

IBM

5100

IBM 5100  
BASIC Introduction



IBM 5100  
Portable Computer  
BASIC Introduction

This manual introduces the IBM 5100 Portable Computer that can be programmed with the BASIC language. It is intended to provide persons using the 5100 with the information necessary to operate the 5100 using the BASIC language.

#### Related Publications

- *IBM 5100 BASIC Reference Manual*, SA21-9217
- *IBM 5100 BASIC Reference Card*, GX21-9218
- *IBM 5100 Communications Reference Manual*, SA21-9215

#### Second Edition (December 1975)

This is a major revision of, and obsoletes, the previous edition SA21-9216-0. Changes are continually made to the specifications herein; any such changes will be reported in subsequent revisions or technical newsletters.

Requests for copies of IBM publications should be made to your IBM representative or the IBM branch office serving your locality.

A form for reader's comments is at the back of this publication. If the form has been removed, address your comments to IBM Corporation, Publications, Dept 245, Rochester, MN 55901.

## Contents

<b>CHAPTER 1. INTRODUCTION . . . . .</b>	<b>1</b>
About This Manual . . . . .	1
About BASIC . . . . .	1
About the 5100 . . . . .	3
Alphabetic Keys . . . . .	3
Numeric Keys . . . . .	3
Operating Keys . . . . .	5
BASIC Command Keywords . . . . .	6
BASIC Statement Keywords . . . . .	6
Arithmetic Operator Keys . . . . .	6
Getting Started . . . . .	6
Entering and Displaying Data . . . . .	7
Correcting Keying Errors . . . . .	11
<b>CHAPTER 2. HOW YOUR 5100 HANDLES ARITHMETIC . . . . .</b>	<b>16</b>
Arithmetic Operators . . . . .	16
The Sequence of Arithmetic Operations . . . . .	21
Positive/Negative Operators . . . . .	17
Arithmetic Constants . . . . .	22
Finding Square Roots . . . . .	23
Variables . . . . .	23
Variables That Stand for Numbers . . . . .	23
Variables That Stand for Characters . . . . .	27
Using Calculation Results . . . . .	28
<b>CHAPTER 3. ENTERING, RUNNING, AND STORING A PROGRAM . . . . .</b>	<b>30</b>
Entering a Program . . . . .	30
Correcting Your Keying Errors . . . . .	31
Running the Program . . . . .	31
Automatic Statement Numbering . . . . .	35
Storing the Program . . . . .	35
Tape Preparations . . . . .	35
SAVE Command . . . . .	38
LOAD Command . . . . .	39
A Review of What You've Done . . . . .	39
<b>CHAPTER 4. HOW TO WRITE A PROGRAM . . . . .</b>	<b>41</b>
The LET Statement . . . . .	41
Using Remarks . . . . .	42
Listing Program Contents . . . . .	44
Branches . . . . .	44
The GOTO Statement . . . . .	44
The IF Statement . . . . .	44
Loops . . . . .	49
<b>CHAPTER 5. OTHER WAYS TO PUT VALUES INTO PROGRAMS . . . . .</b>	<b>58</b>
The READ, DATA, and RESTORE Statements . . . . .	58
The INPUT Statement . . . . .	60
Prompting Your Input . . . . .	61
Entering Character Variables into Programs . . . . .	62
A Review of What You've Done . . . . .	63
<b>CHAPTER 6. MAKING CHANGES TO YOUR PROGRAMS . . . . .</b>	<b>64</b>
Correcting Keying Errors . . . . .	64
Inserting New Lines . . . . .	64
Replacing One Line with Another . . . . .	66
Removing a Line . . . . .	67
Renumbering Statement Lines . . . . .	68
<b>CHAPTER 7. MORE ABOUT THE PRINT STATEMENT . . . . .</b>	<b>69</b>
Making Headings . . . . .	70
Math Calculations in PRINT Statements . . . . .	71
<b>CHAPTER 8. SETTING UP YOUR OWN FORMAT—PRINT USING AND IMAGE STATEMENTS . . . . .</b>	<b>72</b>
Example of Printing . . . . .	74
<b>CHAPTER 9. MORE THINGS YOU CAN DO WITH BASIC . . . . .</b>	<b>77</b>
Some General System Functions . . . . .	77
Conversion Functions and Constants . . . . .	78
Trigonometric Functions . . . . .	78
Logarithms and Exponents . . . . .	79
<b>CHAPTER 10. TAPE DATA FILES . . . . .</b>	<b>80</b>
Activating and Deactivating Files . . . . .	80
Creating a Tape File . . . . .	81
Retrieving a File . . . . .	81
Repositioning Files . . . . .	82
<b>CHAPTER 11. ARRAYS . . . . .</b>	<b>84</b>
Defining an Array . . . . .	86
DIM Statement for One-Dimensional Arrays . . . . .	86
DIM Statement for Two-Dimensional Arrays . . . . .	87
Elements of Arrays . . . . .	87
Assigning Values to Array Elements . . . . .	88
Another Way to Assign Values to Arrays . . . . .	91
Assigning Values to an Entire Array at Once . . . . .	92
Working with Elements of Arrays . . . . .	93

Printing Arrays . . . . .	93
Putting One-Dimensional Arrays Together in a Program . . . . .	94
Two-Dimensional Array. . . . .	95
Arithmetic with Arrays . . . . .	96
Addition and Subtraction with Arrays . . . . .	97
Multiplication and Division . . . . .	97
Averaging Two Sets of One-Dimensional Arrays . .	98

**APPENDIX A. BASIC STATEMENTS AND**

<b>COMMANDS . . . . .</b>	<b>100</b>
BASIC Statements . . . . .	100
BASIC System Commands . . . . .	102
Editing Function . . . . .	102
<b>INDEX . . . . .</b>	<b>103</b>

### ABOUT THIS MANUAL

This manual will show you how to operate the 5100 using the BASIC language. If you are already familiar with the BASIC language, you may be able to skip most of the language-only topics and simply learn how to operate the 5100. If you are not familiar with the BASIC language, you should read the manual from cover to cover while performing the suggested keying operations or examples on your 5100. Not all of the features and functions of the BASIC language are covered in this manual. For more information about the 5100 or the BASIC language, see the *IBM 5100 BASIC Reference Manual*, SA21-9217.

This manual assumes that your 5100 has been installed and checked out. If it's not, use the setup procedure in the *IBM 5100 BASIC Reference Manual* before continuing with this manual.

### ABOUT BASIC

BASIC is an interactive computer language, that is, whatever you enter into the 5100 is processed immediately. BASIC has many built-in functions that allow you to effectively solve your problems. BASIC also allows you to write programs using BASIC language statements and facilities. These programs can be stored on the tape cartridge for later use.

BASIC is a good language to experiment with. Nothing you do from the keyboard can damage the 5100; and the more you experiment, the more you will learn about BASIC and the 5100.

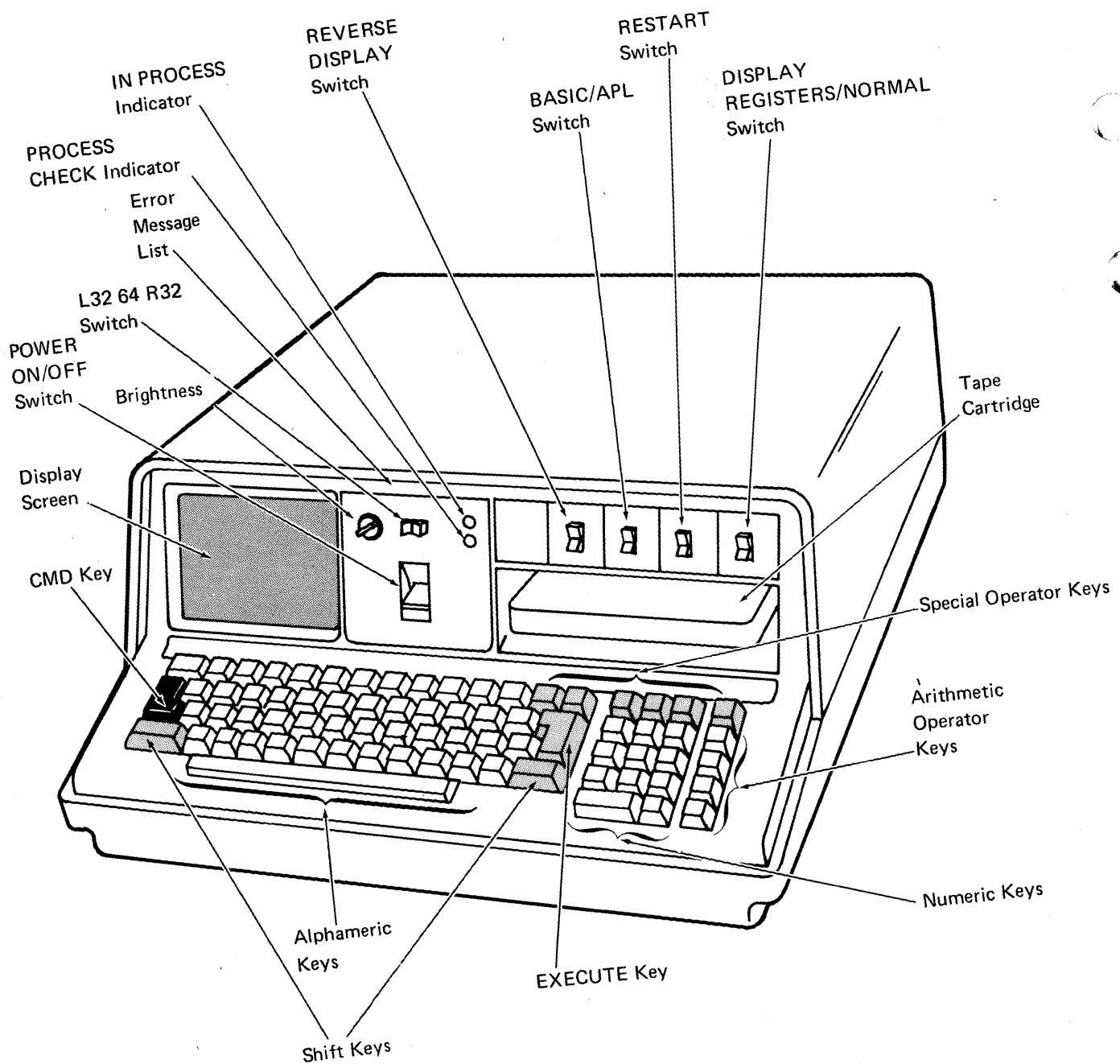


Figure 1. The 5100

## ABOUT THE 5100

The 5100 (Figure 1) is a portable computer designed to help you solve problems. The display screen and indicator lights communicate information to you, and the keyboard and switches allow you to control the operations the 5100 will perform.

Before you begin to use the 5100, you should become familiar with the keys and the control panel (Figure 1). The control panel consists of a series of switches, which will be explained later.

What follows is a brief description of the keys. How you use the keys will be described later. For now, familiarize yourself with the names and locations of keys on the 5100. Figure 2 shows a BASIC-only keyboard, and Figure 3 shows a combined BASIC/APL keyboard.

### Alphabetic Keys

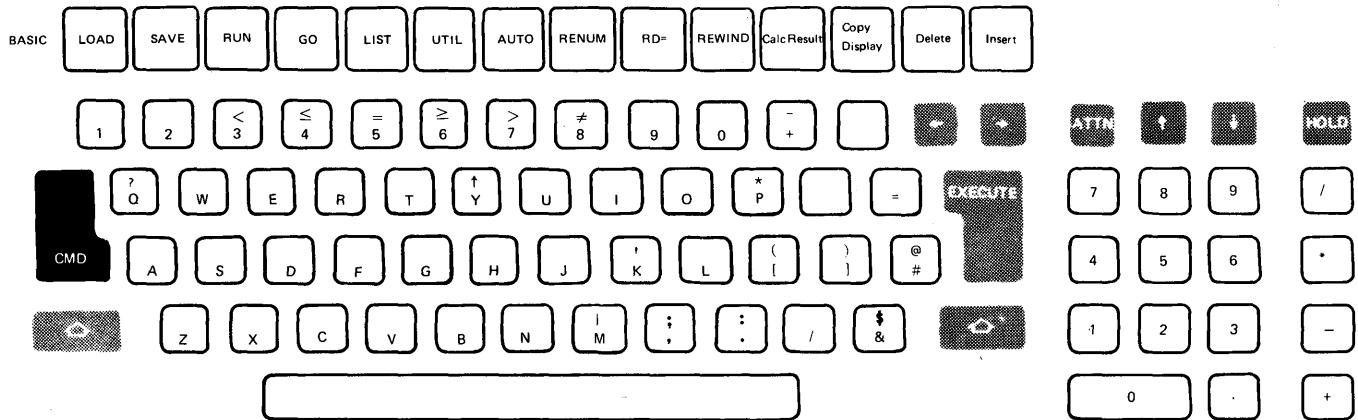
The alpha keys are similar to those on a standard typewriter, except that there are no lowercase characters. The alpha characters are all uppercase, even though they are in the lower shift position on the key. Thus, you *do not* use the shift key (  ) for alpha characters.

If you want to enter an upper shift special symbol, such as ?, you must hold down the shift key, then press the key to enter the special symbol, just as you would to type an uppercase character on an ordinary typewriter.

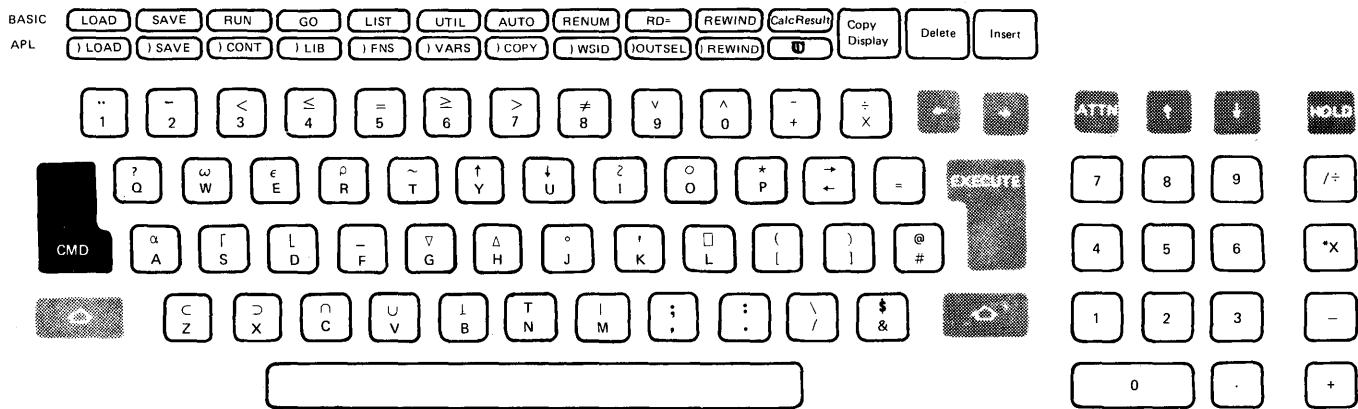
If your 5100 is equipped to operate either BASIC or APL programs, you may be unfamiliar with the symbols appearing at the top of some of the alphabetic keys (Figure 3). For BASIC operations, *even on a BASIC-only machine (Figure 2) where they are not shown on the key top*, these symbols can be displayed or printed, although their APL functions do not apply to BASIC operations. See the *IBM 5100 BASIC Reference Card*, GX21-9218, for the APL characters and their locations.

### Numeric Keys

Either the top row of alphabetic keys or the special calculator arrangement of numeric keys on the right of the keyboard can be used to enter numbers.



**Figure 2. BASIC-only 5100 Keyboard**



**Figure 3. Combined BASIC/APL 5100 Keyboard**

## Operating Keys

The black key labeled CMD; the gray keys with the legend names EXECUTE, ATTN, and HOLD; and the gray keys with the arrows are special operating keys (Figure 4). The gray keys with the arrows and the spacebar (used to enter blank characters) automatically repeat the operation they perform when held down.

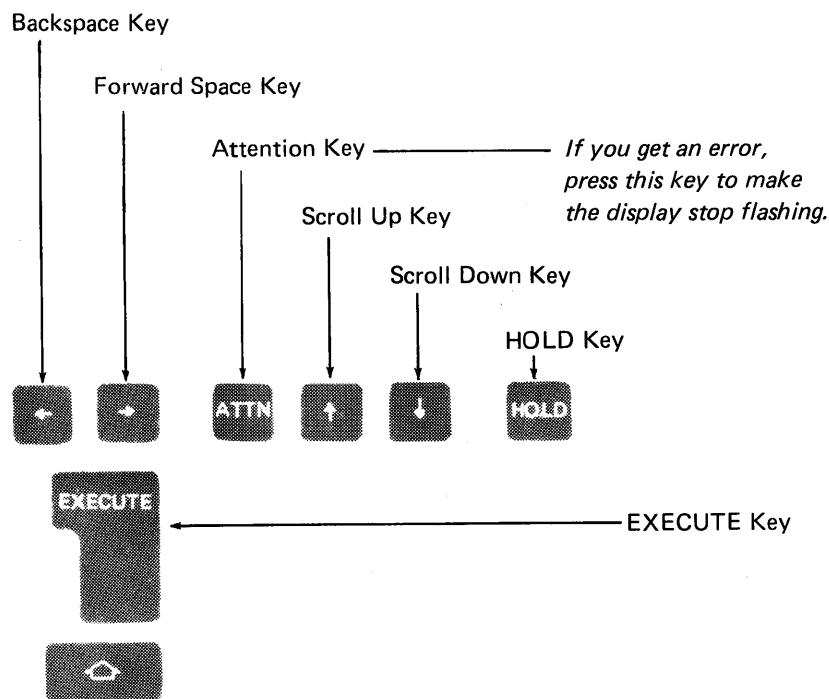


Figure 4. 5100 Special Operating Keys

## **BASIC Command Keywords**

The words listed above the top row of alphabetic keys (1-0) are BASIC command keywords that you can enter by holding down the CMD key and then pressing the number below the desired command. For example, to enter the LOAD command keyword, hold down the CMD key and press 1. These commands and their use are described later.

## **BASIC Statement Keywords**

The words printed on the front of some of the alphabetic keys are BASIC statement keywords. The words will appear on the display screen if you press the key while holding down the CMD key. This permits you to enter an entire word, error-free, with one or two keystrokes.

## **Arithmetic Operator Keys**

The four keys to the right of the calculator arrangement of numeric keys are the arithmetic operator keys that are used to perform division, multiplication, subtraction, and addition. There are also alphabetic keys that perform the same functions. In BASIC the symbol / is used for division, and the symbol \* is used for multiplication.

## **GETTING STARTED**

Make sure the switches on your 5100 are set as follows:

<b>Switch</b>	<b>Setting</b>
L32 64 R32	64
DISPLAY REGISTERS/NORMAL	NORMAL
BASIC/APL (combined machines only)	BASIC

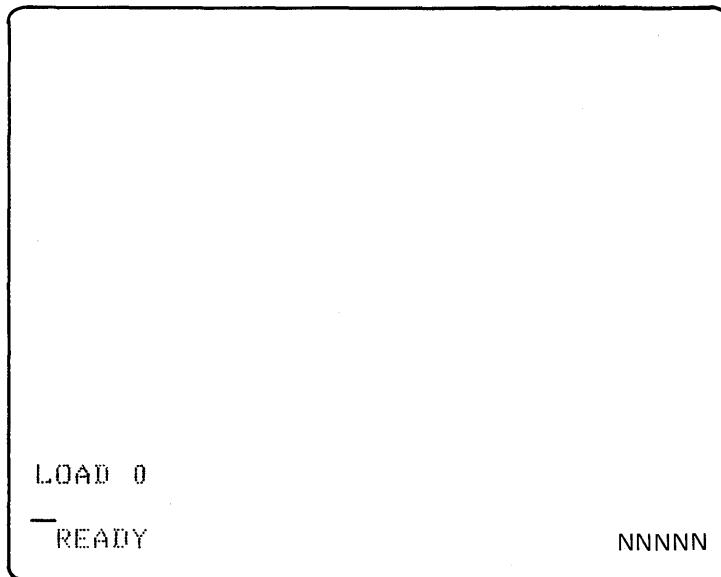
If your 5100 has the BASIC/APL switch, it can execute either BASIC or APL language statements. The language used is selected only during the power up procedure or when the RESTART switch is pressed. Make sure your 5100 is plugged in and turn the power on. If the power is already on, press RESTART and wait a few seconds. During this time, the 5100 performs internal checks to make sure it is operating correctly.

If an error is detected during these checks, the PROCESS CHECK indicator may come on. If the PROCESS CHECK indicator comes on, press RESTART. The 5100 will again perform the internal checks. If the light comes on again, call for service.

The IN PROCESS indicator comes on whenever the display screen is blank, which indicates that the 5100 is doing internal processing.

## ENTERING AND DISPLAYING DATA

First, let's look at the display screen. Your display screen should look like this:

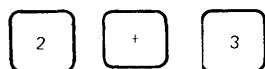


If the READY message does not appear, press RESTART again, and wait a few seconds. If the READY message still does not appear, call for service.

The LOAD 0 (zero) message indicates that the 5100 has a clear work area. The flashing underline () between the LOAD 0 and READY messages is called a cursor. It tells you where the next character you enter will be displayed. The READY message indicates that the 5100 is ready to receive your instructions. The number in the lower right corner, indicated by the NNNNN on the display screen drawing, is the number of character positions (bytes) in the work area available for your instructions and data. This number changes during processing. The number is omitted on the remaining display screen drawings in the manual.

The display screen can contain up to 16 lines of data. The bottom line indicates the status of the 5100 and specifies the number of bytes available in the work area (NNNNN). The line next to the bottom displays the input you are entering from the keyboard. The remaining lines display the preceding 14 lines that have been entered and processed. When the data on the input line is processed, that line is moved up one line, leaving the input line empty so more data can be entered. Up to 64 characters of data can be entered per line.

Now let's enter some data into the 5100. Enter the following problem using the numeric keys and arithmetic operator keys:



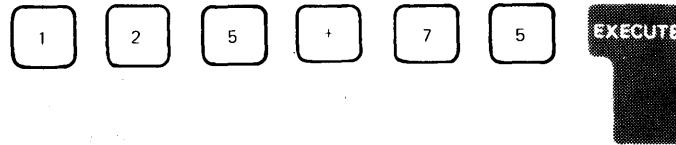
Notice that the characters are displayed as each key is pressed. To process the data you just entered, you must press the EXECUTE key. Press EXECUTE now.

The display screen shows:

A small rectangular box representing a display screen. Inside the box, the word "LOAD" is followed by a space and "0". Below that, "2+3" is displayed. On the next line, the number "5" is shown. At the bottom of the box, the word "READY" is preceded by a short horizontal line.

Notice that the instruction you entered, 2+3, is on the left margin of the display screen, while the answer, 5, is indented one position from the left margin on the next line.

Enter and execute 125+75 by pressing the following keys:



This display screen shows:

LOAD 0  
2+3  
5  
  
125+75  
200  
  
READY

The appearance of your display can be changed by switches on the control panel. The REVERSE DISPLAY switch allows you to change from black characters on a white background to white characters on a black background, or vice versa. Change the switch and select the type of display you feel most comfortable with. You may have to adjust the BRIGHTNESS control switch as you change from one background to the other.

Now watch the display as you set the L32 64 R32 switch to the L32 position. With the switch in this position, the leftmost 32 characters on each line are displayed with a space between each character. With the switch in the L32 position, your display screen shows:

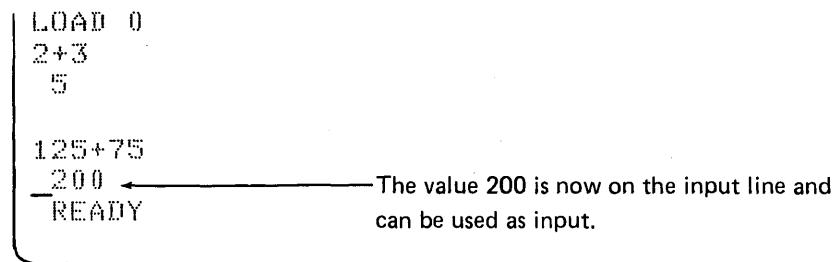
L O A D 0  
2 + 3  
5  
  
1 2 5 + 7 5  
2 0 0  
  
READY

Now set the switch in the R32 position and notice that the display is blank (except for the storage number). In the R32 position, the right-most 32 characters are displayed with a space between each character.

Return the switch to the 64 position, and notice that all characters are displayed without the space in between. For the exercises in the remainder of this book, keep the switch in the 64 position.

There are two gray keys with white arrows above the numeric keyboard. These keys move the display lines (except the status line) up or down. The scroll up key  moves the display lines up one line, and the scroll down key  moves the display lines down one line. Both keys continue to move the display lines if they are held down. Now use the scroll down key  to move the display down two lines.

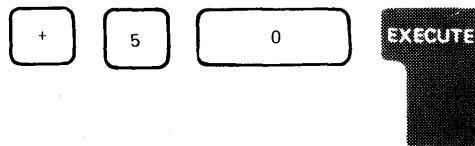
The display screen shows:



LOAD 0  
2+3  
5  
  
125+75  
200 ←  
READY

The value 200 is now on the input line and can be used as input.

Use the forward space key and move the cursor to the right of 200. Notice that the cursor (the underline) is replaced by a flashing character as you space the cursor through the numeric characters. The flashing character serves the same function as the cursor; it indicates the position in the line where input from the keyboard will be displayed. Now, press the following keys:



The display screen shows:

LOAD 0  
2+3  
5  
  
125+75  
200+50  
250  
  
READY

You are now familiar with the format of the display screen. From this point on, only the line or lines being discussed will be shown.

### Correcting Keying Errors

The 5100 has a number of very useful functions that allow you to correct errors made while entering data. On a line-by-line basis, at any time, you can

1. Replace a character
2. Delete a character
3. Insert a character

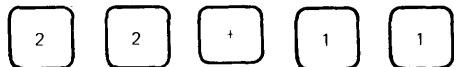
#### Replace a Character

To replace a character, move the cursor with the backspace key 

or forward space key  until it is at the incorrect character. The

cursor moves one character space in the direction of the arrow each time the appropriate arrow key is pressed. These keys will continue to move the cursor if they are held down. The incorrect character is then replaced simply by keying the correct character over the incorrect character. (In some instances, characters can be combined to form a character not on the keyboard; for example, the period and quotation mark combine to make an exclamation mark. If you want to replace one of these characters (the . or ') with the other, you should backspace to the character, press the spacebar to delete the character, backspace again, then enter the desired character.)

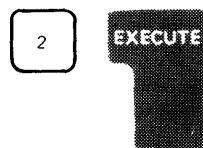
For example, you want to do the problem 22+12, but you press the following keys:



The display screen shows:

22+11...

To correct this error, the cursor must be moved back one position (under the second 1) so that character can be rekeyed. Now, press the backspace key once. To correct the keying error and execute the problem, press the following keys:



#### Delete a Character

To delete a character, you also use the backspace key or the forward space key to move the cursor. Once the cursor is in the position of the character to be deleted (the character is flashing), hold down the CMD key and press the backspace key once. The character is then deleted, and any characters to the right are shifted one position to the left to close up the space left by the deletion.

For example, you want to do the problem 13+45, but you press the following keys:



The display screen shows:

O

123+45...

C

Press the backspace key to move the cursor (flashing character) back to the 2. Look at the labels that appear above the backspace and forward space keys: Delete and Insert. To perform the delete function, hold down the CMD key while you press  once.

C

The display screen shows:

13+45

— This character is flashing.

Now press EXECUTE to execute the problem. Pressing the EXECUTE key processes the entire line regardless of the position of the cursor.

O

#### Insert a Character

To insert a character, position the cursor using the backspace key 

or the forward space key  then hold down the CMD key and press the forward space  key once. This operation moves the flashing

character (and all other characters to the right of it) one position to the right, creating the space you need to insert one character. The cursor is not moved and is now displayed as an underline. To insert the character, simply press the character key. If a character is in the last (64th) position of the line, the insert function is ignored.

For example, you want to do the problem 123\*6, but you press the following keys:

C



C

The display screen shows:

13\*6...

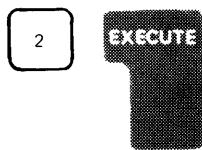
To correct the error, press the backspace key to move the cursor (flashing character) back to the 3. Look at the labels that appear above the backspace and forward space keys: Delete and Insert.

To perform the insert function with the cursor position at the 3, hold down the CMD key while you press  once.

The display screen shows:

1...3\*6

To correct the keying error and execute the problem, press the following keys:



There is one more way to correct a keying error. If you make several errors halfway through the line, you can backspace the cursor to the character following the last correct character and then press the ATTN (attention) key. This causes everything from the cursor position to the end of the line to be cleared from the display screen.

Since the data from the input line is not processed until the EXECUTE key is pressed, you can visually verify any input before it is processed. However, if you do press EXECUTE before you notice a mistake, you must press ATTN; then you can simply enter the input again, or you can use the scroll down key  to move the input back to the input

line and correct it. When the corrections have been made, press EXECUTE again.

O For example, you want to do the problem 135+280, but you entered and executed 134+280. The display screen shows:

134+280  
414

C To correct the original input, press the scroll down key  three times to get the original input back to the input line. The display screen shows:

134+280  
↑  
This character is flashing.

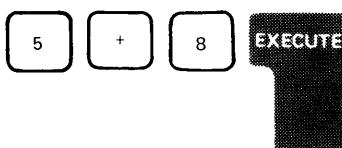
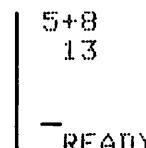
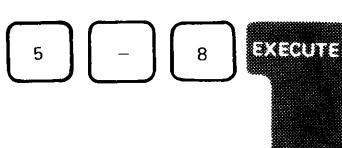
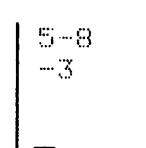
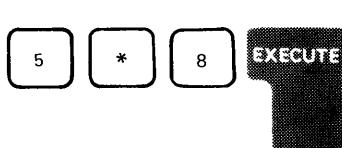
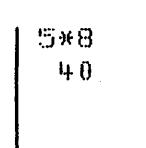
You may now correct the error, and then process the data again by pressing EXECUTE.

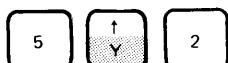
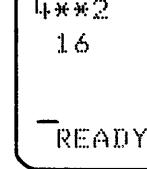
## Chapter 2. How Your 5100 Handles Arithmetic

The following examples show some common, simple arithmetic operations you can do on the 5100.

### ARITHMETIC OPERATORS

Before you begin these simple operations, you should know that some of the arithmetic signs (called *operators*) you are accustomed to using are different when you use the BASIC language. For example, the multiply sign ( $\times$ ) is not used in BASIC. Instead, the asterisk (\*) is used for multiplication. Similarly, the divide sign ( $\div$ ) is replaced by the slash (/) in BASIC. The sign for exponentiation (raising to a power) in BASIC is  $\uparrow$  (the upper shift character on the Y key) or \*\*. Now enter these problems:

Arithmetic	You Press	The Display Screen Shows
Add 5 and 8		
Subtract 8 from 5		
Multiply 5 times 8		

Arithmetic	You Press	The Display Screen Shows
Divide 5 by 8		
5 to the power of 2	 (Hold down the shift key and press the Y key to get ↑ symbol.)	
4 to the power of 2		

## THE SEQUENCE OF ARITHMETIC OPERATIONS

BASIC has a prescribed order of arithmetic execution called arithmetic hierarchy. When two or more operators such as +, -, \*, /, or  $\uparrow$  are used, arithmetic is performed according to this hierarchy. That is, operators with higher priorities are performed first, while operators with the same priority are performed as they are encountered from left to right. The arithmetic hierarchy in BASIC is:

1. Operations enclosed in parentheses
2. Mathematical functions (for example, sine, cosine, or square root)
3. Exponentiation ( $\uparrow$  or  $**$ )
4. Positive/negative operations, which are described later
5. Multiplication and division (\*, /)
6. Addition and subtraction (+, -), which have the lowest priority

When an operation is enclosed in parentheses, it is performed first, even though the operator enclosed in the parentheses may have a lower priority than the operators outside the parentheses. As a result, the prescribed order of execution can be changed if you use parentheses.  
*Operations enclosed in parentheses are executed in BASIC before operations outside parentheses, regardless of the hierarchy of the operators.*

Some examples of arithmetic hierarchy are:

$$3+4/5$$

$$4/5+3$$

In both of these examples, the 4 is first divided by the 5 because the divide operation has the highest priority. The .8 result is then added to the 3, giving a final result of 3.8.

Another example is:

$$(3+4)/5$$

In this example, the 3 and 4 are first added because they are enclosed in parentheses. The result, 7, is then divided by 5, giving a final result of 1.4.

In the expression

$$16+23-4+133-8$$

addition and subtraction occur from left to right in the following sequence:

$$\begin{array}{r} 16 \\ + \underline{23} \\ \hline 39 \quad \text{Interim result} \\ - \underline{4} \\ \hline 35 \quad \text{Interim result} \\ + \underline{133} \\ \hline 168 \quad \text{Interim result} \\ - \underline{8} \\ \hline 160 \quad \text{Final result} \end{array}$$

However, when parentheses are added, the sequence of operations can change:

$$(16+23)-((4+133)-8)$$

In this example, the operations occur in the following sequence:

1. Proceeding from left to right, the expression in the first set of parentheses is evaluated.

$$\begin{array}{r} 16 \\ + 23 \\ \hline 39 \end{array}$$

Interim result 1

2. The parenthetical expression within the second set of parentheses is evaluated next.

$$\begin{array}{r} 4 \\ + 133 \\ \hline 137 \end{array}$$

Interim result 2

3. The second set of parentheses is now evaluated.

$$\begin{array}{r} 137 \\ - 8 \\ \hline 129 \end{array}$$

Interim result 3

4. Finally, the subtraction (having the same priority as addition) is performed and the result is displayed.

$$\begin{array}{r} 39 \\ - 129 \\ \hline - 90 \end{array}$$

Interim result 1  
Interim result 3  
Final result

Although the numbers and operators in this expression are the same as those in the previous example, the parentheses completely change the order of the operations.

Now, determine the result of the following expression by entering the numbers, then pressing the EXECUTE key:

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

The order of the arithmetic operations is:  $4-2=2$ ,  $3\times 2=6$ , and  $4^6=4096$ .

The display screen shows:

$4\uparrow(3*(4-2))$   
4096

Figure 5 shows other examples of arithmetic expressions.

<b>This is the Way it Looks as Arithmetic:</b>	<b>This is the Way it Looks in BASIC:</b>	<b>This is What it Means:</b>
$\frac{a+b+c}{2}$	$(A+B+C)/2$	First add A, B, and C; divide the sum by 2.
$a+\frac{b+c}{2}$	$A+(B+C)/2$	Add B and C, divide the sum by 2; add the result to A.
$3a+4$	$3^* A+4$	Multiply A by 3; then add 4.
$3(a+4)$	$3^*(A+4)$	Add A and 4; multiply the sum by 3.
$x^2+7$	$X\uparrow 2+7$	Square X and add 7.
$(x+7)^2$	$(X+7)\uparrow 2$	Add X and 7; square the quantity.
$\frac{(x+1)^2}{2}$	$(X+1)\uparrow 2/2$	Add X and 1; square the quantity; divide the result by 2.
$\frac{\left(\frac{x^2}{2}\right)(x+y)}{3}$	$(X\uparrow 2/2)*(X+Y)/3$	Square X and divide by 2; add X and Y and multiply by the previous result; divide that result by 3.

**Figure 5. Examples of Arithmetic and BASIC Expressions**

As you can see, the more complicated the arithmetic expression looks, the more complicated the BASIC expression looks. When you are writing BASIC expressions, remember that parentheses must always be in balanced pairs—as many right parentheses as left parentheses. If a statement gets too complicated, you can usually break it down into several simpler statements.

## Positive/Negative Operators

The plus (+) and minus (-) signs indicate that a number is positive or negative. When used for this purpose, the + and - signs have a higher priority in the arithmetic hierarchy than they have when used for addition and subtraction. In the following example:

-2↑2-3

the 2 is raised to the second power, and the minus is assigned to the result before the subtraction.

One rule that you must follow is that *you cannot use two operators sequentially*, except \*\* which is the same as ↑. Sequential operators must be separated by parentheses. This rule applies to both the arithmetic operators (+, -, \*, /, and ↑) and the positive/negative operators (+ and -). For example, enter  $7^{-3}$  as  $7\uparrow-3$ . The flashing display screen shows:

7↑-3  
7↑-3  
↑

ERROR 100

The lower arrow indicates the syntax error. Press ATTN to stop the flashing screen, then enter the corrected expression,  $7\uparrow(-3)$ . A complete description of 5100 error messages is included in the *IBM 5100 BASIC Reference Manual*, SA21-9217. Short descriptions of the error messages are given on the pullout card above the display screen.

You must use parentheses to separate consecutive operators, as in the following examples:

Invalid	Valid
---------	-------

6+-4                  6+(-4)

3\*-1.5                  3\*(-1.5)

8--4                  8-(-4)

Try to solve the following problems using the 5100. Remember to press EXECUTE after entering each problem.

7\*3.5-1.28

23.22 ←

3.2/(-1.6\*4)

-5 ←

1.2↑3\*8/(4-,2)

3637.894737 ←

Answers

## ARITHMETIC CONSTANTS

BASIC has three built-in arithmetic constants to represent the values of:

1. e (natural log) = 2.718281828459
2.  $\pi$  (pi) = 3.141592653590
3.  $\sqrt{2}$  (square root of 2) = 1.414213562373

For example, if you want to use  $\pi$  in an equation, you don't need to type in 3.14 . . . You just use the special BASIC constant. Here are the special symbols that BASIC recognizes for these constants:

For This Constant:	Use This Symbol:
e (natural log)	&E
$\pi$ (pi)	&PI
$\sqrt{2}$ (square root of 2)	&SQR2

You might use one of these constants in a program that calculates the area of a circle (AREA =  $\pi R^2$  is the formula). Your program statement would read:

50 LET A = &PI\*R↑2

You can use these constants anywhere in your programs. In addition, there are special constants in BASIC that are used to convert pounds, inches, and gallons to metric kilograms, centimeters, and liters respectively. Normally, when you want to switch from the U.S. measuring system to the metric system, you multiply the measured quantity by a fixed constant to obtain the equivalent measurement in the metric system. For example, 1 lb equals 0.454 . . . kg, so 2 lbs equal 2\*0.454 . . . kg. With BASIC, instead of remembering what the conversion multipliers are, the 5100 can provide them for you. These constants are:

- &INCM, which has a value of 2.54 (centimeters per inch)
- &LBKG, which has a value of 0.453592 (kilograms per pound)
- &GALI, which has a value of 3.785412 (liters per gallon)

## FINDING SQUARE ROOTS

You can determine square roots automatically with your 5100. Instead of writing your own formula for determining the square root of a number, you use the letters SQR, followed by the number whose square root you want to know enclosed in parentheses. For example, SQR (X) finds the square root of X, where X is 0 or a positive number.

You can use SQR in any of your arithmetic expressions, and the expressions inside the parentheses can involve any kind of arithmetic. For example:

$\sqrt{X+Y}$       is entered as      SQR (X+Y)

$\sqrt{\frac{X+Y+Z}{5}}$       is entered as      SQR ((X+Y+Z)/5)

$\sqrt{\frac{A+X}{2}}$       is entered as      SQR(A+X/2)

Other conversion and trigonometric functions and conversion constants in BASIC are discussed in Chapter 9.

## VARIABLES

Algebraic formulas often contain variables to which you assign your own values when using the formulas. In the formula  $\pi R^2$ , for example, the R is a variable representing the radius. You must assign a value to R when you use the formula.

### Variables That Stand for Numbers

You can *name* a variable in BASIC with a single character of the extended BASIC alphabet (A-Z, @, \$, and #). A BASIC variable can also be named with one of the preceding letters or symbols followed by a single digit (0 through 9). Typical variable names are A2, #9, and B1. You can name a maximum of 319 different variables in BASIC.

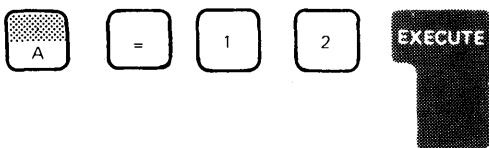
## Assigning Values to Variables

Variables are assigned values by using the equal (=) sign. After you assign a value to a variable, you can use the variable in a calculation. For example, if you enter

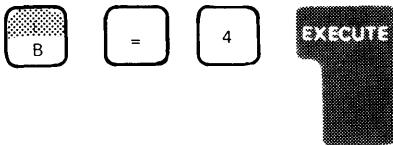


you assign the numeric value of 12 to the variable named B4.

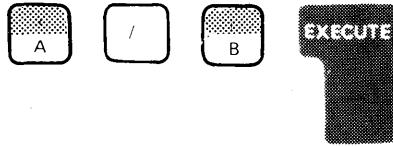
To illustrate this, enter the following:



and



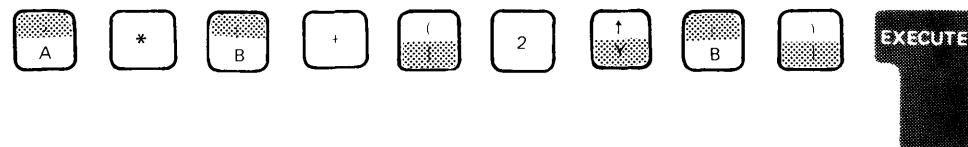
You have named variables A and B and assigned them values of 12 and 4 respectively. You can now use them in the following calculation:



The display screen shows:

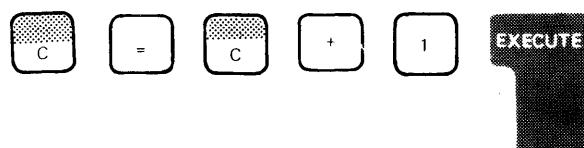
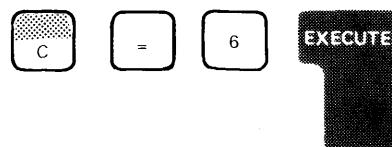
A=1.2  
1.2  
B=4  
4  
A/B  
3  
  
READY

Now press the following keys:



The answer is 64.

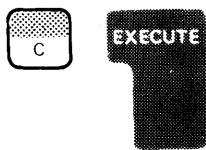
Press the following keys:



Notice that the second expression increases the value of C by 1. This method of changing the value of a variable is commonly used in programming for stepping through a process. This will be described in detail later.

#### Displaying Variable Values

To find the current value of any variable, you can simply enter the variable name and press EXECUTE. For example, key:



The display screen shows the variable and its current value:

C  
7

#### A Note About Numbers

When you use numbers in BASIC, they can be:

- *Integers* (whole numbers) such as:

2, -76, 842, 10000000, or 999111

- *Decimal numbers* such as:

-1.5, 3.7772, 0.00081, or -457.25

- Numbers in *exponential format* such as:

6E7 (meaning  $6 \times 10^7$ ) or 5.4E-3 (meaning  $5.4 \times 10^{-3}$ )

## A Note About Names

When you name variables in BASIC, they can be:

- A single character of the extended BASIC alphabet (A-Z, @, \$, or #) such as:

\$, C, or V

- A single character of the extended BASIC alphabet (A-Z, @, \$, and #) followed by a single number such as:

A4, \$6, \$3, or T3

## Variables That Stand for Characters

While you usually think of variables as standing for numbers, in BASIC you can let a variable stand for combinations of characters such as words or names. If a variable is going to represent a word or a name, it is called a *character variable*. You name character variables with one letter of the extended BASIC alphabet (A-Z, @, \$, and #) followed by a dollar sign (\$). The dollar sign tells the 5100 that the variable is a character variable.

To assign a word or name to a character variable, you enclose the word or name in single quotation marks following the equal sign. For example:

```
A$='HARVEY SMITH'
```

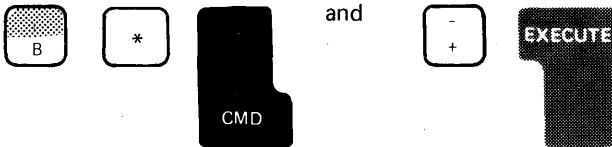
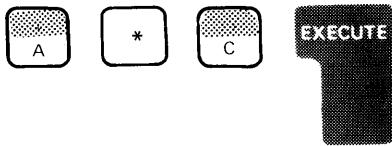
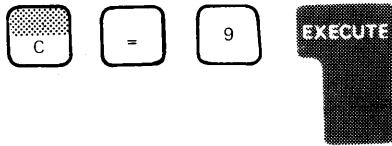
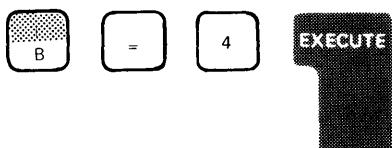
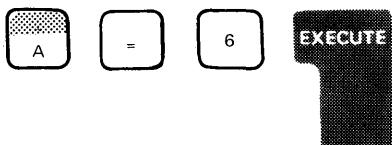
Here are the general rules:

- A character variable is named by a single letter of the extended BASIC alphabet followed by a dollar sign (\$).
- To assign a value to the character variable, enclose the words or names you are assigning in single quotation marks following the equal sign.
- The character limit is 18. If you require more than 18 characters, assign the excess to a second variable. If you enter more than 18 characters for a single variable, the excess is ignored.

## Using Calculation Results

When you are entering a series of expressions in which the result from one expression is used in the next expression, you can use the 

key (while holding down the CMD key) to insert the result of the last expression. This is the calc result function. Notice that Calc Result is listed above the  key. For example, key the following:



The display screen shows:

A=6

6

B=4

4

C=9

9

A\*C

54

B\*( -54)

216

Notice that the 5100 inserted the result of A\*C into the second expression.

The calc result function will always insert the result of the *last* calculator expression (character expressions as well as arithmetic expressions). You must hold down the CMD key while you press  for the calc result

function. Arithmetic results are enclosed in parentheses to avoid any conflict with adjacent operators in case the result is negative. Character results are enclosed in single quotation marks, just as they were when they were assigned.

Up to this point, you have been operating the 5100 as a calculator. Any of the operations described thus far can be performed any time the 5100 is waiting for you to enter input, with two exceptions:

1. When a BASIC program is in operation and it stops for keyboard input required by an INPUT statement, you cannot perform any calculator operations.
2. When you are entering data to create a keyboard-generated data file, you cannot perform any calculator operations.

Any time, other than the exceptions listed, you can enter any of the calculations described.

You can also stop a program during its operation, and change the values of variables, then continue the program. This is extremely useful when checking a program for proper operation.

The following chapters discuss how to program your 5100 using the BASIC language.

## Chapter 3. Entering, Running, and Storing a Program

A program is your way to communicate with the 5100 to solve a problem. The key words in this statement are *communicate* and *to solve a problem*. All programming is oriented toward problem solving. Problems can only be solved by first analyzing the problem, then by formulating the solution. This involves communication. You can communicate with the 5100 using the BASIC or APL language, as opposed to your communicating with other people in the English language. Thus, a program is little more than a means of translating your instructions to solve a problem into a language that the 5100 understands.

### ENTERING A PROGRAM

The following discussion shows you how to enter a BASIC program into the 5100, and then how to execute that program. Also in this chapter, you will learn how to save a program on a magnetic tape cartridge, then load the program back into the 5100 for execution again.

The program you will enter calculates the volume of a cylinder. The volume of a cylinder is found by multiplying the length of the cylinder times the area of the base. Enter the statements just as they are shown in the following example. Don't forget to press EXECUTE after entering each statement. You can enter the statements character-by-character or use the BASIC statement keyword keys with the CMD key. If the 5100 detects an error in a statement you have entered, the keyboard becomes inactive (except for ATTN and HOLD), and the display will flash. To stop the flashing display, press ATTN, then correct the error.

```
0010 REM CYLINDER VOLUME
0020 PRINT 'DIAMETER?'
0030 INPUT D
0040 IF D=0 GOTO 0110
0050 PRINT 'LENGTH?'
0060 INPUT L
0070 A=&PI*(D/2)^2
0080 V=A*L
0090 PRINT 'THE VOLUME IS ',V
0100 GOTO 0020
0110 END
```

Now read your entries on the display screen to see if you have entered the program correctly. If you find a keying error, the next paragraph describes how to correct the error before you run the program. If your 5100 has an attached printer, you can hold down the CMD key and press the  key below Copy Display to get a copy of the displayed

data. The copy display function provides you with printed copy of all 16 lines of data.

### Correcting Your Keying Errors

To correct or change a statement line of a program already entered in the 5100, use the gray scroll keys (  and  ) to position the

incorrect line to be changed on the input line right above the READY message line. When pressed momentarily, these keys move all information on the top 15 lines up or down one line position. When you hold these keys down, the display lines will repeatedly move up or down. When the line you want to change is positioned correctly, which is easy to identify because the first character will be flashing, you can use the forward space or backspace key to position the cursor at the character to be corrected. You can then use the insert and delete functions to make the change. Remember, these functions are activated only when you hold down the CMD key and press  (Insert) or  (Delete). After all changes

have been made to the line, press EXECUTE to reenter the line.

### RUNNING THE PROGRAM

After you have entered the statement lines of the sample program, you are ready to run the program. To run the program, enter RUN, then press EXECUTE. Any error during execution causes the display to flash; the ATTN key must then be pressed. Press ATTN to stop the flashing screen, then correct the error. You will have to enter RUN again to execute the program. When you run the program, the display screen shows:

RUN  
DIAMETER?

?

You will recognize the prompting message DIAMETER? as part of the second statement in the sample program. This is a PRINT statement, which directs information to be displayed.

Note that the bottom question mark is flashing. The flashing question mark is a result of the INPUT statement in the sample program. The INPUT statement causes the question mark to be flashed to indicate that you are to enter information from the keyboard for the program.

Now respond to the request for data to be entered by keying a value for the diameter, then press EXECUTE. You can enter any number of digits you want. The maximum number of digits that can be assigned to any variable (your variable is D for diameter) is 13 digits. You can include a decimal point, which does not count as a digit entry, but you must not enter commas. (Commas indicate multiple variables to the 5100.)

If you enter a decimal number with more than six digits to the right of the decimal point, any digits beyond the sixth are rounded when the answer for volume is displayed. The 5100 is initialized to round numbers at the sixth decimal position. However, the rounding position can be changed to any position from 1 to 13 with the RD= command, which sets the rounding position. To set the 5100 to round all displayed or printed results and calculations at the second decimal position, you would enter: R D = 2.

The rounding command can also be included with the GO and RUN commands as described in the *IBM 5100 BASIC Reference Manual*, SA21-9217. Remember that whenever you turn the power on or press RESTART, the rounding position is set at 6.

All examples in this manual are run with the rounding position set to 6 digits (RD=6). If you change the rounding position, you will get different results.

You must remember that when using any programming language, including BASIC, you are communicating with the machine, telling it what you want it to do. Thus, you should define precisely what the machine does not know to avoid unnecessary problems.

You can enter values for the cylinder volume program as many times as you want. After you enter a value for the length in response to the flashing question mark and press EXECUTE, the 5100 will display the information you specified and compute the answer.

The statements in the sample program are described in the following paragraphs. In addition, Appendix A contains a short definition of all the BASIC statements used in the 5100.

Statement	Meaning
10 REM CYLINDER VOLUME	The REM (remark) statement can appear anywhere in the program, but has no effect on program execution. This statement is used to insert comments into the BASIC program.
20 PRINT 'DIAMETER?'	This PRINT statement specifies that DIAMETER? be displayed. The single quotation marks around DIAMETER? indicate that it is a character constant and that the entire character string is to be displayed.
30 INPUT D	The INPUT statement allows you to assign values from the keyboard to variables when your program is running. In this example, the variable D will receive the value you enter. The 5100 displays a question mark in position 1 of the input line to indicate that keyboard input is expected.
40 IF D=0 GOTO 110	The IF statement transfers control to a specified statement when a specified condition is met. In this statement, the program will terminate when you enter 0 for the diameter. As long as you want to calculate volumes, you can enter values for the diameter and length. When you are finished, however, just enter 0 for diameter and the program goes to statement 110 (END).

<b>Statement</b>	<b>Meaning</b>
<b>50 PRINT 'LENGTH?'</b>	This statement displays the character string LENGTH?.
<b>60 INPUT L</b>	This statement specifies that the variable L will receive the value you enter for length. Again, a question mark will be displayed to indicate that keyboard input is expected.
<b>70 A=&amp;PI*(D/2)<math>\uparrow</math>2</b>	This is an arithmetic expression indicating that the variable A will receive the value of $\pi$ times D (your entry for diameter) divided by 2, and raised to the power of 2.
<b>80 V=A*L</b>	This expression assigns a value to a variable. In this statement, the variable V will receive the value of the variable A times the variable L (your entry for length).
<b>90 PRINT 'THE VOLUME IS', V</b>	This statement indicates that the characters enclosed in single quotation marks (THE VOLUME IS) are to be displayed, followed by the value of the variable V. The value of V was calculated in the preceding statement and is the volume.
<b>100 GOTO 20</b>	The GOTO statement transfers control to a specified statement. In this statement, control is transferred to statement 20. This provides for a number of volume calculations to be made repetitively.
<b>110 END</b>	The END statement indicates the end of execution of a BASIC program and terminates operations.

After you have computed your last volume calculation, you can end the program operation by entering 0 for the requested diameter and pressing EXECUTE.

The numbers preceding the statements are called statement numbers. They are necessary so the 5100 knows the proper sequence of your instructions. BASIC statement numbers in the 5100 can have values from 0001 through 9999. You can use consecutive numbers if you wish, but normally expansion room is left between the statement numbers so that changes can be made more easily (see *Making Changes to Your Program*). When you enter statement numbers, you do not need to include the leading zeros. They will be added by the 5100 when the statement is entered.

### Automatic Statement Numbering

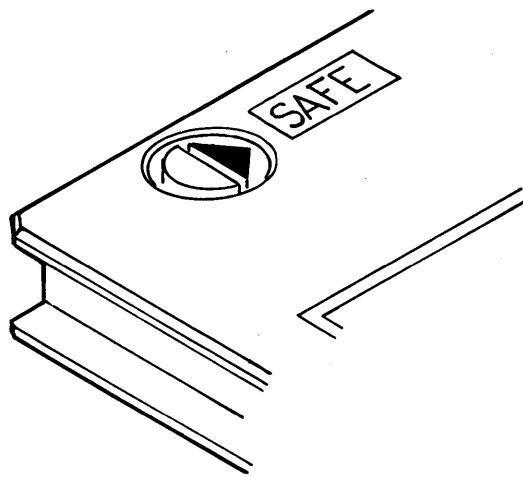
Instead of manually entering the statement numbers in a BASIC program, you can instruct the 5100 to provide statement numbers for you. You can do this with the AUTO command. Simply enter AUTO and press EXECUTE. Notice that the word AUTO is displayed above statement number 0010. From this point on, the 5100 numbers your statements in increments of 10. Automatic numbering continues until you enter anything other than the last statement number (a command word, for example, LIST, in the input line). You can restore automatic numbering by entering AUTO NNNN where NNNN is the statement number you want to begin with.

## STORING THE PROGRAM

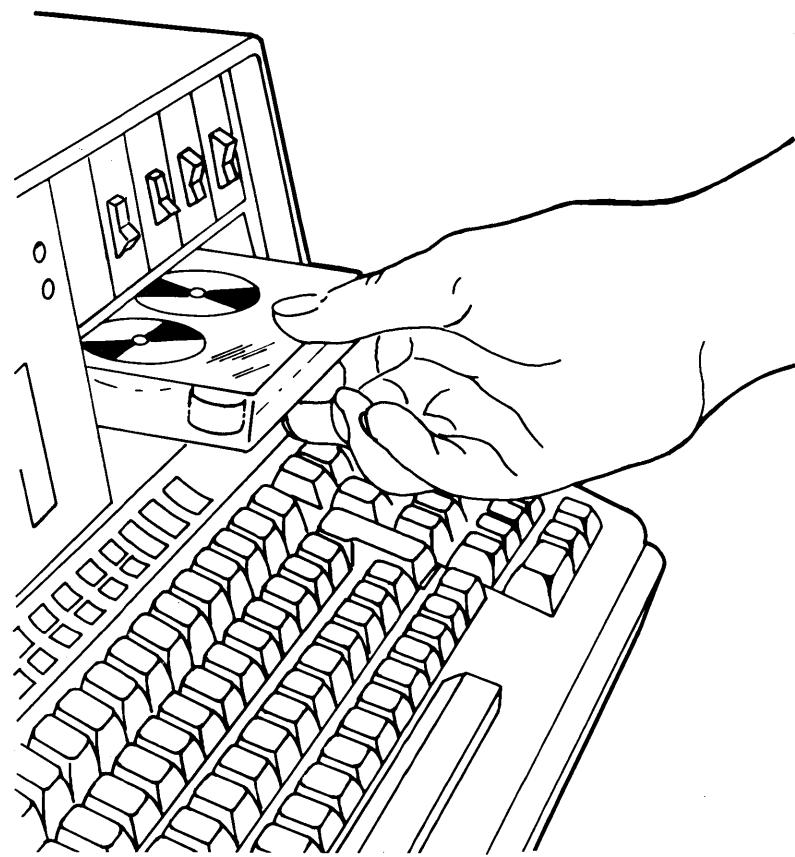
The 5100 magnetic tape cartridge allows you to conveniently store your programs (or data) and have them available for use by following the simple operations described in this section. Before using a tape cartridge, check the tape cartridge security arrow in the corner of the cartridge. Figure 6 shows the arrow pointing to SAFE. When the arrow is in this position, *the tape cannot be written on*. To be able to write on the tape, use a screw driver or a coin to turn the arrow away from SAFE. Figure 7 shows how a magnetic tape cartridge is loaded into the 5100. Press the cartridge in until it is firmly seated.

### Tape Preparations

Before you can use it for programming or data operations, the magnetic tape cartridge must be prepared, which consists of marking the tape to define how much space is to be in each tape *file*. When used in relation to the BASIC 5100, a *file* is the area on a tape that contains one program or a collection of related data items.



**Figure 6. SAFE Arrow**

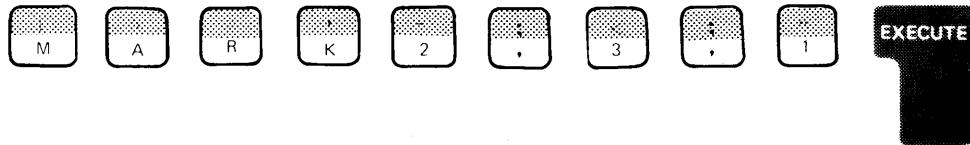


**Figure 7. Inserting a Cartridge**

To prepare a tape with one file of any size, or several files of the same size in the same operation, you enter a MARK command. The MARK command can be entered anytime that READY is indicated on the display screen. It will not interfere with your programs. For our exercise, we are assuming that you are beginning a new tape (no files have been marked). We will mark three files, each containing 2048 or 2K character positions (K is equal to 1024). This exercise provides enough space on tape to contain three programs (one in each file) of approximately 35 statements each. Tape files for the 5100 are numbered sequentially beginning with 1. Should you later decide to add more files, you can do so as long as you do not exceed the physical limits of the tape. A tape contains space for approximately 200K of storage, minus the leading and trailing data for each file, which equals 0.5K per file. Thus, a tape can contain approximately 132 1K files, 44 4K files, or any combination of file sizes up to 200K, including the required 0.5K per file.

*Note:* If you are not using a new tape cartridge, you must first ensure that your tape does not contain important data belonging to someone else. This is necessary because any existing data is erased when you remark the tape. The 5100 displays an error message when you attempt a MARK command to a file that is already marked. To continue marking the file, press ATTN to stop the flashing screen, press the scroll up key once to move the display up one line, enter GO in positions 1 and 2, then press EXECUTE.

To mark the tape in our exercise, press the following keys:



You have now marked the tape for three files of 2K characters each, starting with file 1. The READY message is displayed when the tape preparation is completed. We will now save the cylinder volume program on the tape. If you want to mark additional files, remember that you must begin with file number 4.

## **SAVE Command**

The cylinder volume program can be saved on tape with the SAVE command. To save the sample program, enter SAVE, then enter the number of the file you want to save it in. We will save the program in file 1, so press 1, then press EXECUTE. The READY message will be displayed to tell you when the program is saved on tape. (You needn't be concerned with the numbers following the READY message.) While the program is being saved, you will notice the tape in the cartridge moving back and forth. This is normal, because the 5100 is reading each segment of data after it is written. This ensures that the information is saved correctly.

To prove that the program has been stored on tape and that you can load it back into storage, the program stored in the machine must first be erased.

There are three ways to do this:

1. Enter LOAD0 and press EXECUTE. This clears machine storage and prepares it to accept input from the keyboard or programs loaded from tape. This is the recommended way to clear machine storage.
2. Press the RESTART switch. This restarts the machine to the same status as when the power was turned on. The internal diagnostics are performed again; thus, this method requires 10-15 seconds depending on the amount of storage in your machine. This method is recommended only when the PROCESS CHECK indicator comes on, or when you change from BASIC to APL or APL to BASIC.
3. Set the power switch to OFF, then set it back to ON. The same diagnostics are performed as during RESTART.

To clear the machine, enter LOAD0 and press EXECUTE. To prove the program no longer exists in the machine, enter RUN and press EXECUTE. The 5100 will respond with an error message to let you know this cannot be done because there is no program in storage. Press ATTN to continue.

In order to run the program again, it must first be loaded into storage from where it was stored on the tape. The LOAD command is used to place the program back into storage.

## LOAD Command

To load the cylinder volume program back into 5100 storage, enter LOAD, then enter the number of the file containing the program you want to load (file 1). Complete this sequence by pressing EXECUTE.

The READY message tells you that the program is loaded and can be executed again. Run the program again by entering RUN, then pressing EXECUTE.

Practice the SAVE and LOAD commands by changing the file number when you again save the sample program on tape and load it back into the 5100.

The following commands have been discussed in this chapter:

- AUTO — Automatically numbers BASIC statements.
- MARK — Prepares a tape cartridge for data to be saved.
- LOAD — Loads the 5100 storage with data from tape or data from the keyboard.
- SAVE — Saves the BASIC program in 5100 storage on tape.
- RD= — Specifies rounding of decimal numbers.
- RUN — Executes a BASIC program.

## A REVIEW OF WHAT YOU'VE DONE

After reading this far and doing the exercises described, you should be able to perform the following functions with your 5100:

- Use as a calculator:
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Exponentiation
  - Use of positive/negative operators
  - Use of parentheses in arithmetic hierarchy
  - Use of arithmetic constants

- Correct keying errors:
  - Replace a character
  - Delete a character
  - Insert a character
  - Make corrections in a line
- Use variables:
  - Assign values to variables
  - Display variable values
- Enter short, simple programs:
  - Change program lines
  - Run programs
  - Erase programs
  - Store programs on tape
  - Load programs from tape
- Clear the machine storage

In the following pages, you are going to write more BASIC programs and learn to use some fundamental tools for writing programs. From this point on in the manual we will not show you the keys to press. We will just say to enter and then give you the data you should enter.

### The LET Statement

A LET statement consists of four parts: a statement number, a symbol to the left of an equal sign, an equal sign, and a quantity or a computation (called an *expression*) to the right of the equal sign. In BASIC programming, a LET statement means:

1. To evaluate the expression on the right side of the equal sign, and
2. Assign that value to the symbol on the left side of the equal sign.

In BASIC, you can have statements such as

0030 LET X=X+1

while you couldn't in math. In BASIC this statement means to take whatever value X now has, add 1 to it, and replace the old value of X with this new value.

Incidentally, you can omit the word LET from a LET statement in a program. These two statements

0010 LET X=A+B

0010 X=A+B

mean exactly the same thing. In all our examples, we'll show the word LET, but it's not necessary to include it.

The following program uses simple arithmetic. Try to look at the program as a step-by-step method for solving a particular problem.

### Problem

Last month you went to the dentist and had an examination and X-rays. That cost \$25. You had two teeth filled. That cost \$24. Your insurance will pay for 75% of everything over \$15. How much do you have to pay, and how much does the insurance pay?

### *What to Do*

1. Find the total dentist bill (call it D).
2. Subtract \$15 to find the amount eligible for insurance (call it E).
3. Take 75% of the result (call it I). That's how much the insurance pays.
4. Subtract the insurance money from the total bill D to find out how much money you owe (call this M).
5. Display how much you have to pay and how much the insurance will pay (M and I).

The following BASIC statements, which you will enter later, can be used to solve this problem:

```
0010 LET D=25+24  
0020 LET E=D-15  
0030 LET I=.75*E  
0040 LET M=D-I  
0050 PRINT M,I  
0060 END
```

Notice the PRINT statement. Since you want to know both your payment and the insurance payment, you can specify both M and I in the same statement. Any time you want to display the value of more than one variable, you can use a single PRINT statement if you list the variables and separate them with commas.

### **USING REMARKS**

You can make your programs easier to work with, and easier for other people to use, if you include descriptions of what the statements do in the program. These descriptions are known as *remark statements*. You write remarks as if they were statements in the BASIC program, but they don't serve any function in the execution of the program. They are solely for information. You can insert them anywhere in a program.

To include a remark in a program, you write a BASIC statement called REM. It has a line number like any other BASIC statement. Following the line number, you enter the letters REM followed by any remark you want. Examples of REM statements are:

40 REM THIS PROGRAM COMPUTES BATTING AVERAGES

70 REM AT THIS POINT, PRINT OUT THE RESULTS

10 REM DENTBILL

You'll see other examples of REM statements as you go through this manual.

You should now be ready to enter the program from the keyboard of your 5100. To enter and execute the program, follow the instructions below. Remember to press EXECUTE after each line is entered.

Instructions	Display Screen Shows
Clear storage	LOAD 0
Enter the statements	0010 REM DENTBILL 0020 LET D=25+24 0030 LET E=D-15 0040 LET I=.75*E 0050 LET M=D-I 0060 PRINT M,I 0070 END
Run the program	RUN 23.5                    25.5

Remember that the idea in this program, and any other programs you write, is to break down what you want to do into logical sequential steps. It may help to use this tactic: Ask yourself what is the very first thing I have to do? What is the next? And so on. You can make a list of what you have to do and then convert each item in the list to a BASIC statement. Thus, you will always have a sequence of statements that will solve the problem in an orderly step-by-step fashion.

The dentbill program is not a typical program because it works on only one set of data that is a part of the program. Most programs are written to use many sets of data and use data that is not a direct part of the program. Later program examples will explain this in detail.

## **LISTING PROGRAM CONTENTS**

Since the dentbill program is currently in the 5100 work area, we can now list the statements of the program on the display screen with the LIST system command. Enter LIST, then press EXECUTE. The statements of the dentbill program now appear on the display screen.

A LIST command causes the first 14 lines of the program to be displayed. You can then use the scroll up and down keys to view additional lines of the program.

It isn't necessary to list the entire program each time you want to see a particular part of it. You can list any 14-line portion of the program by entering the last line number you want displayed after the command keyword LIST. For example, LIST 30 would display statement numbers 10, 20, and 30 of the dentbill program.

## **BRANCHES**

The 5100 normally executes programs line by line according to the line numbers of the statements. However, you can vary this sequential order and transfer control to a line number other than the next sequential one. This is called *branching*. Two of the statements you can use for branching are the GOTO and IF statements.

### **The GOTO Statement**

This statement tells the 5100 to go to a specific line number. A GOTO statement at line 20 of a program that tells the 5100 to go to line number 60 would look like this:

0020 GOTO 0060

## The IF Statement

An IF statement can test whether a variable is equal to, greater than, or less than another variable or constant of the same data type. The IF statement includes a GOTO statement. The IF statement operates this way:

1. The IF statement tests the condition you define.
2. If the answer to the test is *yes*, the condition is true; the 5100 will go to the line number that you entered in the IF statement.
3. If the answer is *no*, the 5100 ignores the rest of the IF statement and goes directly to the next sequential line in the program.

Here's an example of an IF statement:

0040 IF X=0 GOTO 0080

In this statement, if X is 0, the 5100 goes to line 80. If X is *not* 0, it goes on to the next line in the program.

The six tests you can make with the IF statement are:

1. Equal to, =
2. Not equal to, ≠ or <>
3. Greater than, >
4. Less than, <
5. Greater than or equal to, ≥ or >=
6. Less than or equal to, ≤ or <=

The 5100 stores <> as ≠, > = as ≥, and < = as ≤; thus, even though you enter <>, the 5100 will display ≠ when you list the statements.

Some examples of IF statements are:

This IF Statement:	Means:
0130 IF X>10 GOTO 0040	If the value of X is greater than 10, go to line 40.
0190 IF Y<21 GOTO 0010	If the value of Y is less than 21, go to line 10.
0010 IF A1 $\geq$ 5 GOTO 0060	If the value of A1 is greater than or equal to 5, go to line 60.
0030 IF A2 $\neq$ X GOTO 0075	If the value of A2 is not equal to the value of X, go to line 75.

The following program examples describe more about how to break down a problem into the BASIC statements required to use your 5100 to solve a problem. Again, as opposed to most typical programs, the sample programs will use data internal to the programs. After you've seen how data within a program can be manipulated, you'll be shown how to supply program data from outside the program.

Although it may not be necessary in all instances, it is a good idea to enter a LOAD0 command before entering any program statements. This ensures that the 5100 work area is clear. Remember also that you can use the AUTO command to provide automatic statement numbering.

#### *Program Example 1*

You are in charge of billing people for orders of dresses. There are two styles, one at \$108 a dozen, and one at \$136 a dozen. On orders of \$500 or over, there is a 10% discount. For the account you are now working on, there are two dozen orders for the first dress, and three dozen orders for the second dress.

The program to determine the bill is:

```
0010 REM PROGRAM TO FIGURE OUT DISCOUNTS ON ORDERS
0020 LET A=2
0030 LET B=3
0040 LET T=A*108+B*136
0050 IF T<500 GOTO 0070
0060 LET D=.1*T
0070 PRINT T,D,T-D
0080 END
```

This program solves the problem in the following steps:

1. It finds the total order (line 40).
2. It tests to see if the total is less than \$500 (line 50). If it is, the program goes to line 70 and displays the total. The discount D will be 0 in this case, and the totals will be displayed.
3. For orders of \$500 or over, the program computes the 10% discount on line 60. Then it continues to line 70 to display the total.

*Note:* D will be 0 when the order is less than \$500 because each time the 5100 starts to execute a program after a RUN command, it automatically sets the value of all the variables in the program to 0. Character variables are set to blanks. This is called *initialization*. The values remain zeros or blanks until a statement in the program assigns a different value. This means that you never have any problem with values being left over from the last time you ran the program. Variables specifically stored in a reserved area of storage, however, retain the last value assigned to them. These values can be passed from one program to another (see *USE* in the *IBM 5100 BASIC Reference Manual*, SA21-9217).

Upon execution of this program, the display screen shows:

624

62.4

561.6

Thus, the total order is \$624, the allowable discount is \$62.40, and the amount to be billed is \$561.60. The cents columns in the dollar figures do not print because the 5100 has no way of determining how many significant digits you want printed. You will be shown how to have numbers printed in the exact format you want in a later chapter.

#### *Program Example 2*

You are moving. You get estimates from two movers and want to know which mover will be cheaper. Mover A charges \$40 an hour and estimates the work will take 5 hours. Mover B charges \$32.50 an hour

and estimates the work will take 8 hours. If both movers cost the same, you'll hire mover B because he has a better reputation. Here is the program:

```
0010 LET A=5*40
0020 LET B=8*32.5
0030 REM TEST TO SEE WHO IS CHEAPER
0040 IF A<B GOTO 0100
0050 REM GO HERE IF B CHEAPER
0060 PRINT 'B CHEAPER OR EQUAL'
0070 PRINT B
0080 STOP
0090 REM GO HERE IF A CHEAPER
0100 PRINT 'A CHEAPER'
0110 PRINT A
0120 END
```

Here is how this program works:

1. In lines 10 and 20, it figures the total cost for each mover.
2. In line 40, it tests to determine which one of two paths to take. Either the program will go to line 100, or it will continue with lines 50, 60, 70, and 80. This test determines which mover is cheaper.
3. If Mover A is cheaper, the program goes to line 100. Line 100 lets you know that Mover A has the contract and displays the total price. Notice line 100. It is a PRINT statement, but it has single quotation marks around the words A CHEAPER. You can write a PRINT statement that displays the words entered if you enclose the words in single quotation marks. If line 100 is executed, the words A CHEAPER will be displayed. Line 110 has no quotation marks. It is a PRINT statement for variable A, and will display the value of variable A. After the PRINT statements, the program ends at line 120.
4. If Mover B is cheaper, the program continues with line 50. Line 60 is a PRINT statement containing the words B CHEAPER OR EQUAL in single quotation marks. If line 60 is executed, the words B CHEAPER OR EQUAL will be displayed. Line 70 will display the value of variable B. After the PRINT statements, the program comes to line 80, a STOP statement, which ends the program.

After you run the program, the display screen shows:

RUN  
A CHEAPER  
200

### Loops

Here is a new problem. You are a rug salesman. All your rugs come in rolls 12 feet wide. Your customers buy rugs in varying lengths depending on how long their rooms are. You want to make a chart of how many square yards of rug are required for rooms of different lengths. The most popular room sizes start at 9 feet long (a 12 by 9 foot rug) and increase a foot at a time (12 by 10, 12 by 11, and so on) until they reach 12 by 20.

You will write a program that computes the number of yards in a 12 by 9 rug, then a 12 by 10 rug, on up to a 12 by 20 rug.

To compute the number of square yards in each of the 12 different sized rugs, you have to find the number of square feet and divide by 9. The main computation step is:

0020 LET Y=(12\*X)/9

where Y is the number of square yards, and X is the length of each different rug. To display the value of Y, you would use this statement:

0030 PRINT Y

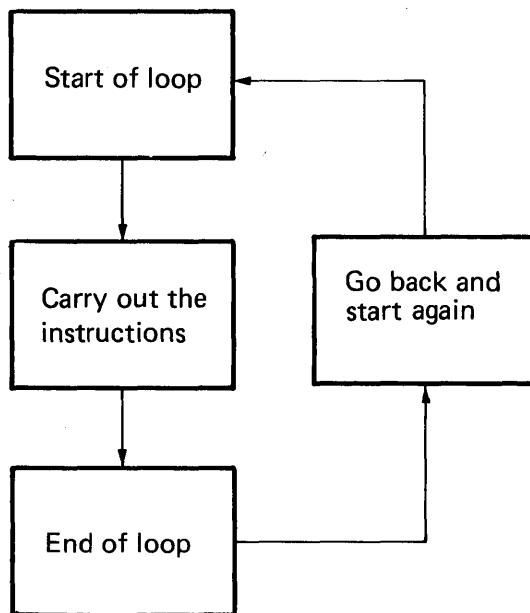
The value of X has to increase by 1, from 9 to 10 on up to 20. You could write a program like this:

0010 LET X=9  
0020 LET Y=(12\*X)/9  
0030 PRINT Y  
0040 LET X=10  
0050 LET Y=(12\*X)/9  
0060 PRINT Y

:

and so on until X=20

This program uses a LET statement to increase the value of X. However, there is a better way. You can make a loop. A loop is just what it sounds like. It is a series of program steps that are repeated. It looks like this:



Two things have to happen to this loop to make it work. It has to have some way to change the values it uses before it loops up to the top and starts again. And it has to have some way to know when to stop, or the program will run indefinitely.

#### First Loop Method

So far you have these program statements:

```
0020 LET Y=(12*X)/9  
0030 PRINT Y
```

You want to start with X equal to 9, so put a LET statement ahead of these two statements assigning 9 as the first value of X. It will also help if you print the value of X with the computed value of Y, so the table will be more self-explanatory. Now, the program looks like this:

```
0010 LET X=9  
0020 LET Y=(12*X)/9  
0030 PRINT X,Y
```

To avoid specifying X=10, X=11, and so on, write a general statement that will keep increasing the value of X by 1. That statement is:

```
0040 LET X=X+1
```

O Remember that while this statement looks peculiar in a mathematical sense, it's perfectly valid in BASIC. It says, "Assign the value of X to be equal to the old value of X plus 1".

C By adding statement 40 to the program, you've changed the value of X and completed the steps required to make the loop operate once. Now you have to add a GOTO statement to go back to the beginning of the loop:

0050 GOTO 20

C You go to line 20 because you only have to go back to the computation step, not to line 10 where you originally set X equal to 9.

C After the 5100 goes to line 20, it computes and displays the yardage again, but this time for X equal to 10. It arrives at line 40 again and changes X to 11; then it goes back to line 20 to compute the next yardage. This process continues, increasing the value of X by 1 after each loop.

#### Ending a Loop

C One thing is missing from an otherwise perfect loop. It never ends. Not at X=20, not at X=30, because X just keeps increasing. If you are sitting in front of your 5100 while this program is running, *you can stop this loop whenever you want to by pressing ATTN*. But this is obviously not an ideal method. You can make the loop stop automatically if you build in a test with an IF statement to see when you've processed enough values of X. In this program, you want the loop to stop when the value of X passes 20. Consider this IF statement:

0050 IF X>20 GOTO 70

C Line 70 will be an END statement.

C The IF test goes *before* the GOTO statement that branches to line 20. If you put it after the GOTO statement, it will never be executed. This is the finished program:

0010 LET X=9  
0020 LET Y=(12\*X)/9  
0030 PRINT X,Y  
0040 LET X=X+1  
0050 IF X>20 GOTO 0070  
0060 GOTO 0020  
0070 END

Looking at this program, you should be able to see that lines 50 and 60 can be combined to make a more efficient program that looks like this:

```
0010 X=9  
0020 Y=(12*X)/9  
0030 PRINT X,Y  
0040 X=X+1  
0050 IF X≤20 GOTO 0020  
0060 END
```

Now, enter and run the program. After you run the program, the display screen shows:

RUN	
9	12
10	13.333333
11	14.666667
12	16
13	17.333333
14	18.666667
15	20
16	21.333333
17	22.666667
18	24
19	25.333333
20	26.666667

You can press HOLD to stop the upward movement of the data. To continue, press HOLD a second time.

There is another way to make a loop in a program. At the beginning of the loop, instead of setting X equal to its first value, you enter the entire range of values that X will use. In the rug example, you would write

```
0010 FOR X=9 TO 20
```

Then you write the statements that solve the problem and print the results:

```
0020 LET Y=(12*X)/9  
0030 PRINT X,Y
```

Then you tell the 5100 to go to the next value of X and repeat the loop:

0040 NEXT X

FOR and NEXT statements always go in pairs: FOR at the beginning of the loop and NEXT at the end. The 5100 automatically repeats the loop as many times as you told it to in the FOR statement. When it finishes, it goes on to the statement following the NEXT statement.

Using the FOR and NEXT statements, the rug program looks like this:

```
0010 FOR X=9 TO 20  
0020 LET Y=(12*X)/9  
0030 PRINT X,Y  
0040 NEXT X  
0050 END
```

In a FOR statement, you can name any arithmetic variable to be the control variable, and you can make its range of values anything you want. The control variable is to the left of the equal sign. The range (to the right of the equal sign) doesn't have to be given in numbers. You can use other variables for the range, for example:

```
0060 FOR J=A TO B  
:  
0120 NEXT J
```

#### Steps

When you write a FOR statement, the 5100 increases the value in steps of 1 (for example, 1 to 2 to 3, or 18 to 19 to 20 to 21). However, sometimes you may want to use just even numbers, or odd numbers, or every tenth number. If your loop requires a value other than steps of 1, you can specify the step value whether you are using FOR and NEXT statements or a LET statement to control the loop.

If you write a loop that uses a LET statement, you can write these LET statements:

0100 LET X=X+2      To change X in steps of 2

0050 LET X=X+10      To change X in steps of 10

If you're using FOR and NEXT statements for the loop, you add the word STEP and the size of the step to the FOR statement. For example:

0010 FOR X=1 TO 25 STEP 2

gives you odd values of X from 1 to 25 (1,3,5,7. . .).

0030 FOR D=10 TO 100 STEP 10

gives you 10, 20, 30, up to 100.

For even values of D from 1 to 20, you would write:

0020 FOR D=2 TO 20 STEP 2

Notice that D is set to 2 because the first even number you want is 2.

If you omit the word STEP and the value from the FOR statement, you automatically get steps of 1. You can also include fractional steps, for example:

0030 FOR I=1 TO 3 STEP .1

#### Loops Within Loops

Here's a problem where two values change: Find the annual amount of interest (A) at the interest rates (I) of 5%, 6%, 7%, 8%, 9%, and 10% on principals (P) ranging from \$100 to \$1000 in steps of \$100.

This problem can be solved by a program that uses two loops—one for changing the interest and one for changing the principal. Do the interest loop first:

```
FOR I=5 to 10
    LET A=(I/100)*P
    PRINT P, I, A
NEXT I
```

This computes  
and displays A,  
P, and I.

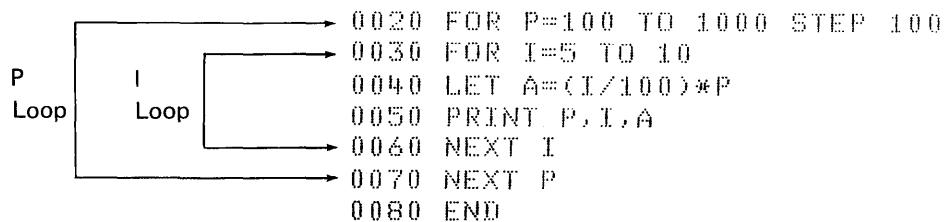
This creates a  
loop for I to vary  
from 5% to 10%.

Now all that's left is to define P, since the program doesn't know where to find the values for P. The P loop has no computations of its own; it only defines the values for P:

FOR P=100 TO 1000 STEP 100

NEXT P

The P loop goes *around* the I loop:



You must put one loop entirely inside the other so that the 5100 will stay in one loop and finish it completely (compute *all* the values for I for a single value of P) before it goes on to the next value of P. In this program, the 5100 starts with P equal to \$100, then it comes to the I loop and sets I equal to 5%. It goes on to compute the interest on \$100 at 5%, 6%, 7%, 8%, 9%, and 10% because it keeps repeating the I loop until I equals 10. When it finishes all the different interests on \$100, it goes to line 70, which is the bottom of the P loop. Here, control loops back to line 20, which increases P to \$200, and starts on the I loop again, this time with P equal to \$200 and with I again ranging from 5% to 10%. The program continues in these loops until all of the values of P have been used. Then you have all the amounts of interest you wanted.

To run the program:

When the display screen shows

READY, enter the statements:

```
LOAD 0
AUTO
0010 REM INTEREST
0020 FOR P=100 TO 1000 STEP 100
0030 FOR I=5 TO 10
0040 LET A=(I/100)*P
0050 PRINT P,I,A
0060 NEXT I
0070 NEXT P
0080 END
```

Run the program

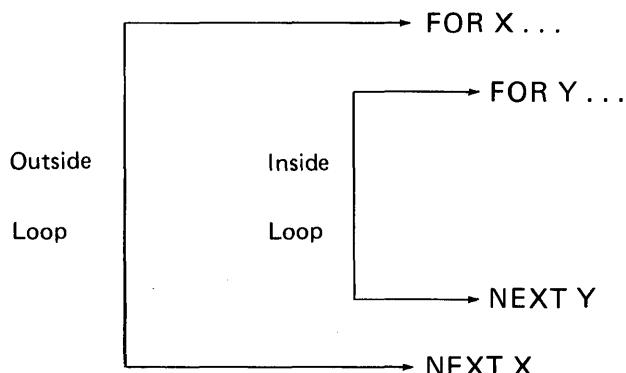
RUN

To see a portion of the program results, press HOLD. To continue execution, press HOLD again.

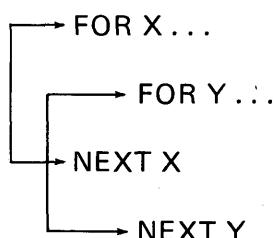
The display screen shows a portion of the output when you pressed HOLD. For example:

800	10	80
900	5	45
900	6	54
900	7	63
900	8	72
900	9	81
900	10	90
1000	5	50
1000	6	60
1000	7	70
1000	8	80
1000	9	90
1000	10	100

Loops within loops must always be nested like this:



so that the inner loop is fully completed each time before the outside one is begun again. Two loops must never overlap like this:



Remember, one loop must always be completely enclosed by the other.

Here is another example of loops. This is a program to find  $X^2$ ,  $X^3$ ,  $X^4$ , and  $X^5$  with X equal to 1 to 10.

The outside  
loop changes  
X from 1 to 10.

The inside loop  
changes Y from  
2 to 5.

```
0010 REM POWERS
0020 FOR X=1 TO 10
0030 FOR Y=2 TO 5
0040 PRINT X,X↑Y
0050 NEXT Y
0060 NEXT X
0070 END
```

Notice that the inside Y loop is fully contained in the outside X loop.  
Run the program as shown:

When the display screen shows  
READY, enter the statements:

```
0010 REM POWERS
0020 FOR X=1 TO 10
0030 FOR Y=2 TO 5
0040 PRINT X,X↑Y
0050 NEXT Y
0060 NEXT X
0070 END
```

Run the program

RUN

The display screen shows  
(use the HOLD key to display  
any 15-line portion of the  
displayed results):

7	16807
8	64
8	512
8	4096
8	32768
9	81
9	729
9	6561
9	59049
10	100
10	1000
10	10000
10	100000

## Chapter 5. Other Ways to Put Values into Programs

In all the programs we've written, we've tried to:

- Write a program to solve the problem using general expressions.
- Supply specific values for the expressions and run the program with the specific values.

The advantage of programming in this way is that the bulk of the program doesn't change every time you want to solve the same problem with different numbers. You only need to change the numbers, not the programmed expression, when you want to run the program using different numbers.

We are now going to look at other ways to supply specific numbers for programs.

### THE READ, DATA, AND RESTORE STATEMENTS

To assign 10 values, say the numbers 1 through 10, to 10 variables, you could use 10 LET statements:

```
0010 LET A=1  
0020 LET B=2  
0030 LET C=3  
:  
0100 LET J=10
```

Using 10 LET statements can be tedious. Another way to enter these numbers is with one DATA statement:

```
0200 DATA 1,2,3,4,5,6,7,8,9,10
```

The DATA statement causes values to be placed in an internal data table. You can use one or several DATA statements to do this. Values in DATA statements are put into the data table sequentially, in the order in which they are entered. The values must be separated by commas. The following set of statements would have the same effect as the single preceding DATA statement:

```
0200 DATA 1,2,3  
0210 DATA 4,5,6  
0220 DATA 7,8,9,10
```

Once the values are in the table, you use the READ statement to assign them to variables. Here's an example:

0200 DATA 1,2,3,4,5,6,7,8,9,10  
0210 READ A,B,C,D,E,F,G,H,I,J

The READ statement locates the values in the data table and assigns them (in order) to the variables—the value 1 to the variable A, 2 to B, 3 to C, and so on.

You don't have to assign all of the values in the data table at one time. For example:

0200 DATA 1,2,3,4,5,6,7,8,9,10  
0210 READ A,B,C

will cause the first three values in the table to be assigned to A, B, and C, respectively. Another READ statement will take up where the last one left off. Thus:

0420 READ D,E,F,G

will assign the values 4, 5, 6, and 7 to D, E, F, and G, respectively.

You must be careful, though, not to try to read more values than the table contains. For example, still another READ statement:

0440 READ H,I,J,K

would be requesting values for four variables when only three numbers (8, 9, and 10) are left in the table. This will cause an error.

If you want, you can use the values in the data table more than once. At any point in your program, you can instruct that values be assigned from the beginning of the table again, even if you haven't read all the values in the table. To go back to the beginning of the table, use the RESTORE statement:

0100 RESTORE

Let's assume that you want to assign the values 1, 2, and 3 to three variables A, B, and C, in that order. Then later in the program you want to assign the same values to D, E, and F. These statements will do just that:

```
0030 DATA 1,2,3,4,5,6  
:  
0060 READ A,B,C  
:  
0100 RESTORE READ FROM START OF DATA TABLE  
0110 READ D,E,F
```

Notice that you can include a comment in the RESTORE statement. The words READ FROM START OF DATA TABLE have no effect on what your program is doing; they merely serve as a reminder to you, when you look at the program, of what the RESTORE statement is doing. Your comment can say anything you want it to say, as long as it fits on one line with the RESTORE statement.

It's important to remember, when using READ and DATA statements, that no matter how many DATA statements you include in your program, only *one* data table is created before any READ statement is executed. The table is created from all the DATA statements in your program, regardless of where they appear—at the beginning, at the end, or scattered throughout. Each of the following three sets of statements has the same effect:

```
0200 DATA 1,2,3  
0210 DATA 4,5,6  
0220 READ H,I,J,K,L,M  
  
0200 READ H,I,J,K,L,M  
0210 DATA 1,2,3  
0220 DATA 4,5,6  
  
0200 DATA 1,2,3  
0210 READ H,I,J,K,L,M  
0220 DATA 4,5,6
```

## THE INPUT STATEMENT

Both the assignment statement (LET) and the DATA statement use constants—unchanging data items that are part of your program—to assign values to variables. You have to know, at the time you're writing your program, what values you want to assign.

The INPUT statement allows a little more flexibility. This statement names the variables that are to receive values, but allows you to wait until you are running your program to actually supply the values. For example:

0050 INPUT X,Y,Z

means that you will supply values from the keyboard for X, Y, and Z when your program is run. You'll know when it's time to supply the values because a flashing question mark will be displayed. When you see this, you should enter your values, one for each variable in the INPUT statement—in this case, three. The values are entered all on one line, separated by commas. Thus, when you've entered the information, the display screen shows:

185,205,191

By entering these numbers, you've assigned 185 to X, 205 to Y, and 191 to Z.

You have to be certain, when entering your values, to enter exactly the same number of values as there are variables in the INPUT statement in your program. The question mark will keep flashing until the correct number of values is entered. If you enter too many values, the excess values are ignored. After the last value is entered, press EXECUTE to continue program execution.

#### Prompting Your Input

Since a lot of time can elapse between the time you write a program and the time you run it, you may have difficulty remembering exactly how many values you have to enter. This is especially true when your program contains more than one INPUT statement. Then you have to keep track of which one comes first.

You can have your program keep track for you by reminding you what has to be entered. All you have to do is include a PRINT statement immediately before the INPUT statement in your program. For example, if your program averages bowling scores, you could use these statements:

0045 PRINT 'ENTER THREE BOWLING SCORES'  
0050 INPUT X,Y,Z

Then, when the program is run, instead of just a question mark appearing when it's time to enter your values, these lines will be displayed:

```
ENTER THREE BOWLING SCORES  
?
```

When you've entered your values, the display screen will show:

```
ENTER THREE BOWLING SCORES  
185, 205, 191
```

You can write any reminder message that you want in the PRINT statement, as long as you enclose it in single quotation marks.

You also have to remember that the PRINT statement has to fit entirely on one line. If your message is so long that it doesn't fit, you might consider using several consecutive PRINT statements:

```
0040 PRINT 'ENTER 12 AVERAGE TEMPERATURES'  
0050 PRINT 'FOR JANUARY TO DECEMBER'  
0060 INPUT M,N,O,P,Q,R,S,T,U,V,W,X
```

### **ENTERING CHARACTER VARIABLES INTO PROGRAMS**

You've been entering numeric variables into programs in this section, but any of the methods you've used will let you supply values for character variables as well. You have already seen how to do this with a LET statement. For INPUT and READ statements, you just use valid character variables where we've been using numeric variables. Also, you must put single quotation marks around the value you're supplying for character variables when you enter the DATA statement or respond to the flashing question mark.

For example, if you want a program to keep track of a person's height and weight, you can enter the person's name, height, and weight with these READ and DATA statements:

```
0010 READ N$,H,W  
0020 DATA 'TOM JONES',6.1,184
```

You could also use an INPUT statement:

0010 INPUT N\$,H,W

and then respond to the flashing question mark like this:

'TOM JONES',6.1,184

### A REVIEW OF WHAT YOU'VE DONE

All of the following methods of assigning values to variables are useful:

- LET statements
- READ, DATA, and RESTORE statements
- INPUT statements with data supplied from the keyboard

You can use a combination of these methods if you have a program where some values don't change, some change occasionally, and others change often.

## **Chapter 6. Making Changes to Your Programs**

It is very important that you be able to make changes in your programs. You may have to change a program to supply values for variables, to make corrections, to add lines, or to remove lines. There are several ways you can change a program, either as you write it or after you write it.

### **CORRECTING KEYING ERRORS**

If you make mistakes while entering your program statements or commands, you already know how to fix them. As you catch the errors, you can:

- Use the backspace or forward space key to position the cursor at the incorrect character, then simply enter the correct character.
- Use the insert or delete function to insert or delete characters.
- Use the scroll up and scroll down keys to position a line to be corrected.
- Press ATTN to delete all characters starting with and to the right of the cursor position.

### **INSERTING NEW LINES**

The following program, called phone, computes charges for local telephone calls. The rate for local calls in this example is 10 cents for the first three minutes or less, and 2 cents for each additional minute or fraction of a minute. We'll write a general program, but we'll purposely omit the actual length of any call. These are the variables we'll use:

T	-	Total length of the call in whole minutes
T1	-	Amount of time over 3 minutes
C	-	Charge for the call

The program is:

```
0010 IF T>3 GOTO 0040
0020 PRINT 'CALL LESS THAN OR EQUAL 3 MIN, 10 CENT CHARGE'
0030 GOTO 0100
0040 LET T1=T-3
0050 LET C=.1+.02*T1
0060 PRINT 'LENGTH OF CALL'
0070 PRINT T
0080 PRINT 'CHARGE FOR CALL'
0090 PRINT C
0100 END
```

Enter this program. After we add a statement to assign a value to T, you'll be able to run the program.

To assign a value to T, you can use READ and DATA statements:

## READ T

## DATA 8 (for an 8 minute call)

You can insert these statements before line 10.

Now enter:

## 5 READ T

Press EXECUTE and enter:

6 DATA 8

and press EXECUTE again.

To see what has been done with these statements, enter the **LIST** command.

The display screen shows:

```
0005 READ T
0006 DATA 3
0010 IF T>3 GOTO 0040
0020 PRINT 'CALL LESS THAN OR EQUAL 3 MIN, 10 CENT CHARGE'
0030 GOTO 0100
0040 LET T1=T-3
0050 LET C=.1+.02*T1
0060 PRINT 'LENGTH OF CALL'
0070 PRINT T
0080 PRINT 'CHARGE FOR CALL'
0090 PRINT C
0100 END
```

The 5100 has taken the two lines and inserted them in the program (as lines 5 and 6) before line 10. By entering a line number and any valid BASIC statement, you have given an instruction. This instruction starts with a line number, and tells the 5100 you are adding a line and where to add it.

Now you can see why it is convenient to have the line numbers increase by 10's; it gives you the chance to insert up to nine new lines between every two original lines.

You can now run the phone program by entering the RUN command.

### **REPLACING ONE LINE WITH ANOTHER**

Let's try a different value for T in the phone program. This time T is 21 minutes. You'll have to change line 6, the DATA statement, to use this new value.

Enter the following statement, then press EXECUTE:

**6 DATA 21**

If you list the program now, the display screen shows:

```
O 0005 READ T  
0006 DATA 21  
0010 IF T>3 GOTO 0040  
0020 PRINT 'CALL LESS THAN OR EQUAL 3 MIN, 10 CENT CHARGE'  
0030 GOTO 0100  
0040 LET T1=T-3  
0050 LET C=.1+.02*T1  
0060 PRINT 'LENGTH OF CALL'  
0070 PRINT T  
0080 PRINT 'CHARGE FOR CALL'  
0090 PRINT C  
0100 END
```

C See what happened? The 5100 replaced the old line 6 with the new line 6.

When you want to replace a line, simply enter the same line number as the line you want to replace and enter the new line. The 5100 replaces the old line in storage with the new one after you press EXECUTE.

Remember that you can use the SAVE command if you want to save the program on tape.

## O REMOVING A LINE

In the phone program, we will now include an INPUT statement so we can run the program with many changing values for T. We can replace the READ statement with an INPUT statement, but the DATA statement must be deleted. To do this, first list the program. Now enter the number of the line you want to delete, then enter DEL and press EXECUTE. To delete line 6, enter 6 DEL, then press EXECUTE. Line 5 can be replaced by the following procedure: enter 5 INPUT T, and press EXECUTE. List the program again, and the display screen shows:

```
O 0005 INPUT T  
0010 IF T>3 GOTO 0040  
0020 PRINT 'CALL LESS THAN OR EQUAL 3 MIN, 10 CENT CHARGE'  
0030 GOTO 0100  
0040 LET T1=T-3  
0050 LET C=.1+.02*T1  
0060 PRINT 'LENGTH OF CALL'  
0070 PRINT T  
0080 PRINT 'CHARGE FOR CALL'  
0090 PRINT C  
0100 END
```

The 5100 has replaced line 5 and deleted line 6.

When you want to delete a line, simply enter the statement number, then enter DEL, and press EXECUTE. A new listing of the program will show the line deleted. You can also use the DEL function to delete several lines. For example, you could delete lines 0070 through 0090 by entering:

0070 DEL 0090

### RENUMBERING STATEMENT LINES

In the phone program, the statement numbers are not sequential by 10's. If you want the numbers to start with 0010 and increase by 10, you can simply use the RENUM command. This command will assign the number 0010 to the INPUT T statement and number the remaining statements from 0020 to 0110.

In addition, the GOTO statements (original lines 10 and 30) will be altered to transfer execution to the appropriate renumbered statement. To see the result of a renumber operation, list the phone program, then enter RENUM and press EXECUTE. After you list the program again, the display screen shows:

```
0010 INPUT T
0020 IF T>3 GOTO 0050
0030 PRINT 'CALL LESS THAN OR EQUAL 3 MIN. 10 CENT CHARGE'
0040 GOTO 0110
0050 LET T1=T-3
0060 LET C=.1+.02*T1
0070 PRINT 'LENGTH OF CALL'
0080 PRINT T
0090 PRINT 'CHARGE FOR CALL'
0100 PRINT C
0110 END
```

## Chapter 7. More About the PRINT Statement

We've seen the PRINT statement used to display the values of variables and to display comments exactly as entered in the statement. We've also seen that to display a comment exactly as you entered it, you must enclose the comment in single quotation marks. You should remember, then, that if you include this line in a program:

```
0050 PRINT 'X'
```

when line 50 is executed, the display screen will show:

X

and *not* the value of X, which would be displayed if line 50 were

```
0050 PRINT X
```

Within a single PRINT statement, you can mix character and arithmetic variables and constants. You must use commas or semicolons to separate the values to be displayed. These separators (delimiters) control spacing of the displayed data. For this example, no separator is required between the variables and character constants. The only instance in which a comma or semicolon is not required is between a character constant and a variable, as in this example.

Here's an example of a program that computes annual interest for any rate and principal that you enter:

```
0010 INPUT R,P  
0020 LET I=(R/100)*P  
0030 PRINT 'THE ANNUAL INTEREST AT 'R' PERCENT ON $'P' IS $'I  
0040 END
```

When you run this program and use values of 7 and 825, here's what you see:

```
RUN  
7,825  
THE ANNUAL INTEREST AT 7 PERCENT ON $ 825 IS $ 57.75
```

If your 5100 has the attached printer, you can specify that the data be printed by entering PRINT FLP in place of PRINT in line 30. All of the capabilities of and restrictions for the PRINT statement also apply to PRINT FLP. A comma must separate the FLP and the first value.

## MAKING HEADINGS

Suppose you have a loop in a program that computes mileage allowances (at 12 cents a mile) for company auto trips (of 10 to 50 miles in steps of 5 miles):

```
0010 FOR X=10 TO 50 STEP 5  
0020 PRINT X, .12*X  
0030 NEXT X  
0040 END
```

When you run this program, the display screen shows:

RUN	
10	1.2
15	1.8
20	2.4
25	3
30	3.6
35	4.2
40	4.8
45	5.4
50	6

You can make headings for these columns by entering a PRINT statement before the loop:

```
0005 PRINT 'MILES', 'MILEAGE ALLOWANCE'
```

When you run the program again, the display screen shows:

RUN	MILES	MILEAGE ALLOWANCE
	10	1.2
	15	1.8
	20	2.4
	25	3
	30	3.6
	35	4.2
	40	4.8
	45	5.4
	50	6

You will be shown later how you could change the mileage allowance column to include trailing zeros, which will make it more readable as dollars and cents.

### MATH CALCULATIONS IN PRINT STATEMENTS

The PRINT statement allows you to include math calculations along with variables and words. Therefore, if you just want a calculation done and the result displayed, you can do it in a single PRINT statement. For example, you can write

```
0010 INPUT X  
0020 PRINT X,X12
```

instead of writing

```
0010 INPUT X  
0020 LET Y=X12  
0030 PRINT X,Y
```

## Chapter 8. Setting Up Your Own Format—PRINT USING and Image Statements

A more flexible way to display results is to use a statement called PRINT USING. This statement allows you to display variables using a particular format. You specify the format in a separate BASIC statement called an image statement. The PRINT USING statement is used together with an image statement. The image statement can appear anywhere in a program and can be used by any number of PRINT USING statements.

The image statement is a BASIC statement, but it looks different from the other BASIC statements. Following the statement number is a colon (:). After the colon, you enter the exact wording and format that you want your results to have. You leave room for any variable values by using # signs where the values belong. When the program is run, the 5100 substitutes real values for the # signs and displays the data in the image statement (including any blank spaces you leave).

A sample PRINT USING statement and its image statement are:

```
0020 PRINT USING 30,G,N  
0030 :GROSS SALES ARE ###.## AND NET PROFITS ARE ###.##
```

Notice that single quotation marks were not used around the character data in line 0030. This is true of all image statements, unless, of course, you want quotation marks to appear in the displayed or printed data.

This is the way these two statements work together:

0020 PRINT USING 30,G,N

The line number of the image statement.      A comma, which is required.      The variables, separated by commas, whose values are going to be displayed or printed in the format specified in the image statement.

0030 :GROSS SALES ARE ###.## AND NET PROFITS ARE ###.##

This identifies the image statement; it does not appear when the statement is executed.      The value of G will be inserted here when the line is displayed or printed instead of ###.##.      The value of N will be inserted here when the line is displayed or printed instead of ###.##.

If these two statements were part of a program, with G equal to 103.72 and N equal to 21.45, at the time you ran the program, the 5100 would display:

GROSS SALES ARE 103.72 AND NET PROFITS ARE 21.45  
↑  
Display Position 1

An image statement always begins with a colon. When you enter the # signs, as stand-ins for variable values, you are really telling the 5100 how many spaces to leave for the values. If a value needs more space than you indicated, a row of asterisks will be displayed or printed instead of the value when the statement is executed. In these statements:

0040 PRINT USING 50,Z,Z↑3  
0050 :Z IS ## AND Z CUBED IS ##

there are only two spaces indicated for each value. If Z is 3, the displayed result would be:

Z IS 3 AND Z CUBED IS 27

But if Z is 5, then  $5^3$  is 125, which is three spaces long. The displayed result would be:

Z IS 5 AND Z CUBED IS \*#\*

The asterisks mean that the answer was too long for the space you indicated.

When you use # signs to indicate space for your variable values in an image statement, you can also control how many decimal places you want the value to have when it is displayed or printed. You do this by inserting a decimal point in the string of # signs wherever you want the decimal point to go in the result. For example, when you are using dollars and cents, insert a decimal point two places from the right, as in this example:

0100 PRINT USING 110,S  
0110 :THE SERVICE CHARGE FOR YOUR ACCOUNT IS \$###.##

If S were 23.47, the result would be displayed as follows when you ran the program:

THE SERVICE CHARGE FOR YOUR ACCOUNT IS \$ 23.47

If your system has a printer, you can use the PRINT USING FLP statement to format your data. All the capabilities of the PRINT USING statement also apply to the PRINT USING FLP statement.

### Example of Printing

There are three sales representatives in your department, and you are responsible for making a monthly report showing their sales figures for each week of the month. You can write a program that will automatically print the report for you in an attractive format.

This is how the variables in the program are named. Each sales representative is assigned a letter: A, B, and C. Starting with the first sales representative, the variables are:

A\$ — Name  
A1 — Sales total for week 1  
A2 — Sales total for week 2  
A3 — Sales total for week 3  
A4 — Sales total for week 4  
A5 — Sales total for the month

and so on for sales representatives B and C

There are these other variables:

M\$ — Name of the month  
W\$, X\$, Y\$, and Z\$ — Last day of each week in the month  
T1, T2, T3, and T4 — Totals for everybody for each week  
T5 — Total for everybody for the month

For this program, you have to supply the month's sales figures, the name of the month, and the last day of each week covered. The report is then printed automatically. Now enter the following program:

```

0010 REM PROGRAM FOR PRINTING MONTHLY SALES REPORT
0020 LET A$='ADLER'
0030 LET B$='BIPPLE'
0040 LET C$='CUBBINS'
0050 PRINT 'ENTER MONTH, LAST DAY OF EACH WEEK'
0060 INPUT M$,W$,X$,Y$,Z$
0070 PRINT 'ENTER FIGURES FOR ADLER'
0080 INPUT A1,A2,A3,A4
0090 PRINT 'ENTER FIGURES FOR BIPPLE'
0100 INPUT B1,B2,B3,B4
0110 PRINT 'ENTER FIGURES FOR CUBBINS'
0120 INPUT C1,C2,C3,C4
0130 LET A5=A1+A2+A3+A4
0140 LET B5=B1+B2+B3+B4
0150 LET C5=C1+C2+C3+C4
0160 PRINT USING FLP,0170,M$
0170 :MONTHLY SALES REPORT FOR MONTH OF #####
0180 PRINT FLP
0190 PRINT USING FLP,0200
0200 :SALESMAN           WEEK ENDING          TOTAL
0210 PRINT USING FLP,0220,W$,X$,Y$,Z$
0220 :      ##      ##      ##      ##
0230 PRINT FLP
0240 PRINT USING FLP,0270,A$,A1,A2,A3,A4,A5
0250 PRINT USING FLP,0270,B$,B1,B2,B3,B4,B5
0260 PRINT USING FLP,0270,C$,C1,C2,C3,C4,C5
0270 : #####,##,##,##,##,##,##,##,##,##,##,##,##,##,##,##
0280 LET T1=A1+B1+C1
0290 LET T2=A2+B2+C2
0300 LET T3=A3+B3+C3
0310 LET T4=A4+B4+C4
0320 LET T5=A5+B5+C5
0330 PRINT FLP
0340 PRINT USING FLP,0350,T1,T2,T3,T4,T5
0350 :TOTALS    #####,##,##,##,##,##,##,##,##,##,##,##,##,##,##,##
0360 END

```

Note that a PRINT FLP statement with nothing after it causes a blank line to be printed. This form of the statement is used in lines 0180, 0230, and 0330 to include blank lines in the printed report.

If your system has a printer, you can run the program by entering a RUN command and pressing EXECUTE. If your system does not have a printer, enter a RUN command, then enter P=D. For example: RUN P=D, then press EXECUTE. This command directs all printed output to the display screen.

During a sample running of this program, the display screen showed:

RUN  
ENTER MONTH, LAST DAY OF EACH WEEK

'JULY', '7', '14', '21', '28'  
ENTER FIGURES FOR ADLER

12.50, 500.00, 400.00, 895.50  
ENTER FIGURES FOR BIPPLE

34.50, 78.90, 500.00, 100.00  
ENTER FIGURES FOR CUBBINS

300.00, 800.00, 700.00, 43.25

For this program the printed or displayed output was:

MONTHLY SALES REPORT FOR MONTH OF JULY

SALESMAN	WEEK ENDING			TOTAL	
	7	14	21		28
ADLER	12.50	500.00	400.00	895.50	1808.00
BIPPLE	34.50	78.90	500.00	100.00	713.40
CUBBINS	300.00	800.00	700.00	43.25	1843.25
TOTALS	347.00	1378.90	1600.00	1038.75	4364.65

## Chapter 9. More Things You Can Do With BASIC

### SOME GENERAL SYSTEM FUNCTIONS

The following system functions provide you with the functions described:

ABS(X)	Gives the absolute value of X.
INT(X)	Gives the integer part of X.
RND or RND(X)	Generates a random number between 0 and 1.
SGN(X)	Determines the sign of variable X, and returns a value of -1, 0, or +1, depending on whether X is negative, zero, or positive.

To use these functions, you just substitute the name of your own variable for the X inside the parentheses. You can also include expressions inside the parentheses, for example:

INT(X<sup>12</sup>+Y\*12)

You might use SGN(X) to find out if X is positive:

SGN(X)

The RND function is a little different from the other functions. You can use RND alone, without a value, to generate a random number between 0 and 1. Each subsequent use of RND in the program will generate a new random number. However, if you rerun the program with a new RUN command, the random numbers generated will be the same as the numbers generated the first time you ran the program. To avoid this, you can use RND(X) to generate different sets of random numbers each time you run your program. The value of X is used by the process that develops the random number. If you want a random number that is a whole integer instead of a decimal number between 0 and 1, multiply the result of RND or RND(X) by a constant (depending on what range you want the random numbers to have); and then use the INT function to make the result an integer.

For example:

INT(RND\*10)      Produces a random number between 0 and 9.

INT(RND\*100)      Produces a random number between 0 and 99.

INT(RND\*1000)      Produces a random number between 0 and 999.

## CONVERSION FUNCTIONS AND CONSTANTS

BASIC has some built-in ways to convert values from one measuring system to another. In addition to &LBKG, &GALI, and &INCM, which were discussed in Chapter 2, BASIC provides:

DEG(X) — Gives the number of degrees in X radians.

RAD(X) — Gives the number of radians in X degrees.

## TRIGONOMETRIC FUNCTIONS

BASIC has functions that automatically perform trigonometric operations for you. Just substitute your own variable or expression where the variable X appears in the following list:

SIN(X) — Gives the sine of X radians.

COS(X) — Gives the cosine of X radians.

TAN(X) — Gives the tangent of X radians.

COT(X) — Gives the cotangent of X radians.

SEC(X) — Gives the secant of X radians.

CSC(X) — Gives the cosecant of X radians.

ASN(X) — Gives the arc sine (in radians) of X.

ACS(X) — Gives the arc cosine (in radians) of X.

ATN(X) — Gives the arc tangent (in radians) of X.

**O** These functions deal in radians. If your program measures angles in degrees instead of radians, combine the RAD or DEG functions with these functions to keep the results in degrees. For example, to find the sine of D degrees, you can use this statement:

0040 LET S=SIN(RAD(D))

**C** Or to find the arc sine of X in degrees instead of radians, you can use this statement:

0070 LET A=DEG(ASN(X))

## **C** LOGARITHMS AND EXPONENTS

BASIC also has functions that automatically take logarithms and calculate exponents for you:

EXP(X) — Gives the natural exponent of X ( $e^X$ ).

LGT(X) — Gives the logarithm of X to the base 10.

LOG(X) — Gives the logarithm of X to the base e.

LTW(X) — Gives the logarithm of X to the base 2.

## Chapter 10. Tape Data Files

A file is a collection of related data items that are stored together.  
All the items are stored in sequential order.

### ACTIVATING AND DEACTIVATING FILES

Files must be activated or *opened* before they can be used within a program. A file must be opened by an OPEN statement in a program. The following example shows the format of an OPEN statement:

```
0050 OPEN FL1,'E80',2,IN
```

FL1 is the file reference, which can be from FL0 to FL9, but must be the same as the file reference in the GET or PUT statement. This file reference does not identify the file being read or written. 'E80' is the device address of the tape unit built in the 5100. The number 2 specifies which physical file on tape is going to be used. This number can be from 1 through 132, and can be specified as a variable. The word IN indicates that the file is to be used for retrieving data items from the file for use in the program.

If a file were to be used with PUT statements, it could be opened as an output file with this statement:

```
0100 OPEN FL1,'E80',2,OUT
```

Normally, a file is deactivated or *closed* by the system after execution of your program. However, if you want to switch an input file to output (or vice versa) and continue to use it in the same program, you must deactivate it by using the CLOSE statement before reopening it. (If you did not use the CLOSE statement and attempted to use an output file for input or vice versa, execution of your program would be terminated.) The CLOSE statement deactivates the file; a subsequent OPEN statement opens (reactivates) the file for its new use and repositions it at its beginning. Under ordinary circumstances, the CLOSE statement is optional, and the system will close a file at the end of program execution. The one time that the CLOSE statement is required is if you use the same file for *both* input and output operations in the same program.

## CREATING A TAPE FILE

The following compound interest program can be used to produce an output listing containing 600 values in 200 lines:

```
0010 PRINT 'ENTER PRINCIPAL'
0020 INPUT P
0030 PRINT FLP, 'TIME', 'RATE', 'AMOUNT'
0040 FOR T=1 TO 10
0050 FOR R=1 TO 20
0060 LET A=P*(1+R/100)^T
0070 PRINT FLP, T,R,A
0080 NEXT R
0090 NEXT T
0100 END
```

The PRINT statement in this program is executed 200 times to produce an output listing containing the values. These values could be grouped as an output file on tape. In fact, instead of printing them, you could store them in the file and use them later. By adding an OPEN statement and substituting a PUT statement for the PRINT statement (line 0070), you can create a tape file; for example:

```
0025 OPEN FL1,'E80',2,OUT
0070 PUT FL1,T,R,A
```

This PUT statement instructs the 5100 to put the values contained in the variables T, R, and A into the file that is defined in the OPEN statement with the same file reference (FL1). As far as the 5100 is concerned, both PUT and PRINT mean output; the only difference is whether the output goes to a tape file or to the printer or display screen. Semicolons cannot be used to separate variables in a PUT statement; use only commas.

## RETRIEVING A FILE

To access data in a tape file, you use the GET statement, which is the input counterpart to the PUT statement.

To access the first set of values from the file created with the preceding PUT statement, you can use the following GET statement:

```
0020 GET FL1,T,R,A
```

This statement assigns the first three values contained in the file to the variables T, R, and A. It is not necessary to use the same variable names that were used when the file was created; for example, we could assign these values to variables X, Y, and Z. The important requirement is that the values in the file and the variables to which they are assigned must be of the same type—arithmetic variables for arithmetic values, and character variables for character values.

The file reference (FL1) must be the same as the file reference in the OPEN statement that defines the specific tape file. You must first close the output file and reopen it for input:

```
0010 CLOSE FL1  
0050 OPEN FL1, 'E80',2,IN
```

After the first GET statement is executed, the file is positioned at the next value. Thus, a second GET statement referring to FL1 would access the next three values in the file. If we wanted to access all the values stored previously, we could issue the GET statement 200 times, or enclose one GET statement in a loop as follows:

```
0050 OPEN FL1,'E80',2,IN  
0060 FOR X=1 TO 200  
0070 GET FL1,T,R,A  
0080 PRINT T,R,A  
0090 NEXT X
```

These statements would print the 200 values for each T, R, and A.

## REPOSITIONING FILES

You may have an occasion to use an input file or an output file more than once in the same program. The RESET statement allows you to reposition the file without deactivating it (deactivation is necessary only when the function of a file is changed from input to output or vice versa). For example:

```
0020 OPEN FL4,'E80',4,IN  
0030 GET FL4,X,Y,Z,Q,R,S  
:  
0100 RESET FL4  
0110 GET FL4,X,Y,Z,Q,R,S  
:  
0150 RESET FL4  
0160 GET FL4,X,Y,Z,Q,R,S  
:
```

Between statements 0030 and 0100, the variables X, Y, Z, Q, R, and S could be used in one set of calculations and their values changed. By repositioning the file, the original values in the file could again be made available and put into variables X, Y, Z, Q, R, and S for different calculations or uses between statements 0110 and 0150, and again between statement 0160 and the end of the program. Actually, the RESET statement used in this way functions for files in the same way that the RESTORE statement functions for the data table created by the DATA statement.

To add data to the end of the file, you can reset it to its end by using the RESET statement with the END keyword:

0200 RESET FL1 END

This statement positions FL1 to the end of the last data item in the file. PUT statements appearing after statement 0200 will place additional values in the file. In effect, RESET END allows you to build onto a file.

## Chapter 11. Arrays

With the BASIC language, you can keep groups of similar data (arithmetic or character) together by organizing them into arrays. An array is a collection of data items that is referred to by a single name.

Arithmetic arrays are named by a single letter of the extended alphabet. Thus, the letter A can stand for a single arithmetic variable or an arithmetic array or both, while the symbol A2 can only stand for a single arithmetic variable. A single letter stands for an array only when it has been defined in a DIM (dimension) statement, which is described later. All elements of an arithmetic array are initially set to 0 when the program is executed.

Character arrays, like simple character variables, are named by a single letter of the extended alphabet followed by the dollar sign (\$). Each element of a character array is 18 characters in length. Each element is initially set to 18 blank characters when program execution begins.

BASIC arrays can be either one or two dimensions. A one-dimensional array can be thought of as a row of successive data items. A two-dimensional array can be thought of as a rectangular matrix of rows and columns. A representation of a one-dimensional array A containing four elements is:

Array A			
A(1)	A(2)	A(3)	A(4)

A representation of a two-dimensional array B with four rows and three columns is:

Array B		
B(1,1)	B(1,2)	B(1,3)
B(2,1)	B(2,2)	B(2,3)
B(3,1)	B(3,2)	B(3,3)
B(4,1)	B(4,2)	B(4,3)

To illustrate the use of one- and two-dimensional arrays, suppose you are keeping weather statistics on the average temperature and the inches

of rainfall for 12 months. You can write a program to keep each set of that data in arrays:

- Names of the months
- Average temperature for each month
- Total rainfall for each month

You can arrange the data as three one-dimensional arrays:

Array 1	Array 2	Array 3
Names of Months	Average Temperature	Rainfall
January	28	3.47
February	31	2.10
March	35	2.95
April	49	4.82
May	60	3.02
June	64	2.87
July	75	2.04
August	81	1.89
September	71	2.74
October	59	2.90
November	46	1.85
December	37	2.35

Or as one two-dimensional array:

Array 1		
Month	Temp	Rainfall
1	28	3.47
2	31	2.10
3	35	2.95
4	49	4.82
5	60	3.02
6	64	2.87
7	75	2.04
8	81	1.89
9	71	2.74
10	59	2.90
11	46	1.85
12	37	2.35

The second example is really a modified combination of the three one-dimensional arrays. The first column has been changed to the numeric representation of the months because the names of the months (character data) cannot be included in the same array with numeric data.

It will be much easier to use the weather data if we keep it together in three one-dimensional arrays, or in one two-dimensional array, than it would be if we considered it as 36 separate variables. This chapter will show you how to work with arrays in BASIC programs.

## DEFINING AN ARRAY

When you want to work with an array, you must first tell the 5100 that you are using an array and not ordinary variables. This is called *defining* your array. Defining the array merely involves telling the 5100 how big the array is going to be so the 5100 can leave room for it, and telling the 5100 what kind of data will be in it. (Later on you enter the data, but this is not part of defining the array.)

The data for your arrays can be numeric data or character data. You can define an array to contain either kind of data, but it must contain *only* one kind of data. You can't mix characters and numbers in a single array. That's why we used the numbers of the months instead of their names when we put the weather data in a two-dimensional array.

An array composed of numbers is called an *arithmetic array*. It is named by a single letter of the extended alphabet such as A or T.

An array composed of character data is called a *character array*. It is named by a single letter of the extended alphabet followed by a dollar sign (\$); for example, N\$ or Q\$.

To define either kind of array, you use a statement called DIM. In the DIM statement, you name the array and include the size of it in parentheses after the name.

### DIM Statement for One-Dimensional Arrays

For a one-dimensional array, the size is a single number. Thus, to define an arithmetic one-dimensional array A with 12 elements, your DIM statement is:

0010 DIM A(12)

To define character array N\$ with 20 elements, your statement is:

0010 DIM N\$(20)

To define both together, your statement is:

0010 DIM A(12), N\$(20)

### DIM Statement for Two-Dimensional Arrays

For two-dimensional arrays, the size is two numbers, one for each dimension. The first number is the number of rows in the array; the second number is the number of columns in the array.

To define array W with 12 rows and 3 columns, the DIM statement is:

0010 DIM W(12,3)

Character array A\$ with 3 rows and 4 columns is defined by:

0010 DIM A\$(3,4)

You can define all your arrays in a single DIM statement. You can also mix definitions of one- and two-dimensional arrays in a single DIM statement.

### ELEMENTS OF ARRAYS

Each individual item in an array is called an *element* of the array. When you want to refer to a particular element of an array, instead of to the whole array itself, you talk about the position of that element in the array. For example, if you want to refer to the third element of one-dimensional array H, you would refer to it as H(3). To refer to the element in the first row and third column of array W, you use W(1,3). The position goes in parentheses after the name of the array. For two-dimensional arrays, the first number is always the number of the row, the second number is always the number of the column.

If we look at the weather example as three one-dimensional arrays, we can call the array with the names of the months M\$, the array of temperature data T, and the array of rainfall data R. If we consider the weather data as one two-dimensional array, called W, the numbers of the months are in column 1, the temperature data is in column 2, and the rainfall data is in column 3. If you wanted to refer to January in a program statement, you would refer to either M\$(1) or W(1,1).

Here are all the months and the way you refer to them in arrays M\$ and W:

This Month:	Is in this Position:	
	In Array M\$	In Array W:
1 (January)	M\$(1)	W(1,1)
2 (February)	M\$(2)	W(2,1)
3 (March)	M\$(3)	W(3,1)
4 (April)	M\$(4)	W(4,1)
5 (May)	M\$(5)	W(5,1)
6 (June)	M\$(6)	W(6,1)
7 (July)	M\$(7)	W(7,1)
8 (August)	M\$(8)	W(8,1)
9 (September)	M\$(9)	W(9,1)
10 (October)	M\$(10)	W(10,1)
11 (November)	M\$(11)	W(11,1)
12 (December)	M\$(12)	W(12,1)

Note that the month names are not used in array W.

If we include the temperature and rainfall data, the first element in each one-dimensional array—M\$(1), T(1), R(1)—or the first row in array W—W(1,1), W(1,2), W(1,3)—will be data for January; the second element in each one-dimensional array, or the second row in W, will be data for February; and so on.

So far, however, there is no data in any of the arrays. We have only defined the names and sizes. After you define an array, the 5100 sets the values of *all* its elements to 0 (for arithmetic arrays) or blanks (for character arrays).

### Assigning Values to Array Elements

To assign values to array elements (the names of the months, the temperatures, or the rainfall), you use the methods of assigning values that you've been using all along.

#### LET Statements

You can use a LET statement to assign a value to an element of an array. So if the average temperature for January is 28°, you could

write either of these statements:

```
0020 LET T(1)=28  
0020 LET W(1,2)=28
```

This method is acceptable if you only have a few values to assign, but it will take forever if the array is large. In the weather example, we would need 36 separate LET statements to assign all the data to the arrays. Nevertheless, the LET statement is handy if you only want to assign a few values, or if you want to change a value you have already assigned.

Remember that if you are assigning a value to an element of a *character array*, you enclose the characters you are assigning in single quotation marks. For example:

```
0020 LET M$(1)='JANUARY'
```

#### DATA and READ Statements

Another way to assign values is to use DATA and READ statements. You use these the same way you do for variables. For example:

```
0020 READ M$(1),M$(2),M$(3)  
0030 DATA 'JANUARY','FEBRUARY','MARCH'
```

or

```
0020 READ W(1,1),W(2,1),W(3,1)  
0030 DATA 1,2,3
```

Again, when you are using large amounts of data, listing them all separately in a READ statement is not practical. In this example, you can take advantage of a FOR-NEXT loop to assign values:

```
0020 FOR I=1 TO 12  
0030 READ T(I)  
0040 NEXT I  
0050 DATA 28,31,35,49,60,64,75,81,71,59,46,37
```

or

```
0020 FOR I=1 TO 12  
0030 READ W(I,2)  
0040 NEXT I  
0050 DATA 28,31,35,49,60,64,75,81,71,59,46,37
```

These statements assign all the average temperature data to array T or to the second column of array W. (For array W, since we are assigning values *only* to the second column, we used a constant of 2 in the READ statement.) You can't avoid specifying 12 values in the DATA statement, but a loop like this makes the READ statement easier to handle.

When assigning values to array W, you could, in fact, use one READ statement and two loops to assign *all* the data at once. It would look like this:

```
0020 FOR I=1 TO 12  
0030 FOR J=1 TO 3  
0040 READ W(I,J)  
0050 NEXT J  
0060 NEXT I
```

Arranged this way, the loops let you enter the data for each row of the array in succession. Your DATA statements might look like this:

```
0070 DATA 1,28,3.47  
0080 DATA 2,31,2.10  
0090 DATA 3,35,2.95
```

We've entered the data for each row of array W in a separate DATA statement because it is easier to visualize the data that way. You could, however, string out the data so that more than one row appears in a DATA statement like this:

```
0070 DATA 1,28,3.47,2,31,2.10,3,35,2.95 . . .
```

This way you could enter as many data items in each DATA statement as will fit on a line. The important thing is that the data must appear in the same *order* as if you were entering it row by row.

#### INPUT Statements

You can use INPUT statements to assign values from the keyboard to array elements. You can list all the array element names in the INPUT statement, or you can write a FOR and NEXT loop—similar to the ones for READ—to specify the names of the elements that are to receive values.

O For example, you can assign values to the one-dimensional rainfall array R with this statement:

0020 INPUT R(1),R(2),R(3),R(4),R(5)

C or with these statements:

0020 FOR I=1 TO 12  
0030 INPUT R(I)  
0040 NEXT I

C You can assign the rainfall data to the third column of array W with these statements:

0020 FOR I=1 TO 12  
0030 INPUT W(I,3)  
0040 NEXT I

O As with the READ statement, you can write a double loop for an INPUT statement so that you can supply all the data for array W at once. In all instances, the 5100 flashes a question mark on the display screen when the system is ready for you to enter the data from the keyboard. However, if your INPUT statement is in a loop, the 5100 flashes a question mark each time the loop is executed. This means you supply one item of data, wait for the next question mark, supply the next item of data, and so on. You will have to enter the data one item at a time, waiting for a question mark between each entry.

#### Another Way to Assign Values to Arrays

O Instead of using a loop with a READ or INPUT statement to assign values, you can write a READ or INPUT statement such as:

0020 MAT READ M  
0030 MAT INPUT N

O These statements tell the 5100 to read in values for the *entire* array. The letters MAT stand for the word matrix.

C This method of assigning values with a MAT READ statement has no effect on your DATA statements. Thus, to assign the temperature data to one-dimensional array T, you could write these statements:

0020 MAT READ T  
0030 DATA 28,31,35,49,60,64,75,81,71,59,46,37

If you use a MAT READ W statement, you would have to enter the data for the entire array in DATA statements. You assign the data row by row with these statements:

```
0020 MAT READ W  
0030 DATA 1,28,3.47  
0040 DATA 2,31,2.10  
0050 DATA 3,35,2.95
```

or with these statements:

```
0020 MAT READ W  
0030 DATA 1,28,3.47,2,31,2.10,3,35,2.95
```

If you use a MAT INPUT statement to assign values to an array, the 5100 will signal you with a flashing question mark, as usual, when it is ready for you to enter data from the keyboard. If you are supplying values for a one-dimensional array, just type in all the values on a single line. If you are supplying values for a two-dimensional array, type in all the data row by row. Remember that the values must be separated by commas.

### Assigning Values to an Entire Array at Once

If you want every element of an array to have the same value, such as all 1's or all 0's, you can assign that value to each element of the array with the following statement:

```
0030 MAT A=(0)
```

You could also assign to every element of an array the value of a variable or the value of an arithmetic expression with this statement:

```
0050 MAT T=(X)
```

or this statement:

```
0060 MAT M=(X+Y*Z)
```

The value you are assigning must be enclosed in parentheses so that the 5100 knows it is not the name of another array.

If you omit the parentheses, you can make one array an identical copy of another array by using this statement:

```
0070 MAT R=S
```

In this statement, you don't use parentheses because you are, in fact, referring to another array in this assignment statement.

This method of assigning values is limited, however. You can't use the following statement:

0040 MAT R=-S

to set the values of the elements of array R equal to the negative values of the elements of array S. To do that, you would have to write this statement:

0040 MAT R=(-1)\*S

(See *Arithmetic with Arrays* later in this chapter for more information.)

### Working with Elements of Arrays

After you assign values to elements of arrays, you can perform calculations with individual array elements. You use elements of arrays just as you use any variable in any BASIC statement. Nothing is different except that you are keeping a set of variables together for your own convenience in organizing data. Each element still has a value and can act as an independent variable.

### Printing Arrays

Elements of arrays, like ordinary variables, can be used in any PRINT or PRINT FLP statement. Some examples of PRINT and PRINT USING statements that include array elements are:

```
0020 PRINT T(3),T(4),M$(2),W(10,2),X,Y,Z  
0030 PRINT FLP, 'THE AVERAGE RAINFALL FOR JANUARY IS:',W(1,3)  
0080 PRINT USING FLP,90,M$(3),R(3)  
0090 :FOR THE MONTH OF ##### THE RAINFALL WAS #.##
```

In addition, you can print an entire array if you insert MAT before the PRINT statement. For example, the statement

0090 MAT PRINT FLP,T

will print the entire one-dimensional temperature array T. The statement

0060 MAT PRINT FLP,W

will print the entire two-dimensional weather array W. It will be printed row by row.

You cannot enter arrays and ordinary variables together in a MAT PRINT statement.

### Putting One-Dimensional Arrays Together in a Program

Now we'll put the three one-dimensional weather arrays, M\$, T, and R, together in a sample program that will keep all the data and display it when you run the program:

```
0110 REM THIS PROGRAM KEEPS WEATHER DATA
0020 DIM M$(12),T(12),R(12)
0030 FOR I=1 TO 12
0040 READ M$(I),T(I),R(I)
0050 NEXT I
0060 DATA 'JANUARY',28,3.47
0070 DATA 'FEBRUARY',31,2.1
0080 DATA 'MARCH',35,2.95
0090 DATA 'APRIL',49,4,4.82
0100 DATA 'MAY',60,3.02
0110 DATA 'JUNE',64,2.87
0120 DATA 'JULY',75,2.04
0130 DATA 'AUGUST',81,1.89
0140 DATA 'SEPTEMBER',71,2.74
0150 DATA 'OCTOBER',59,2.9
0160 DATA 'NOVEMBER',46,1.85
0170 DATA 'DECEMBER',37,2.35
0180 PRINT USING FLP,0190
0190 : MONTH           AVG TEMP          RAINFALL      FOR 1974
0200 FOR I=1 TO 12
0210 PRINT USING FLP,0230,M$(I),T(I),R(I)
0220 NEXT I
0230 : #####          ######          ####, #####
0240 END
```

This program uses FOR and NEXT loops to simplify handling the large number of values involved in these arrays. Notice that instead of writing a FOR and NEXT loop for each array when we were assigning values to the members, we wrote a single loop that worked across the three arrays instead of completing each 12-element array individually. Of course, the DATA statements had to have their data in the same order.

We also used a loop to display the data. It lets us use a single PRINT USING statement with a single IMAGE statement to print out 12 lines of data.

After you enter the statements and run the program, the display screen shows:

MONTH	Avg Temp	Rainfall	For 1974
JANUARY	28	3.5	
FEBRUARY	31	2.1	
MARCH	35	3.0	
APRIL	49	4.8	
MAY	60	3.0	
JUNE	64	2.9	
JULY	75	2.0	
AUGUST	81	1.9	
SEPTEMBER	71	2.7	
OCTOBER	59	2.9	
NOVEMBER	46	1.9	
DECEMBER	37	2.4	

### Two-Dimensional Array

Now we'll do the same thing with the two-dimensional array W. This time we'll use MAT READ W and MAT PRINT FLP, W statements instead of using loops to assign the weather data and print it. If your system does not have a printer, skip this program because the output exceeds the limits of the screen:

```
0110 REM THIS PROGRAM KEEPS DATA IN A 2 DIM ARRAY
0020 DIM W(12,3)
0030 MAT READ W
0040 DATA 1,28,3.5,2,31,2,1,3,35,3,,4,49,4,8,5,60,3,
0050 DATA 6,64,2,9,7,75,2,,8,81,1,9,9,71,2,7,10,59,2,9
0060 DATA 11,46,1,9,12,37,2,4
0070 PRINT USING FLP,0080
0080 : MONTH          AVG TEMP           RAINFALL   FOR 1974
0090 MAT PRINT FLP,W
0100 END
```

The printed output is:

MONTH	AVG TEMP	RAINFALL FOR 1974
1	28	3.5
2	31	2.1
3	35	3
4	49	4.8
5	60	3
6	64	2.9
7	75	2
8	81	1.9
9	71	2.7
10	59	2.9
11	46	1.9
12	37	2.4

## ARITHMETIC WITH ARRAYS

Suppose, instead of weather data for one year, you have weather data for two years. This data can be in two arrays. You are interested in averaging the temperatures and rainfall over the two years and making new arrays to contain the two-year averages. To see how to do this, let's look at the two sets of temperature data. If you assume that they are in two one-dimensional arrays called A and B, then to find the average temperature for each month over the two years, you have to add the two temperatures for January and divide by 2, add the temperatures for February and divide by 2, and so on.

## O Addition and Subtraction with Arrays

You can do all the addition in one step, adding the entire array A to the entire array B, with this statement:

0010 MAT C=A+B

C Again, the letters MAT stand for matrix. The preceding statement causes each element of array A to be added to the corresponding element of array B and the result to be stored in the corresponding element of array C.

C The same kind of addition statement works if you want to add two-dimensional arrays. If all the weather data for the first year is in two-dimensional array T and for the second year in two-dimensional array U, and you want the result in array V, the statement is:

0040 MAT V=T+U

Each element of array T is added to the corresponding element of array U. This includes the columns with the numbers of the months and the columns with the rainfall.

O Similarly, if you want to subtract each element of an array from the corresponding element of another array, you would write this statement:

0050 MAT C=A-B

The letters MAT always tell the 5100 to work with an entire array. Just remember that you must define all the arrays, including the one which is receiving the results, in a DIM statement at the start of your program. Also, you can only add or subtract when all the arrays named have the same dimensions. You can't, for example, add a 14-element array to a 12-element array.

## O Multiplication and Division

We have seen how to add and subtract array elements. Now what about dividing by 2? Before we can divide, we must see how to multiply, because BASIC doesn't let you divide arrays directly; you can only multiply. You can multiply each element of an array (called A, for example) by a constant, a single variable, or an arithmetic expression with this statement:

0030 MAT C=(2)\*A

The multiplier *always* goes in parentheses so the 5100 knows it is not another array, and it must always go *before* the \*. For division, you merely multiply the array by 1 over the divisor, or by a decimal number such as 0.5. Therefore, to divide each element of array A by 2, you would use this statement:

```
0080 C=(1/2)*A
```

### Averaging Two Sets of One-Dimensional Arrays

If the weather data is kept in two sets of one-dimensional arrays, A and B for temperature and C and D for rainfall, a program for averaging the two sets of data and assigning the results to master arrays T and R might look like this:

```
0010 DIM M$(12),A(12),B(12),C(12),D(12),T(12),R(12)
0020 MAT READ M$
0030 DATA 'JAN','FEB','MAR','APR','MAY','JUNE','JULY','AUG'
0040 DATA 'SEPT','OCT','NOV','DEC'
0050 MAT READ A,B,C,D
0060 DATA 20,21,22,23,24,25,26,27,28,29,30,31
0070 DATA 10,12,14,16,18,20,22,24,26,28,30,32
0080 DATA 2,2,2,2,3,3,3,4,4,4,5,5
0090 DATA 5,5,5,4,4,4,2,2,2,1,3,2
0100 MAT T=A+B
0110 MAT T=(1/2)*T
0120 MAT R=C+D
0130 MAT R=(1/2)*R
0140 FOR I=1 TO 12
0150 PRINT FLP,M$(I),T(I),R(I)
0160 NEXT I
0170 END
```

We defined arrays T and R in the DIM statement on line 10, as well as arrays M\$, A, B, C, and D. Note that we only need one array for the names of the months, no matter how many years of data we have stored in other arrays.

## Averaging Two-Dimensional Arrays

If the two sets of weather data are stored in two-dimensional arrays X and Y, a program for averaging the data might look like this:

```
0110 DIM X(12,3),Y(12,3),W(12,3)
0020 MAT READ X,Y
0030 DATA } 
0040 DATA } Data for Arrays A, B, C, and D
0050 DATA } 
0060 DATA } 
0070 MAT W=X+Y
0080 MAT W=(1/2)*W
0090 MAT PRINT FLP,W
0010 END
```

We defined array W along with arrays X and Y in the DIM statement at the start of the program. Note that the numbers of the months, which are in column 1 of both arrays X and Y, are added in statement 0070 along with the rest of the data in arrays X and Y. But when we divide by 2 in statement 0080, we get back the original numbers 1 through 12.

## Appendix A. BASIC Statements and Commands

A complete list of the statements and commands in the BASIC language that are used for the 5100 is shown below. A brief description of each statement and command is included. Although all the statements and commands are not discussed in this manual, each is described in detail in the *IBM 5100 BASIC Reference Manual*, SA21-9217.

### BASIC STATEMENTS

CHAIN	Ends a program, then loads and begins executing another program.
CLOSE	Deactivates open files.
DATA	Creates an internal data table of values.
DEF	Defines an arithmetic function to be used in the program.
DIM	Specifies the size (dimensions) of an array.
END	Ends a program.
FNEND	Ends an arithmetic function defined in a DEF statement.
FOR	Begins a loop.
GET	Assigns values from a file to variables.
GOSUB	Branches the program to the beginning of a subroutine.
GOTO	Branches the program to a specific statement.
IF	Branches the program depending on specific conditions.
IMAGE	Specifies the format of printed or displayed data.
INPUT	Assigns values from the keyboard to variables during program execution.

<b>O</b>	<b>LET</b>	Assigns values to variables.
<b>C</b>	<b>MAT</b>	Assigns values to all elements of an array.
<b>C</b>	<b>MAT GET</b>	Assigns values from a file to elements of an array.
<b>C</b>	<b>MAT INPUT</b>	Assigns values from the keyboard to elements of an array.
<b>C</b>	<b>MAT PRINT (FLP)</b>	Displays or prints the values of all elements of an array.
<b>C</b>	<b>MAT PRINT USING (FLP)</b>	Displays or prints the values of all elements of an array in a format specified in an IMAGE statement.
<b>C</b>	<b>MAT PUT</b>	Writes the values of all elements of an array into a tape file.
<b>C</b>	<b>MAT READ</b>	Assigns values from the internal data table (see DATA) to elements of an array.
<b>C</b>	<b>NEXT</b>	Ends a loop (see FOR).
<b>C</b>	<b>OPEN</b>	Activates files for input or output.
<b>C</b>	<b>PAUSE</b>	Interrupts program execution.
<b>C</b>	<b>PRINT (FLP)</b>	Displays or prints the values of specified variables, expressions, or constants.
<b>C</b>	<b>PRINT USING (FLP)</b>	Displays or prints the values of specified variables, expressions, or constants in a format defined in an IMAGE statement.
<b>C</b>	<b>PUT</b>	Writes the values of specified variables into a tape file.
<b>C</b>	<b>READ</b>	Assigns values from the internal data table (see DATA) to variables or array elements.
<b>C</b>	<b>REM</b>	Inserts comments or remarks in a program.
<b>C</b>	<b>RESET</b>	Repositions a tape file to its beginning.
<b>C</b>	<b>RESTORE</b>	Causes values in the internal data table (see DATA) to be assigned starting with the first value in the table.

RETURN	Ends a current subroutine.
STOP	Ends a program.
USE	Saves variables to be used by many programs.

### BASIC SYSTEM COMMANDS

AUTO	Automatically numbers BASIC statements.
GO	Resumes execution of a MARK command or program that was halted.
LIST	Displays or prints the contents of storage.
LOAD	Loads storage with data from tape or data from the keyboard. Also see <i>Function Keys</i> in the <i>IBM 5100 BASIC Reference Manual</i> , SA21-9217.
MARK	Prepares a tape cartridge for programs or data to be saved.
MERGE	Combines programs on tape with programs in storage or data on tape with data in storage.
PATCH	Allows loading of patch program or tape recovery program or tape copy.
RD=	Specifies the number of digits at which rounding occurs for displayed or printed results.
RENUM	Renumbers the statements in storage.
REWIND	Rewinds the tape cartridge.
RUN	Executes a BASIC program.
SAVE	Saves the contents of storage on tape.
UTIL	Displays or prints a directory of the contents of the tape. Also transfers control to the communications feature.

### Editing Function

DEL	Deletes a statement or a group of statements from storage.
KEYx,	Allows editing of key groups, where x=0 to 9.

**O** ABS(x) absolute value of x 77  
**C** ACS(x) arc cosine of x (in radians) 78  
 adding to a tape file 83  
 addition 6, 16, 97  
 APL symbols 3, 4  
 array dimensions 84  
 array elements 84, 87, 93  
 arrays 84  
 arithmetic arrays 84, 86  
 arithmetic constants 22  
 arithmetic hierarchy 17  
 arithmetic operator keys 6  
 arithmetic operators 16  
 arithmetic with arrays 96  
 ASN(x) arc sine of x (in radians) 78  
 assigning values 24, 41, 58, 88, 92  
 ATN(x) arc tangent of x (in radians) 78  
 ATTN key 5, 14, 21, 30  
 AUTO command 35  
 automatic statement numbering 35

**C** backspace key 11  
 BASIC/APL switch 6  
 branching 44  
 brightness control 9

**C** calc result function 28  
 centimeters per inch (&INCM) 22  
 character arrays 84, 86  
 character variables 27, 47, 62  
 clearing storage 38  
 CLOSE statement 80  
 closing tape files 80  
 CMD key 5, 12, 13, 29  
 command keywords 6, 102  
 conversion constants 22, 78  
 conversion functions 78  
 copy display function 31  
 COS(x) cosine of x radians 78  
 COT(x) cotangent of x radians 78  
 CSC(x) cosecant of x radians 78  
 cursor 7

**D** DATA statement 58, 89  
 defining arrays 84, 86  
 DEG(x) degrees in x radians 78  
 DEL function 67  
 delete function 13  
 deleting characters 12  
 device address 80  
 DIM statement 84, 86  
 DISPLAY REGISTERS/NORMAL switch 6  
 display screen 7  
 displaying variable values 26  
 division 6, 16, 97  
**E** editing functions 67, 102  
 END statement 34  
 error correction 11, 30, 64  
 EXECUTE key 5, 30  
 EXP(x) natural exponent of x 79  
 exponentiation 16, 17  
 exponents 79

**F** flashing question mark 32, 61, 92  
 FOR statement 53, 90  
 formatting output 72  
 forward space key 10, 11

**G** GET statement 80  
 GO command 37  
 GOTO statement 34, 44

**H** HOLD key 5, 30, 52, 56

IF statement 33, 45, 51  
image statement 72  
IN PROCESS indicator 7  
initializing variables 47  
INPUT statement 32, 60, 90  
insert function 13  
inserting characters 13  
inserting program statements 64  
INT(x) integer part of x 77

keys 3, 4  
kilograms per pound (&LBKG) 22

LET statement 41, 88  
LGT(x) logarithm of x to the base 10 79  
LIST command 44  
liters per gallon (&GALI) 22  
LOAD command 7, 38, 39  
LOG(x) logarithm of x to the base e 79  
logarithms 79  
loops 50, 90  
LTW(x) logarithm of x to the base 2 79  
L32 64 R32 switch 6, 9

MARK command 37  
MAT INPUT statement 91  
MAT PRINT statement 93  
MAT READ statement 91  
mathematical functions 17  
multiplication 6, 16, 97

naming arrays 84, 86  
natural log (e) 22  
negative operations 17, 21  
nested loops 54  
NEXT statement 53, 90  
numeric keys 3, 4  
numeric variables 23, 47

OPEN statement 80  
opening tape files 80  
  
parentheses 17, 18  
pi ( $\pi$ ) 22  
positive operations 17, 21  
POWER ON/OFF switch 6, 38  
PRINT statement 32, 42, 61, 69  
PRINT USING statement 72  
printing arrays 93  
printing blank lines 75  
PROCESS CHECK indicator 7  
prompting message 61  
PUT statement 80

RAD(x) radians in x degrees 78  
raising to a power 16  
RD= command 32  
READ statement 58, 89  
ready message 7  
relational operators 45, 51  
REM (remark) statement 33, 42  
removing program statements 67  
RENUM command 68  
replacing characters 11  
replacing program statements 66  
RESET statement 82  
RESTART switch 6, 7, 38  
RESTORE statement 58  
REVERSE DISPLAY switch 9  
RND[(x)] random number 77  
rounding 32  
RUN command 31, 38  
RUN P=D command 75

**O** SAVE command 38  
scroll down key 10

scroll up key 10  
SEC(x) secant of x radians 78  
SGN(x) sign of x 77  
shift key 3

SIN(x) sine of x radians 78

special characters 3, 11

**C** square root 22, 23  
statement keywords 6, 30, 100

statement numbers 35, 68

status line 8

steps 54

STOP statement 48

storage capacity 7

subtraction 6, 16, 97

system functions 77

**TAN(x)** tangent of x radians 78

tape cartridge 35

tape files 35, 37, 80

**C** variables 23, 86



## READER'S COMMENT FORM

IBM 5100  
BASIC Introduction

SA21-9216-1

### YOUR COMMENTS, PLEASE . . .

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM branch office serving your locality.

Corrections or clarifications needed:

*Page*      *Comment*

I would like a reply.

Name \_\_\_\_\_  
Address \_\_\_\_\_

SA21-9216-1

— Cut Along Line —

Fold

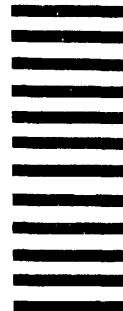
Fold

FIRST CLASS  
PERMIT NO. 387  
ROCHESTER, MINN.

**BUSINESS REPLY MAIL**

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .



IBM Corporation  
General Systems Division  
Development Laboratory  
Publications, Dept. 245  
Rochester, Minnesota 55901

Fold

Fold



**International Business Machines Corporation**  
**General Systems Division**  
**5775D Glenridge Drive N.E.**  
**Atlanta, Georgia 30301**  
**(USA Only)**

**IBM World Trade Corporation**  
**821 United Nations Plaza, New York, New York 10017**  
**(International)**

IBM 5100 BASIC Introduction

Printed in U.S.A.

SA21-9216-1

## READER'S COMMENT FORM

IBM 5100  
BASIC Introduction

SA21-9216-1

### YOUR COMMENTS, PLEASE . . .

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM branch office serving your locality.

Corrections or clarifications needed:

*Page*            *Comment*

I would like a reply.

Name \_\_\_\_\_

Address \_\_\_\_\_

SA21-9216-1

Cut Along Line

Fold

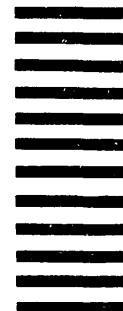
Fold

FIRST CLASS  
PERMIT NO. 387  
ROCHESTER, MINN.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .



IBM Corporation  
General Systems Division  
Development Laboratory  
Publications, Dept. 245  
Rochester, Minnesota 55901

Fold

Fold



International Business Machines Corporation  
General Systems Division  
5775D Glenridge Drive N.E.  
Atlanta, Georgia 30301  
(USA Only)

IBM World Trade Corporation  
821 United Nations Plaza, New York, New York 10017  
(International)

IBM 5100 BASIC Introduction Printed in U.S.A.

SA21-9216-1



International Business Machines Corporation

General Systems Division

5775D Glenridge Drive N.E.

Atlanta, Georgia 30301

(USA Only)

IBM World Trade Corporation

821 United Nations Plaza, New York, New York 10017

(International)