WANG

BASIC PROGRAMMING MANUAL

SYSTEM 2200

Jack Jarvis & Company, Inc.
707 S. W. Washington Street
Portland, Oregon 97205
Telephone: (503) 224-7838

2200 BASIC PROGRAMMING MANUAL

WANG LABORATORIES, INC.
836 NORTH STREET, TEWKSBURY, MASSACHUSETTS 01876, TEL.(617) 851-4111, TWX 710 343-6739, TELEX 94-7421

PREFACE

Wang Laboratories would like to take this opportunity to both congratulate and thank you for purchasing the Wang 2200 system. With its expandable memory (from 4k to 32k), and its powerful BASIC language, the 2200 system offers the user infinite programming possibilities.

The 2200 BASIC Programming Manual is an introduction to the 2200 itself, the BASIC language, and BASIC programming techniques.

In keeping with Wang's progressive philosophy, the 2200A system is expandable with the following peripherals: the 2215 (or 2222) Keyboard, the 2221 (or 2231) High-Speed Printer, the 2201 Output Writer, the 2216 CRT Display Module, the 2227 Telecommunications option and others. The Model 2200B in addition can have the following peripherals: the 2202 Plotting Output Writer, the 2203 Punched Paper Tape Reader, the 2214 Marked Sense Card Reader, the 2230–1 (or 2230–2, 2230–3) Disk Memory, the 2212 Flat Bed Plotter, and the 2207 Teletype Controller. Additional peripherals for the system are being developed and will be announced shortly.

The combination of its extensive memory, BASIC language, and vast peripheral choice makes the 2200 system unique in its class of calculators.

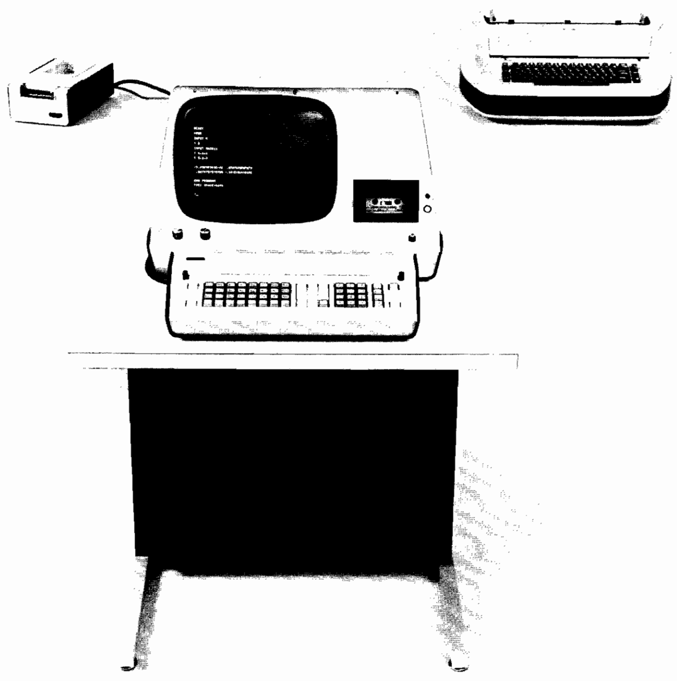


TABLE OF CONTENTS

Page

HOW TO USE THIS MANUAL

The 2200 BASIC Programming Manual is being provided with your 2200 System to enable you to instruct yourself in the operations of the 2200. The manual is designed for the user who is only slightly familiar with BASIC and is not at all familiar with the 2200.

This manual covers only instructions on the basic components of the 2200, namely the Central Processing Unit (2200 CPU), the Cathode Ray Tube Display (CRT—2216) and the BASIC Keyword Keyboard (2215). Instruction in the use of the remaining peripherals (i.e., Printers or Output Writers, etc.) is provided under separate cover for each peripheral purchased.

The Wang 2200 has two operating modes, the immediate mode and the programming mode. Parts I and II of the manual introduce the 2200 using the Immediate Mode, or one-line programming. All the techniques that can be done in the Immediate Mode are discussed; i.e., use of single statement lines and multi-statement lines, how to do simple and complex calculations, how to format the display, and how to do repetitive calculations in a single line (looping).

Once the techniques of the Immediate Mode are mastered, it is then a simple matter to progress to using the 2200 as a Programmable Calculator. Part III of the manual devotes itself entirely to writing Programs on the 2200 and the intricacies of the BASIC language hardwired into the 2200.

In addition to the 2200 BASIC Programming Manual, a Reference Manual is also provided with the equipment. It is not recommended that the Reference Manual be used for instruction purposes, but in the future as a quick refresher once you are familiar with the 2200 or as a means for familiarizing yourself with 2200 BASIC as used by the 2200.

In the Appendices of this Manual, additional information is provided. On the last page of this volume is a Customer Comment form which we at Wang hope you will use for comments and suggestions, then mail to us here at Tewksbury.

Part I
—The Basics of the 2200—

INTRODUCTION

Part I begins your introduction in the basics of the 2200. First, a description is provided on how to install and turn on your 2200 System. The remainder of Part I instructs you in how to enter and execute a single statement line in the Immediate Mode, as well as introducing beginning BASIC. Even though you may be familiar with the BASIC language it is recommended that Part I (and Part II) still be read to acquaint yourself with unique features of the 2200.

CHAPTER 1
EQUIPMENT INSTALLATION AND POWER ON PROCEDURE

A Typical 2200 System

CHAPTER 1
EQUIPMENT INSTALLATION AND POWER ON PROCEDURE

After unpacking and inspecting your equipment, the following procedure is used to install and turn on your 2200 System.

SECTION 1 – 1
INSTALLATION

The basic components of the 2200 are the Central Processing Unit (CPU) and the Power Supply. All other pieces of equipment are considered peripherals and are attached to the CPU. The CPU is divided into two main areas: the memory area and the peripheral attachment area. A connector cord from the CPU attaches to the power supply box. (Fig. 1—1).

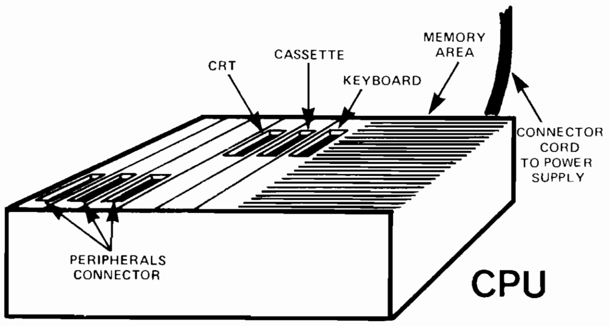
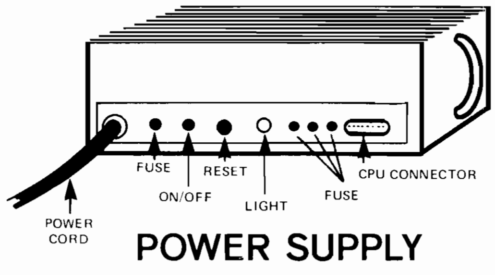


Fig. 1 – 1

The power supply box is described in Fig. 1—1a.

Fig. 1—1a



Install your system as follows:

1. Be sure the ON/OFF switch is OFF on the CPU.
2. Plug the main power cord from the Power Supply Box into a wall outlet.
3. Attach the power cord from the CPU to the Power Supply Box.
4. Plug any peripherals having a power cord (i.e. CRT) into a wall outlet.
5. Attach all peripherals to the CPU (i.e. Keyboard, CRT, Tape Cassette.)
   Be sure the locking clips are fastened when devices are plugged into the CPU.

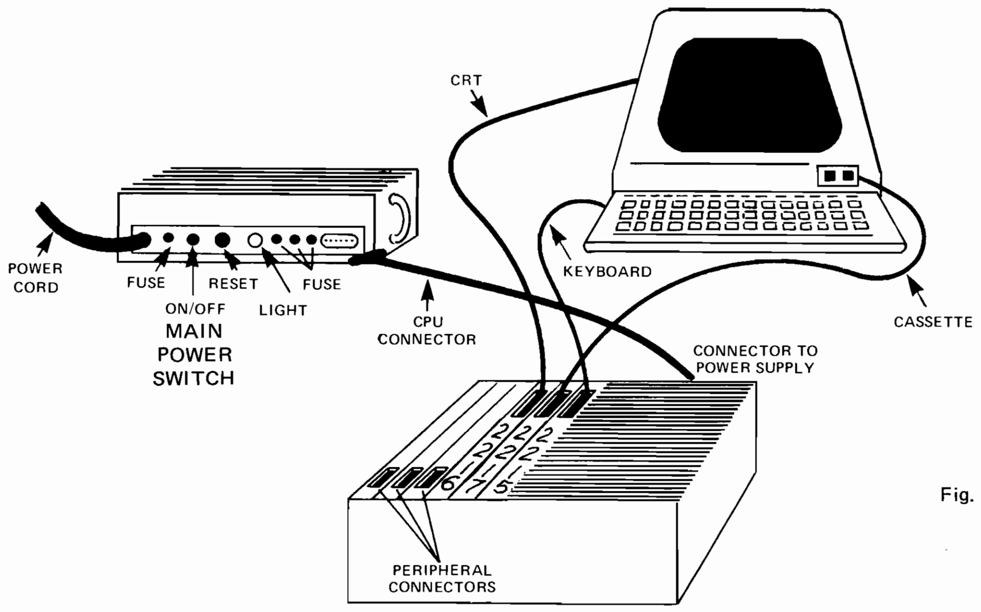


Fig. 1 – 1b

SECTION 1 – 2
"POWER ON" PROCEDURES

After installation is complete the following "power on" procedure is followed.

1. Turn the ON/OFF switches on the peripherals to the ON position (including CRT).
2. Move the ON/OFF switch on the power supply box to the ON position.

NOTE:

When the main power ON/OFF switch is turned ON the system is automatically initialized, that is the memory is cleared and the display appears as shown in Fig. 1–2. This process is called Master Initialization. The system is then ready to use.

```
READY
:_
```

Fig. 1–2

NOTE:

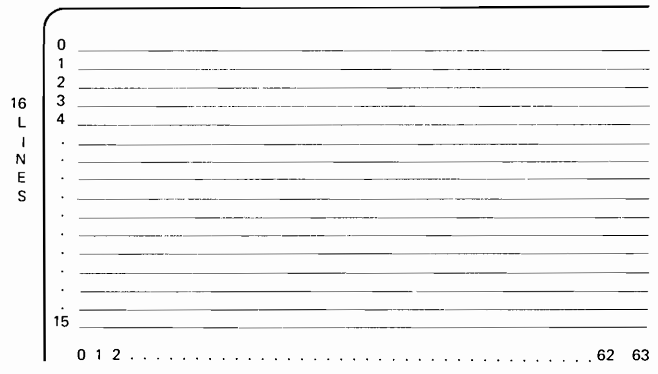
If READY does not appear immediately, leave power ON for 15 sec. and then turn the switch OFF, then ON again. READY will then appear on the CRT.

CHAPTER 2 AN INTRODUCTION TO THE 2200 CRT DISPLAY AND BASIC KEYWORD KEYBOARD

The basic 2200 unit is comprised of three parts – the Central Processing Unit, the Cathode Ray Tube (CRT) Display, and the BASIC Keyword Keyboard. The following sections describe the procedures for using the CRT Display and Keyboard to their best advantage, for solving problems.

SECTION 2 – 1
THE 2200 CRT DISPLAY

The CRT Display is designed to enable the user to more easily write programs and review results. The CRT unit itself is composed of an 8x10½ inch screen, and two controls used to set the brightness and contrast of the output as it appears on the screen. The screen itself has a maximum capacity of 16 lines, each 64 characters in length. (Fig. 2–1). If more than sixteen lines are entered at any one time, each new line is added at the bottom of the CRT, moving the previously entered lines up; the line at the top of the CRT display is replaced by the line directly beneath it.



64 SPACES EACH

Fig. 2 – 1

Figure 2–1a shows the display as it appears after the RESET key has been keyed, or after MASTER INITIALIZATION has taken place.

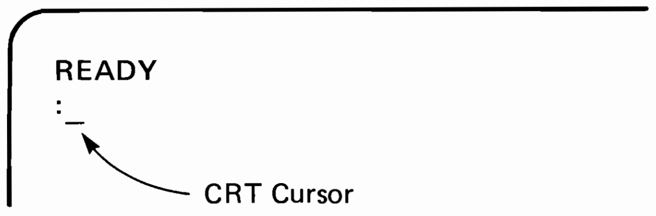


Fig. 2–1a

The combination of READY and the colon tells the operator that no processing is taking place within the 2200, and that the 2200 is now READY to accept new information. The colon must appear in the screen, in order to enter any information into the system.

Next to the colon is a short "hash" mark referred to as the "CRT Cursor". This mark denotes the location where the next input characters will be positioned in the display.

SECTION 2 – 2
THE 2215 BASIC KEYBOARD

THE PURPOSE OF THE 2200 BASIC SYSTEM COMMANDS

In order to use the 2200 effectively, it is necessary to understand the functions of the various sections of the 2215 keyboard. These include not only the 2200 BASIC keywords, mathematical functions and numbers, but what are referred to as "the 2200 BASIC System Commands", located around the perimeter of the keyboard. These commands are used to edit, execute and control the processing of all information entered into the system.

The keyboard is similar to a typewriter keyboard in that there are upper case and lower case keys. To obtain a capital letter on a typewriter you must touch the SHIFT key, which puts the typewriter into upper case, then touch the appropriate key. To obtain lower case letters the typewriter must be in lower case. To generate a single letter (e.g. A, or B) on the 2200, you must SHIFT before touching the appropriate key, as these characters are upper case. To generate certain BASIC key words (e.g. PRINT, END) you must be in lower case, therefore the SHIFT key isn't used. The use of the SHIFT key is explained throughout this manual.

The following explanation of the keyboard with examples is presented in the same logical order as the keystrokes used in evaluating a typical calculation.

THE RESET KEY

The RESET key is located in the upper right-hand corner of the keyboard.

Touch the RESET key.

This does three things:

1. Clears the CRT display, and prints the following in the display:

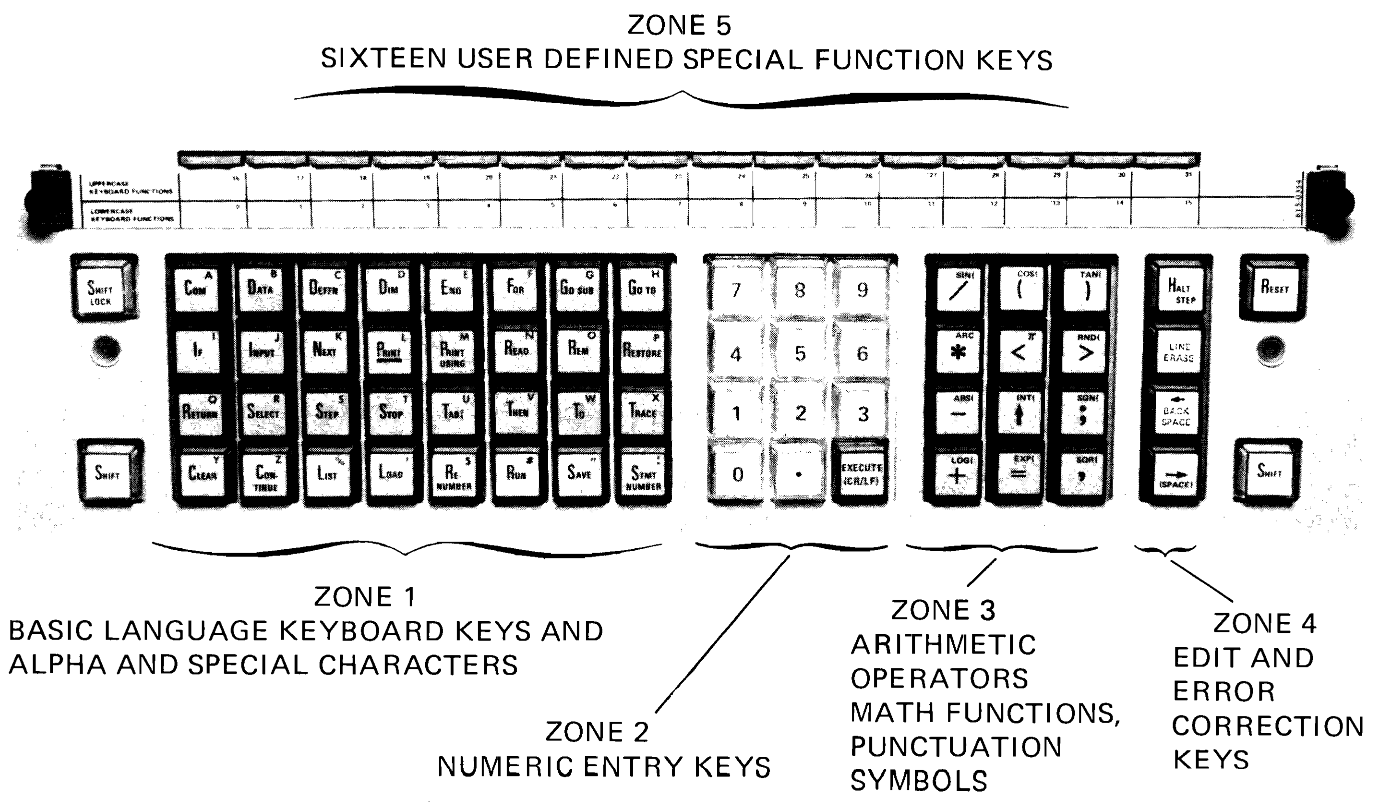
```
READY
:_
```

Fig. 2 – 2

2. Terminates *any* processing taking place.
3. In terminating any processing, the RESET command Unlocks the 2200 BASIC keyboard, allowing the user to enter new instructions from the keyboard. While any processing is taking place, the keyboard is locked.

   The RESET command does not alter the memory in any way.

THE THREE FUNCTIONAL GROUPINGS OF THE 2200 KEYBOARD


ZONE 5
SIXTEEN USER DEFINED SPECIAL FUNCTION KEYS
ZONE 1
BASIC LANGUAGE KEYBOARD KEYS AND ALPHA AND SPECIAL CHARACTERS
ZONE 2
NUMERIC ENTRY KEYS
ZONE 3
ARITHMETIC OPERATORS MATH FUNCTIONS, PUNCTUATION SYMBOLS
ZONE 4
EDIT AND ERROR CORRECTION KEYS

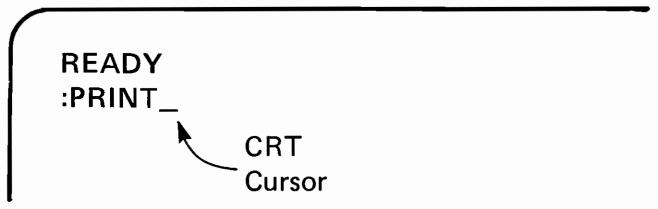
CHAPTER 2 AN INTRODUCTION TO THE 2200 CRT DISPLAY AND BASIC KEYWORD KEYBOARD

In the explanation of the 2200 keyboard, the following problem is solved:

> *Evaluate the expression*
>
> 36X8.25 + TANGENT OF 35 radians and print out the result, labeled "ANSWER".

THE 2200 BASIC KEYWORDS – each of which requires only one keystroke to generate – are the green keys in the 4 row block on the left-hand portion of the keyboard. (See photo on page 6.)

Touch the PRINT key. The CRT display looks like this



```
READY
:PRINT_
```

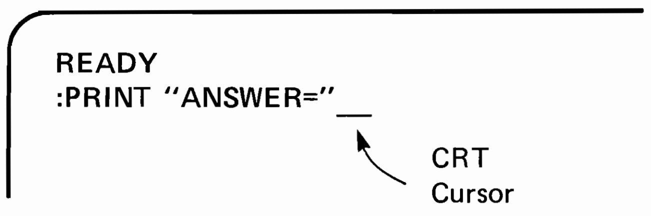
Fig. 2–2a

NOTE:

When using the 2200 as a calculator, the keyword PRINT must always be used to instruct the 2200 to print out the results of the calculations taking place.

All 2200 keywords are lower case (one keystroke, no SHIFT). The top three rows are presented alphabetically for convenience [1]. Each time one of these keys is touched, the entire keyword, plus a following space is displayed on the CRT, and entered into the 2200 system.

These same keys when used in upper case (SHIFT before touching) generate the alphabet, and a number of other 2200 BASIC symbols. For example: depress the SHIFT LOCK key and touch keys " A N S W E R SHIFT = SHIFT".



```
READY
:PRINT "ANSWER="_
```

Fig. 2–2b

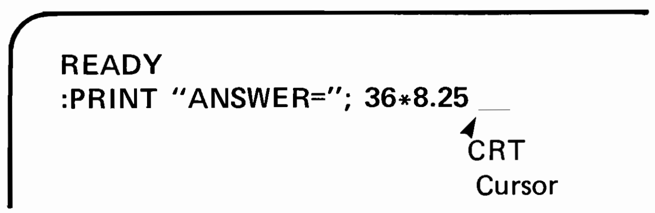
Touching the SHIFT LOCK key locked the keyboard in upper case. Touching the SHIFT key disengaged the SHIFT LOCK key, putting the keyboard back into lower case. For single keystrokes in upper case just touch SHIFT once before the appropriate key.

Any characters included within a set of quotation marks are collectively known as a *literal string.* Although literal strings are not required, they are often useful for labeling output. They are always printed out exactly as they are represented in quotes following a PRINT command.

THE NUMERIC KEYBOARD – the white keys found in the center of the keyboard used to enter all numeric data from the keyboard. (See photo on page 6.)

THE MATHEMATIC KEYBOARD FUNCTIONS – the green keys located to the right of the numeric keys. These include the plus (+), minus (–), multiply (*), divide (/), and power (↑) operations, left and right parentheses, and math functions.

Touch Key ;[2] 3 6 * 8 · 2 5



```
READY
:PRINT "ANSWER="; 36*8.25 _
```

Fig. 2–2c

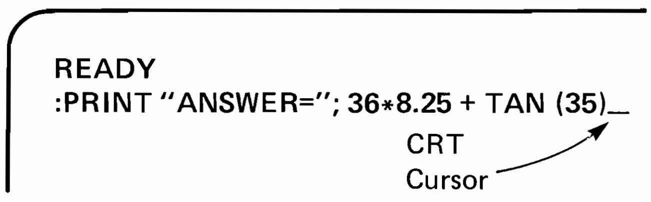
[1] Some of the keywords on the lower row (blue) of this block serve an additional function, as system controls. These keywords are explained in this text, as they are used.

[2] The use of the semicolon will be explained in a later chapter.

CHAPTER 2 AN INTRODUCTION TO THE 2200 CRT DISPLAY AND BASIC KEYWORD KEYBOARD

Each of the keys of the mathematic functions has an upper case function which is generated by depressing the SHIFT key first followed by the appropriate mathematic function key. Again the SHIFT is turned off after touching the desired key.

Touch keys + SHIFT TAN (35)



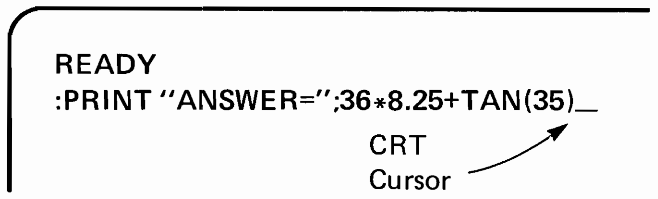
```
READY
:PRINT "ANSWER="; 36*8.25 + TAN (35)_
```

Fig. 2–2d

THE CR/LF–EXECUTE COMMAND [1]

The CR/LF–EXECUTE command is used to process all IMMEDIATE MODE calculations. This is accomplished by touching the CR/LF–EXECUTE key after the complete line has been entered.

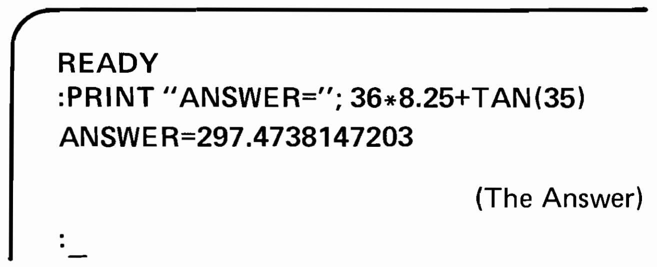
Continuing with the present example,



```
READY
:PRINT "ANSWER=";36*8.25+TAN(35)_
```

Fig. 2–2e

Touch the CR/LF-EXECUTE key.



```
READY
:PRINT "ANSWER="; 36*8.25+TAN(35)
ANSWER=297.4738147203

                              (The Answer)
:_
```

Fig. 2–2f

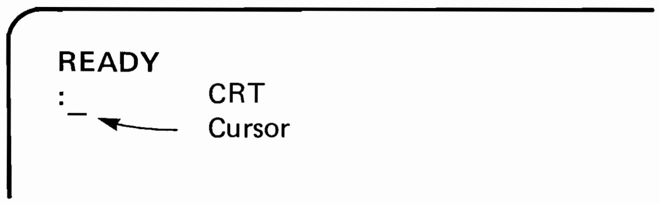
When the CR/LF-EXECUTE key is touched, the line is checked for BASIC grammatical correctness. If the line is not correct, an error message is generated, signalling the mistake. (See Chapter 4 for a full explanation of ERRORS and ERROR MESSAGES.)

If the line is correct the line is immediately executed, and cleared from the memory area. If there was a line number in front of the statement line, the 2200 then waits for its next instruction (as is the case in Programming, discussed in Part III of this manual).

After the line is executed, the cursor on the CRT moves to the first space of the next line and the processing light on the keyboard lights up. The new colon, ":", does not appear until processing has stopped and all output is displayed. The processing light goes out.

Thus, to use the 2200 as a calculator, a 2200 BASIC line must be entered into the 2200 via the keyboard each time it is to be executed, and the CR/LF-EXECUTE key then touched. While the 2200 is processing the line, all the keys on the keyboard (with the exception of the RESET key) are locked. Processing as a calculator continues until one of the following conditions occurs:

1. A result is arrived at implying that all instructions have been carried out. At this point, the keyboard is unlocked and a (:) appears on the CRT.
2. The RESET key is touched while the 2200 is processing. All processing terminates and the 2200 clears the CRT display and prints, READY (Fig. 2–2g).



```
READY
:_
```

Fig. 2–2g

3. A BASIC grammatical or operational error is discovered by the 2200. This causes the appropriate error message to be displayed, and

[1] CR/LF–EXECUTE stands for Carriage Return/Line Feed.

processing terminates. (Appendix A lists all the errors and error messages.)

TO SUMMARIZE – the colon must appear in the display before any line can be entered into the 2200. Either Master Initialization or RESET initializes the display. For an immediate calculation with a printout, the keyword PRINT must precede the calculation. To obtain the result of the line, the CR/LF-EXECUTE key is touched. Once this key is used, the line is lost, and must be rekeyed if the results are to be repeated.

THE CRT CONTROL KEYS

The CRT control keys enable the user to either "edit" any information *before* the CR/LF–EXECUTE key is touched or to obtain special unusual effects on the CRT under program control. The CRT control keys are discussed in this section and in more depth later in this manual.

There are three blue CRT control keys, located along the right-hand edge of the keyboard. (See photo on page 6.)

Touch the $\boxed{\text{space} \rightarrow}$ key several times. This causes the CRT Cursor to move one space to the right, each time the key is touched, enabling the user to enter spaces in 2200 BASIC statement lines.

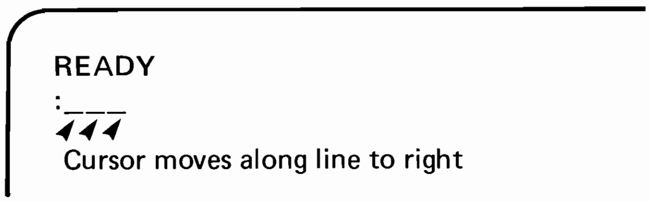


Fig. 2–2h

Touching this key at the end of a line causes the cursor to jump to the first space of the next line.

Touch the $\boxed{\begin{array}{c}\leftarrow\\ \text{BACK}\\ \text{SPACE}\end{array}}$ key several times. This causes the CRT cursor to move one space to the left, each time the key is touched.

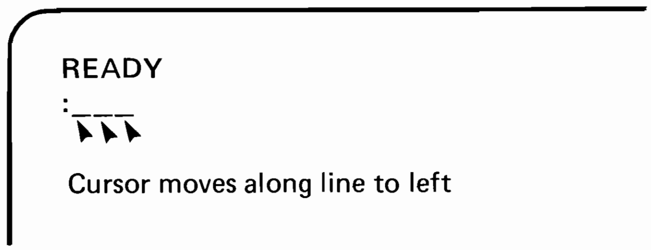


Fig. 2–2i

Now, touch the PRINT, SHIFT and A keys. The CRT display is now

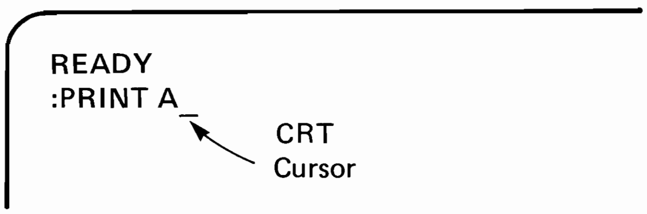


Fig. 2–2j

Touch the ← key once. This deletes only the character "A" from the CRT display.

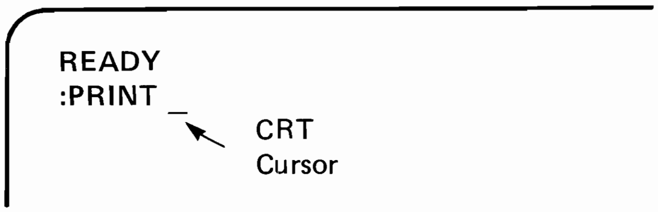


Fig. 2–2k

Touching this key while entering a line causes the CRT cursor to backspace a single space to the left and delete the last keystroke entry from the CRT.

Touch the ← key again. This time, the entire keyword PRINT is deleted because the keyword was entered by touching a single key.

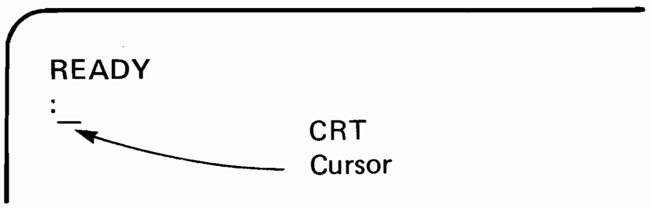


Fig. 2–2l

Backspacing to delete with the ← key deletes either a character or a whole word depending upon how each was generated. The word PRINT when generated by the PRINT key, is considered as a single character and is deleted by striking the

← key only once. But if the word PRINT was generated with upper case letters (e.g. P R I N T) then each time the ← key is touched only one character at a time is erased.

Key PRINT "ANSWERS ARE AS FOLLOWS"

```
READY
:PRINT "ANSWERS ARE AS FOLLOWS"_
                         CRT
                         Cursor
```

Fig. 2–2m

Touch $\boxed{\text{LINE ERASE}}$ key. This causes the display to show

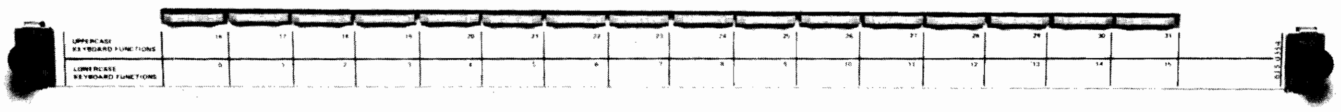
```
READY
:_
        CRT
        Cursor
```

Fig. 2–2n

Touching the $\boxed{\text{LINE ERASE}}$ key erases the entire line at which the cursor is located.

Both the ← key and $\boxed{\text{LINE ERASE}}$ key work as described in both upper and lower case.

THE SPECIAL FUNCTION KEYS



Located at the top of the 2200 BASIC keyboard is a row of 16 keys, known collectively as the Special Function keys. These keys enable the 2200 user to "customize" his calculator to meet specific programming requirements, and to facilitate the entry of special groups of characters, frequently entered into the 2200. Chapter 19 describes the various uses of these keys, and the purposes intended for each.

CHAPTER 2 AN INTRODUCTION TO THE 2200 CRT DISPLAY AND BASIC KEYWORD KEYBOARD

EXERCISES

1. Using the 2200 as a calculator, perform the following calculations. (Be sure to touch the PRINT key each time a new line is entered):

   a. 14 + 6
   b. 8 * 6
   c. 8 - 12.66
   d. - 18 * 4.55
   e. 96/853
   f. 5 ↑ 3
   g. 26 + 5 + 12 + 10
   h. 7 - 12.6 + 8 - .002
   i. 8 * 10 + 6 * 4
   j. 144 ↑ .5

2. Do the following, using the 2200 as a calculator.

   a. i Print the sum of 86.2 and 155.86
      ii Print the result of 8522 minus 1498
      iii Print the product of -57 and 16.6
      iv Print the quotient of 20 divided by 4.25
      v Print the result of 17.3 raised to the 1.6 power
   b. Find the sum of the 10 integers,1 thru 10.
   c. Find the product of the twelve integers, 1 thru 12.
   d. Assuming 365 days in a year, 24 hours in a day, find the number of seconds in a year.

ANSWERS

KEYSTROKES	ANSWER
1. a. PRINT 14+6 CR/LF-EXECUTE	20
b. PRINT 8*6 CR/LF-EXECUTE	48
c. PRINT 8-12.66 CR/LF-EXECUTE	-4.66
d. PRINT -18*4.55 CR/LF-EXECUTE	-81.9
e. PRINT 96/853 CR/LF-EXECUTE	.1125439624853
f. PRINT 5↑3 CR/LF-EXECUTE	125
g. PRINT 26+5+12+10 CR/LF-EXECUTE	53
h. PRINT 7-12.6+8-.002 CR/LF-EXECUTE	2.398
i. PRINT 8*10+6*4 CR/LF-EXECUTE	104
j. PRINT 144↑.5 CR/LF-EXECUTE	12
2. a. i PRINT 86.2+155.86 CR/LF-EXECUTE	242.06
ii PRINT 8522-1498 CR/LF-EXECUTE	7024
iii PRINT -57*16.6 CR/LF-EXECUTE	-946.2
iv PRINT 20/4.25 CR/LF-EXECUTE	4.705882352941
v PRINT 17.3↑1.6 CR/LF-EXECUTE	95.691588717
b. PRINT 1+2+3+4+5+6+7+8+9+10 CR/LF-EXECUTE	55
c. PRINT 1*2*3*4*5*6*7*8*9*10*11*12 CR/LF-EXECUTE	479001600
d. PRINT 365*24*60*60 CR/LF-EXECUTE	31536000

CHAPTER 3
USING THE 2200 AS A CALCULATOR

The material in the following Section explains the use of parentheses and the order of execution of algebraic expressions. The 2200 follows all the standard accepted rules associated with algebra. Even though you may be familiar with algebraic ordering, it is recommended that this section be read.

SECTION 3 – 1
ORDER OF EXECUTION AND THE USE OF PARENTHESES

The exercises included at the end of the last section demonstrated the use of the basic five arithmetic operations: Addition, subtraction, multiplication, division, and exponentiation. In those exercises, for the most part, there was no question about the priority, or order of execution of the arithmetic operators used. As long as the operators are not mixed within an expression, the expression is simply evaluated left to right.

In most cases however, mathematical expressions involve several different operators. For example, consider the expression:

W *X↑Y - Z

How is this expression evaluated? The table below provides the answer

Table 3–1

Operation	Symbol	Order Of Execution (Priority)
Exponentiation	↑	Computed 1st
Division multiplication	/ *	Computed 2nd
Subtraction Addition	- +	Computed 3rd
Using the above priorities, all expressions are evaluated left to right.		

Thus the order of execution of W * X ↑ Y - Z is:

First, X is raised to the power Y. Second, the result is then multiplied by W. Third, and finally, Z is subtracted from the product.

This answers the questions of the order of execution. However, suppose this is not the intended order. In this case, parentheses must be used to indicate the order of execution intended. Thus, if the product of W * X is to be raised to the power Y, the expression would be written as

(W *X) ↑ Y - Z

Fig. 3–1

Or, if X is raised to the power (Y–Z), the expression would be written as

W * X ↑ (Y–Z)

Fig. 3–1a

or, if W is to be multiplied by X↑Y–Z, the expression is written as

W * (X ↑ Y - Z)

Fig. 3–1b

It is evident then, that parentheses are used to alter the order of execution. Parentheses indicate that the enclosed quantities are to be evaluated first. When parentheses are used in an expression, the order of execution becomes:

Table 3 – 2

Operation	Symbol	Order of Execution (Priority)
Expressions within Parentheses	( )	Computed 1st
Exponentiation	↑	Computed 2nd
Division Multiplication	/ *	Computed 3rd
Subtraction Addition	- +	Computed 4th
Using the above priorities, all expressions are evaluated left to right.		

Parentheses have an additional use as well. To calculate 5^{-3}

Touch RESET

Key PRINT 5 ↑ -3 CR/LF-EXECUTE

The CRT display is

```
READY
:PRINT 5 ↑ -3
          ↑ ERR 15

:_
```

CRT
Cursor

Fig. 3–1c

Two mathematical operator symbols cannot appear next to each other; they must be separated using parentheses. Touch RESET and rekey the problem.

```
READY
:PRINT 5 ↑ (-3)
 8.00000000 E- 03

:_
```

Fig. 3–1d

Multiple sets of parentheses can be used as well. For example, in the following expression, three sets of *nested* parentheses are used.

(((7.3 + 4.2) ↑ 2 + 6) ↑ .5 + 17)/22

Fig. 3–1e

An unlimited amount of nesting of parentheses is allowed on the 2200. Parentheses can and should be used whenever there is any question as to the order of execution of an expression. Their use assures that the expression is executed exactly as intended.

However, in using parentheses, there are two rules which must be followed:

First, parentheses must always be balanced – there must be an equal number of right and left parentheses.

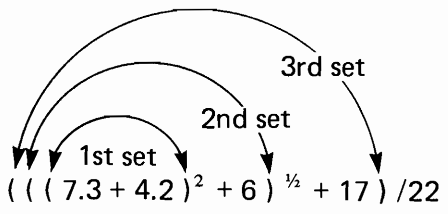


Fig. 3–1f

Second, implied multiplication is not allowed; that is, the expression X * (Y + Z) is correct, while X (Y + Z) is not. When multiplication is intended, the multiply key, (*), must be used.

SECTION 3 – 2
KEYBOARD FUNCTIONS

In addition to the arithmetic operators, 2200 BASIC includes a set of 12 commonly used mathematical functions, found as upper case on the right-hand green keys of the 2200 keyboard. Notice that, with the exception of the π and ARC functions, a left hand (open) parenthesis is included with each function. To use one of these functions then, a right hand (closed) parenthesis must be keyed following the last element of the function argument. To generate an ARCSIN(, ARCCOS(, or ARCTAN(function, the ARC key is touched prior to the appropriate trigonometric function key. Table 3–3 gives an entire list of these keyboard functions.

Table 3 – 3

Keyboard Function	Meaning	Example
[1] SIN(expression)	Find the sine of the expression	SIN(π/3) = .8660254037841
[1] COS(expression)	Find the cosine of the expression	COS(.693↑2) = .8868799122686
[1] TAN(expression)	Find the tangent of the expression	TAN(12)= –.6358599286636
ARC SIN(expression)	Find the arcsin of the expression	ARC SIN (.003)= 3.00000450E-03
ARC COS(expression)	Find the arccosine of the expression	ARC COS (.587)= .943448079441
[2] ARC TAN(expression)	Find the arctangent of the expression	ARC TAN (3.2)= 1.26791145842
	Assign the value (3.14159265359)	4∗#PI=12.56637061436
	Produce a random number between 0 and 1	RND(X)=.8392246586193
ABS(expression)	Find the absolute value of the expression	ABS(7∗3.4+2)= 25.8 ABS(–6.537)=6.537
INT(expression)	Take the greatest integer value of the expression	INT(3.6)=3 INT(–5.22)=–6
SGN(expression)	Assign the value 1 to any positive number, 0 to zero, and –1 to any negative number	SGN(9.15)=1 SGN(0)=0 SGN(–.124)=–1
LOG(expression)	Find the natural logarithm of the expression	LOG(3052)= 8.023552392402
EXP(expression)	Find the value of e raised to the value of the expression	EXP(.33∗(5–6))= .7189237334321
SQR(expression)	Find the square root of the expression	SQR(18+6)=SQR(24)= 4.8989794856

[1] Unless instructed otherwise the function is interpreted in radians. To use degrees, touch SELECT D CR/LF–EXECUTE once. All following trigonometric expressions are then interpreted as degrees. To use grads, touch SELECT G CR/LF-EXECUTE. To reset the 2200 to radian measure, touch SELECT R CR/LF–EXECUTE, or switch the 2200 OFF and ON.

[2] The arctangent notation ATN(is also a recognized function notation, but must be keyed in directly from the keyboard.

CHAPTER 3
USING THE 2200 AS A CALCULATOR

EXAMPLES OF USE OF KEYBOARD FUNCTIONS

1. Find the $\sqrt{114.6+53.47}$
   Key PRINT SHIFT SQR(114.6/53.47) CR/LF-EXECUTE
   The CRT display shows

```
READY
:PRINT  SQR  (114.6/53.47)
 1.4639869882
```

Fig. 3–2

2. Find Log_e 10
   Key PRINT SHIFT LOG(10) CR/LF-EXECUTE

```
READY
:PRINT  LOG(10)
 2.302585092994
```

Fig. 3–2a

3. Find the SIN of 3.289 radians.
   Key PRINT SHIFT SIN(3.289) CR/LF-EXECUTE

```
READY
:PRINT  SIN(3.289)
 -.14687409221
```

Fig. 3–2b

Unless instructed otherwise, trigonometric functions are interpreted in radians. To use degrees, key SELECT D CR/LF–EXECUTE once, before entering the problem. All following trigonometric arguments are then interpreted as degrees. The 2200 can be put into Gradian mode by keying SELECT G CR/LF–EXECUTE.

To reset the 2200 to radian measure from either degrees or gradians, key SELECT R CR/LF–EXECUTE, or switch the 2200 to OFF and ON to reinitialize.

4. Find the COSINE 48°
   Key SELECT SHIFT D CR/LF-EXECUTE
   PRINT SHIFT COS(48) CR/LF-EXECUTE

```
READY
:SELECT D
:PRINT COS(48)
 .669130606358
```

Fig. 3–2c

Return the 2200 to Radian mode by keying SELECT SHIFT R CR/LF–EXECUTE.

5. Find the absolute value of the expression

$$\frac{1.68^2 - 46}{28.5}$$

   Key PRINT SHIFT ABS ((1.68↑2–46)/28.5) EXECUTE

```
READY
:PRINT ABS ((1.68↑2–46)/28.5)
 1.515003508772
```

Fig. 3–2d

SECTION 3 – 3
NUMBERS AND NUMERIC DATA
FLOATING POINT NUMBERS

Up to this point, entering numbers into the 2200 has been accomplished via the numeric keyboard, by keying digits and decimal point in the appropriate sequence. Thus, in entering the number 135.68, the required keystrokes are 1 3 5 . 6 8 in that order. Similarly, entering the number –.0085 requires the keystrokes – . 0 0 8 5 .

In cases such as these, the sign (where necessary) and digits have been entered in a fixed sequence, with the decimal point in its true position. Numbers entered in this manner are known as *fixed point numbers*, and their format is referred to as *fixed point format.*

In fixed point format, numbers with up to thirteen (13) digits, a decimal point, and a sign may be entered into the 2200. When entering a number greater than zero, a plus sign is implied, and need not be entered.

SCIENTIFIC NOTATION

While fixed point format enables the user to enter numbers as large as 9999999999999., or as small as .0000000000001, there are obvious limitations. Not only is 13 digits limiting in size in many cases, but awkward to use as well (to be certain there are the correct number of digits, they all have to be counted).

To alleviate these problems, another format, referred to as floating point, may be used with 2200 BASIC. When *floating point* is used, the number is represented as a fixed point number, multiplied by an integral power of ten. Examples of numbers represented in floating point are:

6.02×10^{24} 5.1×10^{-5}

195×10^{18} $.016 \times 10^{5}$

Fig. 3–3

Notice that in floating point, the decimal point is optional, and as mentioned before, the power of 10 is an integer. Also, for numbers greater than zero, a plus sign is assumed in both the exponent and the fixed point portion of the number, if a sign is not entered. When a floating point number is written with the decimal point after the first non-zero digit (e.g. 5.64E3 as opposed to 56.4E2 or .0564E5) it is said to be in Scientific Notation.

Using floating point in the 2200 requires the use of the letter "E" to signify that an exponent of 10 is being entered. To generate the letter "E", the SHIFT and $\boxed{\text{END}^{\text{E}}}$ keys are used.

Print out the number 5.675×10^{4}
Key PRINT 5.675E4 CR/LF–EXECUTE

```
READY
:PRINT 5.675E4
 56750

:_
```

Fig. 3–3a

Print out the number 15.9×10^{-8}
Key PRINT 15.9E–8 CR/LF–EXECUTE

```
READY
:PRINT 15.9E-8
 1.59000000E-07


:_
```

Fig. 3–3b

Other examples of the correct use of floating point are given in Figure 3–3c.

CHAPTER 3
USING THE 2200 AS A CALCULATOR

NUMBERS AND NUMERIC DATA
VALID USE OF FLOATING POINT NOTATION

6.02×10^{24}	entered as	6.02E24	printed as	6.02000000E+24
195×10^{18}	entered as	195E18	printed as	1.95000000E+20
5.1×10^{-5}	entered as	5.1E-5	printed as	5.10000000E-05
$.016 \times 10^{18}$	entered as	.016E18	printed as	1.60000000E+16
-1.5683×10^{40}	entered as	-1.5683E40	printed as	−1.56830000E+40
$.00641 \times 10^{5}$	entered as	.00641E5 which is equal to 6.41E2	printed as	641

Fig. 3–3c

When entering numbers in floating point notation, each can include up to thirteen digits, a decimal point and sign, and a two-digit positive or negative exponent. However, the keyword PRINT will display only the first nine digits in scientific notation, though the remaining digits are kept internally.

The largest exponent which 2200 BASIC will accept is E99. The smallest exponent is E−99. The values of the exponents must always be integers — no decimals or fractions are allowed. Examples of invalid numbers are given in Figure 3–3d

INVALID USE OF SCIENTIFIC NOTATION

8.7E5.8	Not valid because of the illegal decimal form of the exponent.
103.2E99	Not valid because in reduced form it is equivalent to 1.032E101, an exponent greater than E99.
.87E-99	Not valid because it is equivalent to 8.7E−100.

Fig. 3–3d

CHAPTER 3
USING THE 2200 AS A CALCULATOR

EXERCISES

Evaluate the following expressions on the 2200. Use the keyboard functions wherever possible.

i 575^2

ii $.00575^3$

iii $(-35)^2$

iv $\frac{1}{35}^{4.2}$

v $3^3 \cdot 5^5$

vi $7(8^3 + 5{\cdot}9^2$

vii $\frac{5^2}{6} + \frac{5}{6^2} + \frac{5^2}{6^2}$

viii $\frac{5 \cdot 6^2}{5 + 6^2}$

ix $(2^5)^4$

x $\sqrt{36}$

xi $\sqrt{\frac{1}{36}}$

xii $\sqrt{5 + \sqrt{8 * 9^2}}$

xiii Tangent $\pi/4$ radians

xiv Arc Sine .5

xv Log of $\sqrt{19.5}$

xvi Integer $3.8^2 - \left(\frac{2}{.3^2}\right)^2$

xvii Absolute value of $(18.2 - 16^2)$

xviii Sign of the expression $\frac{\sqrt{14.5-6}}{15.8 \cdot \text{cosine } 22 \text{ radians}}$

CHAPTER 3
USING THE 2200 AS A CALCULATOR

	Problem	Keystrokes	Answers
i	575^2	Print 575↑2 =	330625
ii	$.00575^3$	Print .00575↑3 =	1.90109375E-07
iii	$(-35)^2$	Print (−35)↑2 =	1225
iv	$\frac{1}{35}^{4.2}$	Print (1/35)↑4.2 =	3.27276041E-07
v	$3^3 \cdot 5^5$	Print 3↑3*5↑5 =	84375
vi	$7(8^3+5 \cdot 9^2)$	Print 7*(8↑3 + 5*9↑2) =	6419
vii	$\frac{5^2}{6} + \frac{5}{6^2} + \frac{5^2}{6^2}$	Print 5↑2/6 + 5/6↑2 + 5↑2/6↑2 =	4.999999999999
viii	$\frac{5 \cdot 6^2}{5 + 6^2}$	Print 5*6↑2/(5+6↑2) =	4.390243902439
ix	$(2^5)^4$	Print 2↑5↑4 =	1048576
x	$\sqrt{36}$	Print SQR(36) =	6
xi	$\sqrt{\frac{1}{36}}$	Print SQR(1/36) =	.16666666667
xii	$\sqrt{5+\sqrt{8*9^2}}$	Print SQR(5+SQR(8*9↑2)) =	5.5186813754
xiii	Tangent $\pi/4$ radians	Print TAN #PI/4) =	1
xiv	Arc Sine .5	Print ARC SINE (.5) =	.5235987755982
xv	Log of $\sqrt{19.5}$	Print Log(SQR(19.5)) =	1.485207232792
xvi	Greatest integer $3.8^2 - \left(\frac{2}{.3^2}\right)^2$	Print INT(3.8↑2 −(2/.3↑2)↑2) =	−480
xvii	Absolute Value of $(18.2 - 16^2)$	Print ABS(18.2 − 16↑2) =	237.8
xviii	Sign of the expression $\frac{\sqrt{14.5-6}}{15.8 \cdot \text{COS (22 radians)}}$	Print SGN(SQR(14.5−6)/15.8*COS (22)) =	−1

CHAPTER 4
2200 BASIC ERRORS AND ERROR MESSAGES

The BASIC language was chosen for the 2200 because it is both powerful and easy to use. However, 2200 BASIC does require the user to follow certain rules. For example there are restrictions on the formats of numbers, and on the structure of BASIC statements. There are also restrictions on the magnitude of numbers used as arguments of functions.

To assure that the language is used properly, an extensive set of error detectors and identifiers is incorporated into the system. If an error occurs, the 2200 notes the fact by displaying an error message at the location where the problem is found, and stops all processing. Appendix A explains all the error diagnostics, and after the programmer has decided on the cause of the error, he may reenter the line correctly.

SECTION 4–1
ERRORS AND WHAT CAN BE DONE ABOUT THEM

Altogether, there are three types of errors a 2200 user can make:

1. A SYNTAX ERROR – an improper or illegal format is used in an entered BASIC statement. Examples include the improper use of the BASIC keywords, and unbalanced parentheses.
2. AN ERROR OF EXECUTION – an error resulting from the execution of an otherwise legal BASIC statement. Examples include such things as attempting to take the square root of a negative number.
3. A PROGRAMMING ERROR – The 2200 executes the statement, but the results obtained are not correct, because the wrong information or logic is used by the programmer.

Of these three types, 2200 BASIC error messages cover the first two as follows:

A SYNTAX ERROR results when the required form of a 2200 BASIC statement is violated. Pressing a sequence of keys not recognized as an accepted combination results in this type of error. In general, an error message is displayed as soon as the CR/LF–EXECUTE key is touched. Examples of this type of error include the improper use of verbs, improper number format, the improper use of operators and parenthesis, and the improper use of punctuation.

ERROR EXAMPLE 1:

Key PRINT 3 * SQR (17 CR/LF–EXECUTE

The CRT display shows

```
READY
:PRINT 3 * SQR(17
                ↑ERR 05

:_
```

Fig. 4–1

THE PROBLEM: A missing right parenthesis.

THE SOLUTION: Reenter the line correctly, with balanced parentheses.

ERROR EXAMPLE 2:

Key PRINT PRINT 3↑6.2 CR/LF–EXECUTE

The CRT display shows

```
READY
:PRINT PRINT 3↑6.2
       ↑ERR 15

:_
```

Fig. 4–1a

THE PROBLEM: The PRINT command appears twice in the same statement.

THE SOLUTION: Reenter the statement correctly, using PRINT only once.

ERROR EXAMPLE 3:

Key PRINT 56849.065825785↑.3 CR/LF–EXECUTE

```
READY
:PRINT 56849.065825785↑.3
                          ↑ERR 20

:_
```

Fig. 4–1b

THE PROBLEM: More than 13 digits in the entered number.

THE SOLUTION: Reenter the statement with less digits.

AN ERROR OF EXECUTION results when an illegal arithmetic operation is performed, or the execution of an illegal statement or programming procedure is attempted. This type of error differs from a Syntax Error in the fact that the statement itself uses the proper syntax; however, the *execution* of the statement leads to an error condition.

ERROR EXAMPLE 4:

Key PRINT 18/(5–5) CR/LF–EXECUTE

```
READY
:PRINT 18/ (5–5)
PRINT 18/(5-5)
              ↑ERR 03
:_
```

Fig. 4–1c

THE PROBLEM: Division by zero – a math error.

THE SOLUTION: Recheck the mathematics and reenter the corrected statement.

ERROR EXAMPLE 5:

Key PRINT 1.23E60/4.95E–50 CR/LF–EXECUTE

```
READY
:PRINT 1.23E60/4.95E-50
PRINT 1.23E60/4.95E-50
                      ↑ERR 03

:_
```

Fig. 4–1d

THE PROBLEM: Exponent overflow: the resulting magnitude of the number calculated is greater than 10^{99}.

THE SOLUTION: Recheck the mathematics and reenter the corrected statement.

THE PROGRAMMING ERROR

Assuming that a 2200 BASIC statement or program is free of syntax and execution errors, it is possible that a programming or logic error has been made. For example, if, instead of using

$$\texttt{(-B+SQR(B↑2-4*A*C))/(2*A)} = \frac{-b+\sqrt{b^2-4ac}}{2a}$$

Fig. 4–1e

the following is used:

$$\texttt{-B+SQR(B↑2-4*A*C)/2*A} = -b+\frac{\sqrt{b^2-4ac}}{2a}$$

Fig. 4–1f

would not necessarily be correct.

(Notice the missing parentheses in Fig. 4–1f.) Therefore the 2200 does only what it is told and if a programming error has occurred the 2200 has no way of knowing it.

Part II
—One Line Programming—

INTRODUCTION

Up to this point, all problems and solutions using the 2200 have revolved around the use of single BASIC statements, composed solely of real numbers. Although this type of problem is very common, the material in Part II demonstrates it is by no means the only type of problem found, nor the only type of problem the 2200 is capable of solving. Likewise, the single statement line is also not the only way in which the 2200 may be instructed to solve these problems.

Part II of the 2200 BASIC Programming Manual introduces using the 2200 under program control. These concepts are then covered in depth in Part III of this manual. Included in Part II is an introduction to the use of variables, punctuation in PRINT statements for varying printout formats, and the use of looping techniques.

Basic to this entire section of the manual is the concept of the 2200 *statement line.* A statement line is simply a 2200 BASIC line, composed of one or more BASIC statements. Statements in a statement line are separated by colons, generated by touching the SHIFT, and $\boxed{\text{STMT : NUMBER}}$ keys.

For example, enter the following statement line

```
PRINT 15:PRINT SQR(15):PRINT 15↑(1/3) CR/LF-EXECUTE
```

```
READY
:PRINT 15:PRINT SQR(15):PRINT 15↑(1/3)
 15
 3.8729833462
 2.4662120743
:_
```

Notice that there are three separate statements in the line, and that the statements are executed sequentially. The results are printed in a column under the statement line. In a multiple statement line, the colons serve to denote the beginning of a new statement.

CHAPTER 5
2200 BASIC VARIABLES

SECTION 5–1
THE ASSIGNMENT STATEMENT

Key X = 5 + 19: PRINT X CR/LF–EXECUTE

```
READY
:X = 5 + 19: PRINT X
 24

:_
```

Fig. 5–1

Notice that this line is actually a multi-statement line composed of two separate BASIC statements. In the first statement, the variable "X" is given or *assigned* a value (The expression "5 + 19"). This type of statement is thus known as an *assignment* statement, because the variable on the left is *assigned* a value. The second statement simply says to print out the value of X.

The colon ":" is used to indicate the start of the second statement. Notice that in the assignment statement, an equal sign "=" is used, and that the variable (in this case "X") assigned a value, is on the left side of the equality sign. In an assignment statement, the variable being assigned a value is *always* on the left side of the equality sign. Conversely, the expression or value assigned to the variable is *always* on the right side of the equality sign. The value assigned can be an expression, a constant or another variable. The following section, discussing 2200 BASIC variables illustrates how this is accomplished.

NOTE:
In the BASIC language, the verb LET is quite often used in an assignment statement. Thus the statement line

X = 5 + 19: PRINT X

could also be written

LET X = 5 + 19:PRINT X

with the SAME RESULTS. However, the verb LET is OPTIONAL. This keyword is not included among the 2200 BASIC keyword block. To generate the verb LET, key SHIFT LOCK, L, E, T, SHIFT.

SECTION 5–2
2200 VARIABLES AND VARIABLE NAMES

In general, a variable in 2200 BASIC is a set of characters which represents a data value in the 2200 system. The value assigned to the variable does not change until it is either assigned a new value by an assignment statement, or until the variable is cleared from the system, by using the CLEAR command, as described in Section 5–3. Once the variable is named the 2200 automatically sets aside a storage location for that value.

Although there are many types of 2200 BASIC variables, the most often used are what are known as numeric *scalar variables.*

DEFINITION – A numeric scalar variable is a variable which can represent only one numeric value at a time (as compared to an array variable described in Chapter 14.) The value assigned may be any real number within the range of the 2200 ($\pm 10^{-99}$ to $\pm 10^{+99}$).

In the 2200 BASIC a numeric scalar variable is designated by a letter or a letter followed by a digit. There are 286 variable names available in the 2200 system (A–Z, A0–Z9).

LEGAL NUMERIC SCALAR VARIABLE NAMES

X	B	1 letter or 1 letter followed by one digit
W	A4	
A	Z2	

Fig. 5–2

ILLEGAL NUMERIC SCALAR VARIABLE NAMES

XX WL 5M C75

Fig. 5–2a

Numeric scalar variables can be used with all the arithmetic (+, –, *, /, ↑) operators. Until a numeric scalar variable is used it is not in the system, and no memory area is reserved for it.

Key PRINT Y CR/LF–EXECUTE

The CRT display shows

```
:PRINT Y
 0
: _
```

Fig. 5–2b

The reason a value of zero is printed for "Y" is simply because no other value was given to "Y" by the program. When the 2200 is instructed to print its value, the variable is automatically assigned a value of "0". Therefore, undefined variables automatically take on the value of zero.

Key PRINT X CR/LF–EXECUTE

```
READY
:PRINT X
 24

:_
```

Fig. 5–2c

The value 24 is printed for the variable "X" if CLEAR CR/LF–EXECUTE had not been keyed, since the statement line in Figure 5–1 was executed, and no other value was assigned to the variable "X". The memory retained the value assigned to X. Figure 5–2e gives examples of numeric scalar variables and assignment statements.

A slight variation of the assignment statement enables the user to assign several variables the same value in one statement:

Key X,Y,Z=3.8: PRINT X: PRINT Y: PRINT Z CR/LF–EXECUTE

```
READY
:X,Y,Z=3.8:PRINT X: PRINT Y: PRINT Z
 3.8
 3.8
 3.8
```

Fig. 5–2d

Notice that the variables above are separated by commas. The comma is generated by keying the $\boxed{\text{SQR(,}}$ key.

X	=	5	The variable X is assigned the value of 5
F	=	4/3*#PI*7↑3	The variable F is assigned the value of the expression 4/3*#PI*7↑3
Y3	=	SIN (30)	The variable Y3 is assigned the value of the expression SIN (30)
X	=	X+1	The variable X is increased (incremented) in value by 1
V	=	1/3*#PI*R↑2*H	The variable V is assigned the value of the expression 1/3*#PI*R↑2*H, where the variables R and H have been previously defined.
C	=	SQR(A↑2+B↑2)	The variable C is assigned the value of the expression SQR(A↑2+B↑2), where the variables A and B have been previously defined.

Fig. 5–2e

SECTION 5–3
THE CLEAR COMMAND AND ITS USE IN INITIALIZING MEMORY

In addition to MASTER INITIALIZING the 2200 (Turn on Procedure) the CLEAR command can also be used to clear the 2200 memory. This process is accomplished by touching CLEAR, CR/LF–EXECUTE. When this is done, all variables and program text [1] are removed from memory.

The CLEAR command can be used selectively to clear variables or program text, by following the CLEAR command with the appropriate letter.

:CLEAR CR/LF-EXECUTE	clears all variables from memory, and all program text as well, clears the entire memory.
:CLEAR V CR/LF-EXECUTE	clears all variables, but does not affect program text.
:CLEAR P CR/LF-EXECUTE	clears *only* program text. Does not affect variables.
:CLEAR N CR/LF-EXECUTE	clears only non-common variables. Does not affect common variables or programming text. (The difference between common and non-common variables is discussed in Chapter 20.)

The previous figures illustrated the use of the most elementary multiple statement lines. The following figures demonstrate more advanced multiple PRINT statement lines. Notice in the following example that the use of variables eliminates the need for entering a complex expression more than once.

PROBLEM:

a. Evaluate the expression 33.4∗LOG(18.66) + 5
b. Find the value of the square of the expression, i.e.,
   $(33.4*LOG(18.66) + 5)^2$
c. Find the value of the cube of the expression, less 3, i.e.,
   $(33.4*LOG(18.66) + 5)^3 - 3$

EXPLANATION:
In this three-part problem, the same expression, (33.4∗LOG(18.55) + 5) appears three times. By using a variable, and assigning it the value of this expression, the problem can be simplified.

Key Z = 33.4∗LOG(18.66) + 5 CR/LF–EXECUTE

Then part a) of the problem simply becomes
Key PRINT Z

part b) becomes
Key PRINT Z↑2

and part c) becomes
Key PRINT Z↑3–3

The CRT display shows

```
READY
:Z=33.4*LOG(18.66)+5

:PRINT Z
 102.7411653269

:PRINT Z↑2
 10555.747053

:PRINT Z↑3-3
 1084506.7531

:_
```

Fig. 5–3

[1] As yet the term program text has not been explained. Very simply, all BASIC statement lines making up a program are called program text. What is meant by a program will become clear in subsequent chapters. Program text is stored in a different place in memory.

or, consider the following problem:

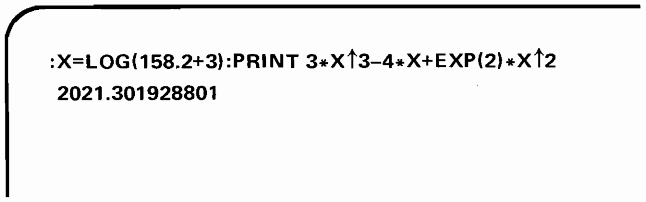
$Y=3*(\log_e (158.2+3))^3 -4*(\log_e (158.2+3)) +$
$e^2 *(\log_e (158.2+3))^2$

If X is assigned the value $(\log_e (158.2)+3)$ then

$$Y = 3X^3 - 4X + e^2 X^2$$

and the statement line needed to solve the problem is simply

```
X=LOG(158.2+3) :PRINT 3*X↑3-4*X+
EXP(2)*X↑2 CR/LF—EXECUTE
```



```
:X=LOG(158.2+3):PRINT 3*X↑3-4*X+EXP(2)*X↑2
 2021.301928801
```

Fig. 5—3a

CHAPTER 5
2200 BASIC VARIABLES

EXERCISES

1. Which of the following are *not* valid 2200 BASIC scalar variables?
   X, K2, B, Y, M12, PQ, ZZ, 2K
2. Which of the following *are* valid floating point numbers?

a. 29E144	d. 9849.92571E96
b. -2.8E-3	e. -13E-99
c. .0000987E-10	f. -13E99

3. Write the following algebraic expressions in a form which the 2200 can evaluate, if they are not useable as they stand.
   a. XYZ
   b. (a-b) (a+b)
   c. $ax^2 - bx + c$
   d. S/T
   e. $\frac{ax+by}{5ab} + 8$
   f. $\frac{(c^3 - d^3) + 2c - d}{c+d}$
4. Using the keyboard functions where applicable, write BASIC output (PRINT) statements to evaluate each of the following:
   (assume that all variables have been previously defined)

   a. $Y = \text{SIN}\ \frac{X}{2}$

   b. $A = \text{COS}\ \frac{X}{4}\ .\ \text{ATN}\ (4X^{-3})$

   c. $Z = e^{-\frac{1}{2}} \sqrt{T+4}$

   d. $B1\ = \text{Log}_e(M1-R1)$

   e. $C = \sqrt{\text{Log}_e\ (\text{Sin}(A/2))}$

CHAPTER 5
2200 BASIC VARIABLES

ANSWERS

1. M12, PQ, ZZ, 2K

2. b, c, d, e

Keystrokes for Exercises 3 and 4

```
3. a)  X * Y * Z
   b)  (A-B) * (A+B)
   c)  A*X↑2-B*X+C
   d)  S/T
   e)  (A*X+B*Y/5*A*B)+8
   f)  ((C↑3-D↑3) + 2*C-D)/C+D


4. a)  Y   = SIN(X/2):PRINT Y
   b)  A   = COS(X/4)*ATN(4*X↑(-3)):PRINT A
   c)  Z   = EXP(-1/2)*SQR(T+4):PRINT Z
   d)  B1  = LOG(M1-R1):PRINT B1
   e)  C   = SQR(LOG(SIN(A/2))):PRINT C
```

CHAPTER 6
INSTRUCTING THE 2200 TO PRINT OUT MORE THAN ONE VALUE PER LINE

Up to this point, every PRINT command executed has resulted in the printing out of a number on a new line; no more than one value has been printed on any one line. While this type of formatting satisfies many output requirements, 2200 BASIC is capable of formatting output in several other ways as well. These include "Zoned" format, "Packed" format and "Tab" format.

SECTION 6–1
WHAT IS ZONED FORMAT?

The idea of zoned format refers to the fact that the CRT display is divided into four 16-space fields or zones.

```
:READY    |             |             |
16 Spaces | 16 Spaces   | 16 Spaces   | 16 Spaces
Zone #1   | Zone #2     | Zone #3     | Zone #4
          |             |             |
          4 X 16 = 64 = Width of CRT
```

Fig. 6–1

Up to this point, all output has been printed one value to a line, in the first zone only.

To generate more than one output value per line, with each value in a separate zone, all variables, numeric values and/or expressions should be included in a single PRINT statement, with COMMAS separating the elements. This is known as ZONED format.

For example, consider the following BASIC statement, which prints out the four integers, 1 through 4, on one line, one value per zone.
Key PRINT 1, 2, 3, 4 CR/LF-EXECUTE

```
READY
:PRINT 1, 2, 3, 4
 1               2               3               4
```

Fig. 6–1a

Notice that a space is left for the implied plus (+) sign, in front of each positive number.

Or consider a BASIC statement line which, for a given radius R prints out the circumference of a circle ($2\pi R$), the area of a circle (πR^2), and the volume of a sphere ($4/3\pi R^3$), all on the same line (Fig. 6–1b).

```
READY
:R=2/3*(5.3+7): PRINT R, 2*#PI*R,#PI*R↑2,4/3*#PI*R↑3
 8.2             51.5221195188   211.2406900274  2309.564877632
```

Fig. 6–1b

Notice that in these zoned PRINT statements, the printout is accomplished by printing one value per zone, beginning in the first zone. The comma between values causes the value to be printed in the next available zone.

Similarly zoned format may be used with alphanumeric characters enclosed in quotation marks (referred to as a "literal character string").

Key PRINT "SQUARE ROOT=", SQR(729) CR/LF-EXECUTE

```
READY
:PRINT "SQUARE ROOT=", SQR (729)

SQUARE ROOT =    27
1st Zone         2nd Zone
```

Fig. 6 – 1c

If the output generated by a PRINT statement overlaps into the next zone, subsequent zoned output starts at the *beginning of the next available zone.*

CHAPTER 6
INSTRUCTING THE 2200 TO PRINT OUT MORE THAN ONE VALUE PER LINE

Key PRINT "THE PROBABILITY IS", 8/14 CR/LF–EXECUTE

```
READY
:PRINT "THE PROBABILITY IS", 8/14

THE PROBABILITY IS                        .5714285714286

        Zone 1          Zone 2                Zone 3
```

Fig. 6–1d

Because the literal string* "THE PROBABILITY IS" is 18 characters in length, and extends into the second zone, the number is printed in zone #3.

If more than four output values are requested in a zoned PRINT statement, (as denoted by more than 3 commas), the output continues in the first zone of the following line.

Key PRINT 5, 5↑2, 5↑3, 5↑4, 5↑5, 5↑6, CR/LF-EXECUTE

```
READY
:PRINT 5, 5↑2, 5↑3, 5↑4, 5↑5, 5↑6
5          25        125              625
3125       15625
```

Fig. 6–1e

Similarly, multiple and/or leading commas may be used in a PRINT statement to shift the printout from zone to zone. The printout shifts one zone for each comma included in the statement as Fig. 6–1f shows. (Note: There is no limit to the number of commas which may be used.)

```
READY
:PRINT 5, 10
 5          10
:PRINT,5,10
             5          10
:PRINT 5 , , 10
 5                      10
:PRINT, 5 , , 10
             5                     10
Zone #1    Zone #2    Zone #3    Zone #4
```

Fig. 6–1f

Commas may also be used as follows:

Key PRINT 5, :PRINT 10 CR/LF-EXECUTE

```
READY        |            |             |
:PRINT 5,    | :PRINT 10  |             |
 5           | 10         |             |
1st Zone     | 2nd Zone   | 3rd Zone    | 4th Zone
```

Fig. 6–1g

Compare this with the following statement line

Key PRINT 5 :PRINT 10 CR/LF-EXECUTE

```
:PRINT 5  :PRINT 10
 5
 10
```

Fig. 6–1h

The difference in the output is caused by the comma "," following the first PRINT statement in Fig. 6–1g, but not included in Fig. 6–1h. The comma signifies that the following PRINT statement is to continue on the same line, but in the beginning of the next zone. If no punctuation is included between the last element of the PRINT statement and the colon, a subsequent PRINT statement signifies the beginning of a new line. The position on a new line is then determined solely by the punctuation and spacing of the subsequent PRINT statement.

*Literal strings are defined as any set of characters enclosed within quotation marks. The quotation marks and letters of the alphabet are generated by touching the SHIFT key and the appropriate (upper case) keyword key, located on the left-hand section of the keyboard. Spaces may be included in a literal string, and are generated by touching the → key.

CHAPTER 6
INSTRUCTING THE 2200 TO PRINT OUT MORE THAN ONE VALUE PER LINE

SECTION 6–2
WHAT IS PACKED FORMAT?

While a zoned format enables the user to print up to four values per line, each in a preset location, a *packed format* results in the user being able to print more than 4 values per line.

To generate packed format, SEMICOLONS are used between each of the variables, numeric values, expressions, and/or literal character strings, instead of commas, in a PRINT statement.

For example, consider the following statement lines and resulting "packed" output:

```
:PRINT 1; 2; 3; 4; 5; 6; 7; 8; 9; 10
 1  2  3  4  5  6  7  8  9  10
:PRINT -1; -2; -3; -4; -5; -6; -7; -8; -9; -10
-1 -2 -3 -4 -5 -6 -7 -8 -9 -10
X = LOG(15*6.98-23.54) : PRINT X; EXP(X); X↑2; EXP (X↑2)
4.3966422514956    81.1599999976     19.32853093     247898743.8929
:_
```

Fig. 6–2

Notice that in packed format, zones are ignored, although, one space is still reserved for the sign (a plus (+) sign is assumed, a minus sign is printed) when printing out numeric values. Also, the semicolon places an extra space after a numeric value. This is to assure some spacing between numbers.

The actual number of data values and/or literal strings which can be printed on a line is dependent upon the length of the alphanumerics themselves. However, the maximum number of characters printed on the CRT on any one line is 64. Any output exceeding this continues in the first space of the following line.

Key in the following 2200 BASIC statement PRINT "SQUARE ROOT="; SQR(729) CR/LF-EXECUTE

```
:PRINT "SQUARE ROOT=";SQR(729)
 SQUARE ROOT =  27
```

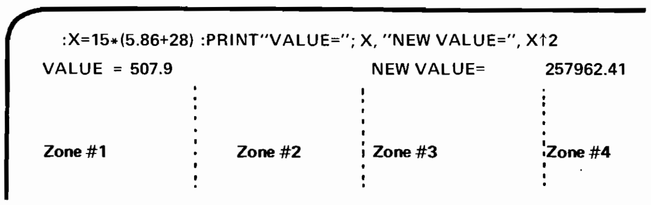
Fig. 6–2a

Notice that a single space is left between the literal printout character "=", and the numeric value 27. The space is included as a "place holder" for the implied plus (+) sign of the numeric. No extra spaces are added to the printout of literal strings. The only spaces printed out with literal strings are those included *within* the quotation marks.

SECTION 6–3
MIXING ZONED AND PACKED FORMAT

Zoned and packed formatting can be used together to achieve a wider range of format control. For example, enter the following *mixed* format line

X=15*(5.86+28) :PRINT"VALUE="; X, "NEW VALUE=";X↑2 CR/LF-EXECUTE



```
  :X=15*(5.86+28) :PRINT"VALUE="; X, "NEW VALUE=", X↑2
VALUE  = 507.9                         NEW VALUE=          257962.41
```

Fig. 6–3

Notice that the use of the two commas caused the "NEW VALUE" literal string to be printed out starting in Zone #3

MIXING ZONED AND PACKED FORMATS
EXAMPLES OF ZONED AND PACKED PRINT STATEMENTS

```
:Z=123.987:Y=.168:Z=-32

:PRINT X, Y, Z

123.987          .168             -32
```

```
:PRINT X, -14.7,Z

123.987          -14.7            -32
```

```
:PRINT "ANSWERS:",X,Y,Z

ANSWERS:         123.987          .168             -32
```

```
:PRINT "A=", (Z↑2-12*X)/(X*Y+Z)

A=               41.52518884201
```

```
:PRINT X,Y,Z,X+Y+Z,-X,-Y,-Z

123.987          .168             -32              92.155
-123.987         -.168            .32
```

```
:PRINT X,Y,Z:PRINT 2*X, (X+Z)↑Y

123.987          .168             -32
247.974          2.1374996116
```

```
:PRINT X,Y,Z,:PRINT 2*X, (X+Z)↑Y1

123.987          .168             -32              247.974
2.1374996116
```

Fig. 6–3a

1 Notice the "trailing" comma in the first PRINT statement of the line, preceding the colon. "Trailing" punctuation tells the 2200 to continue the next PRINT statement on the same line, if possible.

SECTION 6 – 4
USING THE TAB(COMMAND FOR FORMAT CONTROL

In addition to using zoned and packed format in 2200 BASIC, another formatting tool is available, which is analogous to the "tab stop" found on a typewriter. This formatting tool is the TAB(command, found in the 2200 keyword block on the left-hand side of the 2200 BASIC keyboard.

When the 2200 encounters a TAB(*command in a* PRINT *statement, the CRT cursor or printing device spaces over to the column position indicated within the* TAB(*parentheses (a right-hand parenthesis must be entered) and then proceeds to output the next part of the* PRINT *statement.*

For example

Key PRINT TAB(10); 25 CR/LF–EXECUTE

```
: PRINT TAB (10); 25

            25
:_
```

$\underbrace{\qquad\qquad}_{\text{11 spaces}}$

Fig. 6–4

The 2200 spaces over 11 spaces to column 10, *leaves a space for the implied plus sign* and prints the number 25 in columns 11 and 12.

> **NOTE:**
> *There are 64 columns per line, numbered 0 thru 63. Thus the first column is numbered column #0 and the 64th column is numbered #63. Therefore a TAB(command of TAB (10) actually spaces 11 spaces to column 10, because the 1st column is numbered column #10.*

Or for example,

Key

PRINT TAB (5↑2–3);"ANSWER";SQR (17.3)
CR/LF–EXECUTE

```
:PRINT TAB( 5↑2-3); "ANSWER ="; SQR( 17.3)
                       ANSWER = 4.1593268686
:_
```

$\underbrace{\qquad\qquad\qquad}_{\text{22 spaces (i.e., }5\uparrow2-3 = 22)}$

Fig. 6–4a

In this case, the 2200 evaluates the TAB(expression, spaces to the indicated column (22), prints out the literal string "ANSWER=", evaluates the square root function, and prints out the result.

The contents of the parentheses of a TAB(command can be any algebraic expression. However, only the integer portion of the resulting evaluation is recognized. When using the CRT as the output device, a number greater than 63 in a TAB(command always results in the positioning of the CRT cursor at the first column of the following line; a number less than zero is ignored.

If the printing position of the 2200 is past the requested tab location at the time the TAB(command is encountered, the location of the CRT cursor does not change at all, and the TAB(command is ignored.

For example,

Key PRINT TAB(20);20; TAB(10); 10 CR/LF–EXECUTE

```
:PRINT TAB(20);20; TAB(10);10
                       20  10

        22 spaces      1 trailing space plus
:_                     1 space left
                       for implied "+" sign.
```

Fig. 6–4b

In this example, the 2200 spaces over 21 spaces to column 20, leaves a space for the implied plus sign, prints the number 20 in columns 21 and 22 and then continues to the next part of the PRINT statement, another TAB(command. However, this second TAB(command says to space over to column 10. Since the CRT cursor is already past column 10, this TAB(command is ignored, and the 2200 goes to the next part of the PRINT statement, which says to print the number 10. A "packed" format results.

Thus, to obtain a printout of the number 10 at column 10, and the number 20, at column 20 in the above example, the PRINT statement must be rearranged as follows:

Key PRINT TAB(10); 10; TAB(20); 20

```
:PRINT TAB(10); 10; TAB(20);20
            10         20
  11 spaces

             22 spaces
```

Fig. 6–4c

NOTE:

When figuring what number to use within the parentheses of a TAB(command, you must remember that the number of spaces the cursor moves is always one more than the number indicated within the parenthesis, because of the way the columns are numbered. (i.e. column 11 means 12 spaces.) When printing numbers another space is left for an implied plus sign. This means if you use a TAB(20) to print a positive number the number is actually printed in column 21, 21 spaces plus another space for the sign. Therefore 22 spaces are left blank from the beginning of the line.

CHAPTER 6
INSTRUCTING THE 2200 TO PRINT OUT MORE THAN ONE VALUE PER LINE

EXERCISES

I Using Commas and Semicolons

1) The 2200 BASIC Statement Line

```
A = 2.1: B = 3.1: C = 4.1: PRINT C,B,A CR/LF—EXECUTE
```

produces which output line?

```
a)  3.1        2.1        4.1
b)  C          B          A
c)  4.1        3.1        2.1
d)  C=2.1      B=3.1      A=4.1
e)  2.1        3.1        4.1
```

2) If W = 5, write a statement line using a single PRINT statement and the W variable which (when raised to various powers) will produce the following output:

```
5              25             125            625
↑              ↑              ↑              ↑
1st Column     17th Column    33rd Column    49th Column
```

3) How could the statement line in Exercise (1) be written to produce the following output format?

```
a)  2.1 )
    3.1 }   In Print-Zone #1
    4.1 )

b)  2.1 )
    3.1 }   In Print-Zone #3
    4.1 )
```

4) If G = −2, write a statement line using a single PRINT statement and the G variable which will produce the following output:

```
   -2     4   -8    16   -32    64   -128    256
↑      ↑     ↑     ↑     ↑     ↑     ↑      ↑
0th    two   one   two   one   two   one    two
Column spaces space spaces space spaces space spaces
```

5) Knowing that commas and semicolons can be mixed in a PRINT statement, how would a 2200 BASIC statement line be written which would produce the following output: **NOTE**— Do in two different ways — (**HINT**: The statement uses literal strings.)

```
X = 18.3           Y=1.20000000E−04
↑                  ↑
in 1st Column      in 17th Column
```

II Use TAB(commands instead of commas to complete Exercises 2 and 5.

CHAPTER 6
INSTRUCTING THE 2200 TO PRINT OUT MORE THAN ONE VALUE PER LINE

ANSWERS

I 1) Output Line #c):

```
4.1            3.1            2.1
```

2) W=5 :PRINT W,W↑2, W↑3, W↑4, CR/LF-EXECUTE

3) a) A=2.1: B=3.1: c=4.1: PRINT A: PRINT B: PRINT C CR/LF–EXECUTE

b) A=2.1: B=3.1: C=4.1: PRINT A

3) a) A=2.1: B=3.1: C=4.1: PRINT A: PRINT B: PRINT C CR/LF–EXECUTE

b) A=2.1: B=3.1: C=4.1: PRINT,,A: PRINT,,B: PRINT,,C CR/LF–EXECUTE

4) G=–2 :PRINT G; G↑2; G↑3; G↑4; G↑5; G↑6; G↑7; G↑8 CR/LF-EXECUTE

5) PRINT "X="; 18.3, "Y="; 1.2E–4 CR/LF–EXECUTE

X = 18.3: Y = 1.2E–4: PRINT "X="; X, "Y="; Y CR/LF–EXECUTE

II 2) PRINT W; TAB(16); W↑2; TAB(32); W↑3; TAB(48); W↑4 CR/LF–EXECUTE

5) X=18.3: Y=1.2E–4:PRINT "X=";X;TAB(16); "Y="; Y CR/LF–EXECUTE

CHAPTER 7
USING THE 2200 WITH REITERATIVE PROCEDURES (LOOPING)

Thus far in the discussion of the 2200, the execution of a statement line has resulted from the sequential execution of each 2200 BASIC statement in a line. Until the entire line was completed after parts of a line or even an entire line needs to be repeated over and over again. This is called looping. This chapter introduces the FOR and NEXT statements which allows you to loop or perform repetitive procedures.

SECTION 7 – 1
FOR/NEXT LOOPING

Suppose a summation procedure (such as for integers 1 thru 25) is required. This problem can be solved by entering one very long PRINT statement:

PRINT 1+2+3+4+5+6+7+8+9....24+25 CR/LF

which would produce the answer, 325.

However, this method is both tedious and time consuming. It is also very limited. What happens when the summation of the integers 1 through 1000 is wanted?

Instead of solving such a problem by repetitively entering numbers, 2200 BASIC enables the user to easily accomplish this process with FOR/NEXT statements. This same summation would be written using the FOR/NEXT statements as shown in Fig. 7–1 and the result in Fig. 7–1a.

$$\underbrace{\textbf{FOR X=1TO25:}}_{\text{COUNTER VARIABLE}}\ \underbrace{\textbf{Y=Y+X:}}_{\text{SUMMATION OF COUNTER}}\ \underbrace{\textbf{NEXT X:}}_{\text{INCREMENT COUNTER}}\ \textbf{PRINT X, Y}$$

CR/LF—EXECUTE

Fig. 7–1

```
:FOR X=1TO25:Y=Y+X:NEXT X:PRINT X,Y
 25                 325

:_
```

Fig. 7–1a

The FOR-TO *statement serves to start the loop and determines the number of times the loop is to be repeated. In this example, the process is to be repeated 25 times, therefore X = 1 to 25. This is called the counter.*

The NEXT *statement increments the counter and serves to test for the outer boundary of the loop. When the loop repeats 25 times in this example the* NEXT *statement tests to see if this number has been reached and if so terminates or stops the looping.*

Fig. 7–1b explains a little more about how the loop is set up.

Analysis shows that for each value which the variable X, the counter, is assigned ranging, 1 thru 25, the value of the variable Y is increased by that value of X. Thus, the variable X acts as a "counter", and the variable Y acts as the summation of X. The NEXT X statement then signals the 2200 to increment the variable X again, and the process continues until the "counter" variable – in this case X – is assigned a value outside the range set up in the FOR-TO statement.

At this point, the loop is "exhausted", and the 2200 continues processing the rest of the line. In this example the loop goes through 25 iterations, after which a PRINT statement is encountered, which prints the final value of X, and Y. (If the PRINT command were included inside the loop instead of outside, the values of X and Y would be printed each time the 2200 executed the loop.)

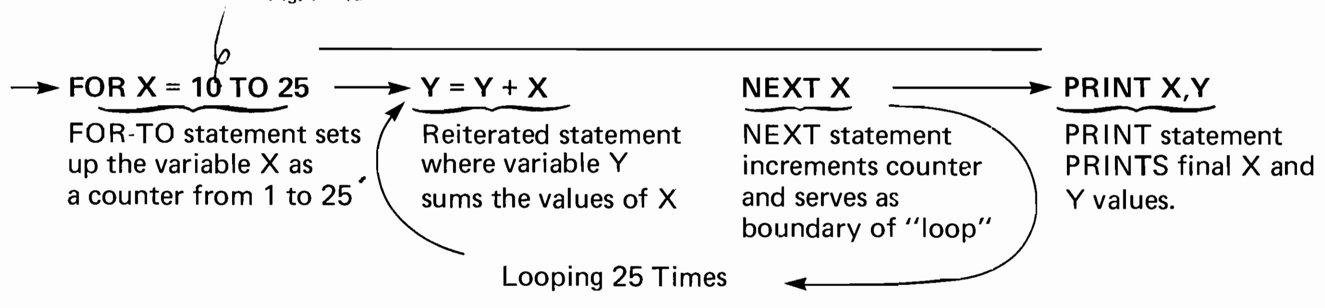

FOR X = 10 TO 25
FOR-TO statement sets up the variable X as a counter from 1 to 25
Y = Y + X
Reiterated statement where variable Y sums the values of X
NEXT X
NEXT statement increments counter and serves as boundary of "loop"
PRINT X,Y
PRINT statement PRINTS final X and Y values.
Looping 25 Times

Fig. 7–1b

CHAPTER 7
USING THE 2200 WITH REITERATIVE PROCEDURES (LOOPING)

The general format of a FOR-TO/NEXT statement is as depicted in Fig. 7–1d.

FOR	[numeric scalar variable]	=	[starting value composed of any legal expression]	TO	[loop limit composed of any legal expression]	STEP	[increment size composed of any legal expression]	NEXT	[numeric scalar variable]

Fig. 7–1d

In general practice, the expressions which define the range of values of the FOR statement are integers, with the STEP size – an optional part of the statement – included only when the step size is other than "+1"; when the STEP size is not included, it is assumed to be "+1", as it was in the above example. (STEP is a 2200 BASIC keyword, generated by touching the STEP key, located in the keyword key block) The variable given in the NEXT statement must be the *same* variable given as the FOR-TO statement variable.

EXAMPLES OF FOR-TO/NEXT STATEMENT PAIRS

FOR X = 5 to 500 NEXT X	X takes on the 496 successive values 5,6,7....498,499,500
FOR X = 5 TO 500 STEP 5 NEXT X	X takes on the 100 successive values 5,10,15....495,500
FOR A = SQR(2)+2 TO SQR(100)+2 NEXT A	A takes on 9 successive values 3.414, 4.414, 5.414,....9.414, 10.414, 11.414
FOR W = 100 TO 25 STEP–2 NEXT W	W takes on the 38 successive values 100, 98, 96....28, 26
FOR G =–10 to +10 STEP .4 NEXT G	G takes on the 51 values –10, –9.6, –9.2....–.4,0,+.4,....+9.6, +10

Because of the range of values and the step size, each of the variables in the above FOR-TO/NEXT statement pairs takes on a number of successive values. However, consider what happens in the following situations:

FOR Z = 10 TO 15 STEP 10 NEXT Z	In the loop Z takes on the single value 10: loop is executed only once.
FOR Y6 = 52.3 TO 100 STEP –1 NEXT Y6	In the loop, Y6 takes on the single value 52.3: Loop is executed only once.
FOR T = 10 TO 1 NEXT T	In the loop T takes on a single value 10: Loop is executed only once.

In these last three examples the loop is executed only once because of the increment defined in the STEP. In the loop [For Z = 10 to 15 STEP 10], the first loop occurs but if you added 10 to 10, you get 20, the limit of the loop is only 15, therefore this is exceeded or can't be reached and the 2200 stops.

CHAPTER 7
USING THE 2200 WITH REITERATIVE PROCEDURES (LOOPING)

SECTION 7–2
SPECIAL PRINT SITUATIONS WITH LOOPING

As mentioned in the discussion following Fig. 7–1b inclusion of the PRINT statement inside the loop causes the 2200 to print out a running account of the variables X and Y. If the statement line is as follows, the output consists of a single column:

Key CLEAR CR/LF [1]
Key
FOR X =1TO25:Y=Y+X:PRINT Y:NEXT X
CR/LF – EXECUTE

```
READY
:FOR X=1TO25:Y=Y+X:PRINT Y: NEXT X
1
3
6
,
,
,
,
276
300
325
```

Fig. 7–2

The reason for this format is the structure of the PRINT statement: No punctuation is included in the PRINT statement following the variable Y. Thus, each execution of the PRINT command causes a printout on a separate line [2]

The format of this output can be changed to zoned or packed, by including after the variable name, a comma or semicolon respectively in the PRINT statement:

Key CLEAR CR/LF–EXECUTE [1]
Key
FOR X=1TO25:Y=Y+X:PRINT Y,:NEXT X
CR/LF–EXECUTE

This generates zoned output, as Fig. 7–2a shows.

```
READY
:FOR X = 1 TO 25: Y = Y + X: PRINT Y,: NEXT X
1               3               6               10
15              21              28              36
45              55              66              78
91              105             120             136
153             171             190             210
231             253             276             300
325
:_
```

Fig. 7 – 2a

To generate packed output (Fig. 7–2b),
Key CLEAR CR/LF–EXECUTE [1]
Key
FOR X=1TO25:Y=Y+X:PRINT Y; :NEXT X
CR/LF–EXECUTE

```
READY
:FOR X =1 TO 25: Y = Y + X: PRINT Y;: NEXT X
 1  3  6  10  15  21  28  36  45  55  66  78  91  105  120  136
 153  171  190  210  231  253  276  300  325
:_
```

Fig. 7 – 2b

Packed output created by semicolon following last element in PRINT statement.

[1]The CLEAR command is used here to clear the variable Y from the system. Otherwise, when the new statement line is executed, the initial value of Y will be 325, left over from the execution of Fig. 7–1b.

[2]When this statement line is executed, the results are displayed on the CRT with such rapidity, that the output is difficult to follow, unless the 2200 is instructed to "pause" after every line of output. To instruct the 2200 to pause for "n" 1/6 seconds after every line of output.
Key   SELECT P n CR/LF–EXECUTE, where n is any integer between 0 and 9, and each "n" represents a 1/6 second pause.

SELECT P 4	=	pause 4 x 1/6 sec., or 2/3 seconds after each line of output.
SELECT P 9	=	pause 9 x 1/6 sec., or 1½ seconds after each line of output.

To turn off a "Pause",
key   SELECT P CR/LF–EXECUTE

CHAPTER 7
USING THE 2200 WITH REITERATIVE PROCEDURES (LOOPING)

SECTION 7–3
CRT PLOTTING USING A FOR-TO/NEXT LOOP AND THE TAB(COMMAND

Knowing that the TAB(command can be used for positioning output, a simple CRT plotting routine can now be written using a FOR-TO/NEXT loop, and a PRINT statement with a TAB(command.

For example, the following statement line causes the 2200 to plot a diagonal line of 11 plus signs (+) down the CRT, starting in the first colunn:

Key
FOR X=0TO10:PRINT TAB(X);"+": NEXT X
CR/LF–EXECUTE

The CRT display shows

```
READY
:FOR X=0 TO 10:PRINT TAB(X);"+":NEXT X
+
 +
  +
   +
    +
     +
      +
       +
        +
         +
          +
:_
```

Fig. 7–3

Each time the variable X is assigned an integer value, the PRINT TAB(statement says to tab to the column denoted by the value of X, and print a "+", then move to the next line and continue the process. This happens a total of eleven times, producing the diagonal line of eleven "+"'s down the CRT display.

If a semicolon is included in the PRINT statement following the "+", the result is different.

Key
:FOR X=0TO10:PRINT TAB(X);"+";: NEXT X
CR/LF–EXECUTE

```
READY
:FOR X=0 TO 10:PRINT TAB(X);"+";:NEXT X

+++++++++++

:_
```

Fig. 7–3a

This is because the last semicolon in the PRINT statement signifies that the printout is to continue on that line, instead of starting on a new line each time the loop is executed.

Consider now the function f(x) = X^2. If graphed on regular graph paper, the result is a parabola. The same thing can be accomplished on the 2200 CRT in one statement line (Fig. 7–3b):

Key
FOR X=1TO8:PRINT TAB(X↑2–1);"0": NEXT X
CR/LF–EXECUTE

```
READY
:FOR X = 1 TO 8: PRINT TAB (X↑2-1); "0": NEXT X

o
   o
        o
               o
                        o
                                   o
                                                o
                                                               o
:_
```

Fig. 7–3b

The result above is for positive values of X. The number 8 was chosen because 8 ↑ 2 – 1 = 63, which just fits within the range of the CRT screen.

CHAPTER 7
USING THE 2200 WITH REITERATIVE PROCEDURES (LOOPING)

By changing the range of the values of the FOR-TO/NEXT loop, a complete parabola can be generated. For example

Key

FOR X = -6 to 6:PRINT TAB(X↑2);X↑2: NEXT X
CR/LF–EXECUTE

produces the results found in Fig. 7–3c.

```
READY
:FOR X = -6 TO 6 :PRINT TAB( X↑2) ;X↑2: NEXT X

                                                  36
                                     25
                         16
                9
         4
     1
  0
     1
         4
                9
                         16
                                     25
:_                                                36
```

Fig. 7–3c

The PRINT command can be used also to skip lines or to complete partially used lines. For example

Key

FOR X=1 TO 5: PRINT X: PRINT: NEXT X
CR/LF–EXECUTE

```
:FOR X=1 TO 5: PRINT X: PRINT: NEXT X
 1

 2

 3

 4

 5

:_
```

Fig. 7–3d

Notice the skipped lines in the printout. The statement line has two PRINT statements, the second of which causes the extra spacing in the printout.

Now key

FOR X=1TO5:PRINT X;:NEXT X:PRINT"DONE"
CR/LF–EXECUTE

```
:FOR X=1TO5:PRINT X;:NEXT X:PRINT "DONE"
 1    2    3    4    5  DONE

:_
```

Fig. 7–3e

Key

FOR X=1TO5:PRINT X;:NEXT X:PRINT:PRINT "DONE" CR/LF–EXECUTE

```
:FOR X=1TO5:PRINT X;:NEXT X:PRINT:PRINT "DONE"
 1    2    3    4    5

 DONE

:_
```

Fig. 7–3f

The extra PRINT statement is the reason for the difference in the output format. Without the extra PRINT statement, the literal string "DONE" is printed on the same lines as the digits 1 through 5. (The trailing semicolon in the PRINT X statement has held the cursor on that line) The blank PRINT statement in the second example causes the CRT to skip to the start of the next line before executing the next PRINT statement (PRINT "DONE").

Part III —Programming The 2200—

INTRODUCTION TO PROGRAMMING

The previous seven chapters illustrated the use of the 2200 as a calculator (i.e. the immediate mode). The immediate mode obtains fast results for one-time calculations.

Multi-statement lines in the immediate mode add the ability to generate lists, or tables of results. For example, FOR 1=10 to 20:PRINT I, I↑5, I↑7:NEXT I.

Now desired is a method of entering a set of 2200 BASIC statements once, and executing them as many times as necessary. The 2200 enables the user to enter a set of 2200 BASIC statements to be stored in the system's memory, displayed on the CRT, and/or saved on cassette tapes.

Consider Fig. A, a BASIC program, and compare it to Fig. B

Fig. A

```
10 FOR X = 1 TO 25
20 Y = Y+X
30 NEXT X
40 PRINT X,Y
```

Fig. B

```
FOR X = 1 TO 25: Y=Y+X:NEXT X:PRINT X,Y
```

What do these two examples have in common? What are their differences?

First, both figures contain the same four statements in the same order. However, Fig. A is written in column form with numbers in front of each statement (i.e., statement line numbers), whereas Fig. B is written in line form with colons between statements[1]. In Fig. A, the statement line numbers define the order of the statement lines, and also identify each one. Fig. A is called a 2200 BASIC program which, when entered into the 2200, can be "executed" over and over again, by simply pressing RUN CR/LF-EXECUTE; the statement line in Fig. B, because it is an immediate mode statement line, must be reentered every time it is needed.

Thus, a 2200 BASIC program is a group of BASIC statements entered into the 2200, with statement line numbers to identify each statement line. The program, once it is entered, can be executed as many times as needed.

1 In actual practice more than one statement can be included (separately) in each program line of a 2200 BASIC program, bu using colons for separators. However, for reasons of simplicity, each statement in this example is given on a separate line.

CHAPTER 8
PROGRAMMING AND USING THE 2200

SECTION 8–1
THE BASICS OF ENTERING AND EXECUTING A PROGRAM IN MEMORY

Writing a 2200 BASIC program involves entering a set of 2200 BASIC statements into the 2200, *with statement line numbers.* Before entering a new program into the 2200, it is best to clear the memory area by keying CLEAR CR/LF—EXECUTE.

This assures that the memory area is free of all program text and variables. The 2200 is now ready to accept a new program.

ENTERING A PROGRAM IN MEMORY
Entering a program into memory via the 2200 BASIC keyboard:

1. First, enter the statement line number.
2. Next, enter the BASIC statement line, following the statement line number.
3. Touch the CR/LF-EXECUTE key, which enters the statement line into memory.
4. Repeat the first three steps as many times as needed to enter the entire program.

Result — The 2200 is "programmed" with the statement lines entered.

EXECUTING A PROGRAM IN 2200 MEMORY USING THE RUN COMMAND
Once a program is in memory, keying RUN CR/LF-EXECUTE executes the program. When RUN is executed, the 2200 does the following:

1. Scans the entire program for variable names, and sets aside space in memory for each of them.
2. Initializes all variables to zero[1].
3. Checks to assure that no logic errors have been made in program.
4. Once Steps 1, 2, and 3 above are completed, the program lines are executed sequentially, and all instructions are carried out.

SECTION 8–2
THE STATEMENT LINE NUMBER

As mentioned, the statement line number serves two purposes: first, it denotes the order in which the statements are to be executed; and second, it identifies each individual statement line.

The line number associated with any 2200 BASIC statement can be any positive one, two, three, or four digit number. The numbers may be uniformily spaced (1,2,3... or 10,20,30...) but need not be. Program lines can be numbered 1,5,15,37,42. However, in general practice, the initial sequence is generally 5,10,15,20... or 10,20,30,40..., to allow ample "space" for later insertion of lines.

There are two ways to enter a statement line number:

1. The statement line number can be generated by using the numeric keyboard, or
2. The statement line number can be generated by keying the STMT. No. key, before entering each statement line.

SECTION 8–3
USING THE STMT. NO. KEY

Consider Fig. A again. Notice that the line numbers are incremented by tens. Although these numbers could be entered by using the numeric keyboard, the STMT. NO. key enables the user to accomplish the same results by a single keystroke. Each time the STMT. NO. key is touched at the beginning of a new line, a line number ten more than the highest line number already in memory is placed at the start of the new line.

[1] With the exception of certain variables predefined as common. See Chapter 20 for an explanation of common variables.

Thus, to enter the program in Fig. A into the 2200's memory using the STMT. NO. key, the following procedure is used:

1. Key CLEAR CR/LF—EXECUTE to clear the memory area.
2. Key STMT. NO., followed by FOR X = 1 TO 25 CR/LF—EXECUTE.

   The CRT display shows

```
READY
:10 FOR X = 1 TO 25
:_
```

Fig. 8—3

Notice that keying the STMT. NO. key generates the number "10" followed by a space.

3. Key the STMT. NO. key again, followed by Y = Y + X CR/LF—EXECUTE.

```
READY
:10 FOR X = 1TO 25
:20 Y = Y + X
:_
```

Fig. 8—3a

Notice that touching the STMT. NO. key generates the number "20" followed by a space.

4. To finish entering the program, key in the statement line numbers and statement lines, in the order indicated. The final result should appear as in Fig. 8—3b.

```
READY
:10 FOR X = 1 TO 25
:20 Y = Y + X
:30 NEXT X
:40 PRINT X,Y
:_
```

Fig. 8—3b

RESULT: The program has been entered in sequential line order. Although using the STMT. NO. key assures that the lines are entered in sequential order (10,20,30 etc.), they need not be entered in that order. For instance, if the program were entered as in Fig. 8—3c, the program would be executed in exactly the same order as in Fig. 8—3b. This occurs because the 2200 executes them in numeric order. Entering a program in other than sequential order, however, precludes the use of the STMT. NO. key.

```
READY
:40 PRINT X,Y
:20 Y = Y + X
:10 FOR X = 1 TO 25
:30 NEXT X
:_
```

Fig. 8—3c

SECTION 8—4 EXECUTING THE PROGRAM

To execute the above program

Key RUN CR/LF-EXECUTE

```
READY
:10 FOR X = 1 TO 25
:20 Y = Y + X
:30 NEXT X
:40 PRINT X,Y
:RUN
 25                325

:_
```

Fig. 8—4

When the 2200 executes a RUN command, it reinitializes all non-common program variables. To RUN the program again,

Key RUN CR/LF-EXECUTE a second time.

Notice the final values of X and Y are again printed out (See Fig. 8—4a). Any program can be executed as many times as desired, simply by keying RUN CR/LF-EXECUTE. Specifying a line number after keying RUN (i.e., RUN 30 CR/LF-

EXECUTE) signifies that you can run a program starting at line 30 and continue to the end. The General Form of the RUN command is as follows:

RUN [line number]

where the square brackets indicate that the line number is optional.

```
READY
:10  FOR X = 1 TO 25
:20  Y = Y + X
:30  NEXT X
:40  PRINT X,Y
:RUN
  25          325

:RUN
  25          325

:_
```

SECTION 8–5
CHANGING A PROGRAM IN MEMORY

Once the lines of a program are put into memory, the lines remain there until cleared from the system, or until they are redefined. New lines can be added at any time.

TO REDEFINE A STATEMENT LINE: once it is entered (i.e., after the CR/LF-EXECUTE key is touched), reenter the same statement line number, followed by the new statement line. Thus, to change line 20 from

20 Y = Y + X

to the line

20 Y = Y + X ↑ 2

the new line, with the same line number, must be entered. Keying CR/LF–EXECUTE, erases the previous line and enters the new one (provided both lines have the same line number).

TO DELETE A LINE FROM MEMORY: key in the number of the statement line, and key CR/LF–EXECUTE.

TO INSERT A NEW LINE: enter an appropriate statement line number which can go in the middle of program, followed by the new statement line. For example

Suppose the line

Z = Y – X

is to be included between line 20 and line 30.

```
:10 FOR X = 1 TO 25
:20 Y = Y + X↑2
:30 NEXT X
:40 PRINT X,Y
:_
```

Fig. 8–5

This can be done by entering any statement line number from 21 thru 29, inclusive, generated via the numeric keyboard. The statement line, Z=Y–X, follows the number.

Thus, entering the line

23 Z = Y – X CR/LF-EXECUTE

automatically is entered internally in the proper sequence. Enter the line by keying 23 Z=Y–X CR/LF. Then key LIST EXECUTE (Section 9-6). The CRT shows the complete program with the new line inserted in the proper place (See Fig. 8–5a).

```
:LIST
10 FOR X = 1 TO 25
20 Y = Y + X↑2
23 Z = Y – X
30 NEXT X
40 PRINT X,Y
:_
```

Fig. 8–5a

To delete line 23 from the program, simply key

23 CR/LF-EXECUTE

The line is no longer in memory.

TO ENTER A NEW LINE AT THE END OF A PROGRAM: enter an appropriate line number (or touch the STMT. NO. key), and enter the new line, at anytime.

SECTION 8–6 LISTING A PROGRAM

USING THE LIST KEY

Once a program has been entered, it should be listed, in order to check that all statement lines have been entered in the proper order. On the 2200, a program can be listed by touching LIST CR/LF-EXECUTE

Thus, to list the program in Fig. 8–5a,

LIST CR/LF-EXECUTE

```
READY
:LIST
10 FOR X = 1 TO 25
20 Y = Y + X
23 Z = Y - X
30 NEXT X
40 PRINT X,Y
:_
```

Fig. 8–6

Even if a program is entered "out of order" as, in Fig. 8–3c, the listing will be "in order".

USING LIST S TO DISPLAY 15 LINES AT A TIME

For longer programs (longer than 15 lines) which cannot fit entirely on the CRT display, the use of LIST S CR/LF-EXECUTE is suggested. This causes the 2200 to display the first 15 lines of the program. To continue listing, key CR/LF-EXECUTE and the next 15 lines of the program are listed. This procedure can be continued until the entire program has been listed.

LISTING A PARTICULAR SECTION OF A PROGRAM

A particular line or set of lines can be listed, by specifying which lines are desired in the LIST statement. The general form of a LIST statement:

LIST [S] [line number [, line number]]

NO. OF FIRST STMT. LINE TO BE LISTED	NO. OF LAST STMT. LINE TO BE LISTED

For example, the statement line

LIST 10, 30 CR/LF-EXECUTE

produces the result

```
:LIST 10,30
10   FOR X = 1 TO 25
20   Y = Y + X
23   Z = Y - X
30   NEXT X
:_
```

Fig. 8–6a

If only one particular statement line is desired, include only that statement line number in the LIST statement. Thus, the statement

LIST 20 CR/LF-EXECUTE

produces the result

```
:LIST 20
20 Y = Y + X
:_
```

Fig. 8–6b

Another way to list longer programs is to initiate a Pause (SELECT P) [1] prior to listing a program. This allows the user to scan the program, as it is slowly displayed on the CRT.

1 When this statement line is executed, the results are displayed on the CRT with such rapidity, that the output is difficult to follow, unless The 2200 is instructed to "pause" after every line of output. To instruct the 2200 to pause for "n" 1/6 seconds after every line of output.

Key SELECT P n CR/LF–EXECUTE, where n is any integer between 0 and 9, and each "n" represents a 1/6 second pause.

SELECT P 4 = pause 4 x 1/6 sec., or 2/3 seconds after each line of output.

SELECT P 9 = pause 9 x 1/6 sec., or 1½ seconds after each line of output.

To turn off a "Pause",
key SELECT P CR/LF–EXECUTE

SECTION 8–7
USING THE BASIC STOP STATEMENT

Consider the following program from Fig. 8–3b. Key this program.

```
READY
:10 FOR X = 1 TO 25
:20 Y = Y + X
:30 NEXT X
:40 PRINT X, Y
:_
```

Fig. 8–7

Although these statement lines represent a complete program, an additional statement can be included anywhere in a program to signal the 2200 to stop processing. This statement is called the **STOP** statement, and generally consists of a statement line number, followed by the BASIC keyword STOP.

Continuing with the above program,

Key STMT. NO.   STOP   CR/LF-EXECUTE

When the 2200 executes this STOP statement during the course of the program's execution, the word STOP is printed on the display.

Key RUN CR/LF-EXECUTE

```
READY
:10 FOR X = 1 TO 25
:20  Y = Y + X
:30  NEXT X
:40  PRINT X,Y
:50  STOP
:RUN
25          325

STOP
:_
```

Fig. 8–7a

STOP "d" AND CONTINUE

Any number of STOP statements can be used in a program allowing the user to halt execution at a predetermined place in the program. If a literal string is included in the STOP statement, it is printed when the STOP statement is executed. This capability allows the programmer to insert messages directly into the STOP statement, without adding a separate PRINT statement. For example, consider the outlined program in Fig. 8–7b.

```
READY
:10  FOR X = 1 TO 25
:20  Y = Y + X
:30  NEXT X
:35  STOP "******END OF CALCULATION***"
:40  PRINT X,Y
:50  STOP "THIS IS LINE #50"
```

Fig. 8–7b

When this program is executed, the following display is produced.

```
READY
:10  FOR X = 1 TO 25
:20  Y = Y + X
:30  NEXT X
:35  STOP "******END OF CALCULATION***"
:40  PRINT X,Y
:50  STOP "THIS IS LINE #50"
:RUN

STOP******END OF CALCULATION******
```

Fig. 8–7c

Execution of the STOP statement does not affect any variables or program text. It simply stops program execution.

After program execution has stopped due to a STOP statement, the user can:

1. Use the 2200 as a calculator and immediately execute statement lines, without statement line numbers (immediate mode).
2. Printout a variable in the program for inspection.
3. Redefine a variable used in the program to see how this affects results.
4. Change the program flow, and instruct the 2200 to continue execution at a different program line.

5. Key CONTINUE CR/LF-EXECUTE which continues program execution immediately following the STOP statement.

Continue with the above example by keying CONTINUE CR/LF. Execution continues with line 40:

```
STOP******END OF CALCULATION******
:CONTINUE
25          325

STOP THIS IS LINE #50
:_
```

Fig. 8–7d

SECTION 8–8
USING THE BASIC END STATEMENT IN A PROGRAM

In addition to using the STOP statement, there is another statement which can be used to terminate program execution, known as the END statement. The END statement line consists simply of a statement number, followed by the BASIC keyword END:

```
100 END
```

The END statement is optional in 2200 BASIC. If used the END statement can appear anywhere in a program and performs two functions:

1. Halts program execution.
2. Displays the total amount of unused memory remaining at the time the statement was executed.

NOTE:
Program execution STOPS automatically when all statements are executed. Therefore END or STOP need not be used for this purpose. However if you have several programs in memory, then either a STOP or END statement must be used to separate them.

For example, clear the memory (by touching CLEAR CR/LF-EXECUTE) and reenter the program in Fig. 8–7 again. Include the statement line

```
50 END
```

When the RUN CR/LF–EXECUTE keys are pressed, the following results[1] :

```
READY

:10 FOR X = 1 TO 25
:20 Y = Y + X
:30 NEXT X
:40 PRINT X, Y
:50 END
:RUN
 25          325

END PROGRAM
FREE SPACE = 3324

:_
```

Fig. 8–8

The Free Space number is an integer number representing the approximate number of bytes[2] remaining for storing additional program text or variables. The 2200 requires approximately 700 bytes as a work area while executing statements. These 700 bytes of the 4096 in a 4k machine are not available to the User. (700 bytes are set aside for a work area in all size machines.) Therefore, the above program in a 4k system requires 772 bytes of memory (i.e. 4096–3324=772, 772–700 =72 for the program itself). See appendix B for further discussion on "housekeeping spaces".

1 The memory area represented in that of the 2200, with 4k bytes (4096 bytes) of memory.

2 A byte is comparable to a programming step.

SECTION 8–9
OTHER USES OF THE END STATEMENT

Whenever the END statement is keyed, the 2200 displays the "FREE SPACE" at that time.

Key CLEAR EXECUTE

Thus, if the END statement is keyed *after the 2200 memory area has been cleared*, the CRT display shows the full available memory, since none has been used.

```
READY
:END

END PROGRAM
FREE SPACE = 3398
:_
```

Fig. 8–9

FREE SPACE is always this amount in a 4k system as long as:

1. No variables have been defined [1] and
2. There is no program text in memory [1] .

Thus, executing the statement lines

```
A = 358/41 :PRINT LOG(A)CR/LF-EXECUTE
END CR/LF-EXECUTE
```

produces the result

```
:A = 358/41 :PRINT LOG(A)
 2.166960919696
:END

END PROGRAM
FREE SPACE = 3386
:_
```

Fig. 8–9a

Since the variable (A) requires storage in memory, 12 bytes are lost in the available FREE SPACE.

However, if the statements

```
CLEAR EXECUTE
PRINT LOG(358/41) CR/LF-EXECUTE
END CR/LF-EXECUTE
```

are executed, there is no loss of FREE SPACE, since no program text was used and no variable appeared in the statements: Fig. (8–9b)

```
READY
:PRINT LOG (358/41)
 2.166960919696
:END

END PROGRAM
FREE SPACE = 3398

:_
```

Fig. 8–9b

When entering a program into memory, any variables within the program must be considered along with the actual program text, when figuring the total memory requirements for "running" a program (See appendix B).

1 Variables are stored in a separate section of the memory as compared to program text. Each requires different amounts of storage. See Appendix B for storage requirements.

CHAPTER 9
UNDERSTANDING PROGRAMMING

When a programmer decides to write a program, he (or she) does not sit down and immediately enter it. Rather, a knowledgeable programmer begins by thoroughly analyzing the problem. If careful analysis is done in the beginning, fewer problems will crop up later. Part of this analysis process often includes a flow-chart.

SECTION 9–1
FLOW-CHARTING

A problem should be carefully analyzed and defined before writing a program to solve the problem. In defining the problem, the programmer should –

– First, determine the output needed – the answers wanted.
– Next, determine the data needed, and how to enter it into the program.
– Finally, determine the computations needed to arrive at the answers, including alternative courses of action.

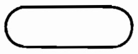
The amount of work required by this last step of analysis depends upon the complexity of the problem. In many cases, a flow-chart of all the processing which is to take place can help simplify the analysis process. A flow-chart helps to crystalize the programmer's thoughts, by allowing one to illustrate on paper the exact order in which processing is to take place.

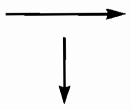
Fig. 9–1 gives some of the standard forms used in flow-charting.

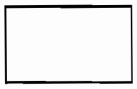
Fig. 9–1a is an example of a flow-chart.

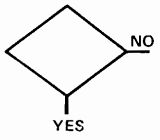
FLOW-CHARTING

FLOW-CHARTING SYMBOLS

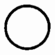

– An oval indicates a starting or stopping operation

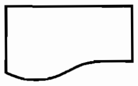

– Arrows indicate the direction of flow through the diagram. Every connecting line should have an arrow on it.


– A rectangular box indicates an operation (i.e., addition, squaring, etc.).



– A diamond indicates a decision (i.e., if YES; if NO), question or comparison.


– A large circle indicates where the program continues at some point. These points are identified by the same letter.


– A printout or display of any type (usually an answer).

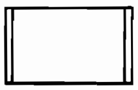

– The Predefined Process Symbol, generally used to represent a Subroutine.

Fig. 9–1

EXAMPLE OF A FLOW CHART

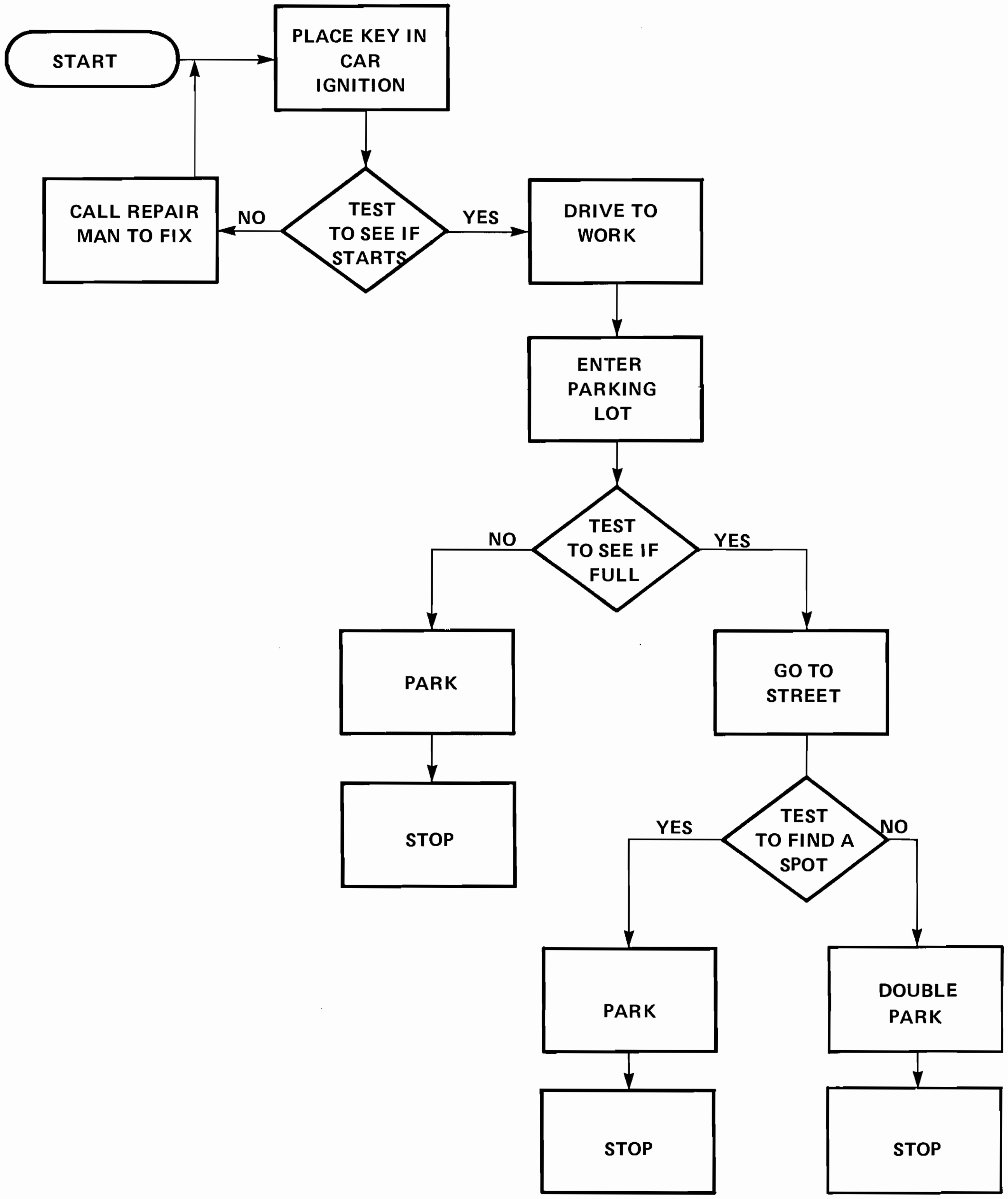


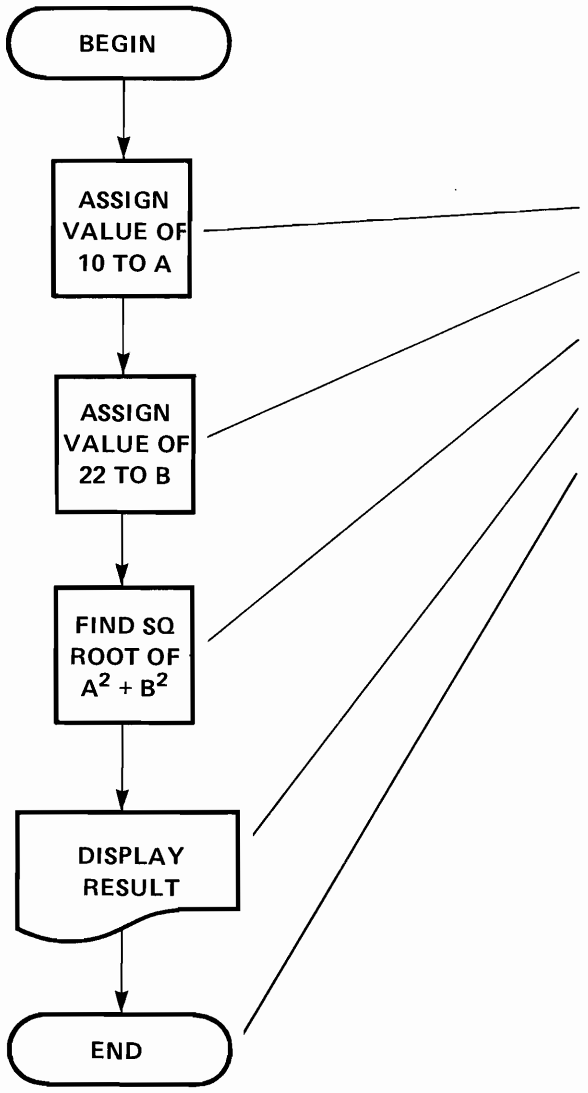
Fig. 9–1a

CHAPTER 9
UNDERSTANDING PROGRAMMING

The following is an example flow chart for solving the problem $C = \sqrt{A^2 + B^2}$, where A is assigned a value of 10, and B a value of 22.

Notice the relationship between the Flow Diagram and the Statements in the Program.

FLOW DIAGRAM

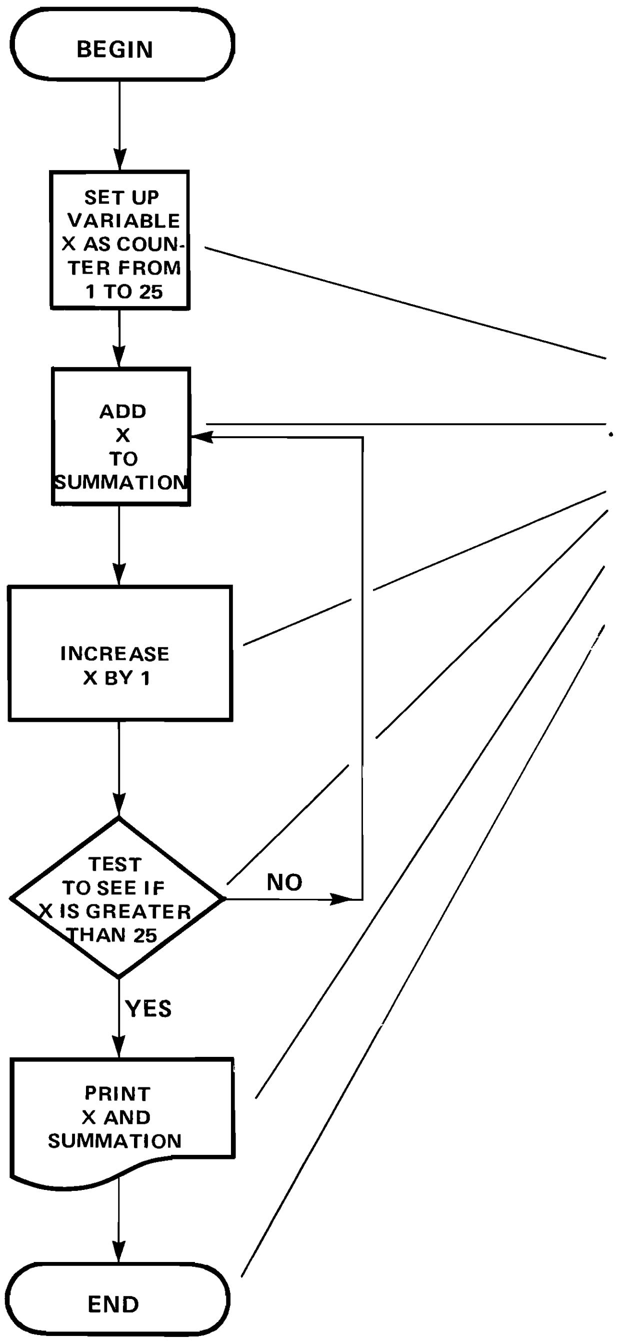


```
10A = 10
20B = 22
30C = SQR (A↑2+B↑2)
40  PRINT A, B, C
50  END
```

```
READY
:10 A=10
:20 B=22
:30 C=SQR (A↑2 + B↑2)
:40 PRINT A,B,C
:50 END
:RUN

 10         22         24.166091947

END PROGRAM
FREE SPACE = 3305
```

Fig. 9—1b

How would a FOR/NEXT loop be represented? Consider the following example, a summation of the first 25 integers.


BEGIN
SET UP VARIABLE X AS COUNTER FROM 1 TO 25
ADD X TO SUMMATION
INCREASE X BY 1
TEST TO SEE IF X IS GREATER THAN 25
NO
YES
PRINT X AND SUMMATION
END

```
10 FOR X = 1 TO 25
20 Y = Y + X
30 NEXT X
40 PRINT X, Y
50 END
```

```
READY
:10 FOR X = 1 TO 25
:20 Y = Y + X
:30 NEXT X
:40 PRINT X, Y
:50 END
:RUN

 25             325

END PROGRAM
FREE SPACE =  3324
```

Fig. 9—1c

Notice that the FOR/NEXT loop has an automatic test built into it. As long as the variable X is less than, or equal to 25, the program flows from step 20 to step 30, and back to step 20 again. A FOR/NEXT loop is an example of a "conditional branch"* because the loop depends (is conditional) upon the value of the variable (here, X) at a given time.

*A conditional branch as well as its definition is discussed in Chapter 12.

Flow charting not only helps to crystalize the programmer's thoughts, but also is a key to understanding how the program accomplishes the results intended. The flow-chart helps a user to understand what the purpose of the program is. The flow-chart is thus an important part of what is known as the *documentation* of a program — explanatory material included with a program to aid others in understanding and executing a program.

SECTION 9–2
THE REMARK (REM) STATEMENT

REM statements are used to insert explanatory comments or remarks into a program and can be included anywhere in a program. Unlike the PRINT, FOR/NEXT, and ASSIGNMENT statements, all of which are executable, the REM statement is nonexecutable — that is; when the 2200 comes upon a REM statement during the course of program execution, it does not execute the statement. The REM statement serves only as a programming aid, however it does take up memory space since it is a statement.

Consider the program in Fig. 9–2, which has three REM statements. (Note that the REM key appears on the 2215 keyboard).

Notice that the REM statements (lines 20, 40, and 60) are not printed when the program is executed and have no effect on the output, they appear only when the program is listed. REM statements do not require quotation marks.

```
READY
:10  PRINT "THIS PROG. COMPUTES THE AREA OF 3 CIRCLES"
:20  REM STANDARD FORMULA FOR AREA IS USED
:30  PRINT "RADIUS", "AREA"
:40 REM FOR/NEXT LOOP USED TO ASSIGN 3 VALUES
:50 FOR R = 5 TO 15 STEP 5
:60 REM AREA COMPUTED IN STATEMENT 70
:70 A = #PI * R ↑ 2
:80 PRINT R, A
:90 NEXT R
:100 END
:RUN


THIS PROGRAM COMPUTES THE AREA OF 3 CIRCLES
RADIUS              AREA
5                     78.53981633975
10                    314.159265359
15                    706.8583470578

END PROGRAM
FREE SPACE 3129

:_
```

Fig. 9–2

CHAPTER 10
THE UNCONDITIONAL BRANCH

As mentioned in Section 9–1, a FOR/NEXT loop is an example of a *conditional branch* – conditional upon the value of the variable in the FOR statement; as long as the FOR variable is within its assigned limits, the NEXT statement causes the program flow to branch back for another iteration.

However, another type of branching is often desired which causes program flow to branch to another location all the time regardless of the assigned values of any variables in the program. Such an *unconditional branch*, in BASIC, is accomplished through the use of the GOTO statement.

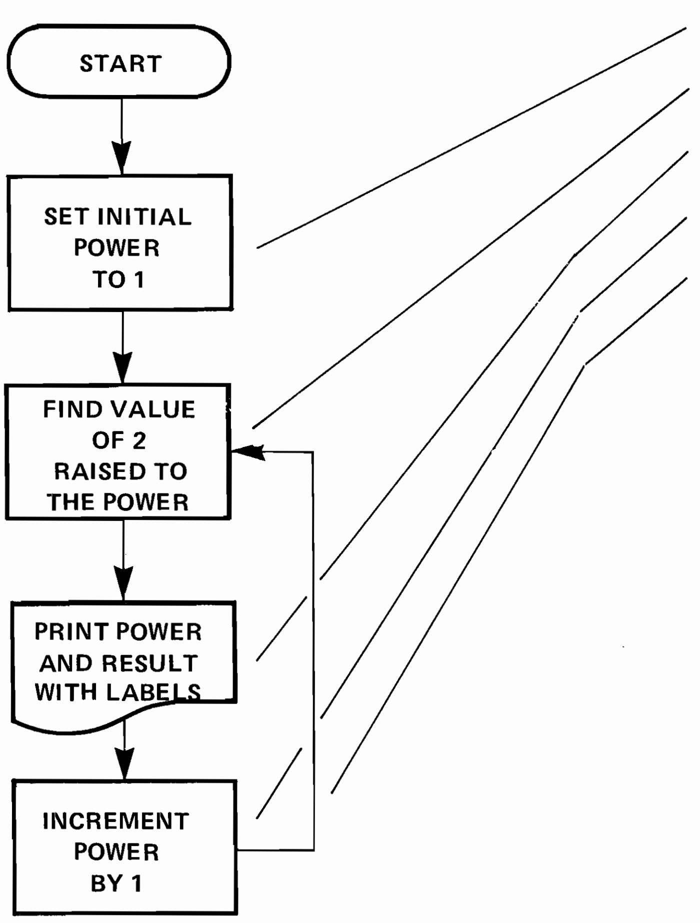
SECTION 10–1
THE GOTO STATEMENT

Consider the program and associated flow chart in Fig. 10–1. Enter the program, SELECT a half-second pause (SELECT P3), and RUN.

Notice that, in the flow chart, the GOTO statement is represented by connecting lines with an arrow head. In this particular program then, the GOTO statement forms an infinite loop; program execution does not terminate by itself.

The General Form of a GOTO statement is

GOTO line number


START
SET INITIAL POWER TO 1
FIND VALUE OF 2 RAISED TO THE POWER
PRINT POWER AND RESULT WITH LABELS
INCREMENT POWER BY 1

```
10 P=1
20 Q=2↑P
30 PRINT "POWER=";P, "Q=";Q
40 P=P+1
50 GOTO 20
```

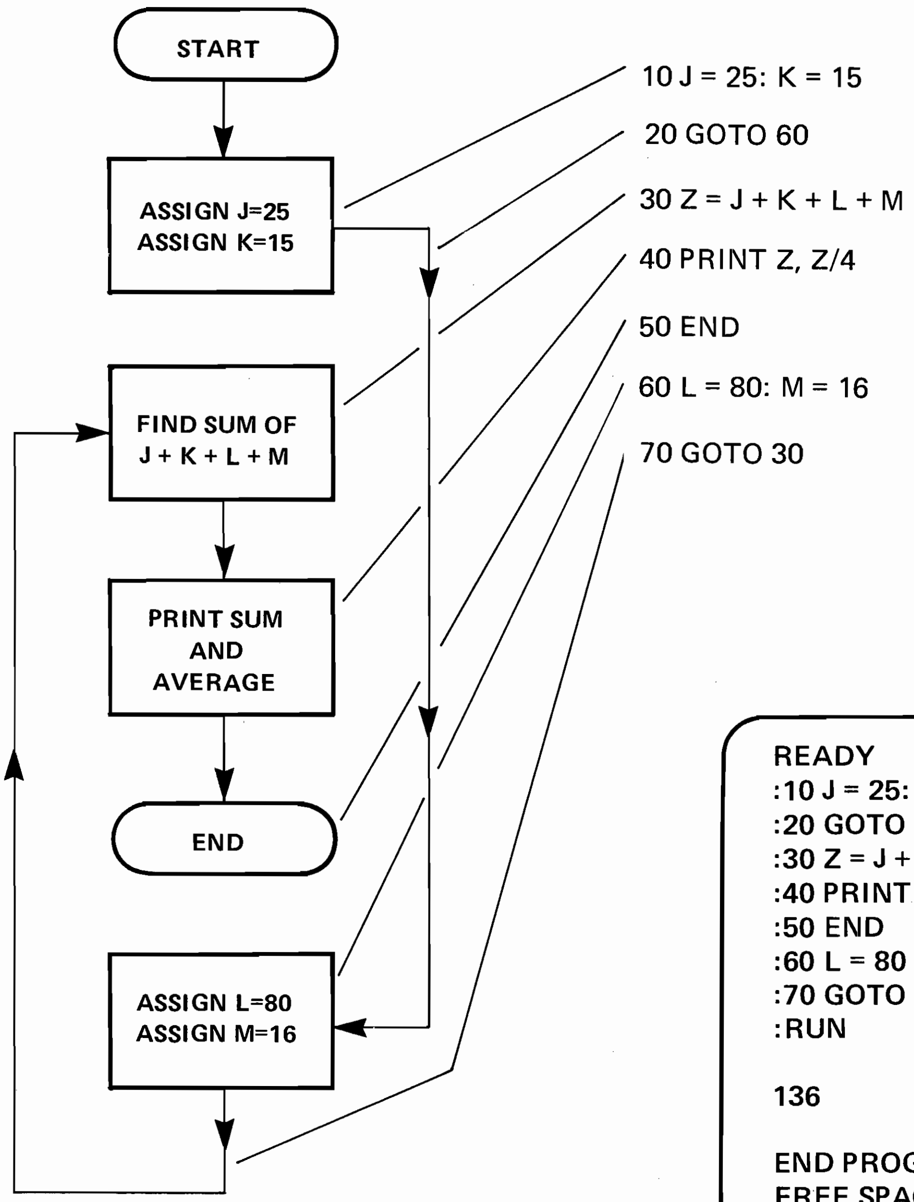
```
READY
:10 P = 1
:20 Q = 2↑P
:30 PRINT "POWER=";P, "Q=";Q
:40 P=P+1
:50 GOTO 20
:SELECT P
:RUN
POWER = 1          Q = 2
POWER = 2          Q = 4
POWER = 3          Q = 8
   .                  .
   .                  .
   .                  .
POWER = 332        Q = 8.74900289E+99
20Q = 2↑P
          ↑ ERR 03
:_
```

Fig. 10–1

CHAPTER 10
THE UNCONDITIONAL BRANCH

Consider Fig. 10–1a which uses two GOTO statements.

This program does not form an infinite loop because of the END statement entered in line 50. Consider what might happen if line 50 were not entered.



```
10 J = 25: K = 15
20 GOTO 60
30 Z = J + K + L + M
40 PRINT Z, Z/4
50 END
60 L = 80: M = 16
70 GOTO 30
```

```
READY
:10 J = 25: K = 15
:20 GOTO 60
:30 Z = J + K + L + M
:40 PRINT Z, Z/4
:50 END
:60 L = 80 : M = 16
:70 GOTO 30
:RUN

136                 34

END PROGRAM
FREE SPACE = 3251

:_
```

Fig. 10–1a

CHAPTER 11
THE DATA AND READ STATEMENTS

Thus far in this manual, we have only looked at one method of assigning values to variables in a program. Generally, each variable that receives a new value requires a separate assignment statement; for example:

```
10  LET A = 17.3
20  B = 23.9 :C = -11.4 :D = 1.3E4
30  LET E = SQR (A*2+B↑2+C↑2+D12)
40  PRINT "E="; E
```

Using separate assignment statements in this way becomes inconvenient if there are many values to be assigned. The DATA statement and the READ statement combine to make the task of assigning many values to variables more efficient.

SECTION 11–1
DATA AND READ STATEMENTS

The DATA statement is used to store numeric and alphanumeric data in a program. The statement can only be used in the program mode, and consists of the BASIC keyword DATA followed by one or more values separated by commas:

```
100 DATA 17.3, 23.9, -11.4, 1.3E4
```

The system automatically sets a data pointer to the location of the first value. It uses this pointer to keep track of the next value to be used in the program.

It does not matter whether all the data is included in one DATA statement or several. The statements below are equivalent to the previous example.

```
100  DATA 17.3
110  DATA 23.9, -11.4
120  DATA 1.3E4
```

Fig. 11–1

The order in which the data appears, however, is important. When the values are stored, they are stored in sequential order as they appear in the program statements. The data pointer is always initially set to the first value stored.

In order to use the values that have been stored, it is necessary to assign variable names to each value before it is used. This is the purpose of the READ statement. The READ statement is composed of the BASIC keyword READ followed by one or more variable names separated by commas (See Fig. 11–1a).

```
10   DATA 17.3, 23.9, -11.4, 1.3E4
20   READ A,B,C,D
```

Fig. 11–1a

The READ statement in line 20 sequentially assigns the four values in the DATA statement to the variables in the READ statement. Thus A = 17.3, B = 23.9, C = –11.4, and D = 1.3E4; these values may now be used in subsequent calculations. All the data does not need to be read at one time with a single READ statement. If fewer values are read than have been stored, the data pointer automatically keeps track of the last value read (See Fig. 11–1b and Fig. 11–1c).

```
READY
:10  DATA 17.3, 23.9, -11.4
:20  DATA 1.3E4
:30  READ A,B
:40  PRINT"A="; A, "B=";B
:50  READ X,Y
:60  PRINT "X="; X, "Y=";Y
:RUN
A=17.3            B=23.9
X=-11.4           Y=13000

:_
```

Fig. 11–1b

CHAPTER 11
THE DATA AND READ STATEMENTS

```
:10  READ A,B,C,D
:20  E = SQR(A↑2+B↑2+C↑2+D↑2)
:30  PRINT "E="; E
:40  DATA 17.3, 23.9, -11.4, 1.3E4
:RUN
E= 13000.03848

:_
```

Fig. 11–1c

SECTION 11–2
USING THE RESTORE STATEMENT

The examples given in Section 11–1 show the READ statement(s) reading data values sequentially from DATA statements. However, once the programs are executed, the data in the DATA statements can not be re-used, unless the programs are RUN again. A method is required that allows the data to be read more than once within a program. This is accomplished through the use of the RESTORE statement.

The RESTORE statement resets the pointer, which allows the data in the DATA statement to be re-used without having to RUN the program again.

GENERAL FORM
RESTORE [expression] [1]

The expression in the RESTORE statement is evaluated by the 2200 and truncated to an integer. The value of the integer represents the position of the next data value to be retrieved by the READ statement. For example, if the value of the expression is 3, the next READ statement will retrieve data, beginning with the third data item stored. If the [expression] is omitted, the next READ statement will retrieve data starting with the first data item stored.

Consider the program in Fig. 11–2. Statement line 20 restores the data pointer, starting at the first value (line 40). Compare this program to the one in Fig. 11–2a, where line 20 restores the data pointer to the third value (line 40).

[1] [ ] = optional, not required.

EXAMPLES OF RESTORE AND RESTORE n

Execution of line 20 restores the entire data string, beginning at the first value.

```
:10  READ M, N, O, P
:20  RESTORE
:30  READ Q, R, S, T, U
:40  DATA 100, 200, 300, 400, 500, 600, 700
:50  PRINT M; N; O; P; Q; R; S; T; U
:RUN

100 200 300 400 100 200 300 400 500

:_
```

Fig. 11–2

Execution of line 20 restores the data string beginning with the 3rd value, 300.

```
:10  READ M, N, O, P
:20  RESTORE 3
:30  READ Q, R, S, T, U
:40  DATA 100, 200, 300, 400, 500, 600, 700
:50  PRINT M; N; O; P; Q; R; S; T; U
:RUN

100 200 300 400 300 400 500 600 700

:_
```

Fig. 11–2a

The RESTORE command can also be used to skip over values in a DATA statement. Consider Fig. 11–2b. Notice that the execution of statement line 20, RESTORE 5, causes the 2200 to skip to the fifth data value for the subsequent READ statement in line 30.

The RESTORE command can thus be used to reset the data pointer to any item in the stored data. In situations where there are multiple data statements, data values are stored sequentially, beginning with the first value in the lowest numbered DATA statement line. Any attempt to RESTORE to a non-existent data value (i.e., – RESTORE 8 or higher in Fig. 11–2b) results in an error message and termination of the program execution.

```
:10 READ M, N
:20 RESTORE 5
:30 READ O, P
:40 DATA 100, 200, 300, 400, 500, 600, 700
:50 PRINT M; N; O; P
:RUN
 100  200  500  600
:_
```

Fig. 11–2b

CHAPTER 12
MAKING DECISIONS

One of the most important capabilities of a calculating/computing system is the ability to test values and make decisions. When such a test or decision occurs within a program, the resulting program flow is made *conditional* upon the relationship tested.

In BASIC this concept is known as a *conditional branch*. This chapter serves to introduce the concept of the conditional branch as it is used in 2200 BASIC.

SECTION 12–1
THE USE OF THE IF/THEN STATEMENT

Among the programming concepts introduced in the flow-charting section of Chapter 9, was the idea of a *decision*. Briefly, a decision is represented in a flow-chart as a diamond (Fig. 12–1)

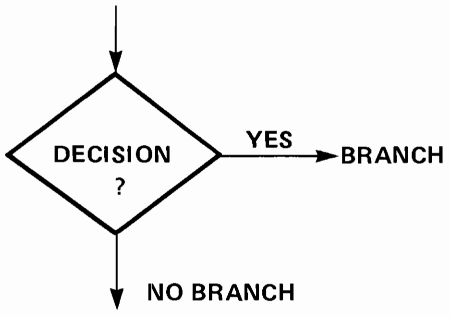


Fig. 12–1

Although a FOR/NEXT loop has a built-in test, BASIC enables the programmer to directly specify other decisions, giving an even greater programming flexibility.

Consider the flow-chart in Fig. 12–1a.

What is happening?

1. A value is read for a variable S (from a DATA statement).
2. The value is tested. If it is greater than 10, the value is squared and assigned to the variable T. If it is less than or equal to 10, the value is cubed, and assigned to the variable T.
3. The values of S and T are printed.
4. The 2200 is directed back to read another value for S, and the procedure is repeated, until there are no more data values.

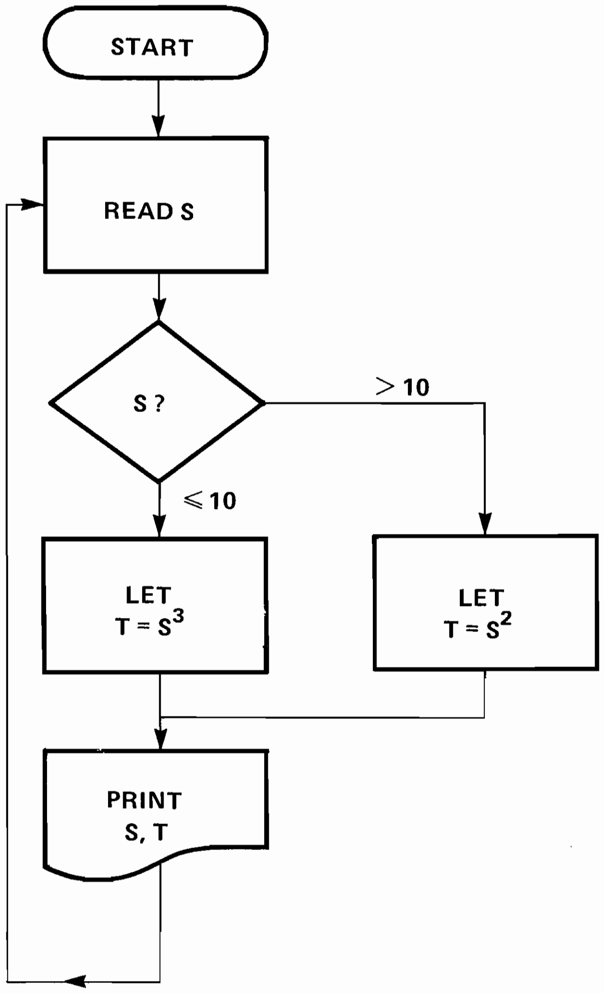


Fig. 12–1a

CHAPTER 12
MAKING DECISIONS

What are the program lines which correspond to the flow-chart? The READ S process is a READ statement; the assignment and PRINT statements are familiar. The test however, is accomplished through the use of an IF/THEN statement. The associated program is given in Fig. 12—1b.

The Program

```
10 READ S

20 IF S> 10 THEN 60

30 T = S↑3

40 PRINT S, T

50 GO TO 10

60 T=S↑2

70 GO TO 40

80 DATA 2, 5, 11, 3

90 DATA -8, 13, 10
```

The CRT Display

```
READY
:10 READ S
:20 IF S> 10 THEN 60
:30 T = S↑3
:40 PRINT S, T
:50 GO TO 10
:60 T = S↑2
:70 GO TO 40
:80 DATA 2, 5, 11, 3
:90 DATA -8, 13, 10
:RUN

 2              8
 5              125
 11             121
 3              27
-8             -512
13              169
10              1000
10 READ  S
           ↑ ERR 27

:_
```

Fig. 12—1b

Analyzing the program line by line:

Statement line 10 says to READ a value from a DATA statement. Statement line 20 contains the *conditional* IF/THEN branch. The 2200, in executing statement line 20 in the program, looks at the value of S, which has been read, and checks whether the condition in the statement is true. If the statement is true, the 2200 *branches* to the number mentioned in the statement. If the statement is not true, the 2200 *goes to the next BASIC statement in sequence,* whether the statement is in the same statement line, or is in the following statement line.

The first value for S in this program is 2; since 2 is not greater than 10, the 2200 *does not* jump to line number 60. It goes instead to the next *statement* in the program; in this program, the next statement is statement line 30.

Observe that the use of IF and GO TO statements force the 2200 to follow the paths shown in the flow-chart. Note the placement of the GO TO statements in line numbers 50 and 70.

There are 7 decisions made during the execution of this program. In two of the decisions, the 2200 goes to statement line 60; in five, the 2200 goes to statement line 30. In this example, when the 2200 runs out of data, processing stops and an error message is printed.

Thus, in this example, the IF/THEN statement allowed the 2200 to differentiate between values less than 10, and values greater than or equal to 10. The general format of an IF/THEN statement is illustrated in Fig. 12–1c.

GENERAL FORMAT

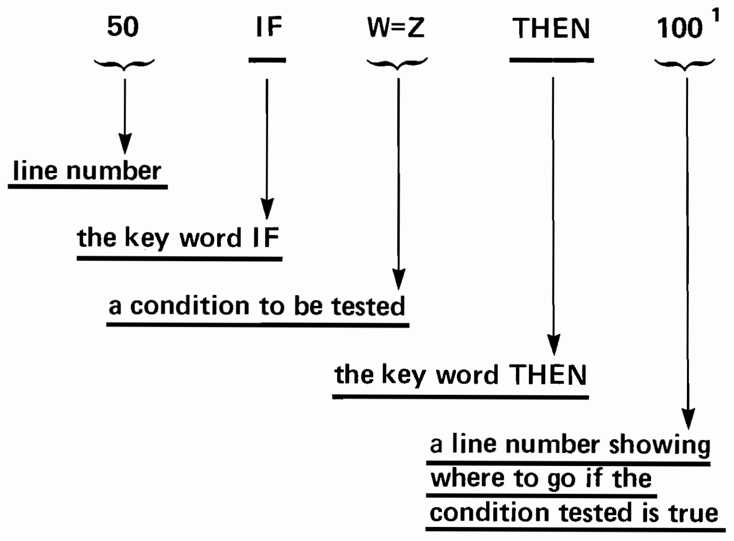


Fig. 12–1c

The key part of the IF/THEN statement is the condition to be tested. The condition is always composed of three parts:

1. The "subject" -- part to be tested
2. The "object" -- part the test is made against
3. The "relation" -- type of comparison to be made

– The subject and object are expressions and must appear on either side of the relation.

– The relation can be any one of the six given in Fig. 12–1d.

[1] The line number following the keyword THEN must be somewhere in the program.

Relation		Generated by	
=	equals	=	key
>	greater than	>	key
<	less than	<	key
> =	greater than or equal	> and =	keys
< =	less than or equal	< and =	keys
<>	not equal	< and >	keys

Fig. 12—1d

EXAMPLES OF LEGAL IF/THEN STATEMENTS

```
10  IF  X>Y THEN 50
15  IF  T6<14 THEN 80
20  IF  16>1.5 * T THEN 80
35  IF  A↑B<>C↑D THEN 14
50  IF  SQR (M + 7) - L<=0 THEN 100
```

Fig. 12—1e

EXAMPLES OF THE IMPROPER USE OF IF/THEN STATEMENTS

```
25 IF W = X GO TO LINE 70
```

- improper form - should be

```
20 IF W = X THEN 70
```

```
40 IF Y <7 THEN GO TO 50
```

- improper form - should be

```
40 IF Y <7 THEN 50
```

```
20 IF C = 4 * B THEN 21
21 PRINT C
```

If the value of C *does* equal the value of 4 * B, the 2200 jumps to line number 21 because the 2200 finds that the condition to be tested is true. Observe, though, that if C *does not* equal 4 * B, the 2200 goes to the same place. Thus, in this example, the IF/THEN statement is meaningless.

Fig. 12—1f

Given the previous information, reconsider the program in Fig. 12—1b. How can this program be altered to enable the 2200 to sense when all appropriate data has been read? One way is to add on an extra item of test data (for example, 9999) which can be tested immediately after the READ statement. If the value read was found to be the test value, execution would be halted, via a transfer in the program, to an END or STOP statement. Otherwise, the program would continue to be processed.

The operation described above is a common programming procedure. To change the program three additional statements must be included: an IF/THEN statement, another DATA statement, and an END or STOP statement. Fig. 12—1g shows the changed program, and the results. Notice the position in the program of the IF/THEN and END statements. (the only requirement for the extra DATA value is that it be the last data value available in the program).

Fig. 12—1h and 1i represent examples of the use of the RESTORE command with an IF/THEN test for a test entry of 9999. In the following example (Fig. 12—1h), a number of students have taken a test, and the difference between each grade and the average grade of the class is desired for each student.

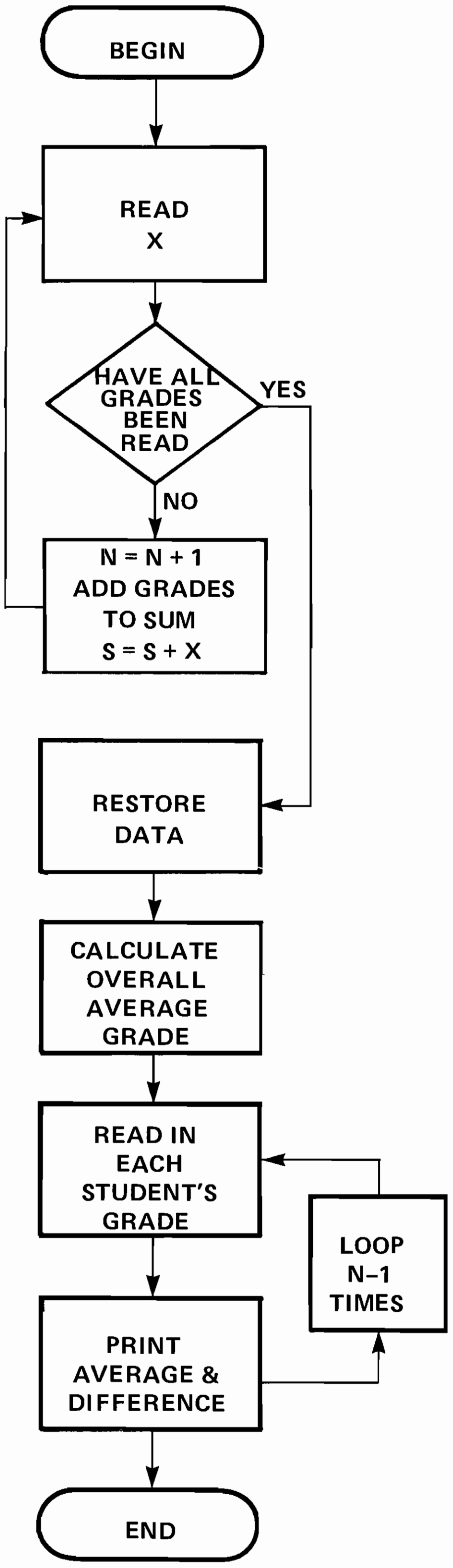
A DATA statement is used to read the grades and calculate the average of the class. A test data value is included at the end of the DATA string, and an IF/THEN statement is used to test for this value, which signifies all data values have been read. Following the reading of all data, the RESTORE statement restores the data for another execution. A FOR/NEXT loop is used for this process, with the number of loops controlled by the number of grades previously read in the program (i.e., N). The differences are calculated, and all the data is printed.

```
READY
:10 READ S
:15 IF S = 9999 THEN 110
:20 IF S > 10 THEN 60
:30 T = S ↑ 3
:40 PRINT S, T
:50 GOTO 10
:60  T = S ↑ 2
:70 GOTO 40
:80 DATA 2, 5, 11, 3
:90 DATA -8, 13, 10
:100 DATA 9999
:110 END
:RUN

 2          8
 5          125
 11         121
 3          27
-8         -512
 13         169
 10         1000

END PROGRAM
FREE SPACE = 3226

:_
```

Fig. 12—1g

```
10   S,N = 0
20   READ X
30   If X = 9999 then 80
40   N = N + 1
60   S = S + X
70   GO TO 20
80   RESTORE
90   A = S/N
100  FOR I = I to N
105  READ X
110  PRINT I, X, X-A, A
120  NEXT I
130  DATA 84, 63, 77, 93, 47, 72
140  DATA 86, 58, 75, 66, 9999
150  END
```

Fig. 12—1h


BEGIN
READ X
HAVE ALL GRADES BEEN READ
YES
NO
N = N + 1 ADD GRADES TO SUM S = S + X
RESTORE DATA
CALCULATE OVERALL AVERAGE GRADE
READ IN EACH STUDENT'S GRADE
LOOP N–1 TIMES
PRINT AVERAGE & DIFFERENCE
END

```
:RUN
 1          84          11.9          72.1
 2          63         -9.1           72.1
 3          77          4.9           72.1
 4          93          20.9          72.1
 5          47         -25.1          72.1
 6          72         -.1            72.1
 7          86          13.9          72.1
 8          58         -14.1          72.1
 9          75          2.9           72.1
 10         66         -6.1           72.1
END PROGRAM
FREE SPACE=3143

:_
```

Fig. 12—1i

CHAPTER 12
MAKING DECISIONS

Fig. 12–1j is a flow-chart representing the solution to the problem of determining which of three different given numbers is the largest. The values must be compared to one another, which involves the use of an IF/THEN statement.

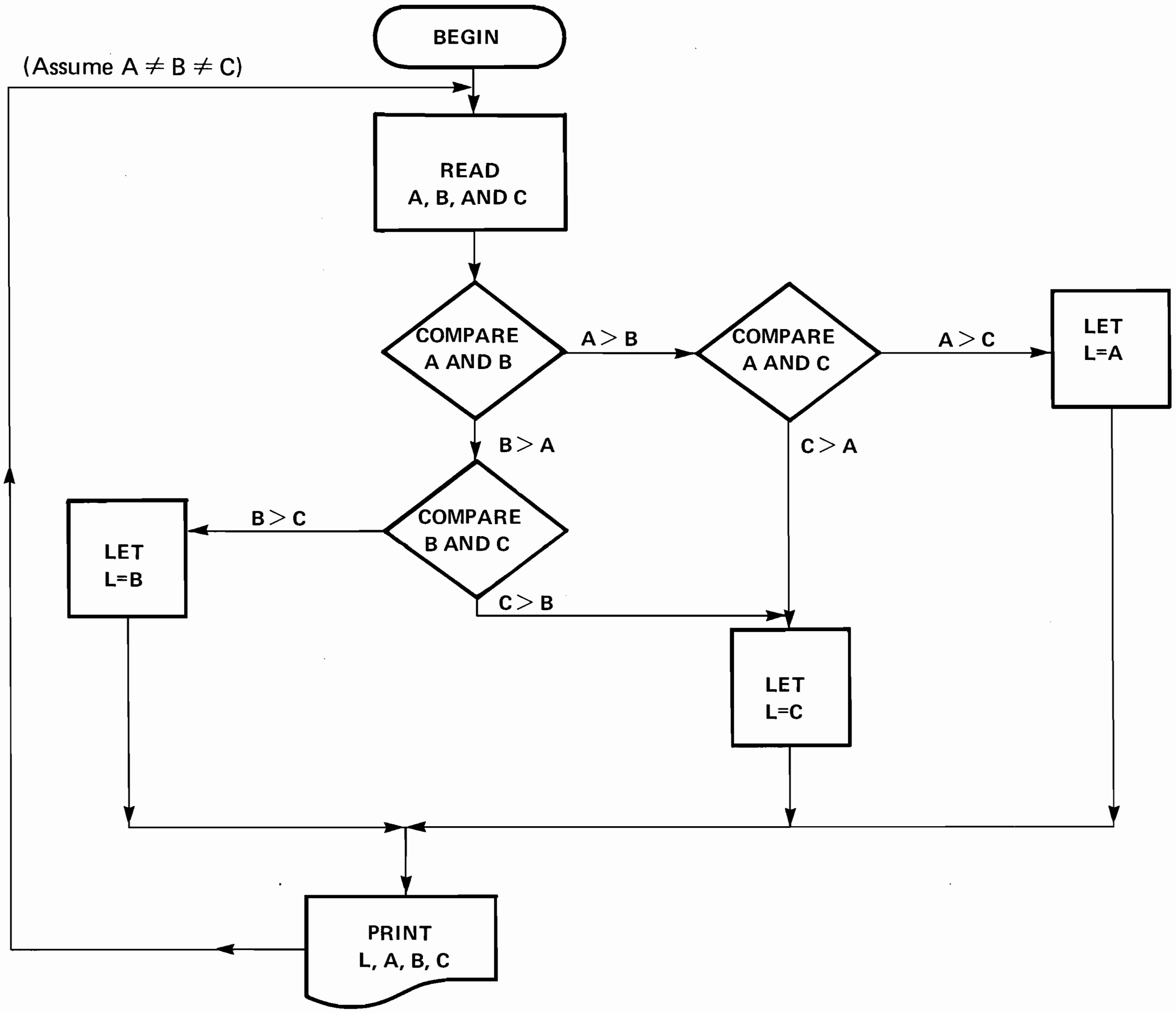


Fig. 12–1j

Since it is assumed that the three values are different, there is no need to test for the "equals" condition. The largest value, when found, is assigned to the variable L, and all four variables are printed. The 2200 is then instructed to go back and repeat the process. Again, as in previous examples, looping terminates when the 2200 runs out of values in the DATA statements. Fig. 12–1k shows the associated program.

The CRT Display

```
READY
:10 READ A, B, C
:20 IF A > B THEN 90
:30 IF B > C THEN 70
:40 L = C
:50 PRINT L, A, B, C
:60 GO TO 10
:70 L = B
:80 GO TO 50
:90 IF A > C THEN 110
:100 GO TO 40
:110 L = A
:120 GO TO 50
:130 DATA 3, 1, 4, 9, 2, 6, 7, 8
:140 DATA 2, 7, 6, 5, 1, 2, 3
:RUN

4      3      1      4
9      9      2      6
8      7      8      2
7      7      6      5
3      1      2      3

10 READ A, B, C

              ↑ERR27

:_
```

Fig. 12–1k

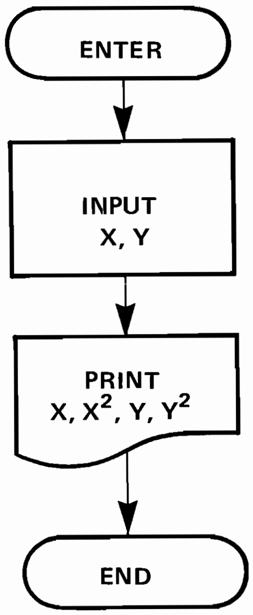
CHAPTER 13
INTERACTIVE PROGRAMMING – USING THE INPUT STATEMENT

By using assignment, READ and DATA statements, a programmer is able to enter all the necessary data, prior to running the program. These statements require that data be contained within the actual program text. Should the programmer wish to alter the data in any way he must change one or more complete statement line. This approach is an effective means of storing constants that remain the same each time the program is executed, but it is not suited to assigning values that may change each time the program is executed. The INPUT statement allows the user to key in data after program execution has begun. The data does not become a part of the program text.

SECTION 13–1
THE INPUT STATEMENT

The BASIC INPUT statement allows the user to enter variable values at selected points in a program, while the program is running. Fig. 13–1 is a simple example of the use of the INPUT statement.

FLOW DIAGRAM



PROGRAM

```
10 INPUT X, Y
20 PRINT X, X↑2, Y, Y↑2
30 END
```

Fig.13–1

When a program with an INPUT statement is executed, the 2200 continues executing line by line until program flow reaches the INPUT statement. The 2200 then stops execution and prints out a question mark "?". The user is then expected to enter data values, one for each variable named in the INPUT statement, separated by commas. When the CR/LF key is touched, program execution continues.

Fig. 13–1a gives the results of executing the program in Fig. 13–1.

```
READY
:10  INPUT  X,Y
:20  PRINT  X, X↑2, Y, Y↑2
:30  END
:RUN
 ? 3, 4
  3       9       4       16

END PROGRAM
FREE SPACE = 3340

:_
```

Fig. 13–1a

Notice that statement line 20 requests a value for two separate variables. Thus, when the program is run, the question mark signifies that two values, separated by a comma, are required. If less than the required number of values (which in this case is two) are given before the CR/LF–EXECUTE key is touched, the 2200 will continue putting "?" 's in the display until all the requested values have been entered. If more values are entered than required, the additional values are ignored. The General Form of the INPUT statement is:

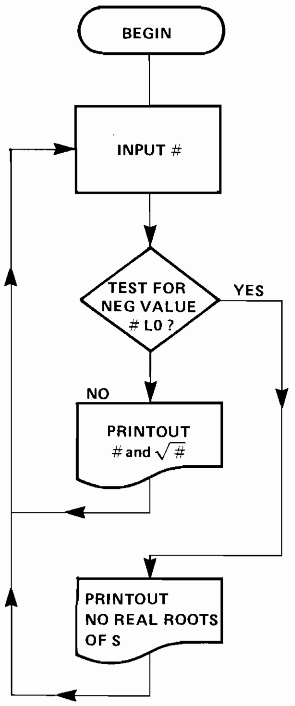
INPUT variable [{variable} ...]
INPUT "literal string", variable [{variable} ...] [1]

where the literal string is used to identify the requested input there is no limit to the number of characters that may appear in the string as long as the maximum line length does not exceed 192 keystrokes.

SECTION 13–2
INPUT WITH AN INCLUDED TEXT STRING

The general form of the INPUT statement allows for the inclusion of a literal string in quotation marks, before the INPUT variable(s). When the statement is executed, the literal string is printed out followed by a question mark. This feature allows the programmer to output a message to the user before the required data is keyed in.

Fig. 13–2 is an example of the use of an INPUT statement containing a literal string. The program itself is set up as an infinite loop, because the GO TO statements at lines 50 and 70 always direct program flow back to the first statement line.



```
10 INPUT "NEXT VALUE", S
20 REM TEST FOR NEG. VALUE
30 IF S < 0 THEN 60
40 PRINT "SQUARE ROOT OF"; S; "IS"; SQR(S)
50 GO TO 10
60 PRINT "NO REAL ROOTS OF"; S
70 GO TO 10
```

```
READY
:10 INPUT "NEXT VALUE", S
:20 REM TEST FOR NEG' VALUE
:30 IF S < 0 THEN 60
:40 PRINT "SQUARE ROOT OF";S;"S";SQR(S)
:50 GOTO 10
:60 PRINT "NO REAL ROOTS OF";S
:70 GOTO 10
:RUN
 NEXT VALUE?_
```

Fig. 13–2

[1] [ ] = optional

CHAPTER 14
ARRAYS, AND ARRAY VARIABLES

In addition to the simple (scalar) numeric variables described and used thus far, 2200 BASIC can define and use *array variables.*

SECTION 14–1 WHAT ARE ARRAYS?

$$\begin{array}{cccccc} & & & \text{COLUMNS} & & \\ & 7 & 6 & 5 & 4 & 9 \\ \text{R} & 1 & \textcircled{9} & 2 & 2 & 3 \\ \text{O} & 6 & 2 & 1 & 4 & 10 \\ \text{W} & 5 & 11 & 4 & 9 & 5 \\ \text{S} & 9 & \textcircled{10} & 6 & 1 & 5 \end{array} \qquad \begin{array}{c}\text{ARRAY}\\ =\text{R}=\end{array} \qquad \begin{array}{ccccc} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ r_{21} & \textcircled{r_{22}} & r_{23} & r_{24} & r_{25} \\ r_{31} & r_{32} & r_{33} & r_{34} & r_{35} \\ r_{41} & r_{42} & r_{43} & r_{44} & r_{45} \\ r_{51} & \textcircled{r_{52}} & r_{53} & r_{54} & r_{55} \end{array}$$

Fig. 14–1

An *array* is simply a set of numbers arranged in a table, such that each number is uniquely identified by its position. In 2200 BASIC, arrays can have either one or two dimensions. In two dimensional arrays, each element is identified by its numerical *row* and *column* position.

Consider Fig. 14–1 which represents a square array, of 5 columns and 5 rows. A square array has the same number of rows as columns.

The array is called R ( ).

Notice that each *element* of the R array on the left is associated with a *subscripted* letter on the right. The circled number 9 is in row 2 and also in column 2. Standard mathematical subscript notation indicates this by "r_{22}". In 2200 BASIC a statement assigning the value of 9 to the element in the 2nd row and 2nd column in this R array is:

R(2,2) = 9

Likewise, the circled number 10 is in row 5, column 2, and is identified as "r_{52}". In 2200 BASIC the notation is simply:

R(5,2)

The first subscript denotes the row, and the second subscript denotes the column, in which the element appears.

An array may consist of either a single row or single column, as in Fig. 14–1a.

$$\begin{array}{cc} \begin{bmatrix} r_1 \\ r_2 \\ \cdot \\ \cdot \\ r_i \end{bmatrix} & [S_1 \; S_2 \; . \; . \; . \; . \; . \; S_j] \\ \text{Column Array} & \text{Row Array} \end{array}$$

Fig. 14–1a

SECTION 14–2 NAMING AND DIMENSIONING ARRAYS

In 2200 BASIC, *array variable* names are the same as simple (scalar) numeric variable names except for the appropriate subscripts enclosed in parentheses. There are therefore 286 array names available. (A–Z, A0–Z9). The subscripts contained in the parentheses describe a particulәr element's position in the array.

EXAMPLES OF LEGAL ARRAY NAMES

A(5)	M(20)
X4(8,6)	P3(10,10)

Fig. 14–2

CHAPTER 14
ARRAYS, AND ARRAY VARIABLES

Before an array or any of its elements can be used space must be set aside in memory for the entire array. This is accomplished using a DIM (dimension) statement. The maximum size of either dimension (i.e. number of rows or number of columns) is 255. The General Form of the DIM statement is as follows:

GENERAL FORM

DIM 'dim element' [$\left\{ \text{'dim element'} \right\}$ ...]

where 'dim element' = $\left\{ \begin{array}{l} \text{numeric array variable} \\ \text{alpha array variable}^1 \text{ [integer]} \\ \text{alpha scalar variable}^1 \text{ [integer]} \end{array} \right\}$ $0 < \text{integer} < 65$

Examples of dimensioned numeric array variables

DIM A(5,2)	Reserves space for a 2 dimensional array of 10 elements (5 X 2)
DIM A(5,2), B(3,1)	Reserves space for two, 2 dimensional arrays of 10 (5 X 2) and 3 (3 X 1) elements
DIM B(6,1), C(3,2), E(5,1)	Reserves space for three, 2 dimensional arrays of 6, 6, and 5 elements respectively.
DIM E1(5)	Reserves space of a single dimension array of 5 elements

Space can be reserved for more than one array in a single DIM statement by separating the entries for array names with commas. The DIM statement must appear before any use of the variables in a program, and the space to be reserved for the array must be explicitly indicated. Subscripts cannot be variables or variable expressions, but must be integer values (1 to 255). The numeric value of the subscript *cannot be a zero.*

Once a numeric array is dimensioned, the initial value of each element is 0, and each element can be used like a regular variable, (Fig. 14–2a) [2] .

```
READY

:10 DIM X(5,5), W(8, 10)
:20 X( 1,3)=25*6.342
:100 IF W(8,5) < 13 THEN 50
:120 Y=W(2,3) * X(3,2)
```

Fig. 14–2a

Except in the DIM statement, array subscripts can be any variable or variable expression whose value is greater than 0 and less than 256. Thus, the subscript can be computed. Fig. 14–2b is an example of a program which computes the position of the array and assigns the value of 5 for each element. In this example, a FOR/NEXT loop is used to change the subscript, from element to element of the array.

```
READY

:10 DIM X( 100)
:20 FOR I = 1 TO 100
:30 X(I) = 5
:40 NEXT I
```

Statement 10 – sets aside 100 spaces in memory for array X.

Statement 20 – sets up a counter where I goes from 1 to 100. I is a variable to represent the subscript of array X.

Statement 30 – assigns the value 5 to every element in the array X where I will go from 1 to 100 ($x_1, x_2, x_3, x_4 \ldots x_{100}$).

Statement 40 – increments the counter by one each time the program loops.

Fig. 14–2b

[1] Alphanumeric variables are discussed in Chapter 16.

[2] The subscript of the first element of an array is always "1", never "0". Thus, X(1) is proper, whereas X(0) is not, and causes an error message to be generated.

CHAPTER 15
NESTED LOOPS

Earlier in this volume the FOR/NEXT loop was introduced and explained. All of the examples showed only one loop within a program. A programming technique called nested loops can be done on the 2200. Nested loops are loops within loops. This chapter uses nested loops to show you how to set up a two-dimensional array in memory and assign values to each element of the array.

SECTION 15–1 NESTED LOOPS

In the last chapter a FOR/NEXT loop was used to define and establish a single dimension array. This did not involve any complex programming; since this type of array was only a single dimension. There was no question as to which element was being referred to, or the order in which the reference was made.

The situation with two dimensional arrays involves somewhat more programming. A natural question at this point is: "If a single dimensional array uses a single FOR/NEXT loop, could two FOR/NEXT loops be used with a two dimensional array?" The answer is "yes". The concept involved is a loop within a loop, and is referred to as *nested loops.*

Consider Fig. 15–1, which shows a nested loop approach to establishing all elements of the array and assigning each the value "1".

```
            10 DIM X(5,3)

            20 FOR B = 1 TO 5
O   I
U   N       30 FOR C = 1 TO 3
T   N
E   E       40 X(B,C) = 1
R   R
            50 NEXT C

            60 NEXT B

            70 END
```

$$X(5,3) = \begin{array}{ccc}
X_{1,1} & X_{1,2} & X_{1,3} \\
X_{2,1} & X_{2,2} & X_{2,3} \\
X_{3,1} & X_{3,2} & X_{3,3} \\
X_{4,1} & X_{4,2} & X_{4,3} \\
X_{5,1} & X_{5,2} & X_{5,3}
\end{array}$$

5 ROWS

3 COLUMNS

Fig. 15–1

When this program is executed, the X array is dimensioned as 5 by 3, and 15 spaces are set aside in memory for this array. Statement line number 20 sets up a FOR/NEXT loop, where the row position is given the name B and a counter is set up from 1 to 5. Statement line # 30 sets up a FOR/NEXT loop, where the column position is given the name C and a counter is set up from 1 to 3. Execution of both of these statements the first time results in the subscripts of array X being assigned the value of $X_{1,1}$. Therefore, when statement number 40 is executed $X_{B,C}$, or $X_{1,1}$ is set to equal 1. Statement number 50 is then executed which results in C being incremented by 1 and tested to see if C has reached three. If not, statement 40 is executed and $X_{B,C}$ now equals $X_{1,2}$ and $X_{1,2}$ is set to equal 1. This continues until the inner loop or C = 3 is satisfied or $X_{1,3}$ = 1. Then statement # 60 is read. Since this is part of the outer (B) loop, B is incremented, by 1 and tested to see if equal to 5, if not B is set to 2 and statement 30, 40, and 50 are repeated. This results in $X_{2,1}$, being set to $X_{2,1}$, $X_{2,2}$, and $X_{2,3}$. This process continues until B is set to 3, 4, and 5 respectively, and is terminated when $X_{B,C}$ = $X_{5,3}$ or the last element of the array is defined and set to equal one. Then statement 70 is executed and the program ends.

Thus the key to understanding nested loops, is that the inner loop goes through an entire processing, for each time the outer loop goes through one process. When the inner loop is finished the program jumps back to the outer loop and the process starts again until the outer loop is completed. There can be any number of nested loops. The only requirements are that each loop have a different counter variable (variables B and C in the previous example), and that the loops not overlap, that is the inner loop must be satisfied or completed before trying to go back to the outer loop, An example of this is shown in Fig. 15–1a.

```
ILLEGAL Loop OVERLAP
┌──── 10   FOR I  = 1 TO 5
│ ┌── 80   FOR J  = 1 TO 3
└─┼─► 100  NEXT I
  └─► 200  NEXT J
```

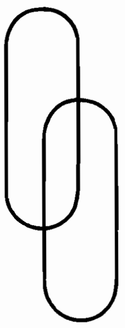


Fig. 15–1a

Fig. 15–1b is another example of a nested loop with an array. In this case, the elements of the 4 by 4 array are assigned different values increasing from 1 to 16.

PROGRAM

```
10 DIM F2 (4,4)
20 FOR I=1 TO 4
30 FOR J=1 TO 4:N=N+1
40 F2 (I,J)=N
50 PRINT F2 (I,J),
60 NEXT J
70 PRINT
80 NEXT I
90 STOP
```

OUTPUT

```
:RUN

 1         2          3          4

 5         6          7          8

 9         10         11         12

 13        14         15         16

 STOP

:_
```

Fig. 15–1b

SECTION 15–2 OTHER USES OF NESTED LOOPS

The use of nested loops is not restricted to arrays. There are many situations where an evaluation of an expression with several variables is required. Nested loops become valuable in these situations, because they enable a programmer to hold one variable at a constant value, while varying the value of another over a range of values. The process can thus automatically be repeated for another constant value, while the second (or third, fourth,...) variable again takes on a set of values.

Fig. 15–2 is an example of such a situation as applied to a mortgage payment calculation. Notice that both NEXT statements appear in the same statement line – line 60. However, the NEXT N statement precedes the NEXT I statement. Thus, the FOR/NEXT N loop is completely within the FOR/NEXT I loop, as required.

SAMPLE PROGRAM – NESTED LOOP

MORTGAGE PAYMENT PROBLEM, letting interest rate and loan period vary

GIVEN: (P)

$$M = \frac{(P)\,\frac{I}{12}}{1 - \left(1 + \frac{I}{12}\right)^{-12N}}$$

where P is the principal or amount borrowed (in dollars)

I is the interest rate which is expressed as a yearly rate; i.e., 6 percent per annum is equivalent to .06.

N is the number of years representing the period of the loan.

M is the amount of the monthly mortgage payment.

IF P = $40,000, then

M = (40000 * I/12)/(1-(1+I/12) ↑ (-N * 12)

If we let interest vary from 7½% to 9% in ½% increments and let the number of years of repayment vary from 20 to 30 yrs. in 5 year increments the problem becomes

```
10  PRINT "AMOUNT BORROWED", "INTEREST RATE", "NO. OF YEARS"; " MO. PYMT."
20  FOR I = .075TO .090STEP .005 :REM INTEREST RATE VARIES
30  FOR N=20TO 30STEP 5 :REM YEARS OF REPAYMENT VARIES OVER EACH INTEREST RATE
40  M = (40000) * ( I/12)/(1-(1+I/12) ↑ (-N*12) )
50  PRINT "$40,000", 100*I, "%", N; TAB(45); "$"; M
60  NEXT N:NEXT I
70  STOP
```

```
:RUN

AMOUNT BORROWED      INTEREST RATE      NO. OF YEARS      MO. PMNT.

$ 40,000                7.5 %                20           $ 322.2372774218
$ 40,000                7.5 %                25           $ 295.5964711202
$ 40,000                7.5 %                30           $ 279.6858034216
$ 40,000                8 %                  20           $ 334.5760276139
$ 40,000                8 %                  25           $ 308.726487759
$ 40,000                8 %                  30           $ 293.5058295595
$ 40,000                8.5 %                20           $ 347.1292933536
$ 40,000                8.5 %                25           $ 322.0908333922
$ 40,000                8.5 %                30           $ 307.5653934374
$ 40,000                9 %                  20           $ 359.8903823416
$ 40,000                9 %                  25           $ 335.6785454527
$ 40,000                9 %                  30           $ 321.849046778

STOP
```

Fig. 15—2

CHAPTER 16
ALPHANUMERIC STRING VARIABLES

Thus far, literal strings have been used as headings or labels for PRINT, STOP, and INPUT statements. These literal strings don't change, and are always printed as represented. String variables are variables whose value are literal strings. The value of string variables can be altered at will.

SECTION 16–1
STRING VARIABLE – NAMES AND CHARACTERISTICS

String variables are distinguished from numeric variables in two ways. First, string variables have different names than numeric variables. A string variable is denoted by a letter or a letter and a digit, followed by a "$" (dollar sign). There are a total of 286 string variable names.

LEGAL STRING VARIABLE NAMES

A4$	**W3$**
X$	**Z6$**

Fig. 16–1

Second, unlike numeric variables, which can only represent numbers, string variables can represent any string of symbols, letters, or digits.

SIZE OF STRING VARIABLES

Until a string variable is assigned a value, it is assumed to consist of one space. This compares to numeric variables, which assume a value of 0, before they are assigned some other value. Unless specified otherwise in a DIM statement, the maximum number of alphanumeric characters a string variable can assume is 16. This compares to the maximum number of digits a numeric variable can assume which is 13. If an attempt is made to assign a literal string of greater length to a string variable, the additional characters are simply ignored.

SECTION 16–2
GIVING STRING VARIABLES VALUES

String variables, like numeric variables are assigned values by Assignment statements, READ/DATA statements, and INPUT statements. Except for INPUT statements, the characters and spaces must all be enclosed in quotes.

Some examples of Assignment statements and READ/DATA statement are shown in Figs. 16–2 and 16–2a.

```
ASSIGNMENT STATEMENT
V$  =  "JOE SMITH"
F$  =  "MAPLE ST."
X4$ =  "G542H-16#"
W$  =  "152,760"
```

Fig. 16–2

```
10 READ A$, B$, C$
 .
 .
80 DATA "OHIO", "MISSOURI", "INDIANA", "NEW YORK",...
```

Fig. 16–2a

The situation with INPUT statements is somewhat different. Alpha characters need not be included in quotation marks. If quotation marks are not used commas and carriage returns act as string terminators, and leading spaces are ignored. Thus, if commas or leading spaces are to be included in the literal string in the INPUT statement, the string must be included in quotes. Fig. 16–2b shows the results of responding to an INPUT request with quotes and Fig. 16–2c without quotation marks. Notice that in the case of no quotes, only the first two parts, as denoted by commas, are picked up. The rest is ignored.

CHAPTER 16
ALPHANUMERIC STRING VARIABLES

```
:10 INPUT Y$, Z$
:20 PRINT Y$ :PRINT Z$
:RUN
? "PARK, MARY J.", "JONES, STANLEY F." CR/LF
PARK, MARY J.
JONES, STANLEY F.

:_
```

Fig. 16–2b
With Quotation Marks

```
:10 INPUT Y$, Z$
:20 PRINT Y$: PRINT Z$
:RUN
? PARK, MARY J., JONES, STANLEY F. CR/LF
PARK
MARY J.

:_
```

Fig. 16–2c
Without Quotation Marks

SECTION 16–3
USING STRING VARIABLES

Once a string variable is given a value, it may be used with all the relational operators, shown in Fig. 12–5.

Fig. 16–3 gives some example of string variables with relational operators.

```
80 IF Z$ = "ABC" THEN 200     IF/THEN EQUALITY STATEMENT
90 W$ = A$                    ASSIGNMENT STATEMENT
130 IF B$ < A$ THEN 150       IF/THEN INEQUALITY STATEMENT
```

Fig. 16–3

However, string variables and strings *cannot* be used with the arithmetic operators (+, –, *, / , ↑), as shown in Fig. 16–3a

ILLEGAL USE OF STRINGS

PRINT C$, C$ ↑ 2	– Strings can't be raised to a power

W$ = "123"
V$ = "456"
	– Literal strings assigned are numeric which is O.K. However, since they are in quotes and are assigned to a string variable, they cannot be arithmetically manipulated.

Y$ = W$ + V$	– Strings can't be added regardless of what characters they represent.

Fig. 16–3a

ALPHANUMERIC ORDERING

A natural question arises at this point, regarding the last example of Fig. 16–3, an IF/THEN inequality statement.

Two string variables are compared by the *relative* alphanumeric characters composing them. The ordering is given in Fig. 16–3b.

Notice that the letters of the alphabet are ordered as expected — A is less than B, is less than C . . . Z. Also notice that the numerals 0 thru 9 are as expected, and numerous symbols fall throughout. When a comparison of strings is made, the strings are compared on a character-by-character basis.

ALPHANUMERIC ORDERING

"LOWEST"

INCREASING RANK ↓

(SPACE)	0	A	P
"	1	B	Q
#	2	C	R
$	3	D	S
%	4	E	T
'	5	F	U
(	6	G	V
)	7	H	W
*	8	I	X
+	9	J	Y
,	:	K	Z
-	;	L	↑
.	<	M	
/	=	N	
	>	O	"HIGHEST"

INCREASING RANK →

Fig. 16–3b

Short strings are filled out with trailing spaces to allow for comparisons with longer strings. These trailing spaces have no effect otherwise. Fig. 16–3c gives some examples of string comparisons.

EXAMPLES OF ALPHANUMERIC COMPARISONS

"JOHN SMITH" < "WILLIAM JONES" — comparison stops after first character: J<W

"SMITH, JOHN" > "JONES, WILLIAM" — comparison stops after first character: S>J

"ABC" < "CBA" — comparison stops after first character: A<C

"ABC" > "-ABC" because of leading space, string comparison stops after first character :"A">"–"

"1921 PARK DRIVE" < "A" because "1" < "A"

"1921 PARK DRIVE" > "-A" because "1" > "-" (space)

"X-" = "X" because "X" is considered to have trailing spaces, filled out for comparison.

"X" > "-X" because "-" (space) < "X"

Fig. 16—3c

Figures 16—3d & 16—3e are examples of a commonly used programming technique, where string variables aid in the creation of conversational programming. Statement 10 is an INPUT statement with literal string, used to indicate that a number is required. The INPUT is assigned to a numeric variable. Statement line 20 then asks whether the user desires the square or cube of the number previously entered. The INPUT is assigned to a string variable. Statement line 30 then checks to see if the request was for a squared number, which if true causes program flow to go to statement line 50, setting the power to 2. If the request of the second INPUT statement is not squared, program flow after statement line 30 continues to line 40, where the response is checked against CUBED. If this condition is met, program flow goes to statement line 60, where the power is set to 3. If the request of the second INPUT statement is neither SQUARED nor CUBED, Program flow comes to the second statement in statement line 40, which causes a printing of the statement BE MORE SPECIFIC, after which program flow is directed back to the second INPUT statement. This allows the program to handle virtually *any* response to the second INPUT statement. A recognizable response (here SQUARED or CUBED) causes the program flow to loop back and continue requesting until a recognized response *is* given.

EXAMPLE OF STRING VARIABLES IN A CONVERSATIONAL PROGRAM

```
10  INPUT "WHAT NUMBER DO YOU WANT TO WORK WITH", X
20  INPUT "DO YOU WANT IT SQUARED OR CUBED", N$
30  IF N$ = "SQUARED" THEN 50
40  IF N$ = "CUBED" THEN 60: PRINT "BE MORE SPECIFIC": GO TO 20
50  P = 2: GOTO 70
60  P = 3
70  PRINT X; N$; "="; X↑P: PRINT
80  GOTO 10
```

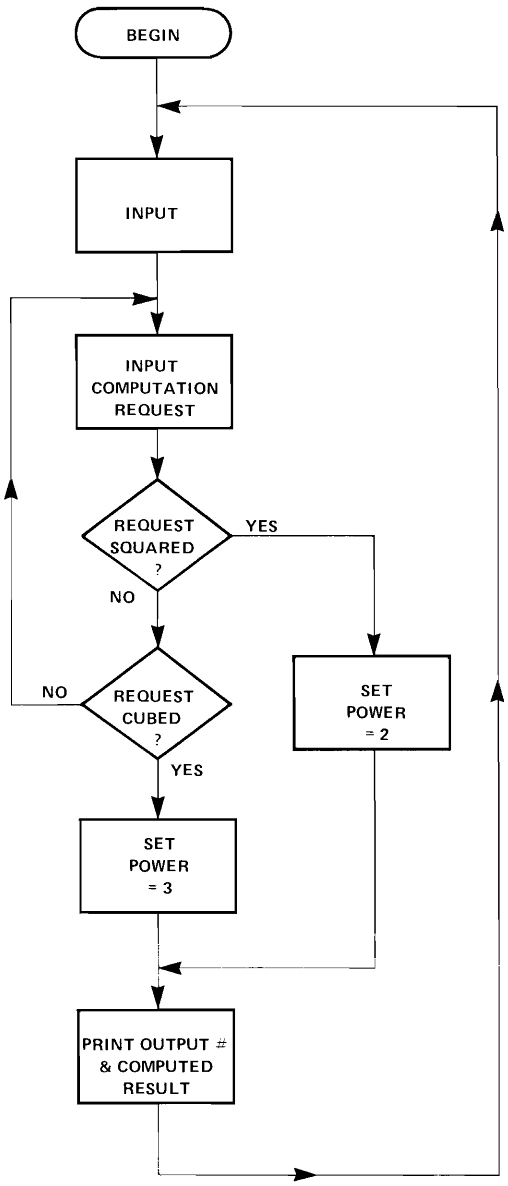


Fig. 16—3d

```
READY
:10 INPUT "WHAT NUMBER DO YOU WANT TO WORK WITH" , X
:20 INPUT "DO YOU WANT IT SQUARED OR CUBED", N$
:30 IF N$ = "SQUARED" THEN 50
:40 IF N$ = "CUBED" THEN 60 :PRINT "BE MORE SPECIFIC" : GOTO 20
:50 P = 2: GOTO 70
:60 P = 3
:70 PRINT X; N$; "="; X↑P: PRINT
:80 GOTO 10
:RUN
WHAT NUMBER DO YOU WANT TO WORK WITH? 5
DO YOU WANT IT SQUARED OR CUBED? SQUARED
5 SQUARED = 25

WHAT NUMBER DO YOU WANT TO WORK WITH?_
```

Fig. 16–3e

Notice also the form of the first PRINT statement in line 70. Both INPUT responses are used as well as the answer.

SECTION 16–4 THE SIZE OF STRING VARIABLES

In SECTION 16–1, it was mentioned that the standard size of a 2200 string variable is 16 characters. If a string variable is shorter than 16 characters, the remaining positions are considered to be trailing spaces. If more than 16 characters are assigned to a string variable, the excess characters beyond 16 are ignored.

2200 BASIC however, allows a user to change the size of a string variable from 16 characters to any number of characters from 1 to 64. The result is a *compacted* or *extended* string variable. This is accomplished, using the DIM statement. The maximum number of characters desired is given, following the string variable name *without* parentheses. In Fig. 16–4, A$ is set to a maximum length of 36 characters, B$ to a maximum of 64, and C$ to a maximum of 7.

```
DIM A$36, B$64, C$7
```

Fig. 16–4

This use of the DIM statement differs from its use as described in SECTION 14–2, in that the string variable still only represents a single "value". In this case however, the "value" in terms of absolute size is changed.

Fig. 16–4a shows the results of using a DIM statement to alter the maximum length of a string variable.

```
READY
:A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"

:PRINT A$
ABCDEFGHIJKLMNOP

:DIM B$5, C$20
:B$, C$="1234567890ABCDEFGHIJKLMN"

:PRINT B$, C$
12345          1234567890ABCDEFGHIJ

:_
```

Fig. 16–4a

Without the DIM statement, assigning a string of 26 characters to a string variable (here A$) results in its picking up only the first 16 characters. However, by changing the length (here B$ to 5, and C$ to 20) results in the string variable picking up its respective maximum number of characters.

STRING ARRAYS
Arrays of string variables can also be dimensioned just as numeric variables can. The maximum subscript is again 255. The maximum number of characters which each string array *element* can assume is 16 characters, unless the size of the string variable is set at another value. Elements of string arrays can be dimensioned from one to sixty-four characters in length. Fig. 16—4b gives both situations.

```
DIM A$(5,5), B$(25)
DIM X3$(5,5)50
```

Fig. 16—4b

In the first DIM statement, the string array A$ ( ) is defined as a two dimensioned array having 5 rows and 5 columns. The string array B$ ( ) is defined as a one-dimensioned array with 25 elements. The maximum length of the individual string elements in A$ ( ) and B$ ( ) is sixteen. The second DIM statement defines a two-dimensional string array of five rows and five columns, in which each element can assume up to fifty characters.

SECTION 16—5
THE STR(FUNCTION

Although string variables cannot be added, subtracted, etc., they can be manipulated. There are two functions which allow the user to analyze the length of string variables and to access the characters which compose them. One of these functions is the STR(function.

The STR(function enables the user to extract, examine and/or replace portions of string variables. The function is generated by keying S T R (

GENERAL FORM

STR{string variable name, expression [,expression]}

Where 1st expression = starting character in string
2nd expression = number of consecutive characters.
(The specification of 2nd expression is optional.)

From the general form of the STR(function,

STR(A$,3,4) means

Starting with the 3rd character of A$, take 4 characters, i.e., the 3rd, 4th, 5th, and 6th.

STR(A$,3) means

Starting with the 3rd character, take remainder of the string A$.

Examples using the STR(function

Assuming B$="ABCDEFGH"

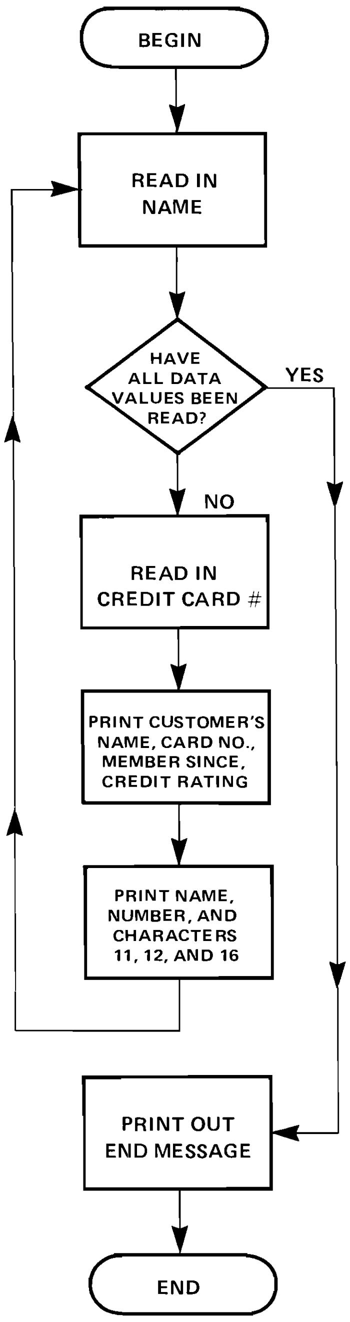
10 A$=STR(B$,2,4)	---A$ is set to "BCDE".
20 STR(A$,4)=B$	---Characters 4 through 16 of A$ are set to "ABCDEFGH".
30 STR(A$,3,3)=STR(B$,5,3)	---The 3rd, 4th, and 5th characters of A$ are set to "EFG".
40 IF STR(B$,3,2)="AB"THEN 100	---Characters "CD" of B$ are compared to the literal string "AB".
50 READ STR(A$,9,8)	---Characters 9 through 16 of A$ receive the next data value.

Fig. 16–5 is a programming example using the STR(function. As the REM statements indicate, the credit card number keyed, via the INPUT statement in line 50, is checked for membership year and credit rating. Both these pieces of information are "imbedded" in the credit card number.

Notice the last 2 DATA values. The ZZZZZ is a piece of test data, checked for by the IF/THEN statement in statement line 60. The variable S serves as a counter, to check the number of transactions processed.

EXAMPLE OF PROGRAM USING THE STR(FUNCTION

```
10 REM CREDIT CARD MEMBERSHIP AND CREDIT CHECK
20 REM DATA TAKEN IN VIA READ STATEMENT
30 S=0 :PRINT
40 INPUT "NAME", N$
50 IF N$="ZZZZZ" THEN 110 :S=S+1
60 INPUT "CREDIT CARD #", C$
70 PRINT "CUSTOMER'S NAME", "CARD NO."," __MEMBER SINCE", "CREDIT RATING"
80 REM CHAR 16 IS CREDIT RATING
90 PRINT N$, C$, "____19"; STR(C$, 9, 2), STR(C$, 16)
100 GOTO 40
110 PRINT "ALL CARDS PROCESSED", S; "TRANSACTIONS"
120 END
```


BEGIN
READ IN NAME
HAVE ALL DATA VALUES BEEN READ?
YES
NO
READ IN CREDIT CARD #
PRINT CUSTOMER'S NAME, CARD NO., MEMBER SINCE, CREDIT RATING
PRINT NAME, NUMBER, AND CHARACTERS 11, 12, AND 16
PRINT OUT END MESSAGE
END

TYPICAL CREDIT CARD

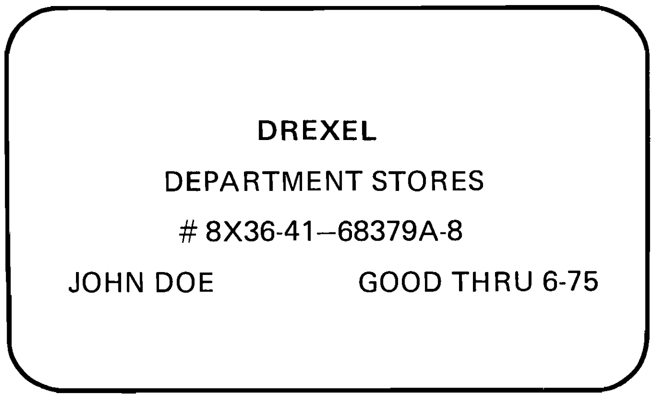


Fig. 16–5

```
:RUN

NAME? "DOE, JOHN"
CREDIT CARD #? 8X36-41-68379A-8
CUSTOMER'S NAME    CARD NO.              MEMBER SINCE    CREDIT RATING
DOE, JOHN          8X36-41-68379A-8         1968         8
NAME? "SMITH, W.C."
CREDIT CARD #? 4X52-61-72594C-5
CUSTOMER'S NAME    CARD NO.              MEMBER SINCE    CREDIT RATING
SMITH, W.C.        4X52-61-72594C-5         1972         5
NAME? "JONES, ROBERT"
CREDIT CARD #? 5X19-71-60127L-1
CUSTOMER'S NAME    CARD NO.              MEMBER SINCE    CREDIT RATING
JONES, ROBERT      5X19-71-60127L-1         1960         1
NAME? "ZZZZZ"
ALL CARDS PROCESSED                      3 TRANSACTIONS

END PROGRAM
FREE SPACE = 2987

:_
```

Fig. 16–6

The STR(function can be used anywhere a string variable is needed and it can be used with string arrays.

SECTION 16–6 THE LEN(FUNCTION

The LEN(function is used to determine the length of an alphanumeric string, excluding trailing blanks. This ability to determine the number of significant characters in a string becomes useful during a number of alphanumeric computations.

The LEN(function gives a numeric value and as such can be used whenever a numeric variable is legal.

The function is generated by keying L E N ( ) .

GENERAL FORM
LEN (string variable)

Some examples of the use of the LEN(function are as follows:

```
READY
:10 DIM B$ (5) 49
:20 B$ (1) = "LAST NAME, FIRST INITIAL, CITY and STATE, ZIP CODE"
:30 PRINT LEN (B$(1))
:RUN


47

:_
```

CHAPTER 16
ALPHANUMERIC STRING VARIABLES

```
READY
:10  LET A$ = "ABCDEF"
:20  PRINT LEN (A$)
:RUN
 6

:_
```

```
READY
:10 INPUT J$
:20 A = 5*SQR (LEN(J$)+2)
:30 PRINT "A = "; A
:RUN
? ABCDEFG
A = 15

:_
```

```
READY
:10 INPUT "WHAT IS YOUR NAME", N$
:20 IF LEN (N$) > 4 THEN 50
:30 PRINT N$; "  IS A VERY SHORT NAME."
:40 GO TO 10
:50 PRINT N$; "  IS A LONG NAME."
:60 GO TO 10
:RUN

? WHAT IS YOUR NAME? MICHAEL
MICHAEL   IS A LONG NAME
WHAT IS YOUR NAME?
```

CHAPTER 17
SUBROUTINES

A subroutine is a program within a program or a group of statements which are to be used over and over again. Rather than writing these statements into the program each time they are used, they can be written once and referred to each time they are needed.

SECTION 17–1 "CALLING" & WRITING SUBROUTINES

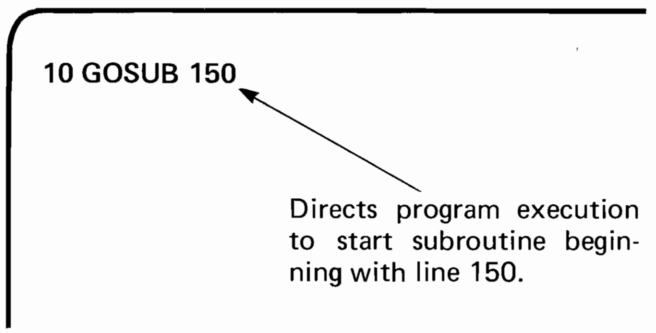
The General Form of a Subroutine is as follows:

GENERAL FORM

GOSUB line number

where line number is the number of the line beginning the subroutine.

Subroutines are accessed using a GOSUB statement (Fig. 17–1).

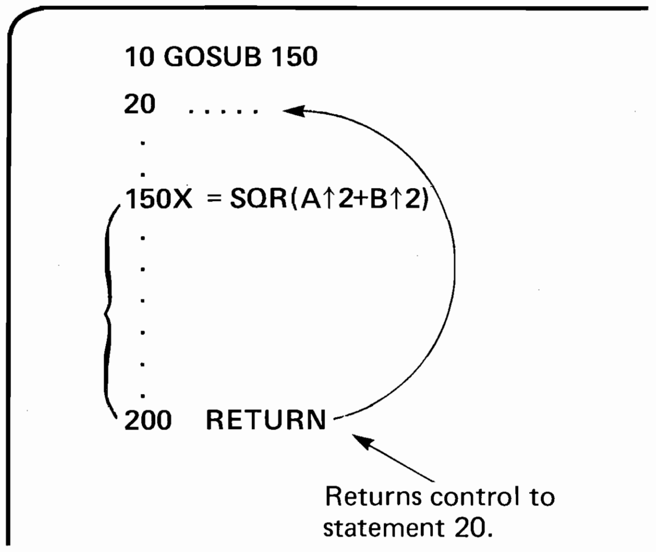


```
10 GOSUB 150
```

Directs program execution to start subroutine beginning with line 150.

Fig. 17–1

The GOSUB statement tells the 2200 to transfer execution of the program to another line (here line #150) which is the first line of the subroutine. All subroutines must end with a RETURN statement (Fig. 17–1a).



```
10 GOSUB 150
20  .....
.
.
150X = SQR(A↑2+B↑2)
.
.
.
.
.
.
200  RETURN
```

Returns control to statement 20.

Fig. 17–1a

The RETURN statement tells the 2200 to go back to the statement immediately following the GOSUB statement which sent the 2200 to that subroutine.

The RETURN statement must be the last executable statement on a line if it is in a multi-statement line. Non-executable statements (e.g. REM, DATA) can be included on the same line after a RETURN (Fig. 17–1b).

EXAMPLES OF LEGAL AND ILLEGAL USE OF RETURN IN MULTISTATEMENT LINES

LEGAL

```
50   RETURN: DATA 5.6, 18, 100
100  X=10: RETURN:REM SUBROUTINE ENDS
```

ILLEGAL

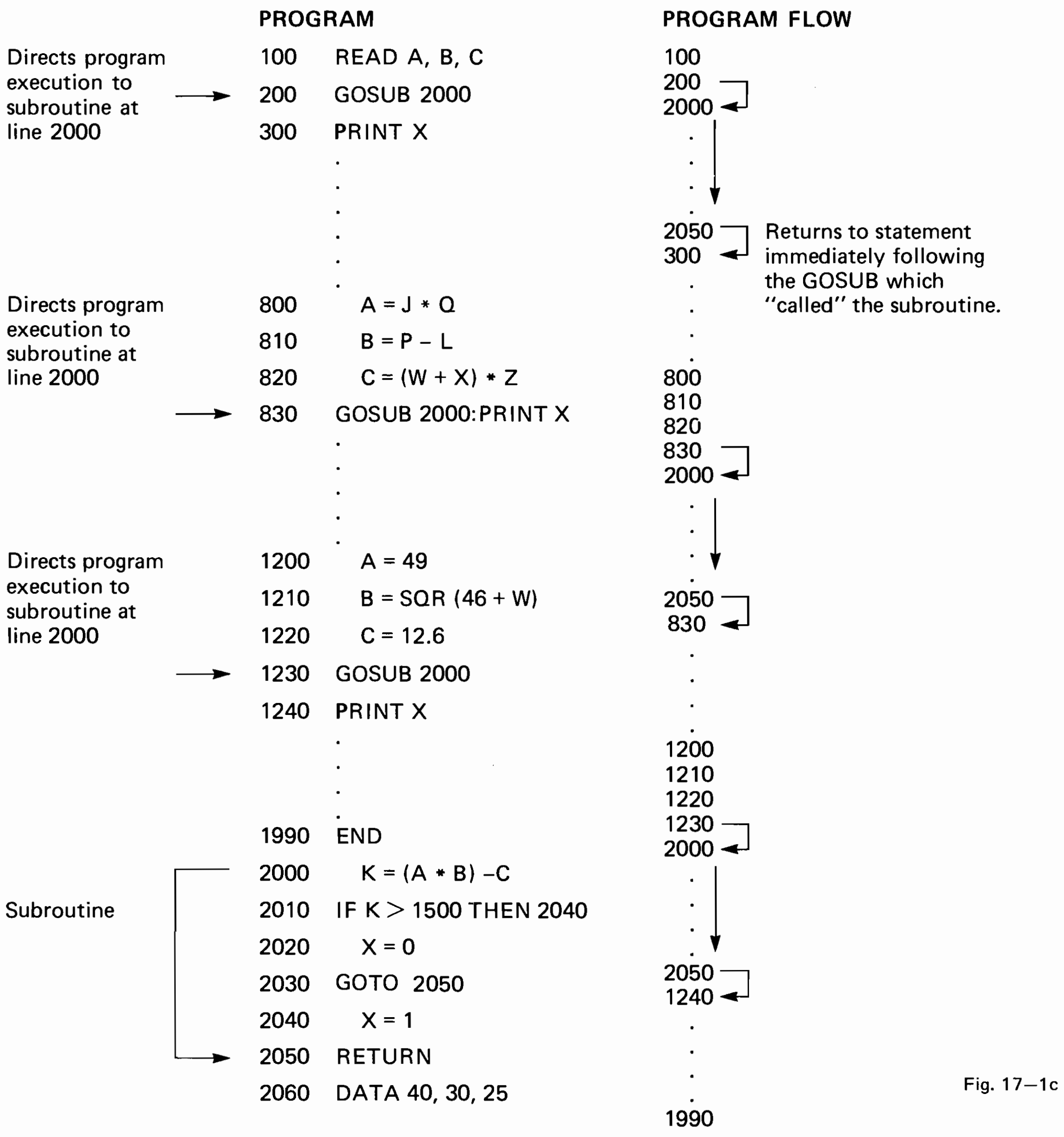
```
150  RETURN: C=SQR(10)
170  RETURN: PRINT X:
```

Fig. 17–1b

Consider the example in Fig. 17–1c which illustrates a program which uses the same subroutine more than once. Notice the flow of the program.

CHAPTER 17
SUBROUTINES

SUBROUTINES


PROGRAM
PROGRAM FLOW
Directs program execution to subroutine at line 2000
Directs program execution to subroutine at line 2000
Directs program execution to subroutine at line 2000
Subroutine
Returns to statement immediately following the GOSUB which "called" the subroutine.

```
PROGRAM

100   READ A, B, C
200   GOSUB 2000
300   PRINT X
      .
      .
      .
      .
      .
      .
800     A = J * Q
810     B = P - L
820     C = (W + X) * Z
830   GOSUB 2000:PRINT X
      .
      .
      .
      .
1200    A = 49
1210    B = SQR (46 + W)
1220    C = 12.6
1230  GOSUB 2000
1240  PRINT X
      .
      .
      .
      .
1990  END
2000    K = (A * B) -C
2010  IF K > 1500 THEN 2040
2020    X = 0
2030  GOSUB  2050
2040    X = 1
2050  RETURN
2060  DATA 40, 30, 25
```

```
PROGRAM FLOW

100
200
2000
.
.
.
.
2050
300
.
.
.
.
800
810
820
830
2000
.
.
.
.
2050
830
.
.
.
.
1200
1210
1220
1230
2000
.
.
.
.
2050
1240
.
.
.
1990
```

Fig. 17–1c

Comment: The 2200 keeps track of the statement following each GOSUB statement so when a RETURN is executed the 2200 returns to the statement immediately following the last executed GOSUB statement.

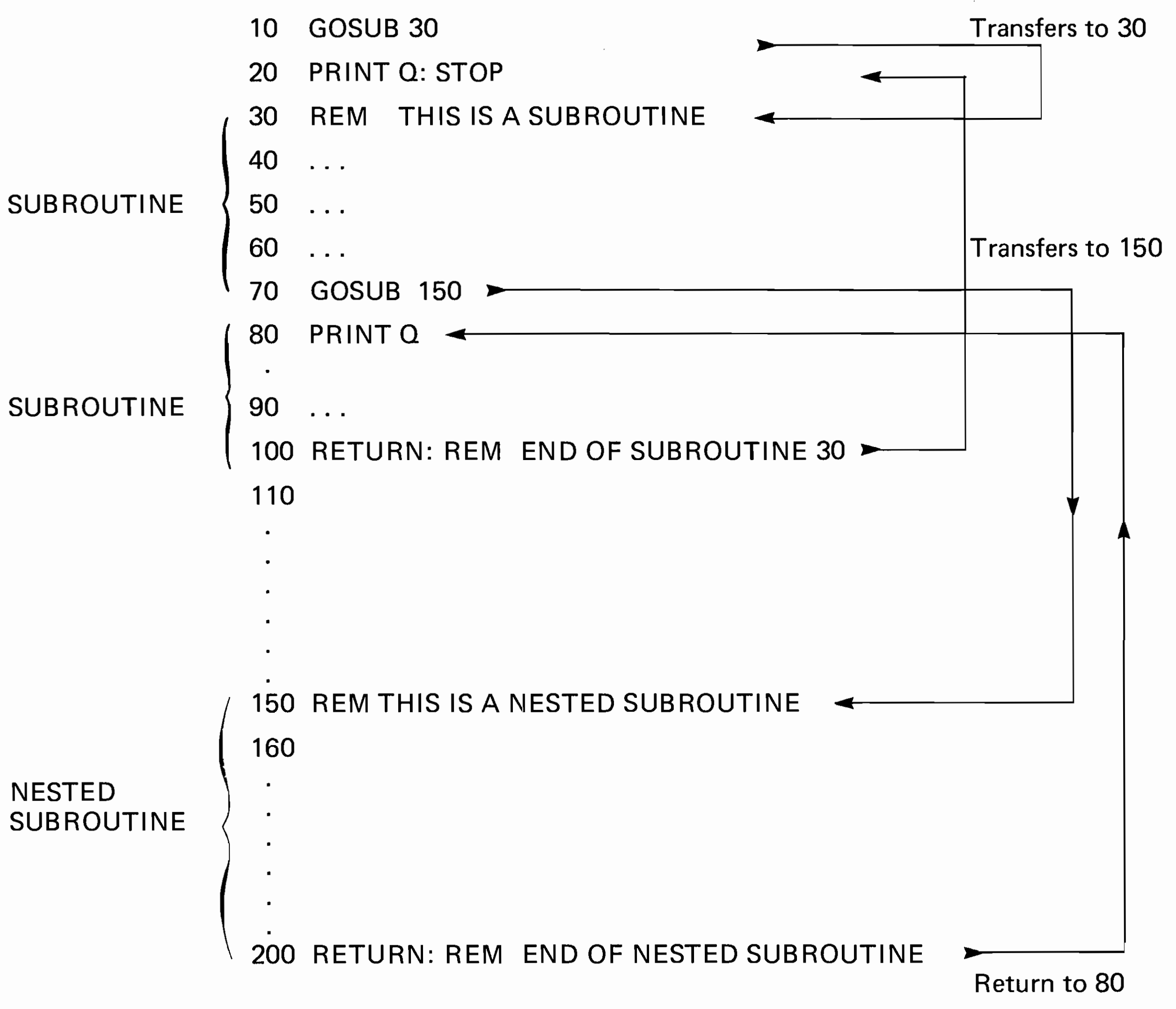
What is happening in this program? The GOSUB statement at line 200 causes the 2200 to jump to statement number 2000. It then executes statements 2000 to 2050, where it encounters a RETURN, and executes the RETURN and returns to the statement following the last executed GOSUB (i.e. 300 PRINT X).

Statements from 300 – 830 are executed and the 2200 branches back again to the subroutine at line 2000. This time line 2050 returns execution to the statement PRINT X in line 830.

SECTION 17–2
NESTED SUBROUTINES

Subroutines can call other subroutines. That is, from within one subroutine you can go to another subroutine and return back to the original subroutine. This is called Nesting Subroutines. Unlimited nesting of subroutines is allowed on the 2200 (Fig. 17–2).

EXAMPLE OF NESTED SUBROUTINES SHOWING PROGRAM FLOW

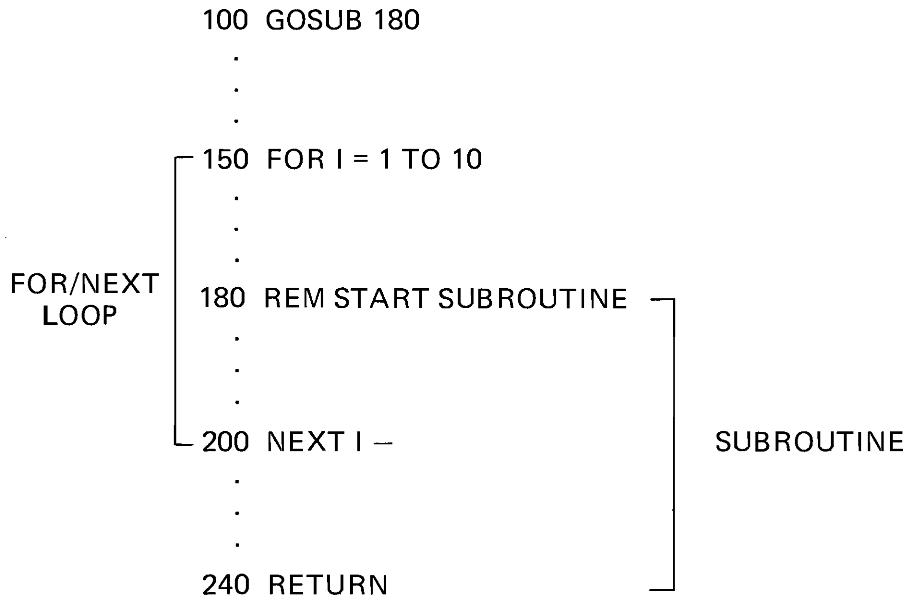


```
10   GOSUB 30
20   PRINT Q: STOP
30   REM   THIS IS A SUBROUTINE
40   . . .
50   . . .
60   . . .
70   GOSUB  150
80   PRINT Q
 .
90   . . .
100  RETURN: REM  END OF SUBROUTINE 30
110
 .
 .
 .
 .
 .
 .
150  REM THIS IS A NESTED SUBROUTINE
160
 .
 .
 .
 .
 .
200  RETURN: REM  END OF NESTED SUBROUTINE
```

Fig. 17–2

SECTION 17–3
ILLEGAL USE OF SUBROUTINES

Subroutines and FOR/NEXT loops should never overlap – that is a subroutine should never contain only one of a matching set of FOR and NEXT statements. For example



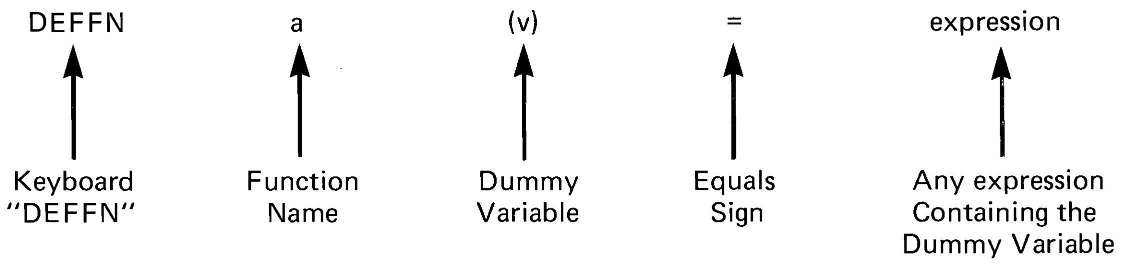
```
100  GOSUB 180
 .
 .
 .
150  FOR I = 1 TO 10
 .
 .
 .
180  REM START SUBROUTINE
 .
 .
 .
200  NEXT I –
 .
 .
 .
240  RETURN
```

Fig. 17–3

CHAPTER 18
SINGLE LINE USER DEFINED FUNCTIONS

The 2200 allows the user to create his own special functions by using the DEFFN statement. The DEFFN statement allows the programmer to define mathematical functions of one variable which can be used like any keyboard function. The General Form is shown in Fig. 18–1.

GENERAL FORM OF DEFFN STATEMENT



where a is any letter or digit
v is a scalar numeric variable

Fig. 18–1

Some examples of user functions are shown in Fig. 18–1a.

EXAMPLES OF USER DEFINED FUNCTIONS

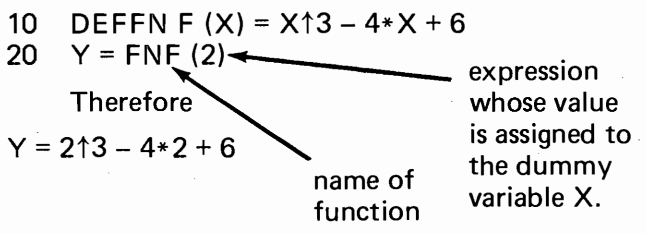
```
10  DEFFN  F(X)  = SQR (X + 9) - X
20  DEFFN  E(G)  = (4*G + 6)/G
30  DEFFN  8(L)  = 42.7↑L
40  DEFFN  2(Y)  = Y*TAN (Y/2)
50  DEFFN  3(A)  =A↑3.6 - A↑2
```

Fig. 18–1a

SECTION 18–1
EXPLANATION OF DEFFN STATEMENT FORMAT AND USES OF DEFFN FUNCTION

1. The function "name" (in Fig. 18–1,) can be any number (0–9) or letter (A–Z) - a total of 36 functions.
2. The "dummy variable" (in Fig. 18–1) can be any numeric scalar variable. It is solely a place holder, and has no effect on a variable of its same name used somewhere else in a program.
3. The function definition (DEFFN statement) can appear anywhere in a program.
4. A function can be used in a program just like any keyboard function. The user function is referenced by using the FNa (expression) format where a is the name of the function and expression is any numeric expression.

CHAPTER 18
SINGLE LINE USER DEFINED FUNCTIONS



```
10   DEFFN F (X) = X↑3 - 4*X + 6
20   Y = FNF (2)
```

Fig. 18–1b

5. The function FNF(X) can be referenced as many times as needed in a program just like any keyboard function. A function cannot refer to itself, but it can refer to other functions (Fig. 18–1c).

```
:10 DEFFN1 (X) = 4*X↑2 + SQR (X + 1)
:20 DEFFN2 (Y) = FN1(Y) + 10
:30 A = 2*FN2(3)
:40 PRINT FN1(3), FN2(3), A
:RUN
 38            48            96

:_
```

Fig. 18–1c

But functions cannot refer to each other (Fig. 18–1d).

```
10   DEFFN A(B)   = 5 + 2 * FN X(B)
20   DEFFN X(Y)   =FNA(Y) - 4
```

Function A refers to function X and function X refers to function X.

Fig. 18–1d

Fig. 18–1e is an example of a program which makes use of a DEFFN function in a FOR/NEXT Loop.

EXAMPLE PROGRAM:

FUNCTION WITH VARIABLE VALUE

```
10   REM LOOP PROGRAM FOR THE AREA OF CIRCLES
20   DEFFN C(R) = π*R↑2
30   PRINT "RADIUS", "AREA"
40   FOR I = 1 TO 10
50   PRINT I, FNC(I)
60   NEXT I
70   END
```

```
READY

:10 REM LOOP PROGRAM FOR THE AREA OF CIRCLES
:20 DEFFN C(R) = #PI*R↑2
:30 PRINT "RADIUS", "AREA"
:40 FOR I = 1 TO 10
:50 PRINT I, FN C(I)
:60 NEXT I
:70 END
:RUN
RADIUS          AREA
 1              3.14159265359
 2              12.56637061436
 3              28.27433388231
 4              50.26548245744
 5              78.53981633975
 6              113.0973355292
 7              153.9380400259
 8              201.0619298298
 9              254.4690049408
 10             314.159265359

END PROGRAM
FREE SPACE = 3246

:_
```

Fig. 18–1e

CHAPTER 19
THE SPECIAL FUNCTION KEYS

At the top of your keyboard there is a group of sixteen Special Function Keys. This unique feature of the 2200 allows the user to customize the 2200 in the following ways: (1) to write and store in memory commonly used character strings for text entry, and access these strings with a single keystroke (i.e. touching a special function key), (2) to write and store special subroutines which can be accessed directly from the keyboard (special function key) or from a program, and (3) to provide argument passing capability. Each of these uses is clearly illustrated in this Chapter. The DEFFN' verb in BASIC allows the user to perform the above mentioned techniques.

SECTION 19–1
GENERAL FORM DEFFN' VERB

The General Form of the DEFFN' verb is as follows:

$$\text{DEFFN' integer} \left\{ \begin{array}{l} \text{"character string"} \\ [(\text{variable [, variable...]})] \end{array} \right\}$$

where:

integer is required and must be 0 ≤ integer < 256. The integer represents the number of the special function key. There are 256 special functions available on the 2200, of which 32 are directly accessible via the keyboard (0 – 31).

"Character string" is optional and can be any string of characters within quotes.

Variable is optional and can be any legal variable name.

The following examples illustrate the uses of the DEFFN' statement mentioned in the Introduction.

EXAMPLE 1 – As used with a character string

```
500 DEFFN' 1 "REWIND"
```

Explanation: Each time the 1 special function key is touched the term REWIND is displayed on the CRT.

EXAMPLE 2 – As used with special subroutines.

```
10 DEFFN' 1
20 PRINT "THIS IS A USER DEFINITION FUNCTION"
30 PRINT "5 SQUARED ="; 5↑2
40 RETURN
```

Explanation: When the 1 special function key is touched the subroutine above is called upon.

EXAMPLE 3 – As used with passing arguments.

```
500 DEFFN' 2 (A, E(3), C$)
```

Explanation: The variable names are specified in the DEFFN' statement which later can be passed.

These examples are discussed in more detail as you progress through the chapter.

SECTION 19–2
DEFFN' WITH COMMONLY USED CHARACTER STRINGS

Often certain strings are used repeatedly in a program. If this string is special to your own applications, it would be nice to be able to have it on a key on the keyboard. With the DEFFN' statement and special Function keys, you in a sense can design your own additional keyboard.

As an example, assume a program is written where the words "Credit Card", "Interest", and "Payments" are used repeatedly. Instead of keying in these characters over and over again you can write them once, store them in memory identified with a DEFFN' integer statement and access them each time by simply touching one of the special function keys.

There are 32 special function keys on the 2200 (0 – 31), 0 – 15 lower case and 16 – 31 uppercase. Therefore you can add 32 special routines and access them through the 32 special function keys on your keyboard.

CHAPTER 19
THE SPECIAL FUNCTION KEYS

Using the example just mentioned assign the specified character strings to special function keys 0, 1 and 2 respectively, as follows:

KEY IN

```
READY

:10 DEFFN' 0 "CREDIT CARD"

:20 DEFFN' 1 "INTEREST"

:30 DEFFN' 2 "PAYMENTS"
```

Each line is now in the memory. Clear the display by touching RESET. Now touch special function key 0, and the SPACE key twice, then special function key 2 and SPACE key twice. The results are shown in Fig. 19–2.

```
READY
:CREDIT CARD    INTEREST    PAYMENTS
```

Fig. 19–2

Each of the character strings is entered into the text each time the appropriate special function key is touched. Up to 32 character strings can be assigned this way.

If you are entering a program where these character strings are used, then you would simply have to touch the appropriate function key to enter the character string, just as if you were entering the words PRINT or DATA or THEN by touching a key on the keyboard.

SECTION 19–3
DEFFN' USED WITH SPECIAL SUBROUTINES

A Subroutine consists of more than one line of programming. Therefore, we shall refer to these special subroutines as "Marked Subroutines". It is required when writing a Marked Subroutine that the first line of the subroutine always be the DEFFN' integer statement and the last line always be a RETURN, as with any subroutine. There is no limit as to the number of lines within the subroutine. Fig. 19–3 is an example of a Marked Subroutine.

```
READY
:10   DEFFN' 1
:20   PRINT "THIS IS A USER DEFINED FUNCTION"
:30   PRINT "5 SQUARED="; 5↑2
:40   RETURN

:_
```

Fig. 19–3

Notice line #10, refers to special function key 1 and that the last line (#40) is a RETURN statement.

Enter this program into memory.

Clear the display by touching RESET. "Call" the special subroutine directly from the keyboard by touching special function key 1. The CRT is shown in Fig. 19–3a.

```
READY
THIS IS A USER DEFINED FUNCTION
5 SQUARED = 25

:_
```

Fig. 19–3a

The purpose of this feature is to allow you to define and write special functions once which can be used over and over again, then accessed by a single keystroke as if they were hardwired into the calculator. This allows you enormous flexibility in customizing your calculator.

As mentioned earlier there are 256 special functions in your calculator. Only the first 32 are directly accessible via the keyboard special function keys. The remaining special functions (also the first 32) can be called under program control. Fig. 19–3b is another example of a Marked Subroutine illustrating the use of a special function under program control.

Marked Subroutine in Memory:

```
40  DEFFN' 100
50  PRINT X, X↑2, X↑3
60  RETURN
```

Fig. 19—3b

This function can not be "called" directly from a keyboard, it must be "called" under program control. In order to "call" a Marked Subroutine in a program the GOSUB' xxx command is used, where xxx is the same integer as in the DEFFN' integer statement.

Fig. 19—3c shows a program which "calls" Marked Subroutine 100.

```
10  X = 10
20  GOSUB' 100
30  END
```

Fig. 19—3c

Notice that the "100" refers to the DEFFN' number not a line number as with a regular GOSUB statement.

The entire program and its results are shown in Fig. 19—3d.

```
READY
10  X = 10
20  GOSUB' 100
30  END
40  DEFFN' 100
50  PRINT X, X↑2, X↑3
60  RETURN
: RUN
 10           100            1000

END PROGRAM
FREE SPACE = 3323

:_
```

Fig. 19—3d

EXAMPLE

An example of a DEFFN' function called from the keyboard is shown in Fig. 19—3e.

```
10  DEFFN'  5: REM  GENERAL QUADRATIC EQUATION SOLUTION
20  INPUT "COEFFICIENTS = A, B, C", A, B, C
30  D = B↑2 - 4*A*C
40  IF D < 0 THEN 80
50  IF D = 0 THEN 70
60  PRINT "X1 ="; (-B + SQR (D) )/(2*A), "X2="; (-B-SQR (D) )/2*A:RETURN
70  PRINT "X1=X2="; -B/(2*A) : RETURN
80  PRINT "X1 and X2 IMAGINARY" : RETURN
```

Fig. 19—3e

To call this program simply touch 5 special function key, or to call this in a program you would use a GOSUB' 5.

EXAMPLE 2

Assume you are writing a program which uses the command "HEX(" in many places throughout the program. Instead of having to key this in every time, write a user defined function for it, call it to the display either from the keyboard or under programming control. This way "HEX(" is entered into current text line each time it is needed. (Fig. 19—3f).

```
100 DEFFN' 1 "HEX("
200 RETURN
```

KEY 1 Special Function Key

```
READY
:HEX(
```

Fig. 19—3f

SECTION 19–4
ARGUMENT PASSING CAPABILITY

The 2200 allows the user to "pass" (i.e. assign) values to variables in a subroutine prior to the subroutines execution. Look at the program in Fig. 19–4 which is a Special subroutine using the DEFFN' integer $\left\{ \text{[(variable , [variable...])]} \right\}$ format.

```
10   DEFFN' 12 (A, B, C)
20   D = B↑2 -4*A*C
30   IF D<0 THEN 70
40   IF D=0 THEN 60
50   PRINT "X1="; (-B+SQR(D))/(2*A); "X2="; (-B-SQR(D))/(2*A) :RETURN
60   PRINT "X1=X2="; -B/(2*A) :RETURN
70   PRINT "X1 and X2 IMAGINARY"
```

Fig. 19–4

Notice the format of the DEFFN' statement contains the variable names to be used within the subroutine and that nowhere in this program are the variables defined. The value each of the variables is to assume in the subroutine is specified, before the function is "called" in the GOSUB' statement. The variables specified in the DEFFN' statement are not dummy variables.

An advantage in using this type of subroutine over a regular subroutine is to save time and programming space. The variables are assigned in the GOSUB' statement whereas in a regular subroutine all variables are assigned prior to calling the subroutine and this could require several lines of text.

The number of the variables entered when calling the function must be the same as the number of variables specified in the function, and must be separated by commas. The value given may be a variable, or any legal algebraic expression. The order in which the values are assigned is the order in which they are presented in the DEFFN' statement. Fig. 19–4a shows a BASIC program in which the variables are defined in the GOSUB' statement.

```
120  GO SUB' 02 (-8,4, 19+SQR(X), COS (Y/A))
 .
 .
 .
 .
 .
 .
 .
290  DEFFN' 02 (A, B, C, D)
300  IF A = D -B THEN 330
 .
 .
 .
350  RETURN
```

Fig. 19–4a

—Variable values are assigned in the same statement as the subroutine "Call", potentially saving several lines.

As said earlier, a major advantage of being able to "pass" arguments in subroutines is the saving of program space. Fig. 19–4b is compared to Fig. 19–4c for assigning values to variables in the same statement as the subroutines "call" and for regular subroutines where the variables must be assigned values prior to the subroutine "call". (Fig. 19–4c.) Notice the difference in program steps needed.

"PASSING" VALUES

```
120 GOSUB' 02 (-8, 4, 19 E SQR(X), COS (Y/A) )
 .
 .
 .
 .
 .
 .
290 DEFFN' 02 (A, B, C, D)
300 IF A = D - B THEN 330
 .
 .
 .
350 RETURN
```

Fig. 19–4b

REGULAR SUBROUTINE

```
READY
:120 W   = -8
:130 S   = 4
:140 Y   = 19 + SQR(X)
:150 Z   = COS (Y/A)
:160 GO SUB 300
   .
   .
   .
:300 IF W = Z - X THEN 330
   .
:350 RETURN
```

Fig. 19–4c

Passing arguments is important for saving space and also so that values can be passed to the subroutine from the keyboard. For special function key entry to a subroutine, arguments are passed by keying them in separated by commas immediately before the special function key is depressed. For example,

```
READY
:10  DEFFN' 0
:20  Y=X*SIN(X) + COS(X)*Y
:30  PRINT Y
:40  RETURN
:12.1 [press Key 0]  Entered from
                     the Keyboard
```

Fig. 19–4d

A maximum of 5 arguments can be passed in this way. Fig. 19–5 shows a flow chart, program and printout as an example of passing values to variables.

SECTION 19–5 EXAMPLE PROGRAM "CALLING" MARKED SUBROUTINE

Program to calculate the number of ways to take "N" objects, "A" at a time.

Known as # of combinations = $C^N_A = \frac{N!}{(N-A)!\,A!}$

```
:10  INPUT "N, A", N, A
:20  GO SUB' 00(N):C=F
:30  GO SUB' 00 (N-A):C=C/F
:40  GO SUB' 00(A):C=C/F
:50  PRINT "THERE ARE";C;"COMBOS."
:60  END
:70  DEFFN' 00(X)
:80  F = 1
:90  FOR J = 1 TO X
:100    F = F*J: NEXTJ
:110    RETURN
:RUN
N, A? 10,4
THERE ARE 210 COMBOS

END PROGRAM
FREE SPACE = 3154

:_
```

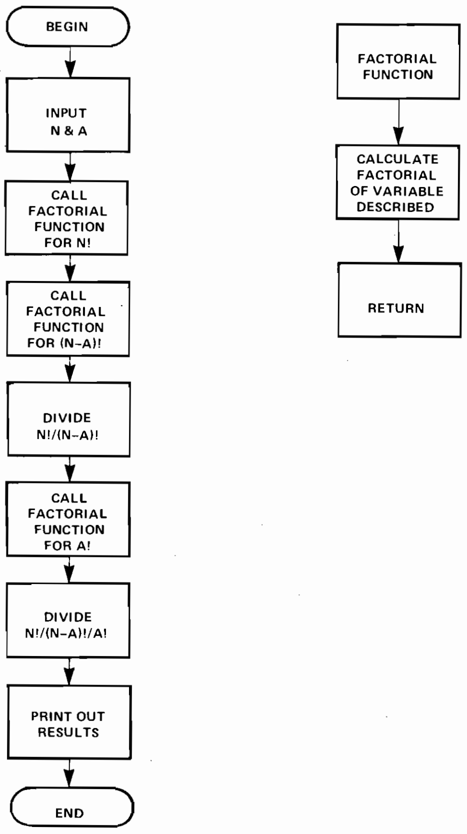

BEGIN
INPUT N & A
CALL FACTORIAL FUNCTION FOR N!
CALL FACTORIAL FUNCTION FOR (N-A)!
DIVIDE N!/(N-A)!
CALL FACTORIAL FUNCTION FOR A!
DIVIDE N!/(N-A)!/A!
PRINT OUT RESULTS
END
FACTORIAL FUNCTION
CALCULATE FACTORIAL OF VARIABLE DESCRIBED
RETURN

Fig. 19–5

CHAPTER 20 PRINTUSING AND % – IMAGE STATEMENT – CONTROLLED FORMATTING OF OUTPUT

Previous discussions of formatting the output involved using the PRINT statement with commas, semicolons and/or the TAB(command. By using a different type of statement, the PRINTUSING statement, output can be edited to fit into a precise image.

SECTION 20–1 GENERAL FORM OF THE PRINTUSING STATEMENT

PRINTUSING operates in conjunction with a referenced IMAGE statement. Print elements in the PRINT USING statement are edited into the print line as directed by the IMAGE statement. Therefore formatting the output according to a pre-determined image requires two statements. The General Form of the PRINTUSING statement is shown in Fig. 20–1.

```
PRINTUSING 'line number' [, {'print element't }...] [;]
where  'line number'  = line number of the corresponding image statement
                        expression
       'print element' = alphanumeric variable
                         literal string in quotes
                     t = comma or semicolon
```

Fig. 20–1

The IMAGE statement is a picture of the exact format to be used for the output line. A % symbol is used to designate the statement as an Image statement. Fig. 20–1a is an example of the two statements.

```
10 PRINTUSING 20, 123.45, 123.45, 123.45, 123.45
```

Line #of IMAGE statement

Data to be outputted or print elements

```
20 % ### ###.# ###.## ###.###
```

Format (Image) of the line

% symbol specifying this as IMAGE statement

Fig. 20–1a

The "#" characters (generated by SHIFT #) are used to represent digits. Decimal points in the IMAGE statement represent where the decimal is to go. Spacing within an Image statement is followed exactly.

Data formats in the Image statement are read sequentially. Extra decimal digits, not given a place in the IMAGE statement are truncated. Fig. 20–1b shows the output from the example in Fig. 20–1a.

```
:RUN
123    123.4    123.45    123.450
```

Fig. 20–1b

SECTION 20–2 OVER AND UNDER FORMATTING IN THE IMAGE STATEMENT

When an Image "over formats" (i.e. more # signs to to the left of the decimal point than necessary) the unneeded left most "#'s" are ignored and leading blanks inserted (Fig. 20–2).

```
:10PRINTUSING 30, 627.6, 958.2, 4567.0
:30% ####.#  #####.#  ####.#
:RUN
627.6           958.2    4567.0
```

overformatted

Fig. 20–2

When an image "under formats" (i.e. fewer # signs to the left of the decimal point than necessary) the #'s are printed in place of the numbers.

```
READY
:10 PRINTUSING 30, 627.6, 958.2
:30% ##.#
:RUN
##.#
##.#
```

underformatted and # signs printed instead of numbers.

Fig. 20–2a

When it is not known exactly how large a number may be, it is best to overformat the Image.

CHAPTER 20 PRINTUSING AND % – IMAGE STATEMENT – CONTROLLED FORMATTING OF OUTPUT

SECTION 20–3 USE OF LITERALS IN AN IMAGE STATEMENT

Literals in an IMAGE statement do not require any quotation marks. Literals are used to label output as in Fig. 20–3 or just to display alphanumerics as in Fig. 20–3a.

```
READY

:50 PRINTUSING 60, .3684, 2.057
:60% AVE.=#.###  ERA = ##.##
:RUN
 AVE. = 0.368   ERA = 2.05

:_
```

Fig. 20–3

```
100 PRINTUSING 110
110 % PROFIT AND LOSS STATEMENT

: RUN
PROFIT AND LOSS STATEMENT

:_
```

Fig. 20–3a

SECTION 20–4 SCIENTIFIC NOTATION IN IMAGE STATEMENT

4 ↑ symbols (↑↑↑↑) are used in the IMAGE statement to represent the Image of the exponent field. The exponent value in the output is adjusted to align the decimal point in the value with the decimal point in the Image. The four ↑'s are assigned as follows:

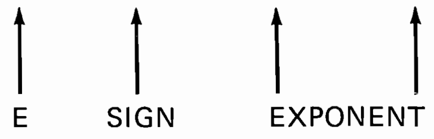


Fig. 20–4 shows the use of modified Scientific Notation in the IMAGE statement. Notice the adjustment of the exponent.

```
READY

:100 PRINTUSING 150, 5.376E8, 2.13E-5, 2.6E-9
:150% #.###↑↑↑↑ .###↑↑↑↑ ##↑↑↑↑
:RUN
5.376E+08   .213E-04  26E-10

:_
```

Fig. 20–4

SECTION 20–5 COMMAS IN IMAGE STATEMENT

Commas can be used in images to improve the readibility of formatted numbers (Fig. 20–5).

```
READY

:100 PRINTUSING 150, 1362594, 3726.59
:150% #,###,###.##      #,###.##
:RUN
1,362,594.00      3,726.59

:_
```

Fig. 20–5

SECTION 20–6 REUSING AN IMAGE STATEMENT

When an IMAGE statement contains only one Image, this Image is reused for each piece of data in the PRINTUSING statement (Fig. 20–6).

(OUTPUT)

```
READY
:100 PRINTUSING 200, 1, 2, 3
:200%   #.#
:RUN
 1.0
 2.0
 3.0

:_
```

Fig. 20–6

The comma used to separate the elements in the PRINTUSING statement, causes the output to appear on a new line each time the image statement is re-used.

If semi-colons are used to separate elements in the PRINTUSING statement, the output continues on the same line, with a packed format resulting. (Fig. 20–6a)

(OUTPUT)

```
READY

:300 PRINTUSING 400, 4; 5; 6
:400%   #.#
:RUN
 4.0 5.0 6.0

:_
```

Fig. 20–6a

SECTION 20–7 USING "+" OR "–" IN AN IMAGE STATEMENT

If the IMAGE statement starts with a "+" sign, the correct (plus or minus) sign is always printed preceding the first significant digit in the Image. See Fig. 20–7.

(OUTPUT)

```
READY

:10 PRINTUSING 20, 15.62, -158.936, -4.1
:20% + ###.##
:RUN
 +15.62
 -158.93
 +352.00
 -4.10

:_
```

Fig. 20–7

If the IMAGE statement begins with a "–" sign, the minus sign for the negative expression is edited into the print line immediately preceding the first significant digit. No sign is included if the number is positive (Fig. 20–7a).

(OUTPUT)

```
READY

:30 PRINTUSING 40, 15.62, -158.936, 352, -4.1

:40%  -###.##

:RUN

   15.62
 -158.93
  352.00
    -4.10

:_
```

Fig. 20–7a

If no sign is included in the Image no problems result if all the numbers are positive. But if negative numbers are to be outputted the minus sign is put into the printout and the entire number is shifted one space to the right. (Fig. 20–7b).

(OUTPUT)

```
READY
:10 PRINTUSING 20, 57.25, -57.25, 326.1, -326.1, -859
:20% ###.##
:RUN
 5 2 . 5
-  5 7 . 2 5
3 2 6 . 1 0
- 3 2 6 . 1 0
- 8 5 9 . 0 0
```

Fig. 20–7b

It is for this reason that a recommendation is made to include signs in the IMAGE statement especially when negative numbers are outputted.

SECTION 20–8 USING $ IN AN IMAGE STATEMENT

When a $ (dollar sign) is used to start an image, a dollar sign is printed in the output of the specified data either

(1) immediately preceding the first significant digit if the number is positive, or
(2) if the number is negative, immediately preceding the minus (–) sign which is just to the left of the first significant digit (Fig. 20–8).

(OUTPUT)

```
READY
:140 PRINTUSING 150, 98.42, 764.27, 2,523, -5.75, -300
:150% $####.##
:RUN
  $98.42
 $764.27
   $2.52
 $- 5.75
$-300.00
```

Fig. 20–8

SECTION 20–9 PRINTING OUT STRINGS AND STRING VARIABLES WITH PRINTUSING

The # symbol should be the only symbol used to define the images of literal strings and string variables. The numeric editing characters ($ – + , .) can be used, but they are recognized only as # symbols.

The 2200 replaces each character in the image with a text string character (i.e. an alphanumeric character) (Fig. 20–9).

(OUTPUT)

```
READY
:10 A $ = "ABCDEF"
:20 PRINTUSING 30, A$, A$, "STUVWXYZ"
:30% #######  $#.###   -###
:RUN
ABCDEF    ABCDEF    STUV

:_
```

Fig. 20–9

Text strings which are over formatted are left justified, and the format is filled out with blanks (1st Image in Fig. 20–9).

If the text string is longer than the format (underformatted) the string is truncated on the right (Fig. 20–9 third Image).

SECTION 20–10 ARRAYS WITH PRINTUSING

The PRINTUSING format can be used to generate the output in an array. Fig. 20–10 is an example of a program which both defines and prints out a 4 X 4 array with each element in the array being printed to a pre-set image (statements 60 + 70).

As you can see from this chapter the PRINTUSING format is a powerful tool for printing output for almost any desired image. The examples in this chapter, illustrate individual features of PRINT-USING, but many features can be combined in a single PRINTUSING statement.

PROGRAM

```
10  DIM F2(4,4)
20  N = .005
30  FOR I = 1 TO 4
40  FOR J = 1 TO 4:N = N + 1
50  F2(I,J) = N
60  PRINTUSING 70, F2(I,J);
70  % +##.###
80  NEXT J
90  PRINT
100 NEXT I
105 PRINT
110 STOP
```

```
:RUN
+1.005   +2.005   +3.005   +4.005

+5.005   +6.005   +7.005   +8.005

+9.005   +10.005  +11.005  +12.005

+13.005  +14.005  +15.005  +16.005

STOP

:_
```

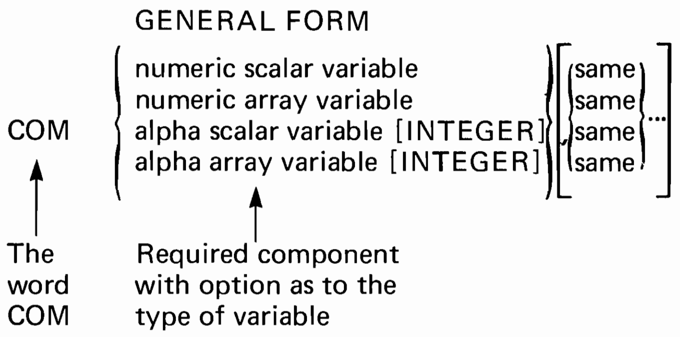
Fig. 20–10

CHAPTER 21
USE OF THE COMMON (COM) STATEMENT

The COM statement allows a programmer to store variables in memory for use in a subsequent program or to use variables from a previous program. The COM statement is used for two purposes: (1) with programs which are too long for the 2200's memory where the program is RUN in sections and data either inputted or generated is common to all sections of the program. By using the COM statement, the data does not have to be entered by hand each time a section of the program is run and (2) to designate data as common to be used with separate and distinct programs so as not to have to key in the data as each program is run. There are some definite rules which must be followed when using COM statement with programs. They are explained in this chapter.

SECTION 21–1
GENERAL FORM AND DESCRIPTION OF THE COM STATEMENT

The General Form of a COM statement is shown in Fig. 21–1.



GENERAL FORM
COM
numeric scalar variable
numeric array variable
alpha scalar variable [INTEGER]
alpha array variable [INTEGER]
same
same
same
same
...
The word COM
Required component with option as to the type of variable

Fig. 21–1

Some examples of COM statements are:

EXAMPLE 1

```
10  COM A(10),   B(3,3),   C2
numeric          numeric   numeric
array            array     scalar
variable         variable  variable
```

EXAMPLE 2

```
10  COM C,       M1$       B$(2,2)32
numeric          alpha     alpha array
scalar           scalar    variable
variable         variable
```

The COM statement allows array definition identical to the DIM statement for array variables. Both array and scalar variables can be included in one COM statement.

SECTION 21–2
STORAGE AND USE OF CLEAR WITH COMMON VARIABLES

STORAGE

A separate part of memory is set aside for the storage of common variables when a COM statement is used.

```
20  COM  B(3,3), D(2)
```

Fig. 21–2

CLEARING VARIABLES

Because common and non-common variables are stored separately as well as variables being stored separately from program text, a programmer is able to clear memory of (1) only non-common variables, (2) all variables (common and non-common), (3) all variables and programming text, (4) or all non-common variables and programming text. This is done with the use of the CLEAR command followed by a letter. The letters N, V, or P desiginates what is to be cleared.

CLEAR N EXECUTE clears only non-common variables. A programmer might use this if he is running several different programs with common data, to clear out any non-common variables generated with each program.

CLEAR V EXECUTE clears the memory of *all* variables (common and non-common). This is used by a programmer to clear memory of variables but leaves program text intact.

CLEAR P EXECUTE clears memory of program text only, leaving intact all variables. This is used when more than one program is being used with the same variables.

CLEAR EXECUTE clears the entire memory of *all* variables and *all* programming text.

SECTION 21–3 USING COM STATEMENTS IN PROGRAMS

Before COM statements are used in any programs two general rules must be followed. They are:

1. COM must be the first executable program line(s) in a program.
2. Common variables must be defined before any non-common variables are defined.

In the following example two programs are written which require the same data for execution. This data is therefore designated as Common. Fig. 21–3 shows both these programs. Notice the first statement of each program is a COM statement.

```
            Program 1
: 140 COM  A, B, C$6
: 150 PRINT  C$
: 160 FOR  I  =  A  to  B
: 170 PRINT I↑2, I↑3
: 180 NEXT I
: 190 END
            Program 2
: 200 COM A, B, C$6
: 210 PRINT C$
: 220 PRINT "C="; SQR(A↑2+B↑2)
: 230 END
```

Fig. 21–3

In order to execute these programs the common variables in statements 140 and 200 must be assigned values first.

Fig. 21–3a shows a short program which both defines and assigns values to the common variables. This must be done before the programs in Fig. 21–3 are executed.

```
:10  COM A, B, C$6
:20  A = 5: B = 10: C$ = "ANSWER"
```

Fig. 21–3a

Fig. 21–3b shows the result of executing the programs in Fig. 21–3.

```
:RUN 140   (Program 1)
ANSWER
25          125
36          216
49          243
64          512
81          729
100         1000

END PROGRAM
FREE SPACE = 3190

:RUN 200   (Program 2)
ANSWER

C = 11.180339887

END PROGRAM
FREE SPACE = 3190

:_
```

Fig. 21–3b

In the introduction to this chapter it was said that COM statements are used often with lengthy programs too large for memory. These programs are usually stored on some external storage device (e.g. Tape Cassette) and loaded into memory in segments. The loading of subsequent segments of the program after the first segment is loaded and executed is directed by the program itself. This is done with the LOAD EXECUTE command, and is called program chaining. It is an important programming technique which is very often used with the COM statement. (See 2200 Reference Manual on Tape Cassette Operations.)

CHAPTER 22
DEBUGGING

As discussed in Chapter 3 of Part I, the 2200 error diagnostics are capable of detecting syntax and execution errors in a program. Programming errors — those in which the program doesn't do what it should — are the responsibility of the programmer. A program error is commonly referred to as a bug. Several methods are available on the 2200 to help the programmer debug his program. This chapter discusses the various techniques a programmer can draw upon as an aid to debugging programs.

SECTION 22–1
HINTS FOR DEBUGGING A PROGRAM

The following suggests several rules which if followed could save a programmer much time in getting a program to run.

Rule 1 — Debugging begins before a program is even written i.e.

1. Make sure you know how to solve the problem.
2. "Play Computer" — go through a hand calculation first.
3. Trace through the flow chart before converting to a program.

Rule 2 — Prevent problems before they happen i.e.

1. Break program down into logical blocks.
2. Make sure all lines are entered correctly, and in the proper sequence.
3. After all blocks of the program are working well, then go back and economize.
4. Test out the program by running through it with "test" data. i.e. both data for which the answer is known, and a representative sample of real data, where possible.

Rule 3 — If a problem does exist, be logical in your approach — debugging is as much a logical art as programming is.

1. a. Quite often, the values which variables assume at the end of a program can tell you where to look for the problem.
   b. By using simple PRINT statements in the immediate mode to check the values of all variables after the program has run.
   c. Compare these values to what the expected value should be.
2. Check all equations — be sure they have been entered correctly with proper constants, variables, operators, and parentheses.
3. If program uses subscripted variables, be sure that proper subscripts are used, and that rows and columns have not been confused.
4. Be sure GOTO, GOSUB, and IF/THEN statements branch to the correct locations.
5. Re-check IF/THEN statements for proper tests and proper arguments. Quite often, the variable or expression tested against a critical (decision) value, never attains that value. RESULT: Branch is never executed.
6. In programs which use several subroutines or user functions, check to see that the proper subroutine or function is called, and that the proper arguments are passed.

OTHER APPROACHES

1. Print out intermediate results at various key locations, to check for correct variable values.
2. Use the HALT/STEP key to step through the execution of a program.
3. Use the TRACE feature to trace the variable values and branching in the program.
4. Include STOPs in program; when problem is discovered reexecute problem section with trace.

SECTION 22–2
USING HALT/STEP AS A DEBUGGING AID

The HALT/STEP key enables a 2200 user to execute a program statement by statement. There are two ways of stepping through a program. The first way is to touch the HALT/STEP key during execution of a program. This causes the 2200 to finish executing the statement it is presently at, display the line, its results, and stop executing the program. Touching HALT/STEP again causes the next statement in the program flow to be displayed and executed. Thus, you can step through a program statement by statement. The second way is to step through a program from a preselected line number by executing a GOTO 'line number' command followed by touching the HALT/STEP key one or more times (Fig. 22–2).

PROGRAM IN MEMORY

```
:10  J = 25
:20  K = 15
:30  GOTO 60
:40  PRINT J + K + L
:50  END
:60  L = 80
:70  GOTO 40
```

STEPPING THROUGH A PROGRAM

```
Key GOTO 20   EXECUTE
Key HALT/STEP
    20K = 15          displayed
Key HALT/STEP
    30 GOTO 60        displayed
Key HALT/STEP
    60L = 80          displayed
Key HALT/STEP
    70 GOTO 40        displayed
Key HALT/STEP
    40 PRINT J + K + L
    120 (Result of Execution of Line 40)
Key HALT/STEP
    50 END

    END PROGRAM Execution of Line 50
    FREE SPACE = 3291

    :_
```

Fig. 22–2

Notice in the above procedure as the HALT/STEP is touched, the next line is displayed along with the results (if any) of executing that line (e.g. lines 40 and 50). The next line that is displayed is always the next line to be logically executed in the program.

SECTION 22–3
HALT/STEP USAGE WITH MULTI-STATEMENT LINES

Multi-statement lines are executed one statement at a time, each time the HALT/STEP key is touched. Statements not yet executed are displayed with the executed statement (Fig. 22–3).

PROGRAM IN MEMORY

```
:10  X = 5 : Y = 10 : Z = 15
:20  PRINT X*Y - Z : STOP
```

HALT/STEP THROUGH PROGRAM

```
Key GOTO 10    EXECUTE
Key HALT/STEP
    10X = 5 : Y = 10 : Z = 15      displayed
Key HALT/STEP
    10:Y = 10 : Z = 15             displayed
Key HALT/STEP
    10::Z = 15                     displayed
Key HALT/STEP
    20 PRINT X * Y - Z : STOP
    35
Key HALT/STEP               Execution of
    20:STOP                 1st and 2nd
                            statements of
    STOP                    line 20
    :_
```

Fig. 22–3

When the HALT/STEP key is used with multi-statement lines, the first time the key is touched, the entire line is displayed, showing only the execution of the 1st statement, the next time the HALT/STEP key is touched, the 1st statement of the line is eliminated and the execution of the 2nd statement is shown, this continues until all statements are shown and executed.

SECTION 22–4
OTHER USES OF HALT/STEP KEY

(1) After the HALT/STEP key is touched, the 2200 can be used as a calculator to do side calculations, to check the value of any variables already defined in the program, or to redefine any variable(s) previously defined in the program.

(2) After the HALT/STEP key is touched as many times as required to check program flow, normal execution can be continued by keying

CONTINUE   CR/LF-EXECUTE

(3) If the operator attempts to HALT/STEP through a program after (a) a text or table overflow error has occurred, (b) a variable is defined which has not previously been defined, (c) any CLEAR command has been used, (d) program text has been added to, deleted, altered, or renumbered, or (e) the RESET key is touched, an error message is printed out, and execution does not continue.

(4) HALT/STEP, rather than RESET, should always be used to interrupt program execution - Use RESET to stop execution only if HALT/STEP fails.

SECTION 22–5 USE OF PROGRAM TRACE

While the HALT/STEP command gives the program flow statement by statement, the TRACE command allows the programmer to expand the output of a program. The TRACE command can be initiated either from the keyboard or from a program. To turn off a TRACE, key TRACE OFF CR/LF (OFF is keyed with upper case letters). In program text, TRACE is turned ON and OFF as shown in Fig. 22–5. (i.e. TRACE and TRACE OFF can be used as ordinary program statements)

```
100 TRACE
      .
      .
      .
      .
200 TRACE OFF
```

Fig. 22–5

Once the TRACE is turned ON it remains ON throughout Program execution until turned off by a TRACE OFF command. When the TRACE is ON (1) any variable which receives a new value during execution (e.g. with LET, READ, or FOR) is printed out and (2) TRANSFER TO xxx is printed out when a program transfer is made to another sequence of statements as a result of a GOTO, GOSUB, IF/THEN, NEXT and RETURN statements. Fig. 22–5a shows several TRACE examples.

CHAPTER 22
DEBUGGING

TRACE EXAMPLES

Assume each of the following examples are part of a separate program where the variables are already defined. The PRINTOUT shown is therefore only for the individual statement shown.

STATEMENT	PRINTOUT
Example #1	
30 X = 52 + SQR (81)	X = 61
Example #2	
70 READ A, B, X (22)	A = 9.4
	B = 64.27
	X ( ) = 2.824
Example #3	
100 GO TO 200	TRANSFER TO 200
Example #4	
30 GOSUB 80	
40 FOR I = 1 TO 25	
.	.
.	.
.	.
.	.
.	.
.	.
190 RETURN	TRANSFER TO 40
Example #5	
50 IF A > B THEN 90	TRANSFER TO 90

Fig. 22—5a

Using TRACE as part of program is illustrated in Fig. 20—5b.

PROGRAM

```
10   TRACE
20   Y = 21.5
30   IF X = 86 THEN 60
40   X = 4*Y
50   GO TO 30
60   TRACE OFF
70   STOP
```

RESULTING PRINTOUT

```
:RUN
 Y  = 21.5
 X = 86
 TRANSFER TO 30
 TRANSFER TO 60
 STOP
```

Fig. 22—5b

SECTION 22–6
USING HALT/STEP AND TRACE TOGETHER

You can trace a program one step at a time by combining the HALT/STEP procedure with TRACE MODE. Fig. 22–6 shows an example of a program which is TRACED step by step.

PROGRAM

```
 READY
:10 READ X, Y: Z = X*Y
:20 IF Z > 100 THEN 40
:30 GOTO 10
:40 PRINT Z
:50 STOP
:60 DATA 5,10,15,20,25,30
```

TO TRACE BY STEPPING

```
Key TRACE CR/LF
Key GOTO 10 CR/LF
Key HALT/STEP
    10 READ X,Y: Z = X*Y
    X = 5
    Y = 10
Key HALT/STEP
    10 : Z = X*Y
    Z = 50
Key HALT/STEP
    20 IF Z > 100 THEN 40
Key HALT/STEP
    30 GO TO 10
    TRANSFER TO 10
Key HALT/STEP
    10 READ X,Y: Z = X*Y
    X = 15
    Y = 20
Key HALT/STEP
    10 : Z = X*Y
    Z = 300
Key HALT/STEP
    20 IF Z > 100 THEN 40
    TRANSFER TO 40
Key HALT/STEP
    40 PRINT Z
    300
Key HALT/STEP
    50 STOP

    STOP
    :_
```

Fig. 22–6

SECTION 22–7
RENUMBERING A PROGRAM

Often when debugging a program or even in entering a program, statements need to be inserted between other statements. Statements are easily inserted if they are numbered 10, 20, 30 etc. where you can insert additional statements between 10 and 20 (e.g. 11, 12 etc.) and 20 and 30. However, when statements are numbered close together there may be no room for inserting additional statements.

With the RENUMBER Key, you can have the 2200 automatically renumber program statements in any fashion you wish. Not only are the statement lines renumbered but all references to statement numbers within the program are renumbered automatically. Another reason you may wish to renumber a program is to clean up a listing for appearance sake.

CHAPTER 22
DEBUGGING

THE RENUMBER statement has several options. (Fig. 22–7)

GENERAL FORM

RENUMBER [line number]	[, line number]	[, integer]
1st 'line number' specifies what line to start renumbering with. All lines ⩾ to this line number are renumbered. If no line number is specified all program lines are renumbered.	This 'line number' specifies what the new starting line number should be. If none is specified, it will equal the increment between line numbers.	The integer specifies what the increment between line numbers should be. If no increment is used, lines are automatically incremented by 10.

Fig. 22–7

Take as an example the following program (Fig. 22–7a) and note the several different ways you can renumber the same program using the options mentioned above. Key in this program.

PROGRAM

```
:1 FOR X = 1 to 10
:2 PRINT X, X ↑ 2, X ↑ 3
:3 NEXT X
:4 GOTO 1
:5 END
```

Fig. 22–7a

EXAMPLE 1

```
Key RENUMBER CR/LF
    LIST CR/LF
    RESULT

    10   FOR X = 1 TO 10
    20   PRINT X, X ↑ 2, X ↑ 3
    30   NEXT X
    40   GO TO 10
    50   END
```

Comment: All program lines are renumbered in increments of 10. The first line number of the resulting program is 10.

EXAMPLE 2

```
Key RENUMBER 20, 12, 10 CR/LF
    LIST CR/LF
  10 FOR X = 1 TO 10
  12 PRINT X, X ↑ 2, X ↑ 3
  22 NEXT X
  32 GO TO 1
  42 END
```

Comment: All program lines beginning with line 20 are numbered in increments of 10. The new line number for 20 is 12.

EXAMPLE 3

```
Key RENUMBER, 5,5 CR/LF
    LIST CR/LF
     5 FOR X = 1 TO 10
    10 PRINT X, X ↑2, X ↑ 3
    15 NEXT X
    20 GOTO 5
    25 END
```

Comment: All program line numbers are renumbered since the 1st parameter is omitted. The new starting line number is 5 and the increment is 5.

EXAMPLE 4

```
Key RENUMBER, , 15 CR/LF
    LIST CR/LF
    15   FOR X = 1 TO 10
    30   PRINT X, X ↑ 2, X ↑ 3
    45   NEXT X
    60   GOTO 15
    75   END
```

Comment: All program line numbers are renumbered because the 1st parameter is omitted. The increment is 15 and the new starting line number equals the increment.

CHAPTER 23
THE HEXADECIMAL FUNCTION [HEX( )]

The HEX function allows the user to output hexadecimal codes to any peripheral on the 2200. Every character or command related to a peripheral is expressed in a unique 2-digit HEX code. The HEX function gives the user the capability to control any feature of a peripheral such as moving the CRT cursor around for plotting or outputting characters that do not appear on the keyboard (e.g. @ or?).

SECTION 23–1
WHAT IS A HEX CODE?

A HEX code is based upon the Hexadecimal counting system. The Hexadecimal System unlike the Decimal System (base 10) is to the base 16. In the Decimal System the digits used are 0–9, while in the Hexadecimal system the digits used are 0–9 and A–F. The numbers in the Hexadecimal System are: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E, and F. Combinations of these digits as with combinations of 0–9 in the decimal system, are used to represent all numbers. Fig. 23–1 shows how counting is done in the Hexadecimal System as compared to the Decimal System.

HEXADECIMAL	DECIMAL
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15
10	16
11	17
12	18
13	19
14	20
15	30
16	31

Fig. 23–1

The Hexadecimal System is used with many computers in their internal design. On the 2200 all characters or commands outputted to a peripheral are represented by a 2-digit HEX code. There are 256 such codes. Fig. 23–1a illustrates these codes.

HEX CODES

00	20	40	60	80	A0	C0	E0
01	21	41	61	81	A1	C1	E1
02	22	42	62	82	A2	C2	E2
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
09	29	49	69	89	A9	C9	E9
0A	2A	4A	6A	8A	AA	CA	EA
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
0F	2F	4F	6F	8F	AF	CF	EF
10	30	50	70	90	B0	D0	F0
11	31	51	71	91	B1	D1	F1
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
19	39	59	79	99	B9	D9	F9
1A	3A	5A	7A	9A	BA	DA	FA
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
1F	3F	5F	7F	9F	BF	DF	FF

Fig. 23–1a

CHAPTER 23
THE HEXADECIMAL FUNCTION [HEX( )]

SECTION 23–2
FORMAT OF HEX FUNCTION IN A BASIC STATEMENT LINE

The 2-digit code used in the HEX function consists either of a two digit number, each digit from 0–9, a letter and a digit, or two letters, where the digit is from 0–9 and the letter is from A–F (Fig. 23–1a).

A BASIC statement line can contain any number of HEX codes. If more than one HEX code is used in a line in sequence there are two ways of writing the line. Fig. 23–2 shows the same line written the two different ways:

```
: 10  PRINT HEX (0909); "ABC"
              or
: 10  PRINT HEX (09); HEX (09); "ABC"
```

Fig. 23–2

The Code HEX (09) causes the CRT cursor to move 1 space to the right. In these examples, you want to produce two spaces in the line before the letters "ABC" are printed. Both examples do this. In the first example, the codes are combined in one set of parentheses. In the second example, the codes are written separately [HEX (09); HEX (09)] Either way is correct. A comma or semicolon is used as punctuation to separate the codes for a zoned or packed format.

SECTION 23–3
SPECIAL CHARACTERS AND CURSOR CONTROLS GENERATED WITH HEX CODES

As already mentioned, every letter, digit and symbol on the keyboard has a corresponding HEX code. The movement of the cursor, right or left, up or down for example can be programmed using HEX codes. Table 23–3 shows a list of the special characters and cursor movement codes.

The complete list of HEX codes for the CRT (2216) is given in Appendix F.

Each peripheral available for the 2200 has a corresponding set of HEX codes.

Some of these codes produce the same results on all peripherals, while others either have no effect or generate a different character. For the listing of the codes for a specific peripheral, see the Reference Manual provided with the purchase of the equipment.

TABLE 23–3

CHARACTER/ CURSOR DIRECTION	HEX CODE	CHARACTER/ CURSOR DIRECTION	HEX CODE
Cursor home	HEX (01)	]	HEX (5D)
Clears screen and Cursor home	HEX (03)	↑	HEX (5E)
		←	HEX (5F)
Bell	HEX (07)	#	HEX (23)
Cursor left	HEX (08)	%	HEX (25)
Cursor right	HEX (09)	' (Apostrophe)	HEX (27)
Cursor down ↓ (Line Feed)	HEX (0A)	*	HEX (2A)
		, (Comma)	HEX (2C)
Cursor up ↑ (Reverse Index)	HEX (0C)	/	HEX (2F)
		:	HEX (3A)
CR/LF	HEX (0D)	;	HEX (3B)
!	HEX (21)	<	HEX (3C)
"	HEX (22)	=	HEX (3D)
&	HEX (26)	>	HEX (3E)
?	HEX (3F)	(	HEX (28)
@	HEX (40)	)	HEX (29)
[	HEX (5B)		
\	HEX (5C)		

CHAPTER 23
THE HEXADECIMAL FUNCTION [HEX( )]

SECTION 23–4
PLOTTING EXAMPLE

The following example prints a running account of the variable I. The HEX function in line 30 is used to return the cursor to the same line after each printout. HEX (OC) is a CURSOR UP command.

```
:10 I = 0
:20 PRINT "COUNT=";I
:30 PRINT HEX(OC); :REM---CURSOR UP
:40 I = I+1
:50 GOTO 20
:RUN
```

$\underbrace{\text{COUNT}}_{\text{fixed on screen}} = \underbrace{329}_{\text{a running total}}$

Part IV
Appendices

CODE 01

Error: **Text Overflow**

Cause: All available space for BASIC statements and system commands has been used.

Action: Shorten and/or chain program by using COM statements, and continue. The compiler automatically removes the current and highest-numbered statement.

Example:

```
:10 FOR I   = 1 TO 10
:20 LET X   = SIN(I)
:30 NEXT I
     ....
     ....
     ....
:820 IF Z   = A-B THEN 900
↑ERR 01
```

(the number of characters in the program exceeded the text table limit when line 820 was entered)

User must shorten or segment program.

CODE 02

Error: **Table Overflow**

Cause: All available space for internal compiler tables has been used (storage of variables, values, etc.) or an endless program loop was encountered.

Action: Shorten or correct and/or chain the program by using COM statements and continue.

Example:

```
:10 DIM A(19), B(10,10), C(10,10)
:RUN
↑ERR 02
```

(the table space required for variables exceeded the table limit for variable storage as line 10 was processed)

User must compress program and variable storage requirements.

CODE 03

Error: **Math Error**

Cause:

1. EXPONENT OVERFLOW. The resulting magnitude of the number calculated was greater than or equal to 10^{100}. (+ , – , * , / , ↑ , TAN , EXP).
2. DIVISION BY ZERO.
3. NEGATIVE OR ZERO LOG FUNCTION ARGUMENT.
4. NEGATIVE SQR FUNCTION ARGUMENT.
5. INVALID EXPONENTIATION. An exponentiation, (X↑Y) was attempted where X was negative and Y was not an integral, producing an imaginary result, or X and Y were both zero.
6. ILLEGAL SIN, COS, OR TAN ARGUMENT. The function argument exceeds 2π X 10'' radians.

Action: Correct the program or program data.

Example:

```
:PRINT (2E + 64)   /   (2E – 41)
PRINT  (2E + 64    /   (2E – 41)
                            ↑ERR 03          (exponent overflow)
```

CODE 04

Error:	**Missing Left Parenthesis**
Cause:	A left parenthesis (( ) was expected.
Action:	Correct statement text.

Example:

```
:10 DEF FNA V) = SIN(3*V-1)
               ↑ERR 04
:10 DEF FNA(V) + SIN(3*V-1)         (Possible Correction)
```

CODE 05

Error:	**Missing Right Parenthesis**
Cause:	A right ( )) parenthesis was expected.
Action:	Correct statement text.

Example:

```
:10 Y = INT(1.2↑5
                  ↑ERR 05
:10 Y = INT(1.2↑5)                  (Possible Correction)
```

CODE 06

Error:	**Missing Equals Sign**
Cause:	An equals sign (=) was expected.
Action:	Correct statement text.

Example:

```
:10 DEF FNC(V)-V+2
              ↑ERR 06
:10 DEF FNC(V)=V+2                  (Possible Correction)
```

CODE 07

Error:	**Missing Quotation Marks**
Cause:	Quotation marks (") were expected.
Action:	Correct statement text.

Example:

```
:10 PRINT "ERROR
                ↑ERR 07
:10 PRINT "ERROR"                   (Possible Correction)
```

CODE 08

Error:	**Undefined FN Function**
Cause:	An undefined FN function was referenced.
Action:	Correct program to define or reference the function correctly.

Example:

```
:10 X=FNC(2)
:20 PRINT "X";X
:30 END
:RUN
10 X=FNC(2)
     ↑ERR 08
:05 DEFFNC(V)=COS(2*V)              (Possible Correction)
```

CODE 09

Error:	**Illegal FN Usage**
Cause:	More than five levels of nesting were encountered when evaluating an FN function.
Action:	Reduce the number of nested functions.

Example:

```
:10 DEF FN1(X)=1+X       :DEF FN2(X)=1+FN1(X)
:20 DEF FN3(X)=1+FN2(X)     :DEF FN4(X)=1+FN3(X)
:30 DEF FN5(X)=1+FN4(X)     :DEF FN6(X)=1+FN5(X)
:40 PRINT FN6(2)
:RUN
10 DEF FN1(X)=1+X        :DEF FN2(X)=1+FN1(X)
              ↑ERR 09
:40 PRINT 1+FN5(2)                  (Possible Correction)
```

CODE 10

Error:	**Incomplete Statement**
Cause:	The end of the statement was expected.
Action:	Complete the statement text.

Example:

```
:10 PRINT X"
            ↑ERR 10
:10 PRINT "X"
      OR
:10 PRINT X                         (Possible Correction)
```

CODE 11

Error:	**Missing Line Number or Continue Illegal**
Cause:	The line number is missing or a referenced line number is undefined; or the user is attempting to continue program execution after one of the following conditions: A text or table overflow error, a new variable has been entered, a CLEAR command has been entered, the user program text has been modified, or the RESET key has been pressed.
Action:	Correct statement text.

Example:

```
:10 GOSUB 200
            ↑ERR 11
:10 GOSUB 100                       (Possible Correction)
```

CODE 12

Error:	**Missing Statement Text**
Cause:	The required statement text is missing (THEN, STEP, etc.).
Action:	Correct statement text.

Example:

```
:10 IF I+12*X,45
              ↑ERR 12
:10 IF I=12*X THEN 45               (Possible Correction)
```

CODE 13

Error:	**Missing or Illegal Integer**
Cause:	A positive integer was expected or an integer was found which exceeded the allowed limit.
Action:	Correct statement text.
Example:	:10 COM D(P)

```
:10 COM D(P)
           ↑ERR 13
:10 COM D(8)                    (Possible Correction)
```

CODE 14

Error:	**Missing Relation Operator**
Cause:	A relational operator (<, =, >, <=, >=, <>) was expected.
Action:	Correct statement text.
Example:	

```
:10 IF A-B THEN 100
          ↑ERR 14
:10 IF A=B THEN 100             (Possible Correction)
```

CODE 15

Error:	**Missing Expression**
Cause:	A variable, or number, or a function was expected.
Action:	Correct statement text.
Example:	

```
:10 FOR I=, TO 2
          ↑ERR 15
:10 FOR I=1 TO 2                (Possible Correction)
```

CODE 16

Error:	**Missing Scalar**
Cause:	A scalar variable was expected.
Action:	Correct statement text.
Example:	

```
:10 FOR A(3)=1 TO 2
           ↑ERR 16
:10 FOR B=1 TO 2                (Possible Correction)
```

CODE 17

Error:	**Missing Array**
Cause:	An array variable was expected.
Action:	Correct statement text.
Example:	

```
:10 DIM A2
          ↑ERR 17
:10 DIM A(2)                    (Possible Correction)
```

CODE 18

Error: **Illegal Value for Array Dimension**

Cause: The values assigned for array dimensions exceed the allowable limits; a dimension is greater than 255; an array variable subscript exceeds the defined dimension.

Action: Correct the program.

Example:

```
:10 DIM A(2,3)
:20 A(1,4) = 1
:RUN
 20 A(1,4) = 1
          ↑ERR 18
:10 DIM A(2,4)                (Possible Correction)
```

CODE 19

Error: **Missing Number**

Cause: A number was expected.

Action: Correct statement text.

Example:

```
:10 DATA L
           ↑ERR 19
:10 DATA 1                    (Possible Correction)
```

CODE 20

Error: **Illegal Number Format**

Cause: A number format is illegal.

Action: Correct statement text.

Example:

```
:10 A=12345678.234567                     (More than 13 digits of mantissa)
                     ↑ERR 20
:10 A=12345678.23456          (Possible Correction)
```

CODE 21

Error: **Missing Letter or Digit**

Cause: A letter or digit was expected.

Action: Correct statement text.

Example:

```
:10 DEF FN.(X)=X↑5-1
           ↑ERR 21
:10 DEF FN1(X)=X↑5-1          (Possible Correction)
```

CODE 22	
Error:	**Undefined Array Variable**
Cause:	An array variable was not referenced as previously defined in this program or as a common variable in another program (i.e., an array variable has been referenced both as a 1-dimensional and as a 2-dimensional array).
Action:	Correct statement text.
Example:	

```
:10 A(2,2) = 123
:RUN
 10 A(2,2) = 123
      ↑ERR 22
:1 DIM A(4,4)                      (Possible Correction)
```

CODE 23	
Error:	**No Program Statements**
Cause:	A RUN command was entered but there are no program statements.
Action:	Enter program statements.
Example:	

```
:RUN
 ↑ERR 23
```

CODE 24	
Error:	**Illegal Immediate Mode Statement**
Cause:	An illegal verb or transfer in an immediate execution statement was encountered.
Action:	Re-enter a corrected immediate execution statement.
Example:	

```
 IF A = 1 THEN 100
 ↑ERR 24
```

CODE 25

Error:	**Illegal GOSUB/RETURN Usage**
Cause:	There is no companion GOSUB statement for a RETURN statement.
Action:	Correct the program.

Example:

```
:10 FOR I+1 TO 20
:20 X=I*SIN(I*4)
:25 GO TO 100
:30 NEXT I: END
:100 PRINT "X=";X
:110 RETURN
:RUN
X=- .7568025

110 RETURN
          ↑ ERR 25
:25 GOSUB 100                (Possible Correction)
```

CODE 26

Error:	**Illegal FOR/NEXT Usage**
Cause:	There is no companion FOR statement for a NEXT statement.
Action:	Correct the program.

Example:

```
:10 PRINT "I=";I
:20 NEXT I
:30 END
:RUN
I = 0
 20 NEXT I
        ↑ERR 26
:5 FOR I=1 TO 10             (Possible Correction)
```

CODE 27

Error:	**Insufficient Data**
Cause:	There is unsufficient data for READ statement requirements.
Action:	Correct program to supply additional data.

Example:

```
:10 DATA 2
:20 READ X,Y
:30 END
:RUN

 20 READ X,Y
          ↑ERR 27
:11 DATA 3                   (Possible Correction)
```

CODE 28

Error:	**Data Reference Beyond Limits**
Cause:	The data reference in a RESTORE statement is beyond the existing data limits.
Action:	Correct the RESTORE statement.
Example:	

```
:10 DATA 1,2,3
:20 READ X,Y,Z
:30 RESTORE 5
     ....
     ....
     ....
:90 END
:RUN
 30 RESTORE 5
                ↑ERR 28
:30 RESTORE 2                 (Possible Correction)
```

CODE 29

Error:	**Illegal Data Format**
Cause:	The data format for an INPUT statement is illegal (format error).
Action:	Reenter data in the correct format starting with erroneous number or terminate run with the RESET key and run again.
Example:	

```
:10 INPUT X,Y
     ....
     ....
     ....
:90 END
:RUN
:INPUT
:1A,2E-30
  ↑ERR 29
:12,2E-30                     (Possible Correction)
```

CODE 30

Error:	**Illegal Common Assignment**
Cause:	A COM statement variable definition was preceded by a non-common variable definition.
Action:	Correct program, making all COM statements the first numbered lines.
Example:	

```
:10 A=1 :B=2
:20 COM A,B
:99 END
:RUN

 20 COM A,B
          ↑ERR 30
:10[CR/LF—EXECUTE]            (Possible Correction)
:30 A=1 :B=2
```

CODE 31

Error: **Illegal Line Number**

Cause: The 'statement number' key was pressed producing a line number greater than 9999; or in renumbering a program with the RENUMBER command a line number was generated which was greater than 9999.

Action: Correct the program.

Example:

```
:9995 PRINT X,Y
:[line number key]
 ↑ERR 31
```

CODE 33

Error: **Missing HEX Digit**

Cause: A digit or a letter from A - F was expected.

Action: Correct the program text.

Example:

```
:10 SELECT PRINT 00P
                   ↑ERR 33
:10 SELECT PRINT 005          (Possible Correction)
```

CODE 34

Error: **Tape Read Error**

Cause: The system was unable to read the next record on the tape; the tape is positioned after the bad record.

CODE 35

Error: **Missing Comma or Semicolon**

Cause: A comma or semicolon was expected.

Action: Correct statement text.

Example:

```
:10 DATASAVE #2   X,Y,Z
                 ↑ ERR 35
:10 DATASAVE #2,X,Y,Z         (Possible Correction)
```

CODE 36

Error: **Illegal Image Statement**

Cause: No format (e.g. #.##) in image statement.

Action: Correct the statement text.

Example:

```
:10 PRINTUSING 20, 1.23
:20% AMOUNT =
:RUN
:10 PRINTUSING 20,1.23
                  ↑ERR 36
:20% AMOUNT = #####           (Possible Correction)
```

CODE 37

Error:	**Statement Not Image Statement**
Cause:	The statement referenced by the PRINTUSING statement is not an image statement.
Action:	Correct the statement text.
Example:	

```
:10 PRINTUSING 20,X
:20 PRINT X
:RUN
:10 PRINTUSING 20,X
                   ↑ERR37
:20% AMOUNT = $#,###.##        (Possible Correction)
```

CODE 38

Error:	**Illegal Floating Point Format**
Cause:	Fewer than 4 up arrows were specified in the floating point format in an image statement.
Action:	Correct the statement text.
Example:	

```
:10 % ##.##↑↑↑
              ↑ ERR 38
:10 % ##.##↑↑↑↑
```

CODE 39

Error:	**Missing Literal String**
Cause:	A literal string was expected.
Action:	Correct the text.
Example:	

```
:10 READ A$
:20 DATA 123
:RUN
20 DATA 123
        ↑ERR 39
20 DATA "123"                  (Possible Correction)
```

CODE 40

Error:	**Missing Alphanumeric Variable**
Cause:	An alphanumeric variable was expected.
Action:	Correct the statement text.
Example:	

```
:10 A$, X = "JOHN"
       ↑ERR 40
:10 A$, X$ = "JOHN"
```

CODE 41

Error:	**Illegal STR(Arguments**
Cause:	The STR(function arguments exceed the maximum length of the string variable.
Example:	

```
:10 B$ = STR(A$, 10, 8)
                     ↑ERR 41
:10 B$ = STR(A$, 10, 6)        (Possible Correction)
```

CODE 42	
Error:	**File Name Too Long**
Cause:	The program name specified is too long (a maximum of 8 characters is allowed).
Action:	Correct the program text.
Example:	:SAVE "PROGRAM#1"

```
:SAVE "PROGRAM#1"
                        ↑ERR 42
:SAVE "PROGRAM1"                (Possible Correction)
```

CODE 43	
Error:	**Wrong Variable Type**
Cause:	During a DATALOAD operation a numeric (or alphanumeric) value was expected but an alphanumeric (or numeric) value was read.
Action:	Correct the program or make sure proper tape is mounted.
Example:	:DATALOAD X, Y

```
:DATALOAD X, Y
              ↑ERR 43
:DATALOAD X$, Y$                (Possible Correction)
```

CODE 44	
Error:	**Program Protected**
Cause:	A program loaded was protected and, hence, cannot be SAVED or LISTED.
Action:	Execute a CLEAR command to remove protect mode, (but, program will be scratched).

CODE 45	
Error:	**Statement Line Too Long**
Cause:	A statement line may not exceed 192 keystrokes.
Action:	Shorten the statement line being entered.

CODE 46	
Error:	**New Starting Statement Number Too Low**
Cause:	The new starting statement number in a RENUMBER command is not greater than the next lowest statement number.
Action:	Reenter the RENUMBER command correctly.
Example:	50 REM – PROGRAM 1

```
50 REM – PROGRAM 1
62 PRINT X, Y
73 GOSUB 500
:
:RENUMBER 62, 20, 5
                ↑ERR 46
:RENUMBER 62, 60, 5             (Possible Correction)
```

CODE 47	
Error:	**Illegal Or Undefined Device Specification**
Cause:	The #n device specifications in a program statement is undefined.
Action:	Define the specified device numbers.
Example:	:SAVE #2

```
:SAVE #2
       ↑ERR 47
:SELECT #2 10A
:SAVE #2                        (Possible Correction)
```

CODE 48

Error:	**Undefined Keyboard Function**
Cause:	There is no mark (DEFFN') in a user's program corresponding to the keyboard function key depressed.
Action:	Correct the program.
Example:	**:[keyboard function key #2]** **↑ERR 48**

CODE 49

Error:	**End of Tape**
Cause:	The end of tape was encountered during a tape operation.
Action:	Correct the program or make sure the tape is correctly positioned.
Example:	see below

```
100 DATALOAD X, Y, Z
                   ↑ERR 49
```

CODE 50

Error:	**Protected Tape**
Cause:	A tape operation is attempting to write on a tape cassette that has been protected (by tab being punched out).
Action:	Mount another cassette or "unprotect" the tape cassette by covering the punched tab with masking tape.
Example:	see below

```
SAVE /103
         ↑ERR 50
```

CODE 51

Error:	**Illegal Statement**
Cause:	The 2200 does not have the micro-program in it to process this BASIC statement.
Action:	Do not use this statement.

CODE 52

Error:	**Expected Data (Nonheader) Record**
Cause:	A DATALOAD operation was attempted but the cassette was not positioned at a data record.
Action:	Make sure the correct tape cassette is mounted and positioned correctly.

CODE 53

Error:	**Illegal Use of HEX Function**
Cause:	The HEX(function is being used in an illegal situation. The HEX function may not be used in a PRINTUSING statement.
Action:	Do not use HEX function in this situation.
Example:	see below

```
:10 PRINTUSING 200, HEX(F4F5)
                        ↑ ERR 53
:10 A$ = HEX(F4F5)
:20 PRINTUSING 200,A$           (Possible Correction)
```

LISTING OF ERROR MESSAGES

CODE 01	TEXT OVERFLOW
CODE 02	TABLE OVERFLOW
CODE 03	MATH ERROR
CODE 04	MISSING LEFT PARENTHESIS
CODE 05	MISSING RIGHT PARENTHESIS
CODE 06	MISSING EQUALS SIGN
CODE 07	MISSING QUOTATION MARKS
CODE 08	UNDEFINED FN FUNCTION
CODE 09	ILLEGAL FN USAGE
CODE 10	INCOMPLETE STATEMENT
CODE 11	MISSING LINE NUMBER OR CONTINUE ILLEGAL
CODE 12	MISSING STATEMENT TEXT
CODE 13	MISSING OR ILLEGAL INTEGER
CODE 14	MISSING RELATION OPERATOR
CODE 15	MISSING EXPRESSION
CODE 16	MISSING SCALAR
CODE 17	MISSING ARRAY
CODE 18	ILLEGAL VALUE FOR ARRAY DIMENSION
CODE 19	MISSING NUMBER
CODE 20	ILLEGAL NUMBER FORMAT
CODE 21	MISSING LETTER OR DIGIT
CODE 22	UNDEFINED ARRAY VARIABLE
CODE 23	NO PROGRAM STATEMENTS
CODE 24	ILLEGAL IMMEDIATE MODE STATEMENT
CODE 25	ILLEGAL GOSUB/RETURN USAGE
CODE 26	ILLEGAL FOR/NEXT USAGE
CODE 27	INSUFFICIENT DATA
CODE 28	DATA REFERENCE BEYOND LIMITS
CODE 29	ILLEGAL DATA FORMAT
CODE 30	ILLEGAL COMMON ASSIGNMENT
CODE 31	ILLEGAL LINE NUMBER
CODE 33	MISSING HEX DIGIT
CODE 34	TAPE READ ERROR
CODE 35	MISSING COMMA OR SEMICOLON
CODE 36	ILLEGAL IMAGE STATEMENT
CODE 37	STATEMENT NOT IMAGE STATEMENT
CODE 38	ILLEGAL FLOATING POINT FORMAT
CODE 39	MISSING LITERAL STRING
CODE 40	MISSING ALPHANUMERIC VARIABLE
CODE 41	ILLEGAL STR(ARGUMENTS
CODE 42	FILE NAME TOO LONG
CODE 43	WRONG VARIABLE TYPE
CODE 44	PROGRAM PROTECTED
CODE 45	STATEMENT LINE TOO LONG
CODE 46	NEW STARTING STATEMENT NUMBER TOO LOW
CODE 47	ILLEGAL OR UNDEFINED DEVICE SPECIFICATION
CODE 48	UNDEFINED KEYBOARD FUNCTION
CODE 49	END OF TAPE
CODE 50	PROTECTED TAPE
CODE 51	ILLEGAL STATEMENT
CODE 52	EXPECTED DATA (NONHEADER) RECORD
CODE 53	ILLEGAL USE OF HEX FUNCTION

EXTIMATING PROGRAM MEMORY REQUIREMENTS

STEP 1: Extimate Program Text Size

Each BASIC program line requires the following amount of storage space (in STEPS or BYTES).

a. Line number	=	5
b. Each BASIC word	=	1
c. Each referenced line number	=	3
d. Remaining text material (including CR/LF-EXECUTE)	=	1 per keystroke

Rather than counting the requirements for each individual text line, you can estimate the average length of groups of lines, or use the following guidelines:

1. Simple, single statement lines	:	approximately 18 bytes per line
2. Multi-statement lines	:	approximately 27 bytes per line

STEP 2: Estimate Array Variable Storage Requirements

Look for COM and DIM statements in the program. Estimate storage for numeric arrays and string arrays separately.

a. Numeric arrays: Calculate the total number of elements in each numeric array. For one-dimensional arrays, this is the dimensioned subscript; for two dimensional-arrays, it is the product of the two subscripts.

EXAMPLE: Given the statement DIM A(5), B(6,3) (or COM A(5), B(6,3)), the array A( ) has five elements and the array B( ) has 18.

Each element requires eight bytes of storage. Therefore, the arrays A( ) and B( ) require 40 bytes and 144 bytes respectively.

b. String arrays: Each string array element requires sixteen bytes of storage unless otherwise specified in COM or DIM statements. If a maximum length is specified (in DIM or COM) for a string array, each element in that array requires the specified number of bytes (or steps) of memory.

EXAMPLE: Given the statement DIM N$(30), P$(6,4), R$(7,10)4, T$(6)24, the arrays require the following amounts of storage:

N$( )	=	30 elements X 16 bytes/element	=	480 bytes
P$( )	=	6 X 4 elements X 16 bytes/element	=	384 bytes
R$( )	=	7 X 10 elements X 4 bytes/element	=	280 bytes
T$( )	=	6 elements X 24 bytes/element	=	144 bytes

STEP 3: Estimate Scalar Variable Storage

a. Numeric scalar variables: Each numeric scalar variable requires four bytes for the variable name plus eight bytes for the value of the variable.

b. String scalar variables: Each string scalar variable requires five bytes for the name plus sixteen bytes for the value, if a maximum size is not specified in a DIM or COM statement. If a maximum size is specified, the variable requires five bytes for the name plus the specified number of bytes (or steps).

EXAMPLE: Given the statements

```
10   DIM B$3 (or 10 COM B$3)
20   LET C$ = "NAME, ADDRESS"
```

The variable B$ requires 5 bytes for the name plus 3 bytes for the value. The variable C$ requires 5 bytes for the name plus 16 bytes for the value.

STEP 4: Total Memory Used By The Program

Add the results of STEPS 1 - 3.

STEP 5: Total Memory Required For Execution

Add the result in STEP 4 to the number of bytes used in the BASIC system scratch area (approximately 700 bytes).

PROGRAM MEMORY REQUIREMENTS

(WORKSHEET)

PROGRAM TITLE

STEP 1:	Program Text Storage		
	a. Number of program lines	=	
	b. Average number of bytes per line	=	
	c. APPROXIMATE NUMBER OF TEXT BYTES (a*b)	=	
STEP 2:	Array Variable Storage		
	d. Total number of numeric array elements	=	
	e. Total number of bytes required (8 X 'd')	=	
	f. Total bytes required for all string arrays	=	
	g. TOTAL MEMORY FOR ARRAYS (e+f)	=	
STEP 3:	Scalar Variable Storage		
	h. Total number of numeric variables	=	
	i. Total storage required (12 bytes X 'h')	=	
	j. Total storage for string variables	=	
	k. TOTAL MEMORY FOR SCALAR VARIABLES (i+j)	=	
STEP 4:	Program Storage		
	l. 'c' + 'g' + 'k'	=	
STEP 5:	Total Program Execution Requirements		
	m. BASIC system scratch area	=	700

TOTAL MEMORY REQUIRED ('l' + 'm') = bytes

APPENDIX C

CRT (CATHODE RAY TUBE MODEL 2216)

Unit Size:	Height	14 inches	(35.6cm)
	Depth	16 inches	(40.6cm)
	Width	21½ inches	(54.6cm)
Display Size:	Height	8 inches	(20.3cm)
	Width	10½ inches	(26.7cm)

Capacity:
16 lines
64 characters/line

Character Size:

	Height	.20 inches	(.51cm)
	Width	.12 inches	(.30cm)
Weight:	36 lbs.	(14.4kg)	

Power Requirements:
115 or 230 VAC ± 10%
50 or 60 Hz

Cable 12′

TAPE DRIVE (MODEL 2217)

Stop/Start Time:
.09/.05 second

Capacity: 522 bytes/ft. (1712 bytes/m)

Recording Speed: 7.5 IPS (19.04 cm/sec.)

Search Speed: 7.5 IPS (19.04 cm/sec.)

Transfer Rate: 326 char/sec. (approx.)

Inter-record Gap: .6 inches (1.52cm)

(Capacity and Transfer Rate include Gaps and Redundant Recording)
Cable 12′

KEYBOARD

Size:	Height	3 inches	(7.62cm)
	Depth	10 inches	(25.4cm)
	Width	17½ inches	(44.5cm)
	Weight	7 lbs.	(2.8kg)

Cable 12′

CPU (CENTRAL PROCESSING UNIT MODEL 2200)

Built-in Functions

Mathematical & Trigonometric Functions

e^x	e to the power of x
Log	Natural Log
SQR	Square Root
π	Pi
Sin	Sine
Cos	Cosine
Tan	Tangent
Arc Sin	Inverse Sine
Arc Cos	Inverse Cosine
Arc Tan	Inverse Tangent
RND	Random Number Generator
ABS	Absolute Value of a Number.
INT	Integer Value of a Number.
SGN	1, 0, or +1 if a number is negative, 0, or positive.

(Trigonometric Functions in Degrees, Radians and Grads)

Alphanumeric Functions

STR	Selection of one or more characters in an alphanumeric string.
HEX	Hexidecimal Values.
LEN	Length of Alphanumeric Variable.

Variable Formats
Scalar Numeric Variable.
Numeric 1 and 2 dimension Array Variables.
Scalar Alphanumeric String Variable.
Alphanumeric 1- and 2- dimensional String Arrays.

Average Execution Times (Milliseconds)

Add/Subtract	.8
Multiply/Divide	3.8/7.4
Square Root/E^x	46.4/25.3
$\text{Log}_e x/X^y$	23.2/45.4
Integer/Absolute Value	.24/.02
Sign/Sine	.25/38.3
Cosine/Tangent	38.9/78.5
Arctangent	72.5
Read/Write Cycle	1.6 μ sec.

(Average Execution times were determined using random number arguments with 13 digits of precision. Average Execution times will be faster in most calculations with arguments having fewer significant digits.

Capacity

Memory Size	4,096 program steps (expandable to 32K)
Peripheral Capacity	6 (expandable in increments of 5) limit 36
Dynamic Range	10^{-99} to 10^{+99}
Subroutine Stacking	No Limit
Weight	60 lbs. (24 kg)
Power Requirements	115 or 230 VAC ± 10%; 50 or 60 Hz

AVAILABLE PERIPHERALS

2201	Output Writer
2202	Plotting Output Writer [1]
2203	Punched Paper Tape Reader [1]
2212	Flat Bed Plotter [1]
2214	Marked Sense Card Reader [1]
2215	Basic Keyboard Module
2216	CRT Display Module
2217	Single Magnetic Tape Cassette Reader/ Recorder
2216/2217	Combined CRT Display/Single Magnetic Tape Cassette Module
2219	I/O Extension Chassis
2221	High-Speed Printer 132 Column
2222	Alphanumeric Keyboard Module
2230-1	Disk Memory (1,228,800 bytes) [1]
2230-2	Disk Memory (2,457,600 bytes) [1]
2230-3	Disk Memory (4,915,200 bytes) [1]
2231	High-Speed Printer 80 Column
2207	Teletype Controller [1]
2227	Standard Telecommunications option

[1] Peripherals used with the 2200B only.
A 2200A can be upgraded to a 2200B upon request at a nominal charge.

DEVICE ADDRESSES FOR 2200 PERIPHERALS

(For further detail, see the individual peripheral manuals)

I/O DEVICE CATEGORIES		DEVICE ADDRESS (S) [1]
KEYBOARDS [2]	(2215, 2222)	001, 002, 003, 004
CRT UNITS [2]	(2216)	005, 006, 007, 008
CASSETTE DRIVES	(2217)	10A, 10B, 10C, 10D, 10E, 10F
HIGH-SPEED PRINTERS	(2221, 2231)	215, 216
OUTPUT WRITERS	(2201)	211, 212
PLOTTERS	(2202, 2212)	413, 414
DISK DRIVES	(2230-1, -2, -3)	310, 320, 330
CARD READERS	(2214)	517
HIGH-SPEED PAPER TAPE READER	(2203)	618
TELETYPES AND TELE-COMMUNICATION LINES	(2207) (2227)	01A, 01C, 01E INPUT 01B, 01D, 01F OUTPUT

[1] In some cases more than one device address is listed for each device category. Unless otherwise noted, each peripheral device is assigned a unique address; device addresses are assigned sequentially.

[2] All peripherals in this category are assigned to lowest device address shown. They may, however, be assigned unique addresses by customer request.

HEXADECIMAL CODES FOR 2216 (CRT)

CODE	CHARACTER
HEX (01)	Cursor home
HEX (03)	Clears screen and cursor home
HEX (08)	Backspace
HEX (0A)	Cursor down ↓ (line feed)
HEX (0C)	Cursor up ↑ (reverse index)
HEX (0D)	CR/LF
HEX (20)	Space
HEX (21)	!
HEX (22)	"
HEX (23)	#
HEX (24)	$
HEX (25)	%
HEX (26)	&
HEX (27)	' (apostrophe)
HEX (28)	(
HEX (29)	)
HEX (2A)	*
HEX (2B)	+
HEX (2C)	, (comma)
HEX (2D)	– (minus)
HEX (2E)	.
HEX (2F)	/
HEX (30)	0
HEX (31)	1
HEX (32)	2
HEX (33)	3
HEX (34)	4
HEX (35)	5
HEX (36)	6
HEX (37)	7
HEX (38)	8
HEX (39)	9
HEX (3A)	:
HEX (3B)	;
HEX (3C)	<
HEX (3D)	=
HEX (3E)	>
HEX (3F)	?
HEX (40)	@
HEX (41)	A
HEX (42)	B
HEX (43)	C
HEX (44)	D
HEX (45)	E

CODE	CHARACTER
HEX (46)	F
HEX (47)	G
HEX (48)	H
HEX (49)	I
HEX (4A)	J
HEX (4B)	K
HEX (4C)	L
HEX (4D)	M
HEX (4E)	N
HEX (4F)	O
HEX (50)	P
HEX (51)	Q
HEX (52)	R
HEX (53)	S
HEX (54)	T
HEX (55)	U
HEX (56)	V
HEX (57)	W
HEX (58)	X
HEX (59)	Y
HEX (5A)	Z
HEX (5B)	[
HEX (5C)	\
HEX (5D)	]
HEX (5E)	↑
HEX (5F)	←

INDEX

To help us to provide you with the best manuals possible, please make your comments and suggestions concerning this publication on the form below. Then detach, fold, tape closed and mail to us. All comments and suggestions become the property of Wang Laboratories, Inc. For a reply, be sure to include your name and address. Your cooperation is appreciated.

TITLE OF MANUAL:

COMMENTS:

Fold

Fold

WANG

Fold

FIRST CLASS
PERMIT NO. 16
Tewksbury, Mass.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

– POSTAGE WILL BE PAID BY –

WANG LABORATORIES, INC.
836 NORTH STREET
TEWKSBURY, MASSACHUSETTS 01876

Attention: Marketing Department

Fold

Cut along dotted line.

Printed in U.S.A.

WANG LABORATORIES (CANADA) LTD.
180 Duncan Mill Road
Don Mills, Ontario M3B 1Z6
TELEPHONE (416) 449-7890
TELEX 06-21-7549

WANG EUROPE, S.A.
Buurtweg 13
9412 Ottergem
Belgium
TELEPHONE: 053/74514
TELEX: 26077

WANG ELECTRONICS LTD.
40-44 High Street
Northwood, Middlesex, England
TELEPHONE Northwood 27677

WANG FRANCE SARL
47, Rue de la Chapelle
Paris 18, France
TELEPHONE 203.27.94 or 203.25.94

WANG LABORATORIES GMBH
Moselstrasse No. 4
6000 Frankfurt am Main
West Germany
TELEPHONE (611) 23-00-40

WANG SKANDINAVISKA AB
Fredsgaten 17
S-172-23
Sundbyberg 1, Sweden
TELEPHONE 08-98-12-45

WANG NEDERLAND B.V.
Damstraat 2
Utrecht, Netherlands
TELEPHONE 030-930947

WANG PACIFIC LTD.
61, King Yip Street, 1st Floor
Kwun Tong, Kowloon, Hong Kong
TELEPHONE 3-434231/2

WANG INDUSTRIAL CO., LTD.
110-118 Kuang-Fu N. Rd.
Taipei, Taiwan
Republic of China
TELEPHONE 784181-3

WANG GESELLSCHAFT MBH
Grinzinger Allee 16
1190 Vienna 19
Austria
TELEPHONE (0222) 32.42.43

WANG COMPUTER PTY. LTD.
25 Bridge Street
Pymble, NSW 2073
Australia
TELEPHONE 449-6388

WANG INTERNATIONAL TRADE, INC.
836 North Street
Tewksbury, Massachusetts 01876
TELEPHONE (617) 851-4111
TWX 710-343-6769
TELEX 94-7421

PHI COMPUTER SERVICES
836 North Street
Tewksbury, Massachusetts 01876
TELEPHONE (617) 851-4111
TWX 710-343-6769
TELEX 94-7421

24 Mill Street
Arlington, Massachusetts 02174
TELEPHONE (617) 648-8550

WANG LABORATORIES, INC.
836 NORTH STREET, TEWKSBURY, MASSACHUSETTS 01876, TEL (617) 851-4111, TWX 710 343-6769, TELEX 94-7421

Printed in U.S.A.
700-3059A
7-73-1.5M
Price $10.00