BASIC is 1.70141E+38, the smallest positive number is 2.9387E-38.

Examples:

BASIC Scientific Notation	Exponential Notation	Number
1.5684E+Ø6	$(1.5684)_x(1Ø^6)$	1,568,4ØØ
-1.5684E+Ø6	$(-1.5684)_x(1Ø^6)$	-1,568,400
1.5684E-Ø3	$(1.5684)x(10^{-3})$	0.0015684
-1.5684E-Ø3	$(-1.5684)_x(10^{-3})$	-0.0015684

In all formats, a space is printed after the number. BASIC checks to see if the entire number will fit on the current line. If not, it issues a carriage return and prints the whole number on the next line.

b. Variables

A variable name represents symbolically any number or string which is assigned to it. The value of a variable may be assigned explicitly by the programmer or may be assigned as the result of calculations in a program. Before a variable is assigned a value, its value is assumed to be zero (numbers) or blanks (strings). String variables have special names. See section 4.

A numeric variable name may be any length, but any alphanumeric characters after the first two are ignored. The first character must be a letter. No reserved

words may appear as variable names or within variable names. The following are examples of legal and illegal BASIC variables:

Legal	Illegal
A	%A (first character must be alphabetic.)
Z1	LET (reserved word)

The first two characters of all variable names in a program (indirect mode) or session (direct mode) must be unique.

c. Array Variables. It is often advantageous to refer to several variables by the same name. In matrix calculations, for example, the computer handles each element of the matrix separately, but it is convenient for the programmer to refer to the whole matrix as a unit. For this purpose, BASIC provides subscripted variables, or arrays. The form of an array variable is as follows:

VV(<subscript>[,<subscript>...])

where VV is a variable name and the subscripts are integer expressions. Subscripts may be enclosed in parentheses or square brackets. An array variable may have as many dimensions as can be defined in a single DIM statement of up to 72 characters. Subscripts must be between 0 and 32767.

Examples:

A(5)	The sixth element of array A. The first element is A(Ø).
ARRAY(I,2*J)	The address of this element in a two-dimensional array is determined by evaluating the expressions in parentheses at the time of the reference to the array and truncating to integers. If I=3 and J=2, this refers to ARRAY(3,4).

The DIM statement allocates storage for array variables and sets all array elements to zero. The form of the DIM statement is as follows:

DIM VV(<subscript>[,<subscript>...])

where VV is a legal variable name. Subscript is an integer expression which specifies the largest possible subscript for that dimension. Each DIM statement may apply to more than one array variable. Some examples follow:

113 DIM A(3), B(1,1)

The array A may contain four values, referred to as A(0), A(1), A(2), and A(3). The array B defines a 4 cell, 2-dimensional matrix with each element referenced as shown:

B(0,0) B(1,0)

B(0,1) B(1,1)

```
110 INPUT N
111 DIM AA(N)
```

The array AA is dynamically dimensioned during program execution. That is, N+1 value positions are allocated, where N is input each time the program is run. These positions are referenced as AA(0), AA(1), AA(2), ... AA(N).

Any integer expression may be used to dimension an array or matrix dynamically. When the program is run, the expression is evaluated, the results truncated to an integer value N, and N+1 positions are allocated for that dimension in the array. If no DIM statement has been executed before an array variable is found in a program, BASIC assumes the variable to have a maximum subscript of 1Ø (11 elements) for each dimension in the reference. A BS or SUBSCRIPT OUT OF RANGE error message will be issued if an attempt is made to reference an array element which is outside the space allocated in its associated DIM statement. For example: 50 LET A(11)=COS(X) when A has been dimensioned by 20 DIM A(10). A BS error can also occur when the wrong number of dimensions is used in an array element reference. For example:

3Ø LET A(1,2,3)=X when A has been dimensioned by 1Ø DIM A(2,2)

A DD or REDIMENSIONED ARRAY error occurs when a DIM statement for an array is found after that array has been dimensioned. This often occurs when a DIM statement appears after an array has been given its default dimension of 1Ø.

d. Operators and Precedence. BASIC provides a full range of arithmetic and logical operators. The order of execution of operations in an expression is always according to their precedence as shown in the table below. The order can be specified explicitly by the use of parentheses in the normal algebraic fashion.

Table of Precedence

Operators are shown here in decreasing order of precedence. Operators listed in the same entry in the table have the same precedence and are executed in order from left to right in an expression.

1. Expressions enclosed in parentheses ()
2. ^exponentiation. Any number to the zero power is 1. Zero to a negative power causes a /Ø or DIVISION BY ZERO error.
3. - negation, the unary minus operator
4. *,/ multiplication and division
5. +,- addition and subtraction
6. relational operators

=	equal
<>	not equal
<	less than
>	greater than
<=,=<	less than or equal to
>=,=>	greater than or equal to

7. NOT logical, bitwise negation
8. AND logical, bitwise disjunction
9. OR logical, bitwise conjunction

Relational operators may be used in any expression. Relational expressions have the value either of True (-1) or False (Ø).

e. Logical Operations. Logical operators may be used for bit manipulation and Boolean algebraic functions. The AND, OR, and NOT operators convert their arguments into sixteen bit, signed, two's complement integers in the range -32768 to 32767. After the operations are performed, the result is returned in the same form and range. If the arguments are not in this range, an FC (ILLEGAL FUNCTION CALL) error message will be printed and execution will be terminated. Truth tables for the logical operators appear below. The operations are performed bitwise, that is, corresponding bits of each argument are examined and the result computed one bit at a time. In binary operations, bit 15 is the most significant bit of the two-byte word, and bit Ø is the least significant.

AND

X	Y	X AND Y
1	1	1
1	Ø	Ø
Ø	1	Ø
Ø	Ø	Ø

OR

X	Y	X OR Y
1	1	1
1	Ø	1
Ø	1	1
Ø	Ø	Ø

NOT

X	NOT X
1	Ø
Ø	1

The following examples of logical operations use the numbers 1, 2, 4, 16, and 63. The numbers are written in binary notation as follows:

Number	Binary Form
1	1
2	10
4	100
16	10000
63	111111

16 AND 63

```
    111111
AND  10000
    010000  = 16
```

2 AND 4

```
     100
AND   10
     000  = 0
```

16 OR 63

```
    111111
OR   10000  = 63
    111111
```

2 OR 4

```
     100
OR    10   = 6
     110
```

The NOT operator produces a "one's complement" of the variable, i.e. -(variable +1). For example, NOT 0 = -(0+1) = -1, and NOT 1 = -(1+1)= -2.

f. The LET statement. The LET statement is used to assign a value to a variable. The form is as follows:

```
LET <VV>= expression
```

where VV is a variable name and the expression is any valid BASIC arithmetic, logical, or string expression. Examples:

```
1ØØ  LET V=X     the value of X is assigned to variable V.
11Ø  LET I=I+1   the '=' sign here means 'is replaced by ....'.
                 That is, the value of I is incremented by 1.
```

The word LET in a LET statement is optional, so algebraic equations such as:

```
12Ø V=.5*(X+2)
```

are legal assignment statements.

A SN or SYNTAX ERROR message is printed when BASIC detects incorrect form, illegal characters in a line, incorrect punctuation or missing parentheses. An OV or OVERFLOW error occurs when the result of a calculation is too large to be represented by BASIC's number formats. All numbers must be within the range 1E-38 to 1.7Ø141E38 or -1E-38 to -1.7Ø141E38. An attempt to divide by zero results in the /Ø or DIVISION BY ZERO error message.

For a discussion of strings, string variables and string operations, see section 4.

2-2 Branching, Loops and Subroutines.

a. Branching. In addition to the sequential execution of program lines, BASIC provides for changing the order of execution. This provision is called branching and is the basis of programmed decision making and loops. The statements in BASIC which provide for branching are the GOTO, IF...THEN and ON...GOTO statements.

1. GOTO is an unconditional branch. Its form is as follows:

GOTO <mmmmm>

After the GOTO statement is executed, execution continues at line number mmmmm.

2. IF...THEN is a conditional branch. Its form is as follows:

IF <expression> THEN <mmmmm>

where the expression is a valid arithmetic, relational or logical expression and mmmmm is a line number. If the expression is evaluated as non-zero (TRUE), BASIC continues at line mmmmm. Otherwise, execution resumes at the next line after the IF...THEN statement.

An alternate form of the IF...THEN statement is as follows:

IF <expression> THEN <statements>

where the statements are any BASIC statements. Examples:

```
1Ø IF A=1Ø THEN 4Ø:  If the expression A=1Ø is true, BASIC branches to
   line 4Ø.  Otherwise, execution proceeds at the next line.
```

15 IF A<B+C OR X THEN 1ØØ: If A is less than B + C, or if X is not equal to zero, execution proceeds at line 100. If A is greater than B + C and X=0, then the next program statement is executed.

2Ø IF X THEN 25: If X is not zero, the statement branches to line 25.

3Ø IF X=Y THEN PRINT X: If the expression X=Y is true (its value is non-zero), the PRINT statement is executed. Otherwise, the PRINT statement is not executed. In either case, execution continues with the line after the IF...THEN statement.

35 IF X=Y+3 GOTO 39: Equivalent to "IF X=Y+3 THEN 39".

3. ON...GOTO provides for another type of conditional branch. Its form is as follows:

ON<integer expression>GOTO<list of line numbers>

After the value of the expression is truncated to an integer, say I, the statement causes BASIC to branch to the line whose number is Ith in the list. The statement may be followed by as many line numbers as will fit on one line. If I=0 or is greater than the number of lines in the list, execution will continue at the next line after the ON...GOTO statement. I must not be less than zero or greater than 255, or an FC or ILLEGAL FUNCTION CALL error will result.

b. Loops. It is often desirable to perform the same calculations on different data or repetitively on the same data. For this purpose, BASIC provides the FOR and NEXT statements. The form of the FOR statement is as follows:

FOR<variable>=<X>TO<Y>[STEP<Z>]

where X, Y and Z are expressions. When the FOR statement is encountered for the first time, the expressions are evaluated. The variable is set to the value of X which is called the initial value. BASIC then executes the statements which follow the FOR statement in the usual manner. When a NEXT statement is encountered, the step Z is added to the variable which is then tested against the final value Y. If Z, the step, is positive and the variable is less than or equal to the final value,

or if the step is negative and the variable is greater than or equal to the final value, then BASIC branches back to the statement immediately following the FOR statement. Otherwise, execution proceeds with the statement following the NEXT. If the step is not specified, it is assumed to be 1. Examples:

```
10 FOR I=2 TO 11              The loop is executed 10 times with the
                              variable I taking on each integral value
                              from 2 to 11.
20 FOR V=1 TO 9.3             This loop will execute 9 times until V is
                              greater than 9.3
30 FOR V=10*N TO 3.4/Z STEP SQR(R) The initial, final and step expressions
                                   need not be integral, but they will be
                                   evaluated only once, before looping begins.
40 FOR V=9 TO 1 STEP -1       This loop will be executed 9 times.
```

FOR...NEXT loops may be nested. This is, BASIC will execute a FOR...NEXT loop within the context of another loop. An example of two nested loops follows:

```
100 FOR I=1 TO 10
120 FOR J=1 TO I
130 PRINT A(I,J)
140 NEXT J
150 NEXT I
```

Line 130 will print 1 element of A if I=1, 2 if I=2 and so on. If loops are nested, they must have different loop variable names. The NEXT statement for the inside loop variable (J in the example) must appear before that for the outside variable (I). Any number of levels of nesting is allowed up to the limit of available memory.

The NEXT statement is of the form:

NEXT[<variable>],<variable>...}]

where each variable is the loop variable of a FOR loop for which the NEXT statement is the end point. NEXT without a variable will match the most recent FOR statement. In the case of nested loops which have the same end point, a single NEXT statement may be used for all of them. The first variable in the list must be that of the most recent loop, the second of the next most recent, and so on. If BASIC encounters a NEXT statement before its corresponding FOR statement has been executed, an NF (NEXT WITHOUT FOR) error message is issued and execution is terminated.

c. Subroutines. If the same operation or series of operations are to be performed in several places in a program, storage space requirements and programming time will be minimized by the use of subroutines. A subroutine is a series of statements which are executed in the normal fashion upon being branched to by a GOSUB statement. Execution of the subroutine is terminated by the RETURN statement which branches back to the statement after the calling GOSUB. The format of the GOSUB statement is as follows:

GOSUB<line number>

where the line number is that of the first line of the subroutine. A subroutine may be called from more than one place in a program, and a subroutine may contain a call to itself or to another subroutine. Such subroutine nesting is limited only by available memory. Subroutines may be branched to conditionally by use of the ON...GOSUB statement, whose form is as follows:

ON <integer expression> GOSUB<list of line numbers>.

The execution is the same as ON...GOTO except that the line numbers are those of the first lines of subroutines. Execution continues at the next statement after the ON...GOSUB upon return from one of the subroutines.

d. OUT OF MEMORY errors. While nesting in loops, subroutines and branching is not limited by BASIC, memory size limitations restrict the size and complexity of programs. The OM or OUT OF MEMORY error message is issued when a program requires more memory than is available. See Appendix C for an explanation of the amount of memory required to run programs.

2-3. Input/Output.

a. INPUT, INSTR$. The INPUT statement causes data input to be requested from the terminal. The format of the INPUT statement is as follows:

INPUT<list of variables>

The effect of the INPUT statement is to cause the values typed on the terminal to be assigned to the variables in the list. When an INPUT statement is executed, a question mark (?) is printed on the terminal signalling a request for information. The operator types the required numbers or strings separated by commas, and types a carriage return. If the data entered is invalid (strings were entered when numbers were requested, etc.) BASIC prints 'REDO FROM START?' and waits for the correct data to be entered. If more data was requested by the INPUT statement then was typed, ?? is printed on the terminal and execution awaits the needed data. If more data was typed than was requested, the warning 'EXTRA IGNORED' is printed and execution proceeds. After all the requested data is input, execution continues normally at the statement following the INPUT. An optional prompt string may be added to an INPUT statement.

INPUT["<prompt string>";]<variable list>

Execution of the statement causes the prompt string to be printed before the question mark. Then all operations proceed as above. The prompt string must be enclosed in double quotation marks (") and must be separated from the variable list by a semicolon (;). Example:

1ØØ INPUT "VALUES"; X,Y causes the following output:

VALUES?

The requested values of X and Y are typed after the ?. A carriage return in response to an INPUT statement will cause execution to continue with the values of the variables in the variable list unchanged.

The INSTR$ function allows you to read characters typed from the keyboard while leaving the screen unchanged. The format of the INSTR$ function is:

INSTR$(X)

The X argument gives the number of characters of input to accept. Nothing is displayed on the screen when the INSTR$ call is encountered. The program pauses until

X keys are pressed on the keyboard (the keys pressed are not displayed on the screen), and the function returns the string of characters entered.

b. JOYSTICK INPUT. The functions JOY, POT, and FIRE are used to read the position of the joystick, the value of the potentiometer knob, and the hit button respectively. See section 6-3, Intrinsic Functions for a full description of joystick input capabilities.

c. PRINT. The PRINT statement causes the computer to print data. The simplest PRINT statement is:

PRINT

which prints a carriage return. The effect is to skip a line. The more usual PRINT statement has the following form:

PRINT<list of expressions>

which causes the values of the expressions in the list to be printed. String lietrals may be printed if they are enclosed in double quotation marks (").

The position of printing is determined by the punctuation used to separate the entries in the list. BASIC divides the printing line into zones of 14 spaces each. A comma causes printing of the value of the next expression to begin at the beginning of the next 14 column zone. A semicolon (;) causes the next printing to begin immediately after the last value printed. If a comma or semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line according to the conditions above. Otherwise, a carriage return is printed. The TAB and SPC functions may also be used to set spacing.

d. OUTPUT. The OUTPUT statement may be used to display data or text at any screen position in any color. The form of the OUTPUT statement is

OUTPUT<expression>,<x>,<y>,<color>

The value of<expression> is displayed on the screen at location <x>, <y>. The <color> parameter may be 0, 1, 2 or 3 and specifies the COLOR array position which identifies the color to use for display. See the COLOR statement in section 5-2. The parameter <x> may range between 0 and 112, the parameter <y> between 0 and 77. Format for display of numeric values of <expression> is described in section 2-1.

e. PLOT. The PLOT statement may be used to display a dot of color at a given screen location. The form of the PLOT statement is:

PLOT <x>, <y>,<color>

The <x> , <y> parameters identify the screen coordinates at which to plot <color>, $0 \leq x \leq 112$ and $0 \leq y \leq 77$. The <color> parameter may be 0, 1, 2 or 3 and specifies the COLOR array position which identifies the color to be plotted. See the COLOR statement in section 5-2.

f. WINDOW. The WINDOW statement is used to restrict screen scrolling to part of the screen. The form of the WINDOW statement is:

WINDOW <y>

The parameter <y> gives the number of pixels to leave available at the bottom of the screen for normal scrolling. The remainder of the screen is made available for undisturbed display using the PLOT and OUTPUT statements. To allow scrolling space for N lines of text use the formula

<y>=(N+1)*6

g. DATA, READ, RESTORE

1. The DATA statement. Numerical or string data needed in a program may be written into the program statements themselves, input from cassette tape or read from DATA statements. The format of the DATA statement is as follows:

DATA <list>

where the entries in the list are numerical or string constants separated by commas.

The effect of the statement is to store the list of values in memory in coded form for access by the READ statement. Examples:

```
10 DATA 1,2,-1E3,.04
20 DATA "WHITE","SOX"
```

2. The READ statement. The data stored by DATA statements are accessed by READ statements which have the following form:

READ <list of variables>

where the entries in the list are variable names of the appropriate data type separated by commas. The effect of the READ statement is to assign the values in the DATA lists to the corresponding variables in the READ statement list. This is done one by one from left to right until the READ list is exhausted. If there are more names in the READ list than values in the DATA lists an OD (OUT OF DATA) error message is issued. If there are more values stored in DATA statements than are read by a READ statement the next READ statement that is executed begins with the next unread DATA list entry. A single READ statement may access more than one DATA statement and more than one READ statement may access the data in a single DATA statement.

An SN (SYNTAX ERROR) message can result from an improperly formatted DATA list. The line number in the error message will refer to the actual line of the DATA statement in which the error occurred.

3. RESTORE statement. After the restore statement is executed, the next piece of data accessed by a READ statement will be the first entry of the first DATA list in the program. This allows re-READing the data.

h. CSAVEing and CLOADing Arrays. Numeric arrays may be saved on cassette or loaded from cassette using CSAVE* and CLOAD*. The formats of the statements are:

CSAVE* <array name>

and

CLOAD*<array name>

When an array is written out or read in, the elements of the array are written out with the leftmost subscript varying most quickly, the next leftmost second, etc:

```
DIM A(1Ø)
CSAVE*A
```

writes out A(Ø),A(1),...A(1Ø)

```
DIM A(1Ø,1Ø)
CSAVE*A
```

writes out A(Ø,Ø),A(1,Ø)...A(1Ø,Ø),A(1Ø,1)...A(1Ø,1Ø)

Using this fact, it is possible to write out an array as a two dimensional array and read it back in as a single dimensional array, etc.

3. FUNCTIONS

BASIC allows functions to be referenced in mathematical function notation. The format of a function call is as follows:

<name>(<argument>)

where the name is that of a previously defined function and the argument is an expression. Only one argument is allowed. Function calls may be components of expressions, so statements like

```
1Ø LET T=(F*SIN(T))/P and
2Ø C=SQR(A 2+B 2+2*A*B*COS(T))
```

are legal.

3.1 Intrinsic Functions

BASIC provides several frequently used functions which may be called from any program without further definition.

For a list of intrinsic functions, see section 5-3.

3.2 User-Defined Functions

The DEF statement. The programmer may define functions which are not included in the list of intrinsic functions by means of the DEF statement. The form of the DEF statement is as follows:

DEF<function name>(<variable>)=<expression>

where the function name must be FN followed by a legal variable name and the variable is a 'dummy' variable name. The dummy variable represents the argument variable (the value in the function call). Only one argument is allowed for a user-defined function. Any expression may appear on the right side of the equation, but it must be limited to one line.

User-defined string functions are not allowed. Examples:

```
12 DEF FNRAD(DEG)=3.14159/18Ø*DEG
25 DEF FNFT(A)=A/12
```

When called with the measure of an angle in degrees, returns the radian equivalent.

When called with a number of inches, returns equivalent number of feet.

A function may be redefined by executing another DEF statement with the same name. A DEF statement must be executed before the function it defines may be called. For a list of formulae for mathematical functions, see Appendix C.

3-3 Errors

An FC error (ILLEGAL FUNCTION CALL) results when an improper call is made to a function. Some places this might occur are the following:

1. a negative array subscript. LET A(-1)=Ø, for example.
2. an array subscript that is too large (>32767)
3. negative or zero argument for LOG
4. negative argument for SQR
5. A^B with A negative and B not an integer
6. improper arguments to MID$, LEFT$, RIGHT$, TAB, SPC, COLOR, JOY, POT, FIRE, PLOT, OUTPUT, INSTR$ or ON...GOTO.

b. An attempt to call a user-defined function which has not previously appeared in a DEF statement will cause a UF error (UNDEFINED USER FUNCTION).

c. A TM or TYPE MISMATCH error will occur if a function which expects a string argument is given a numeric value or vice-versa.

4. STRINGS

Expressions may either have numeric values, or they may be strings of characters. BASIC provides a complete complement of statements and functions for manipulating string data. Many of the statements have already been discussed so only their particular application to strings will be treated in this section.

4-1 String Data

A string is a list of alphanumeric characters which may be from 0 to 255 characters in length. Strings may be stated explicitly as constants or referred to symbolically by variables. String constants are delimited by quotation marks at the beginning and end. A string variable name ends with a dollar sign ($).

Examples:

A$="ABCD" Sets the variable A$ to the four character string "ABCD"

B9$="14A/56" Sets the variable B9$ to the six character string "14A/56"

AA$="E$" Sets the variable AA$ to the two character string "E$"

Strings input to an INPUT statement need not be surrounded by quotation marks.

String arrays may be dimensioned exactly as any other kind of array by use of the DIM statement. Each element of a string array is a string which may be up to 255 characters long. The total number of string characters in use at any point in the execution of a program must not exceed the total allocation of string space or an OS or OUT OF STRING SPACE error will result. String space is allocated by the CLEAR command which is explained in section 6-2. The FRE function with a string argument returns the number of bytes of free string space.

4-2 String operations

a. Comparison Operators. The comparison operators for strings are the same as those for numbers:

=	equal
<>	not equal
<	less than
>	greater than
=<,<=	less than or equal to
=>,>=	greater than or equal to

Comparison is made character by character on the basis of ASCII codes until a difference is found. If, while comparison is proceeding, the end of one string is reached, the shorter string is considered to be smaller. ASCII codes may be found in Appendix B. Examples:

```
"A"<"Z"    ASCII A is Ø65, Z is Ø9Ø
"1"<"A"    ASCII 1 is Ø49
" A"<"A"   Leading and trailing blanks are significant in string literals.
           ASCII blank is 32.
```

b. String Expressions. String expressions are composed of string literals, string variables and string function calls connected by the + or concatenation operator. The effect of the concatenation operator is to add the string on the right side of the operator to the end of the string on the left. If the result of concatenation is a string more than 255 characters long, an LS (STRING TOO LONG) error message will be issued and execution will be terminated. For example, "THE" + " END" = "THE END". Remember that spaces inside the quotes are significant.

c. Input/Output. The same statements used for input and output of normal numeric data may be used for string data as well.

1. INPUT, PRINT. The INPUT and PRINT statements read and write strings on the terminal. Strings input from the keyboard need not be enclosed in quotation marks, but if they are not, leading blanks will be ignored and the string will be terminated when the first comma or colon is encountered. Examples:

```
1Ø INPUT Z0$,F0$          Reads two strings separated by commas or
                          colons, and assigns them to Z0$ and F0$
                          respectively.
2Ø INPUT X$               Reads one string and assigns it to the
                          variable X$.
3Ø PRINT X$,"HI, THERE"   Prints two strings, including all spaces
                          and punctuation in the second.
```

2. DATA, READ. DATA and READ statements for string data are the same as for numeric data. Strings in DATA statements should be enclosed in quotes, and separated by commas.

4-3. String Functions.

The format for intrinsic string function calls is the same as that for numeric functions. For the list of string functions, see section 5-3.

String function names must end with a dollar sign.

5. LISTS AND DIRECTORIES

5-1 Commands. Commands direct BASIC to arrange memory and input/output facilities, to list and edit programs and to handle other housekeeping details in support of program execution. BASIC accepts commands after it prints 'OK' and is at command level.

CLEAR

Sets all program variables to zero.

CLEAR <integer expression>

Same as CLEAR but sets string space (see 4-1) to the value of the integer expression. If no argument is given, string space will remain unchanged. When BASIC is loaded, string space is set to 50 bytes. CLEAR may be used as a program statement.

CLOAD[<string expression>]

Causes the current program to be deleted, all variables to be cleared, and the program on cassette tape designated by the first five characters of <string expression> to be loaded into memory. If no string is given, the first available program is loaded. The cassette READ button must be pressed before the CLOAD can begin.

CLOAD*<array name>

Loads data from cassette tape into the specified array. CLOAD* may be used as a program statement. The cassette READ button must be pressed before the CLOAD* can begin.

CONT

Continues program execution after a Control/C has been typed or a STOP or END statement has been executed. Execution resumes at the statement after the break occurred unless input from the terminal was interrupted. In that case, execution resumes with the reprinting of the prompt (? or prompt string). CONT is useful in debugging, especially where an 'infinite loop' is suspected. An infinite loop is a

series of statements from which there is no escape. Typing Control/C causes a break in execution and puts BASIC in command level. Direct mode statements can then be used to print intermediate values, change the values of variables, etc. Execution can be restarted by typing the CONT command, or by executing a direct mode GOTO statement, which causes execution to resume at the specified line number.

In BASIC, execution cannot be continued if a direct mode error has occurred during the break. Execution cannot continue if the program was modified during the break.

CSAV [<string expression>]

Causes the program currently in memory to be saved on cassette tape under the name specified by the first five characters of <string expression> . If no string is given, the program is stored with no name. Both the READ and WRITE cassette buttons must be pressed before the CSAVE is issued.

CSAVE*<array name>

Causes the named numeric array to be saved on cassette tape. CSAVE* may be used as a program statement. Both the READ and WRITE cassette buttons must be pressed before the CSAVE* is issued.

LIST [<line number>]

Lists the program currently in memory starting with the given line number. If no line number is given, listing begins with the lowest numbered line. Listing is terminated either by the end of the program or by typing Control/C.

If you wish to interrupt a listing temporarily, type Control/S. The output to the screen will stop until you type Control/Q, when listing will resume from where it was stopped by the Control/S. This facility is particularly useful when reviewing listings which are too long to fit on the screen.

NEW

Deletes the current program and clears all variables. Used before entering a new program from the terminal.

REWIND

Turns on cassette motor to permit repositioning of the tape. To rewind the tape, depress the REW button; to advance the tape rapidly, depress the FAST FORWARD button; to play a tape through the TV speaker, depress the READ button. Do not put the cassette in record mode (READ and WRITE buttons both pressed). Pressing any key on the keyboard signals completion of tape repositioning, and turns off the tape motor. REWIND may also be used as a program statement.

RUN[<line number>]

Starts execution of the program currently in memory at the line specified. If the line number is omitted, execution begins at the first line.

5-2 Statements. The following table of statements is listed in alphabetical order. In the table, X and Y stand for any expressions. I and J stand for expressions whose values are truncated to integers. V and W are any variable names. The format for a BASIC line is as follows:

<nnnnn> <statement>[:<statement>...]

where nnnnn is the line number. Unless otherwise noted, statements may also be issued as direct mode commands.

Name	Format
CLS	CLS

Clears TV screen

COLOR	COLOR<color list>

Selects palette of four colors for plotting and other screen display. Subsequent reference to colors is by position number (0, 1, 2, 3) in <color list>. The first code, color 0, automatically become the background color. The LIST command prints line numbers in color 1 and statements in color 3. Displays from PRINT and INPUT appear in color 3. Available color values are 0=black, 1=red, 2=green, 3=yellow, 4=blue, 5=magenta, 6=cyan, 7=white. For example, COLOR 0, 7, 3, 1

produces a black screen with white line numbers and red statements shown by a LIST. Output from PRINT and INPUT appears in red. The default is COLOR 4, 3, 0, 7. (blue, yellow, black, white).

DATA DATA<list>

Specifies data to be read by a READ statement. List elements can be numerical, integer, or string constants. List elements are separated by commas.

DEF DEF FNV(<W>)=<X>

Defines a user-defined function. Function name is FN followed by a legal variable name. Definitions are restricted to one line (72 characters). User-defined string functions are not allowed. DEF may not be used in direct mode.

DIM DIM <V>(<I>[,J...])[,...]

Allocates space for array variables. More than one variable may be dimensioned by one DIM statement up to the limit of the line. The value of each expression gives the maximum subscript possible for that array dimension. The smallest subscript is Ø. Without a DIM statement, an array is assumed to have maximum subscript of 1Ø for each dimension referenced. For example, A(I,J) is assumed to have 121 elements, from A(Ø,Ø) to A(1Ø,1Ø) unless otherwise dimensioned in a DIM statement.

END END

Terminates execution of a program. END may not be used in direct mode.

FOR FOR<V>=<X>TO<Y>[STEP<Z>]

Allows repeated execution of the same statements. First execution sets V=X. Execution proceeds normally until NEXT is encountered. Z is added to V, then IF Z<Ø and V>=Y, or if Z>Ø and V<=Y, BASIC branches back to the statement after FOR. Otherwise, execution continues with the statement after NEXT.

GOTO GOTO<nnnnn>

Unconditional branch to line number nnnnn.

GOSUB GOSUB<nnnnn>

Unconditional branch to subroutine beginning at line nnnnn.

IF...GOTO IF <X> GOTO<nnnnn>

Same as IF...THEN except GOTO can only be followed by a line number and not another statement.

IF...THEN IF<X>THEN<nnnnn>
or IF<X>THEN<statement>[:statement...]

If value of X<>0, branches to line nnnn or executes statements following THEN. If X=0, proceeds to next line in program.

INPUT INPUT [<string expression;>]<V>[,<W>...]

Causes BASIC to request input from terminal. Values typed on the terminal are assigned to the variables in the list. If no string expression is given, BASIC will print a question mark. If a string expression is given, BASIC will print its string value, then a question mark. Input values should be separated by commas, and terminated with a carriage return. INPUT may not be used in direct mode.

LET LET <V>=<X>

Assigns the value of the expression to the variable. The word LET is optional.

NEXT NEXT [<V>,<W>...]

Last statement of a FOR loop. V is the variable of the most recent loop, W of the next most recent and so on. NEXT without a variable name terminates the most recent FOR loop.

ON...GOTO ON<I>GOTO<list of line numbers>

Branches to line whose number is Ith in the list. List elements are separated by commas. If I=Ø or > number of elements in the list, execution continues at next statement. If I<Ø or >255, an error results.

ON...GOSUB ON<I> GOSUB <list of line numbers>

Same as ON...GOTO except list elements are initial line numbers of subroutines.

OUTPUT OUTPUT<expression>,<x>,<y>,<color>

Displays the value of <expression> at the given <x>, <y> screen coordinates, using the color whose COLOR array position is given by <color>. 0≤<x>≤112, 0 ≤<y>≤77, 0≤<color>≤3. Numeric values for <expression> are printed as described in section 2-1.

PLOT PLOT<x>,<y>,<color>

Plots a dot on the screen at the given <x>, <y> position using the color whose COLOR array position is given by <color>. The screen origin (0,0) is the lower left corner. 0 ≤<x>≤112, 0≤<y>≤77, 0≤<color>≤3.

PRINT PRINT<X>[,<Y>...]

Causes values of expressions in the list to be printed on the terminal. Spacing is determined by punctuation.

PUNCTUATION	Spacing - next printing begins:
,	at beginning of next 14 column zone
;	immediately
other or none	at beginning of next line

String literals may be printed if enclosed by (") marks. String expressions may be printed.

READ READ<V>[,<W>...]

Assigns values in DATA statements to variables. Values are assigned in sequence starting with the first value in the first DATA statement and the first variable in the READ list. Later READs continue from where the last left off, unless the RESTORE statement has been executed.

REM REM[<remark>]

Allows insertion of remarks. Not executed, but may be branched to.

RESTORE RESTORE

Allows data from DATA statements to be reread. Next READ statement after RESTORE begins with first data item of first DATA statement.

RETURN RETURN

Terminates a subroutine. Branches to the statement after the calling GOSUB. RETURN may not be used in direct mode.

REWIND REWIND

Turns on cassette motor to permit repositioning of the tape. To rewind the tape, depress the REW button; to advance the tape rapidly, depress the FAST FORWARD button; to play a tape through the TV speaker, depress the READ button. Do not put the cassette in record mode (READ and WRITE buttons both pressed). Pressing any key on the keyboard signals completion of tape repositioning, and turns off the tape motor.

SOUND SOUND<exp 1>,<exp 2>

Generates various sounds depending on values given for <exp 1>, <exp 2>. SOUND 7,4096 turns off any sound. SOUND 0,24844 produces a siren; SOUND 3,32 a low buzz; SOUND 3,16 a medium buzz; SOUND 3,0 a high buzz. $0 \leq$ <exp 1> ≤ 7, $1 \leq$ <exp 2> ≤ 32767.

STOP STOP

Stops program execution. BASIC enters command level and prints BREAK IN LINE nnnnn. STOP may not be used in direct mode.

TONE TONE <exp 1>,<exp 2>

Generates musical tones with frequency based on 1/<exp 1>, for length of time proportional to <exp 1>*<exp 2>. Avoid the use of negative or large positive values for <exp 1>*<exp 2>, as these will produce extremely long-lasting tones.

WINDOW WINDOW<exp>

Restricts normal screen scrolling to a portion of the screen. The number of pixels of vertical scrolling space is given by <exp>. To compute the number of pixels required to provide N print lines of scrolling space, use the formula (N+1)*6.

5-3 Intrinsic Functions

BASIC provides several commonly used numeric and string functions that may be called from your program without further definition. Individual functions are described in alphabetic order below. All such functions take the basic form:

name(X[,Y...])

where name is the name of the function, and X and Y...are function arguments (variables, expressions, or constants of the appropriate data type). Numeric functions may be used in BASIC commands or statements wherever a numeric expression is permitted, string functions wherever string expressions may be used.

Examples

BASIC Statement Type	Function Example
assignment	LET A = EXP(X)
loop control	FOR I=1 TO ABS(X)
conditional branch	IF COS(X) THEN 250
concatenation	A$="TOTAL=" + STR$(TL)

The following table lists in alphabetic order the name of each intrinsic function, its call format, and a description of its purpose, inputs, and outputs.

Function

ABS ABS(X)

Returns absolute value of expression X. ABS(X)=X if X>=0,-X if X<0.

ASC ASC(X$)

Returns the ASCII code of the first character of the string X$. ASCII codes are in appendix A.

ATN ATN(X)

Returns arctangent(X). Result is in radians in range -pi/2 to pi/2.

CHR$ CHR$(I)

Returns a string whose one element has the ASCII character identified by code I. ASCII codes are in Appendix A.

COS COS(X)

Returns cos(X). X is in radians.

EXP EXP(X)

Returns e to the power X. X must be <=87.3365.

FIRE FIRE(X)

FIRE(0) reads the hit button on the left joystick control, FIRE (1) reads the right hit button. FIRE returns a 0 if the button is pressed. 1 if not pressed.

FRE FRE(Ø)

Returns number of bytes in memory not being used by BASIC or the current program. If argument is a string, returns number of free bytes in string space.

INSTR$ INSTR$(X)

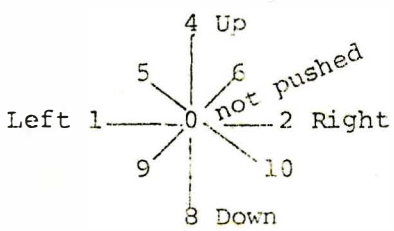
Waits for keyboard input of X characters. No prompt is given, no input is displayed on the screen, and no carriage return is required to terminate input. Returns input as a string of length X and leaves the screen display unaffected. $0 \leq X \leq 255$.

INT INT(X)

Returns the largest integer <=X

JOY JOY(X)

JOY(0) reads the position of the left joystick. JOY(1) reads the right joystick. JOY returns values as indicated below.



LEFT$ LEFT$ (X$,I)

Returns leftmost I characters of string X$.

LEN LEN(X$)

Returns length of string X$. Non-printing characters and blanks are counted.

LOG LOG(X)

Returns natural (base e) log of X. X>Ø

MID$ MID$(X$,I[,J])

Without J, returns rightmost characters from X$ beginning with the Ith character. If I>LEN(X$), MID$ returns the null string. Ø<I<255. With 3 arguments, returns a string of length J of characters from X$ beginning with the Ith character. If J is greater than the number of characters in X$ to the right of I, MID$ returns the rest of the string. Ø<=J<=255.

RND RND(X)

Returns a random number between Ø and 1 using the uniform probability distribution. X>Ø gives the next number in the sequence of random numbers. X=Ø repeats the last number returned. X<0 initiates a new random number sequence. Sequences started with the same negative number will be the same.

POS POS(I)

Returns the screen column position at which next character would print. Leftmost position =Ø.

POT POT(X)

POT(0) reads the knob on the left joystick control; POT(1) reads the right knob. Depending on the position of the knob, POT returns a value between 3 and 200.

RIGHT$ RIGHT$(X$,I)

Returns rightmost I characters of string X$. If I>=LEN(X$), returns X$.

SGN SGN(X)

If X>Ø, returns 1, if X=Ø returns Ø, if X<Ø, returns -1. For example, ON SGN(X)+2 GOTO 1ØØ,2ØØ,3ØØ branches to 1ØØ if X is negative, 200 if X is Ø and 3ØØ if X is positive.

SIN SIN(X)

Returns the sine of the value of X in radians. COS(X)=SIN(X+3.14159/2).

SPC SPC(I)

Generates string of I blanks. Ø<=I<=255.

SQR SQR(X)

Returns square root of X. X must be >=Ø.

STR$ STR$(X)

Returns string representation of value of X.

TAB TAB(I)

Spaces to position I on the screen. Space Ø is the leftmost space, 15 the rightmost. If the carriage is already beyond space I, TAB has no effect. May only be used in PRINT statement.

TAN TAN(X)

Returns tangent(X). X is in radians.

VAL VAL(X$)

Returns numerical value of string X$. If first non-blank character of X$ is not +,-,. or a digit, VAL(X$)=Ø.

5-4 Special Characters

BASIC recognizes several characters in the ASCII font as having special functions in carriage control, editing and program interruption. Characters such as Control/C, Control/S, etc. are typed by holding down the Control key and typing the designated letter.

Typed as	Printed as
:	:

Separates statements in a line.

?	?

equivalent to PRINT statement.

Carriage return

Enters commands, statements, or data into memory and returns print position to beginning of a new line.

(backspace)

Erases last character typed.

Control/C

Interrupts execution of current program or LIST command. Takes effect after execution of the current statement or after listing the current line. BASIC goes to command level and types OK. CONT command resumes execution. See section 6-1.

Control/O

Suppresses all output until an INPUT statement is encountered, another Control/C is typed, an error occurs or BASIC returns to command level.

Control/Q

Causes execution to resume after Control/S. Control/S and Control/Q have no effect if no program or command is being executed.

Control/S

Causes program or command execution to pause until Control/Q or Control/C is typed.

Control/U

Signals that line being input should be ignored; generates carriage return.

5-5 Error Messages

After an error occurs, BASIC returns to command level and types OK. Variable values and the program text remain intact, but the program cannot be continued by the CONT command. All GOSUB and FOR context is lost. The program may be continued by direct mode GOTO, however. When an error occurs in a direct statement, no line number is printed. Format of error messages:

Direct Statement	?XX ERROR
Indirect Statement	?XX ERROR IN YYYYY

where XX is the error code and YYYYY is the line number where the error occurred. The following are the possible error codes and their meanings:

ERROR CODE	EXTENDED ERROR MESSAGE
BS	SUBSCRIPT OUT OF RANGE

An attempt was made to reference an array element which is outside the dimensions of the array, or the wrong number of dimensions are used in an array reference.

CN	CAN'T CONTINUE

Attempt to continue a program when none exists, a direct mode error occurred, or after a modification was made to the program.

DD	REDIMENSIONED ARRAY

After an array was dimensioned, another dimension statement for the same array was encountered. This error often occurs if an array has been given the default dimension of 1Ø and later in the program a DIM statement is found for the same array.

FC	ILLEGAL FUNCTION CALL

The parameter passed to a numeric or string function was out of range. FC errors can occur due to:

1. a negative array subscript (LET A(-1)=Ø)
2. an unreasonably large array subscript (32767)

3. LOG with negative or zero argument

4. SQR with negative argument

5. A^B with A negative and B not an integer

6. Calls to MID$, LEFT$, RIGHT$, TAB, SPC, FIRE, POT, JOY, PLOT, OUTPUT, INSTR$ or ON...GOTO with an improper argument.

ID ILLEGAL DIRECT

Certain statements are legal only in indirect mode. See individual statement entries in section 5-2.

LS STRING TOO LONG

An attempt was made to create a string more than 255 characters long.

MO MISSING OPERAND

An attempt was made to execute an incomplete command, statement, or function calculation.

NF NEXT WITHOUT FOR

The variable in a NEXT statement corresponds to no previously executed FOR statement.

OD OUT OF DATA

A READ statement was executed but all of the DATA statements in the program have already been read. The program tried to read too much data or insufficient data was included in the program.

OM OUT OF MEMORY

Program is too large, has too many variables, too many FOR loops, too many GOSUBs or too complicated expressions. See Appendix C.

OS OUT OF STRING SPACE

String variables exceed amount of string space allocated for them. Use the CLEAR command to allocate more string space or use smaller strings or fewer string variables.

OV OVERFLOW

The result of a calculation was too large to be represented in BASIC's number format. If an underflow occurs, zero is given as the result and execution continues without any error message being printed.

RG RETURN WITHOUT GOSUB

A RETURN statement was encountered before a previous GOSUB statement was executed.

SN SYNTAX ERROR

Missing parenthesis in an expression, illegal character in a line, incorrect punctuation, etc.

ST STRING FORMULA TOO COMPLEX

A string expression was too long or too complex. Break it into two or more shorter ones.

TM TYPE MISMATCH

The left hand side of an assignment statement was a numeric variable and the right hand side was a string, or vice-versa; or a function which expected a string argument was given a numeric one or vice-versa.

UF UNDEFINED USER FUNCTION

Reference was made to a user defined function which had never been defined.

UL UNDEFINED LINE

The line reference in a GOTO, GOSUB, or IF...THEN was to a line which does not exist.

/Ø DIVISION BY ZERO

Can occur with integer division as well as floating point division. Ø to a negative power also causes a DIVISION BY ZERO error.

5-6 Reserved Words

Some words are reserved by the BASIC interpreter for use as statements, commands, operators, etc., and thus may not be used for variable or function names. In addition to the words listed below, intrinsic function names are reserved words.

RESERVED WORDS

AND	NEXT
CLEAR	NOT
CSAVE	ON
CLOAD	OR
COLOR	OUTPUT
CONT	PRINT
DEF	READ
DIM	REM
END	RESTORE
FN	RETURN
FOR	RFWIND
GOSUB	RUN
GOTO	SOUND
IF	STEP
INPUT	STOP
LET	THEN
LIST	TO
NEW	TONE
	WINDOW

APPENDIX A
ASCII CHARACTER CODES

DECIMAL	CHAR.	DECIMAL	CHAR.	DECIMAL	CHAR.
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093	]
008	BS	051	3	094	^
009	HT	052	4	095	<
010	LF	053	5	096	'
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	
040	(	083	S	126	
041	)	084	T	127	DEL
042	*	085	U		

LF=Line Feed FF=Form Feed CR=Carriage Return DEL=Rubout

APPENDIX B
SPACE AND SPEED HINTS

A. Space Allocation

The memory space required for a program depends, of course, on the number and kind of elements in the program. The following table contains information on the space required for the various program elements.

Element	Space Required
Variables	
integer	5 bytes
floating point	6 bytes
string	6 bytes
Arrays. Let N = # of elements, D = # of dimensions.	
integer	2 * (N + D) + 6 bytes
floating point	(4 * N) + (2 * D) + 6 bytes
string	(3 * N) + (2 * D) + 6 bytes
Functions	
intrinsic	1 byte for the call
user-defined	6 bytes for the definition
Reserved Words	1 byte each
Other Characters	1 byte each
Stack Space	
active FOR loop	16 bytes
active GOSUB	5 bytes
parentheses	6 bytes each set
temporary result	10 bytes

B. Space Hints

The space required to run a program may be significantly reduced without affecting execution by following a few of the following hints:

1. Use multiple statements per line. Each line has a 5 byte overhead for the line number, etc., so the fewer lines there are, the less storage is required.

2. Delete unnecessary spaces. Instead of writing

```
10 PRINT X, Y, Z
```

use

```
10 PRINTX,Y,Z
```

3. Delete REM statements to save 1 byte for REM and 1 byte for each character of the remark.

4. Use variables instead of constants, especially inside for loops, and when the same value is used several times. For example, using the constant 3.14159 ten times in a program uses 40 bytes more space than assigning

```
10 P=3.14159
```

once and using P ten times.

5. Using END as the last statement of a program is not necessary and takes five extra bytes.

6. Reuse unneeded variables instead of defining new variables.

7. Use subroutines instead of writing the same code several times.

8. Use the zero elements of arrays. Remember the array dimensioned by

```
100 DIM A(10)
```

has eleven elements, A(0) through A(10).

C. Speed Hints

1. Deleting spaces and REM statements gives a small but significant decrease in execution time.

2. Variables are set up in a table in the order of their first appearance in the program. Later in the program, BASIC searches the table for the variable at each reference. Variables at the head of the table take less time to search for than those at the end. Therefore, reuse variable names and keep the list of variables as short as possible.

3. Use NEXT without the index variable.

4. String variables set up a descriptor which contains the length of the string and a pointer to the first memory location of the string. As strings are manipulated, string space fills up with intermediate results and extraneous material as well as the desired string information.

When this happens, BASIC's "garbage collection" routine clears out the unwanted material. The frequency of garbage collection is inversely proportional to the amount of string space. The more string space there is, the longer it takes to fill with garbage. The time garbage collection takes is proportional to the square of the number of string variables. Therefore, to minimize garbage collection time, make string space as large as possible and use as few string variables as possible.

APPENDIX C
MATHEMATICAL FUNCTIONS

Derived Functions.

The following functions, while not intrinsic to BASIC, can be calculated using the existing BASIC functions:

Function:	BASIC equivalent:
SECANT	SEC(X) = 1/COS(X)
COSECANT	CSC(X) = 1/SIN(X)
COTANGENT	COT(X) = 1/TAN(X)
INVERSE SINE	ARCSIN(X) = ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X) = -ATN X(X/SQR(-X*X+1)) +1.5708
INVERSE SECANT	ARCSEC(X) = ATN(XSQR(X*X-1)) +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X) = ATN(1/SQR(X*X-1)) +(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X) = ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X) = (EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X) = (EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X) = EXP(-X)/EXP(X)+EXP(-X)) *2+1
HYPERBOLIC SECANT	SECH(X) = 2/(EXP(X)+EXP(=X))
HYPERBOLIC COSECANT	CSCH(X) = 2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X) = EXP(-X)/(EXP(X)-EXP(-X)) *2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X) = LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X) = LOG(X+SQR(X*X+-1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X) = LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X) = LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X) = LOG((SGN(X)* SQR(X*X+1)+1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X) = LOG((X+1)/(X-1))/2
A MOD B	A-B*INT(A/B)

APPENDIX D
USING THE CASSETTE TAPE UNIT

Programs may be saved on cassette tape by means of the CSAVE command. CSAVE is used in direct or indirect mode, and its format is as follows;

CSAVE[<string expression>]

The program currently in memory is saved on cassette under the name specified by the first five characters of the string expression. For example, the program named A is saved by CSAVE "A". If no string is given, the program is stored with no name.

After CSAVE is completed, BASIC always returns to command level. Programs are written on tape in BASIC's internal representation. Variable values are not saved on tape, although an indirect mode CSAVE does not affect the variable values of the program currently in memory.

Before using CSAVE, make sure the tape is positioned properly (see REWIND below, for rewinding instructions). In DIRECT mode, depress the READ and WRITE buttons simultaneously, then issue the CSAVE. In indirect mode, the buttons must be pushed prior to executing the calling statement, or the program aborts when it attempts to execute the CSAVE.

Programs may be loaded from cassette tape by means of the CLOAD command, which has the same format as CSAVE. CLOAD is used in direct mode only. The effect of CLOAD is to execute a NEW command, clearing memory and all variable values and loading the specified program into memory. When done reading and loading, BASIC returns to command level.

Before using CLOAD, make sure the tape is positioned properly (see REWIND below for rewinding instructions). In Direct mode, depress the READ button; then issue the CLOAD or vice versa. BASIC does not return to command level after a CLOAD if it could not find the requested program. In that case, the computer will continue to search until it is reset.

BASIC data may be read and written with CSAVE* and CLOAD* commands. The formats are as follows:

CSAVE*<array variable name>

and CLOAD*<array variable name>

See section 2-3d for a discussion of CSAVE* and CLOAD* for array data.

To reposition a cassette tape, you must issue the REWIND command to turn on the tape motor, and depress the REWIND or FFWD button on the built-in cassette unit. In direct mode, you may push the button, then issue the command, or vice versa. In indirect mode, push the button prior to executing the calling statement. Following a direct or indirect REWIND, depress any key on the keyboard to signal that repositioning is complete.

APPENDIX E
CONVERTING BASIC PROGRAMS
NOT WRITTEN FOR THE INTERACT COMPUTER

Though implementations of BASIC on different computers are in many ways similar, there may be some incompatibilities between your BASIC and the BASIC used on other computers.

1. Strings.

A number of BASICs require the length of strings to be declared before they are used. All dimension statements of this type should be removed from the program. In some of these BASICs, a declaration of the form DIM A$(I,J) declares a string array of J elements each of which has a length I. Convert DIM statements of this type to equivalent ones in your BASIC (e.g. DIM A$(J)). BASIC uses "+" for string concatenation, not "," or "&". BASIC uses LEFT$, RIGHT$ and MID$ to take substrings of strings. Some other BASICs use A$(I) to access the Ith character of the string A$, and A$(I,J) to take a substring of A$ from character position I to character position J. Convert as follows:

OLD	NEW
A$(I)	MID$(A$,I,1)
A$ (I,J)	MID$ (A$,I,J-I+I)

This assumes that the reference to a subscript of A$ is in an expression on the right side of an assignment. If the reference to A$ is on the left hand side of an assignment, and X$ is the string expression used to replace characters in A$, convert as follows:

OLD	NEW
A$(I) = X$	A$=LEFT$(A$,I-1) +X$+MID$(A$,I,1)
A$(I,J) =X$	A$=LEFT$(A$,I-1) +X$+MID$(A$,J+1)

2. Multiple assignments.

Some BASICs allow statements of the form:

```
5ØØ LET B=C=Ø
```

This statement would set the variables B and C to zero. In your BASIC, this has an entirely different effect. All the "=" signs to the right of the first one would be interpreted as logical comparison operators. The easiest way to convert statements like this one is to rewrite them as follows.

```
5ØØ B=Ø:C=Ø
```

3. Some BASICs use "/" instead of ":" to delimit multiple statements on a line. Change each "/" to ":" in the program.

4. Programs which use the MAT functions available in some BASICs will have to be rewritten using FOR...NEXT loops to perform the appropriate operations.

APPENDIX F

SAMPLE PROGRAMS

A. Depreciation Program

The program below calculates depreciation using your choice of the straight-line, double-declining balance or sum-of-years-digits method.

```
10 CLS:OUTPUT "DEPRECIATION",15,40,2
20 OUTPUT "ROUTINES",30,30,2
30 OUTPUT "READY?",3,15,2:A$=INSTR$(1)
40 CLS:OUTPUT "1 STRAIGHT-LINE",4,50,2
50 OUTPUT "2 DOUBLE DECLIN.",4,40,2
60 OUTPUT "3 SYD METHOD",4,30,2
70 WINDOW 18:INPUT "METHOD";I
80 IF I<=3 AND I>=1 GOTO 100
90 PRINT"1,2 OR 3 PLEASE":GOTO 70
100 CLS:WINDOW 77
110 PRINT "STARTING":INPUT "  VALUE";V
120 PRINT "USEFUL":INPUT "  LIFE";N
130 CLS:ON I GOTO 140,150,160
140 PRINT "STRAIGHT-LINE":PRINT:GOTO 170
150 PRINT "DOUBLE-DECLINING":PRINT: GOTO 170
160 PRINT "SYD METHOD":PRINT
170 PRINT "YR  DEPR.  VALUE"
180 J=0
190 ON I GOTO 200,300,400
200 REM S-L CALCULATIONS
210 D=V/N
220 V=V-D:J=J+1
230 PRINT J;SPC(1);D;SPC(2);V
240 IF J<N GOTO 220:END
300 REM D-D CALCULATIONS
310 D=(2/N)*V
320 V=V-D:J=J+1
330 PRINT J;SPC(1);D2;SPC(3);V
340 IF J<N GOTO 320:END
400 REM SYD CALCULATIONS
410 F1=V/(N*(N+1)/2)
420 J=J+1:F2=N-J+1
430 D=F1*F2:V=V-D
440 PRINT J;SPC(1);D;SPC(3);V
450 IF J<N GOTO 420:END
```

B. Sort Program

The program below sorts a list of numbers into ascending order using an exchange sort technique.

```
10 REM SORT PROG.
20 CLS
30 PRINT "HOW MANY":INPUT "  NUMBERS";N
40 DIM K(N)
50 CLS:PRINT "ENTER VALUES:"
60 FOR I=1 TO N:INPUT K(I):NEXT I
70 FOR I=1 TO N-1
80 REM FIND NEXT SMALLEST NO.
90 FOR J=I+1 TO N
100 IF K(J)>=K(I) GOTO 130
110 REM EXCHANGE SMALLER NO.
120 K1=K(I):K(I)=K(J):K(J)=K1
130 NEXT J
140 NEXT I
150 PRINT:PRINT "ORDERED LIST:":PRINT
160 FOR I=1 TO N:PRINT K(I):NEXT I
```

C. Craps Game

This program simulates a crap game. Rules are printed if you request them. Toss the dice by pressing the 'CR' key when asked to throw the dice.

```
10 CLS:INPUT "RULES";A$
20 IF A$="YES" OR A$="Y" THEN GOSUB 500
30 REM PLAY CRAPS
40 CLS:PRINT "--THROW--":B$=INSTR$(1)
50 GOSUB 600
60 ON NUM-1 GOTO 70,70,90,90,90,80,90,90,90,80,70
70 CLS
71 PRINT NUM; "--YOU LOSE":GOTO 140
80 CLS
81 PRINT NUM; "--YOU WIN":GOTO 140
90 CLS:OLD=NUM
91 PRINT NUM; "--THROW AGAIN":B$=INSTR$(1)
100 GOSUB 600
110 IF NUM=OLD GOTO 81
120 IF NUM=7 GOTO 71
130 GOTO 91
140 PRINT:INPUT "PLAY AGAIN";A$
150 IF A$="YES" OR A$="Y" GOTO 10
160 END
```

APPENDIX G

TONE Parameters for Generating Music

The TONE command takes two arguments. The first determines the pitch of the tone. The product of the two arguments determines the length of the note. (See section 5-3.) Listed below are values for the first parameter and the pitches they produce for two octaves of the chromatic scale. To produce two different pitches of equal duration, you must calc the second parameters so that the products of the parameters are equal.

For example, suppose you wish to produce a C-E-G triad. From the chart we can see that 168 as the first TONE parameter produces a C. Pick a second parameter that produces a C of the desired length, e.g. TONE 168,150. The product of these parameters is 168 x 150 = 25,200.

The first parameter value for E is 131. The second parameter is calculated by solving the equation 25,200 = 131 x ?. Since 25,200/131 is approximately 192, the command "TONE 131,192" will produce an E about the same length as the C. For the G, calculate the second parameter as 25,200/110, or about 229. Therefore, use TONE 110,229 to produce a G of the same length as the E and C.

To try this example, run the following program:

```
10 TONE 168,150
20 TONE 131,192
30 TONE 110,229
```

Dividing all second parameters by 2 produces tones lasting 1/2 as long, making the notes go faster. Multiplying them by 2 produces notes twice as long, and so on.

```
500 REM RULES
510 CLS:PRINT "RULES FOR CRAPS:":PRINT
520 PRINT "TO WIN:":PRINT
530 PRINT "GET A 7 OR 11":PRINT " ON 1ST THROW"
540 PRINT:PRINT "OR GET 4,5,6,8":PRINT "9 OR 10 THEN"
550 PRINT " MATCH IT":B$=INSTR$(1)
560 CLS:PRINT "TO LOSE:":PRINT
570 PRINT "GET A 2,3 OR 12":PRINT " ON 1ST THROW"
580 PRINT:PRINT "OR THROW A 7"
590 PRINT " BEFORE YOU MATCH":PRINT " YOUR 1ST THROW"
595 RETURN
600 REM DICE THROW
610 A=RND(1):A=INT(6*A)+1
620 B=RND(1):B=INT(6*B)+1
630 NUM=A+B:RETURN
```

D. Circumference and Area of a Circle

The program below calculates the circumference and area of a circle, given the circle's radius. A radius of zero terminates the program.

```
10 CLS:PRINT:INPUT "RADIUS";R
15 IF R=0 THEN END
20 C=2*3.14159*R
30 A=3.14159*R↑2
40 PRINT:PRINT "CIRCUM.=";C
50 PRINT "AREA=";A
60 A$=INSTR$(1)
70 GOTO 10
```

TONE Parameters

NOTE	FIRST TONE PARAMETER
LOW G	224
$G^{\#}$	212
A	200
$A^{\#}$	189
B	179
C	168
$C^{\#}$	158
D	148
$D^{\#}$	139
E	131
F	124
$F^{\#}$	117
MIDDLE C	110
$G^{\#}$	104
A	97
$A^{\#}$	91
B	85
C	80
$C^{\#}$	75
D	71
$D^{\#}$	67
E	63
F	59
$F^{\#}$	55
HIGH G	51