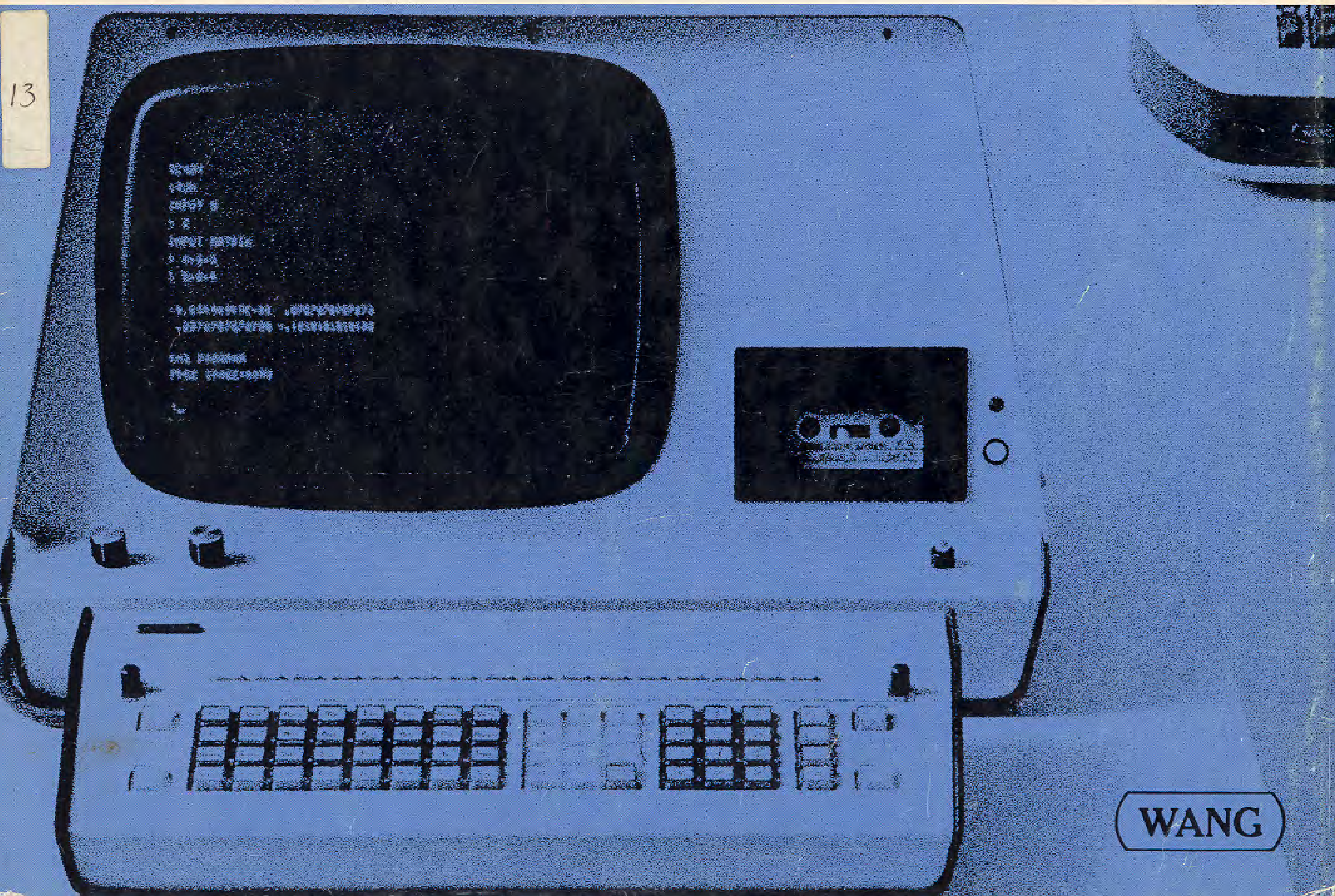
SYSTEM 2200

2200 A/B BASIC PROGRAMMING MANUAL

13



WANG

2200 A/B BASIC Programming Manual

WANG LABORATORIES, INC.
836 NORTH STREET, TEWKSBURY, MASSACHUSETTS 01876, TEL (617) 851-4111, TWX 710 343 6769, TELEX 94 7421

HOW TO USE THIS MANUAL

The System 2200 BASIC Programming Manual is provided with your System 2200 as an instruction aid in the operations of the System 2200. It is designed for the user who is only slightly familiar with BASIC and is not at all familiar with the System 2200.

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

The WANG System 2200 has two operating modes, the Immediate Mode and the Programming Mode. Part I of this manual introduces the System 2200 using the Immediate Mode, or one-line programming. All the operational techniques in the Immediate Mode are discussed; i.e., use of single statement lines and multi-statement lines, how to execute simple and complex calculations, how to format the display, and how to perform 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 System 2200 as a Programmable Calculator. Part II explains the methods for writing programs and the intricacies of the BASIC language hardwired into the System 2200.

Part III discusses additional BASIC language featuers available only with the purchase of a System 2200B. Any of the features mentioned in Part III can be made available by a field upgrade of a System 2200A.

In addition to the System 2200 BASIC Programming Manual, a Reference Manual is provided with your equipment. It is not recommended that the Reference Manual be used for instruction purposes; use it as a quick refresher once you are familiar with the System 2200, or as a means for familiarizing yourself with System 2200 BASIC.

In the Appendices of the manual, additional information is provided. On the last page of this volume is a Customer Comment form to be used by you for comments and suggestions. Please mail the form to our home offices in Tewksbury, Massachusetts. Your comments will be appreciated.

PREFACE

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

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

In keeping with Wang's progressive philosophy, the System 2200A is expandable with the following peripherals: the Model 2215 (or 2222) Keyboard, the Model 2221 (2231) Line Printer, the Model 2261 High Speed Printer, the Model 2201 Output Writer, the Model 2216 CRT Executive Display, the Model 2217 Tape Cassette Drive, and the Model 2227 Telecommunications options and others. The System 2200B, in addition, offers the following peripherals: the Model 2202 Plotting Output Writer, the Model 2203 Punched Paper Tape Reader, the Model 2214 Mark Sense Card Reader, the Model 2230-1 (or 2230-2, 2230-3) Fixed/Removable Disk Drive, the Model 2240 Dual Removable/Flexible Disk Drive, the Model 2243 Triple Removable/Flexible Disk Drive, the Model 2212 Analog Flatbed Plotter, the Model 2207 Telecommunications Interface, the Model 2234 Hopper-Feed Punched Card Reader, the Model 2244 Hopper-Feed Mark Sense/Punched Card Reader, the Model 2232 Digital Flatbed Plotter and others. Additional peripherals for the system are being developed and will be announced shortly.

Several options are being developed for the system. Presently, Option I, the Matrix ROM, Option II, the General I/O ROM, and Option III, Character Editor ROM are available.

Table of Contents

Table of Contents (Continued)

Table of Contents (Continued)

Part I

Using the System 2200 in the Immediate Mode

Part I of this manual covers many of the techniques used with the System 2200 in the Immediate Mode. By Immediate Mode it is meant all results of problems not entered with line numbers are obtained immediately and are not repeatable without reentering the line into the system. All materials presented in Part I are elementary to the System 2200 whether the System 2200 is used in the Immediate Mode or in the Programming Mode (Parts II and III).

Chapter 1

Equipment Installation and Power On Procedure

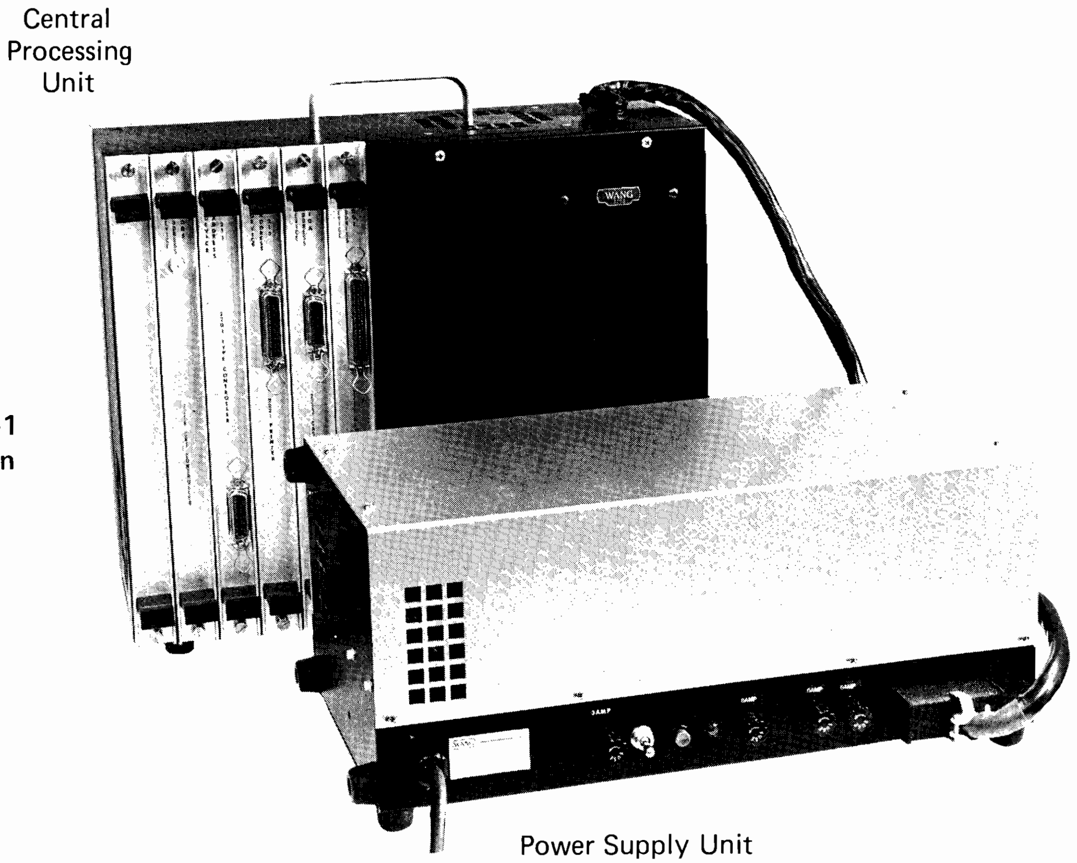
After unpacking and inspecting your equipment, the following procedure is used to install and turn on your WANG System 2200.

Section 1-1 Installation

The basic components of the System 2200 are the Central Processing Unit (CPU) and the Power Supply Unit. All other equipment is considered a peripheral and is 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 Unit.

The Power Supply Unit is pictured to the right.

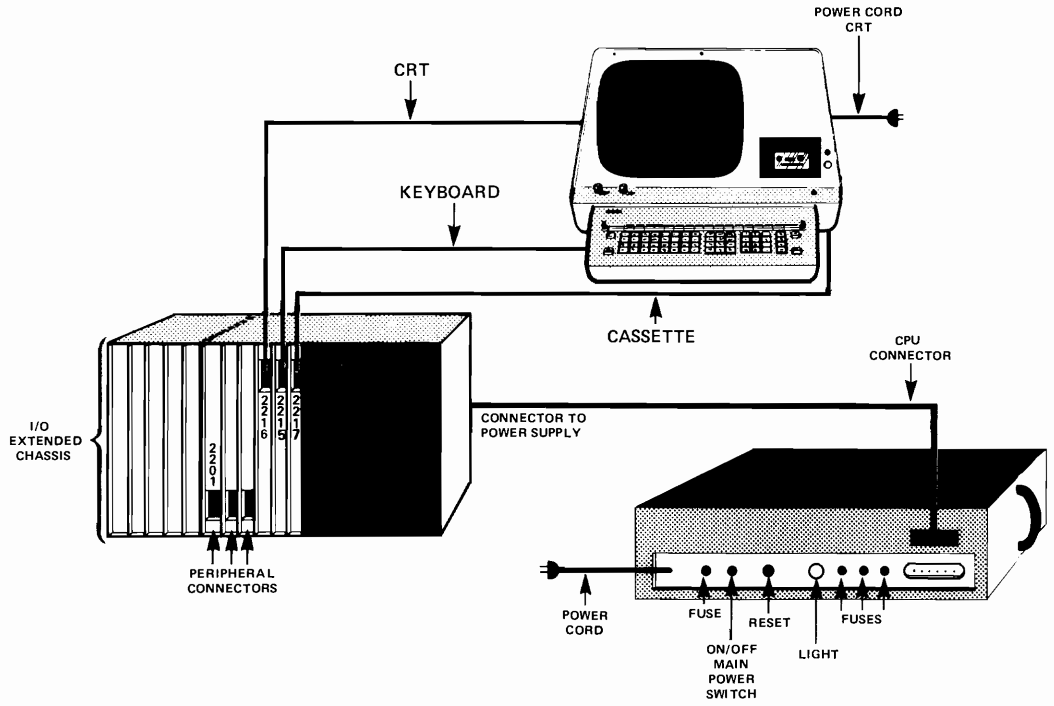


Chapter 1
Equipment Installation and Power On Procedure

INSTALL YOUR SYSTEM as follows:

1. Be sure the ON/OFF switch on the power supply unit is OFF.
2. Plug the main power cord from the Power Supply Unit into a wall outlet.
3. Attach the power cord from the CPU to the Power Supply Unit.
4. Plug any peripherals having a power cord (e.g. CRT) into a wall outlet.
5. Attach all peripherals to the CPU (e.g. Keyboard, CRT, Tape Cassette).

   Be sure the locking clips at the site of attachment are fastened when devices are plugged into the CPU.



TYPICAL INSTALLATION

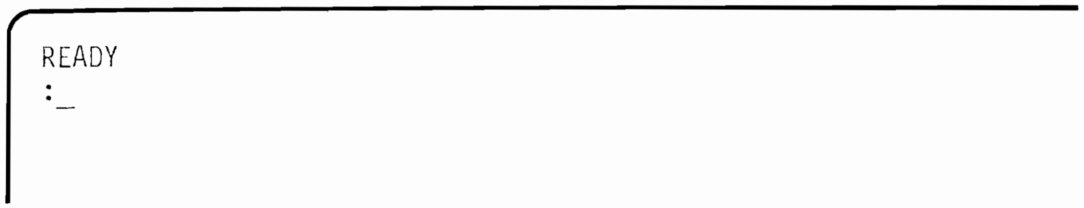
Section 1-2
"Power-On" Procedures

The Power-On procedure is as follows:

1. Turn the ON/OFF switches on all peripherals to the ON position (including CRT).
2. Move the ON/OFF switch on the Power Supply Unit to the ON position.

NOTE:

When the main power ON/OFF switch on the CPU is turned ON the system is automatically initialized, that is the memory is cleared and the display appears as shown to the right. This process is called MASTER INITIALIZATION. The system is then ready to use; if READY does not appear immediately, leave power ON for 15 sec.; turn the switch OFF, then ON again. READY then appears on the CRT screen.



```
READY
:_
```

CRT DISPLAY AFTER MASTER INITIALIZATION

Chapter 2

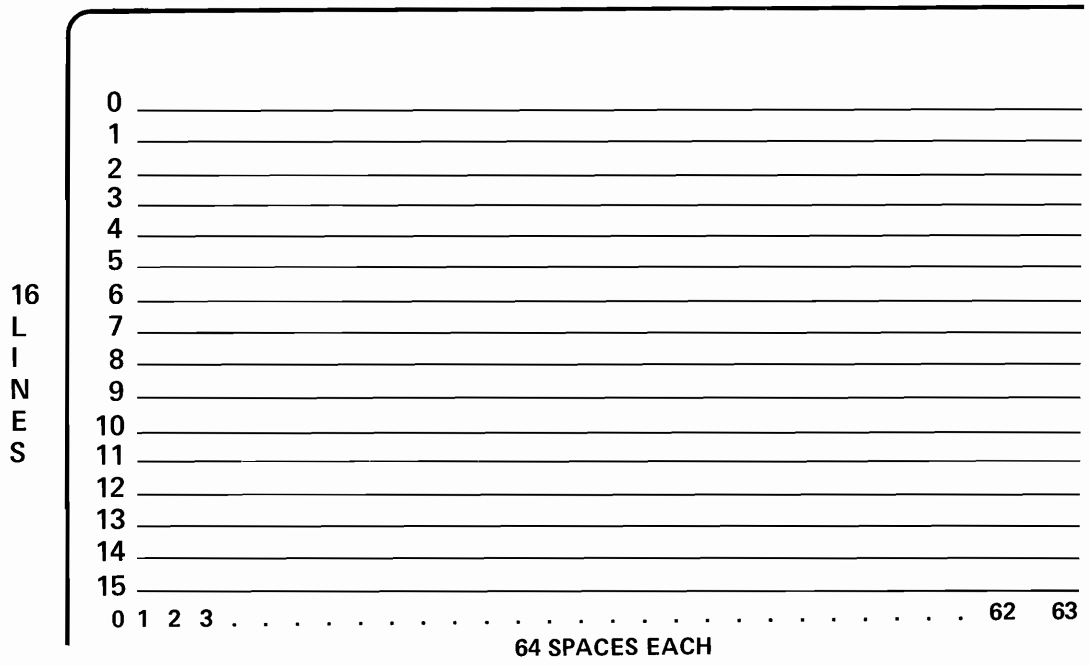
An Introduction - CRT Display and BASIC Keyboard

The following sections describe the procedures for using the CRT display and Model 2215 keyboard to their best advantage for solving problems.

Section 2-1
The System 2200 CRT Display

The CRT display enables the user to more easily write programs and review results. The CRT unit is composed of an 8 x 10-1/2 inch screen, and two controls used to set the brightness and contrast of the output as it appears on the screen. The screen has a maximum capacity of 16 lines, each 64 characters in length. 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.

CRT DISPLAY CAPACITY



THE RESET KEY

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

Touch the RESET key.

After RESET is touched, the display appears as shown to the right.

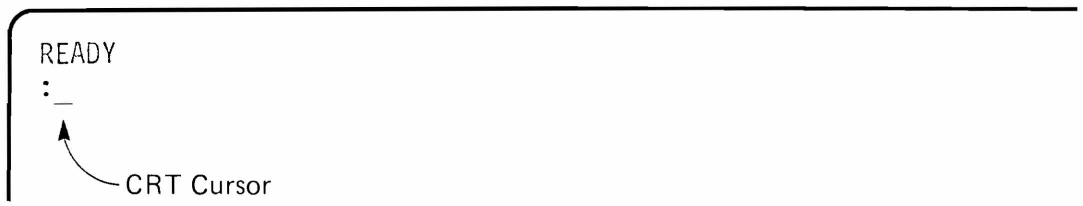
The combination of READY and the COLON tells the operator that no processing is taking place within the System 2200, and that the System 2200 is now READY to accept new information. *The colon must appear on the screen before any information can be entered into the system.*

Next to the colon is a short "hash" mark referred to as the *CRT Cursor.* This mark denotes the location where the next character will be displayed on the screen.

Touching the RESET effects the system in three ways.

1. Clears the CRT display, and prints READY and a colon in the display.
2. Terminates any processing taking place in the System 2200.
3. In terminating any processing, the RESET command unlocks the 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. It is used primarily to clear the screen.



```
READY
:_
```

CRT DISPLAY AFTER RESET

Section 2-2
The Model 2215 BASIC Keyboard

There are 5 zones on the 2215 BASIC keyboard as shown in the photo to the right.

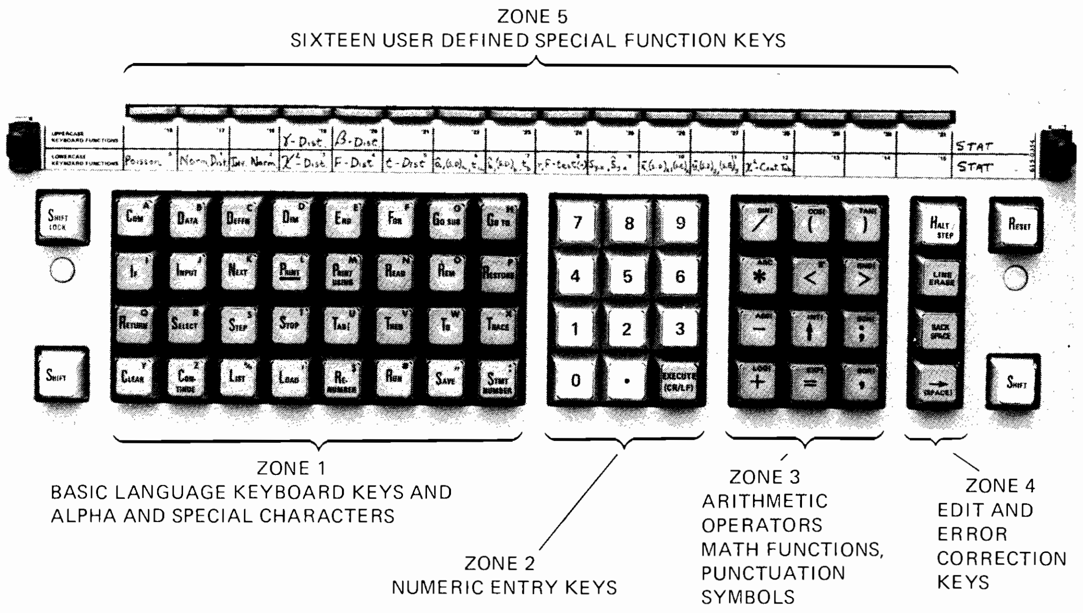
NOTE:

The basic content of this chapter also applies to Systems using the 2222 Alphanumeric Keyboard. However, the 2222 Keyboard is designed differently than the pictured 2215 Keyboard; therefore, the zone descriptions and keystroke instructions differ.

Zone 1 of the 2222 is similar to a typewriter keyboard and all words (except PRINT) must be entered with individual keystrokes. Since the Manual text is geared to Model 2215 users, when the text instructs "Touch NEXT key", the Model 2222 operator must type the individual letters, "N-E-X-T".

Zones 2 and 3 are very similar; zone 4 does not exist on the 2222 Keyboard. The space bar key, backspace key, and here erase key are all in different locations. (See Section I, 2200 Reference Manual, for a complete discussion of the Model 2222 Keyboard.)

Operators of both 2215 and 2222 Keyboards should utilize the problems presented in this manual. Efficient operation of the System 2200 depends on keyboard proficiency.


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

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. To generate a single letter (e.g. A, or B) on the System 2200, you must SHIFT before touching the appropriate key, as these letter characters are upper case. To generate most BASIC keywords (e.g., PRINT, END) you must be in lower case.

THE SYSTEM 2200 BASIC KEYWORDS

Zone 1 of the keyboard consists of a four row block of green keys on the left-hand side of the keyboard. This group of keys is used to generate alpha characters (upper case) and BASIC keywords (lower case) with a single keystroke.

Touch the PRINT key.

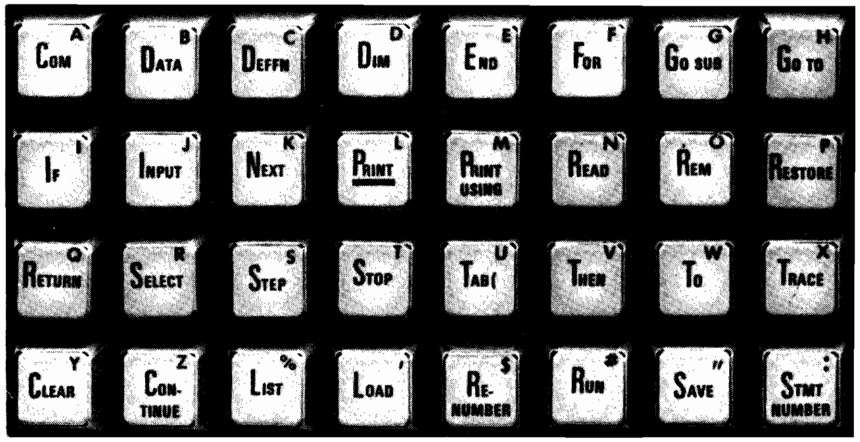
Notice the entire word PRINT plus a following space was generated on the screen with a single keystroke. Most BASIC words can be generated on the screen with a single keystroke (see keyboard).

Touch the SHIFT LOCK key.

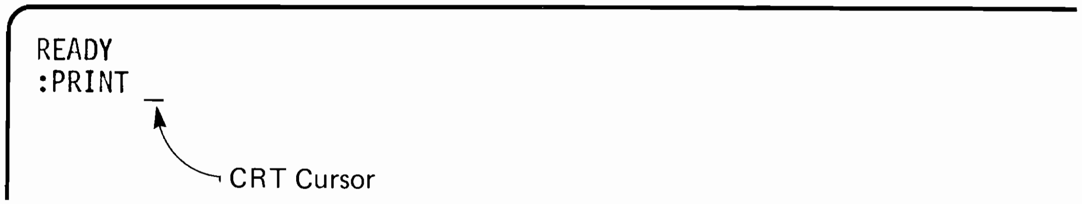
This locks the keyboard in upper case. Notice the light under the SHIFT LOCK key is ON. When this light is ON, it indicates that the keyboard is in upper case.

Touch keys "A N S W E R"

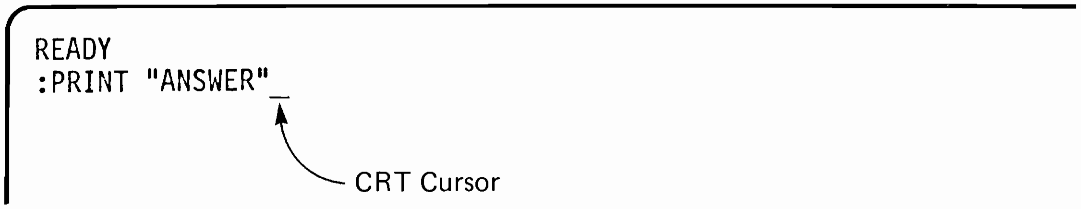
Touch the SHIFT key.



Zone 1



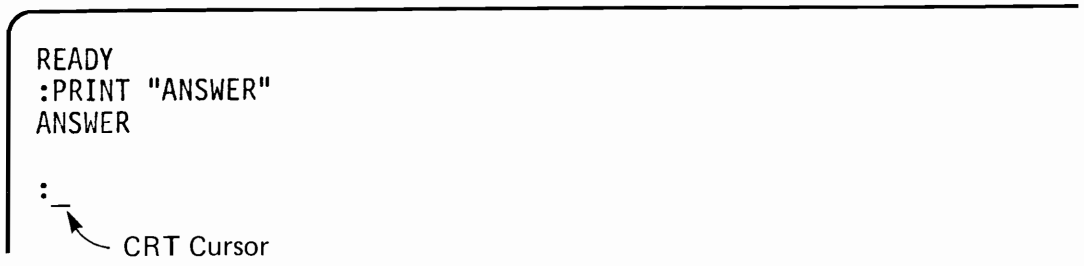
```
READY
:PRINT _
```



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

This unlocks the keyboard and returns the keyboard to lower case. For single keystrokes in upper case touch the SHIFT key once before the appropriate key. The SHIFT key is turned off automatically when the appropriate key is touched.

Touch the CR/LF-EXECUTE[1] key



```
READY
:PRINT "ANSWER"
ANSWER

:_
```

RESULTS DISPLAYED WHEN CR/LF—EXECUTE TOUCHED

When the CR/LF - EXECUTE key is touched, the line is checked for errors. If correct, the line is immediately executed, the results displayed, and is completely cleared.

In order to have the results of a line printed on the display, the keyword PRINT always must be used. In the last example the word "ANSWER" is to be printed. In order to do this the word PRINT must precede the word(s) to be printed.

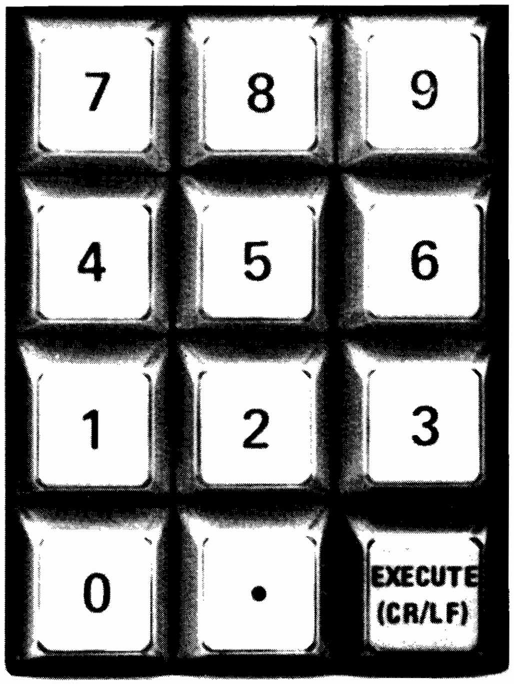
THE NUMERIC KEYBOARD ZONE

The block of white keys in the center of the keyboard (Zone 2) is used to enter all numeric data from the keyboard.

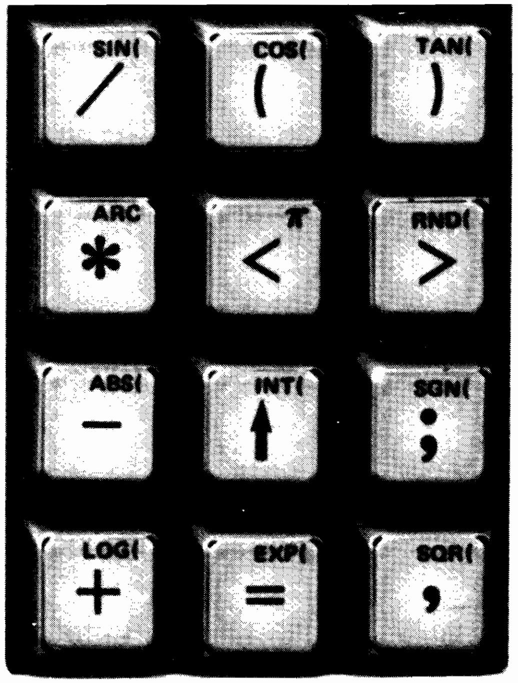
THE MATH KEYBOARD AND MATH FUNCTIONS ZONE

The block of green keys (Zone 3) located to the right of the numeric keys incudes the plus (+), minus (–), multiply (*), divide (/), and power (↑) operations, left and right parentheses, and math functions.

The following problem is solved using both the numeric keys and math function keys.



Zone 2

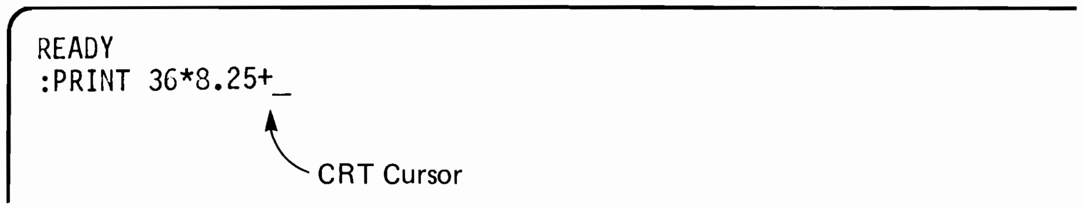


Zone 3

[1]CR/LF means carriage return/line feed. The RETURN key is used on the Model 2222 keyboard.

Find 36 X 8.25 + TAN (35) = ?

Touch Keys PRINT 3 6 * 8 . 2 5 +

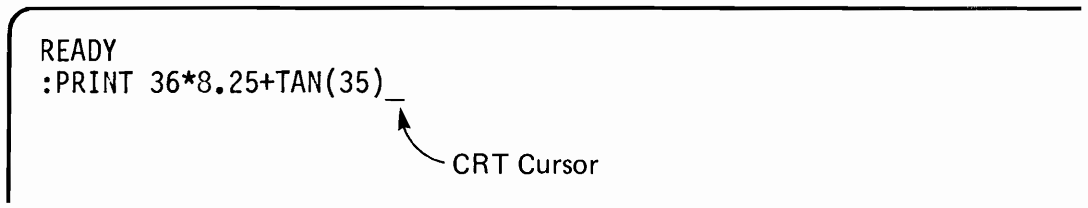


```
READY
:PRINT 36*8.25+_
```

Notice the PRINT key is used first. This must be done in order to see the results of the calculation printed.

The math functions are generated when the keyboard is in upper case.

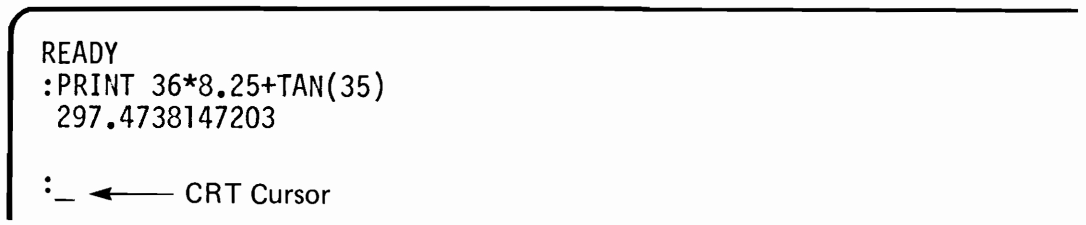
Touch Keys SHIFT TAN (35)



```
READY
:PRINT 36*8.25+TAN(35)_
```

Notice when you touched the TAN(key the left hand parenthesis was automatically generated. You must generate the right hand parenthesis with the right hand parenthesis key. This is true for most of the functions.

Touch the CR/LF-EXECUTE key.



```
READY
:PRINT 36*8.25+TAN(35)
 297.4738147203

:_
```

Once the line is executed the answer appears and the cursor on the CRT moves to the first space of the next line. The new colon (:) did not appear until processing had stopped and all output was displayed. The processing light,which is on during all processing, goes out. Also while any processing is taking place the keyboard is locked. Only the RESET key can stop processing and unlock the keyboard.

REVIEW

1. The keyboard like any typewriter keyboard has upper and lower case keys. Most keys on the Model 2215 keyboard can generate two different inputs, depending on whether used in upper or lower case.
2. There are 4 zones on the Model 2215 keyboard (a 5th zone, Special Function Keys, is discussed in depth later in this text).
3. The keyword PRINT must precede any calculation whose results are to be viewed on the CRT.
4. The CR/LF - EXECUTE key is used to initiate all calculations.

THE CRT CONTROL KEYS & EDIT KEYS

Zone 4 is located at the far right of the keyboard. This zone is made up of CRT control keys and edit keys.

These keys enable the user to edit any information before the CR/LF - EXECUTE key is touched.

Touch RESET

Touch the SPACE key several times.
→

The CRT cursor moves one space to the right each time the space key is touched, enabling the user to enter spaces in System 2200 BASIC statement lines. Touching this key at the end of a line causes the cursor to jump to the first space of the next line.

←
Touch the BACK SPACE key several times.

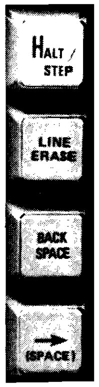
The CRT cursor moves one space to the left each time the Back Space key is touched.

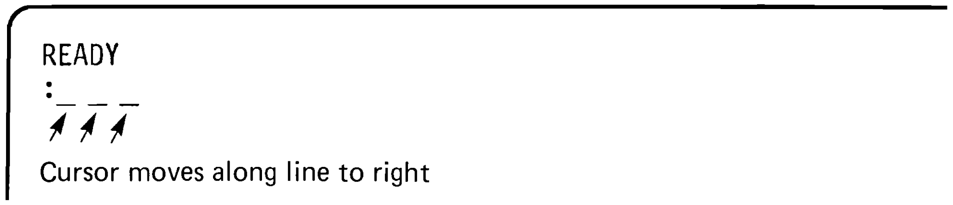
Touch keys PRINT SHIFT A

←
Touch the BACK SPACE key.

This deletes only the character "A" from the CRT display.

Zone 4





```
READY
:_ _ _
```

Cursor moves along line to right

```
READY
:_ _ _
```

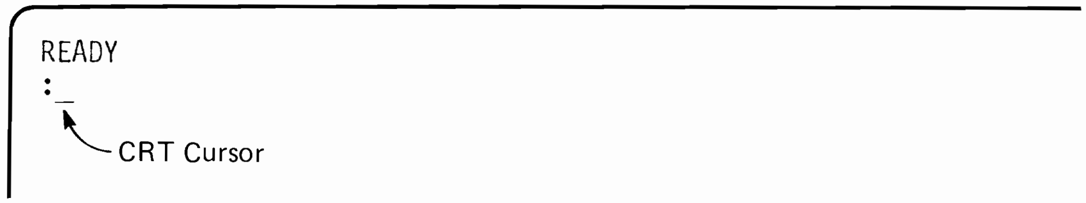
Cursor moves along line to left

```
READY
:PRINT A_
```

```
READY
:PRINT _
```

Touching the Back Space 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 $\overset{\leftarrow}{\text{BACK SPACE}}$ key again.



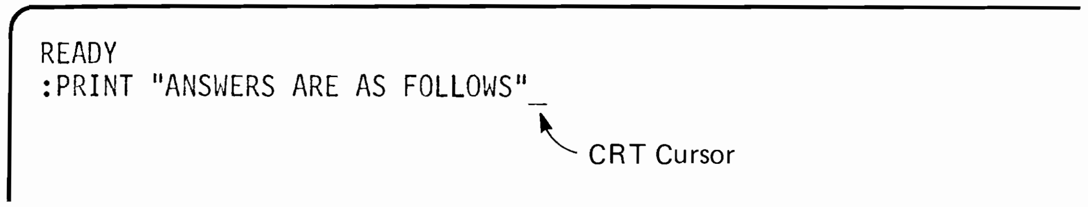
```
READY
:_
```

This time, the entire keyword PRINT is deleted because the word was entered by a single keystroke.

Backspacing to delete with the $\overset{\leftarrow}{\text{BACK SPACE}}$ 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 a single character and is deleted by touching the $\overset{\leftarrow}{\text{BACK SPACE}}$ 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 $\overset{\leftarrow}{\text{BACK SPACE}}$ key is touched only one character at a time is erased.

Touch keys PRINT "ANSWERS ARE AS FOLLOWS"

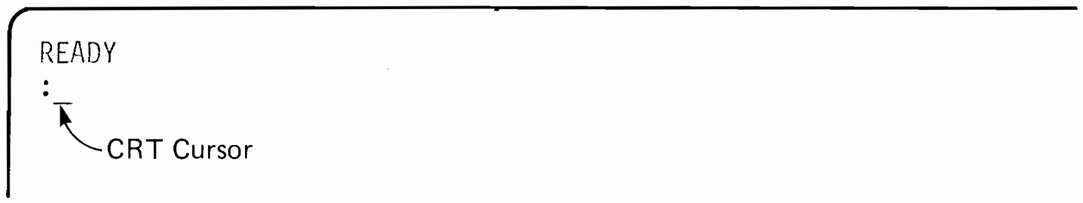


```
READY
:PRINT "ANSWERS ARE AS FOLLOWS"_
```

Touch the LINE ERASE key

Touching the LINE ERASE key erases the entire line at which the cursor is located.

Both the ← BACK SPACE key and the LINE ERASE key work as described in both upper and lower case.



```
READY
:_
```

EXERCISES

1. Using the System 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 System 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	ANSWERS
PRINT 14+6 CR/LF—EXECUTE	20
PRINT 8*6 CR/LF—EXECUTE	48
PRINT 8–12.66 CR/LF—EXECUTE	–4.66
PRINT –18*4.55 CR/LF—EXECUTE	–81.9
PRINT 96/853 CR/LF—EXECUTE	.1125439624853
PRINT 5↑3 CR/LF—EXECUTE	125
PRINT 26+5+12+10 CR/LF—EXECUTE	53
PRINT 7–12.6+8–.002 CR/LF—EXECUTE	2.398
PRINT 8*10+6*4 CR/LF—EXECUTE	104
PRINT 144↑.5 CR/LF—EXECUTE	12
PRINT 86.2+155.86 CR/LF—EXECUTE	242.06
PRINT 8522–1498 CR/LF—EXECUTE	7024
PRINT –57*16.6 CR/LF—EXECUTE	–946.2
PRINT 20/4.25 CR/LF—EXECUTE	4.70588235294
PRINT 17.3↑1.6 CR/LF—EXECUTE	95.691588717
PRINT 1+2+3+4+5+6+7+8+9+10 CR/LF—EXECUTE	55
PRINT 1*2*3*4*5*6*7*8*9*10*11*12 CR/LF—EXECUTE	4790016000
PRINT 365*24*60*60 CR/LF—EXECUTE	31536000

Chapter 3 Calculator Facts

This chapter introduces several important concepts about the System 2200 as a calculator.

The material in Section 3-1 explains the use of parentheses and the order of execution of algebraic expressions. The System 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 chapter demonstrated the use of the basic five arithmetic operations: addition, subtraction, multiplication, division, and raising to a power. In those exercises, for the most part, there were no questions 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 at the right provides the answer.

ORDER OF EXECUTION

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 of Y; second, the result is then multiplied by W; finally, Z is subtracted from the product.

Suppose this is not the intended order. In this case, parentheses must be used to indicate the intended order of execution. Thus, if the product of W * X is to be raised to the power of Y, and Z subtracted, the expression would be written as:

(W * X) ↑ Y – Z.

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

W * X ↑ (Y – Z).

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

W * (X ↑ Y – Z).

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 of the expression is altered. See the table to the right.

ORDER OF EXECUTION

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. To calculate 5^{-3} :

Touch RESET

Touch keys PRINT 5↑–3 CR/LF-EXECUTE

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

ERR 15 tells you that two mathematical operator symbols cannot appear next to each other, they must be separated using parentheses.

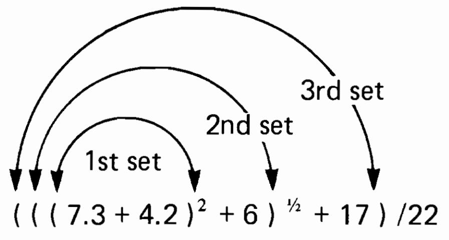
Touch RESET

Touch keys PRINT 5↑ (–3)

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

:_
```

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



An unlimited amount of nesting of parentheses is allowed on the System 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.

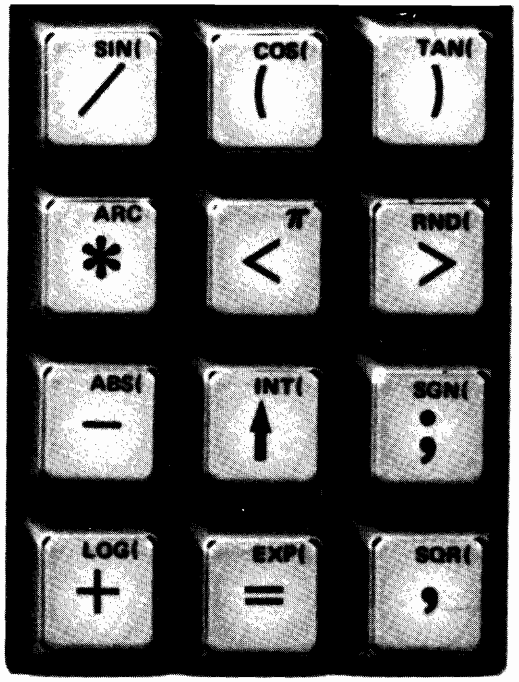
However, in using parentheses, there are two rules which must be followed:

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

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
Using the Keyboard Functions

Zone 3, already mentioned, contains 12 commonly used math functions (all upper case). All except the π and ARC functions have a left hand parenthesis included and require you to key in the right hand parenthesis. To generate an arcsine, arccosine, or arctangent function, the ARC key is touched prior to the appropriate trigonometric function key. The table on the next page gives a complete list of these keyboard functions.



Zone 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 arcsine 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
π (Appears as #PI on CRT display)	Assign the value (3.14159265359)	4*#PI=12.56637061436
RND(expression)	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 not greater than 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(3053)=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 System 2200 to radian measure, touch SELECT R CR/LF–EXECUTE, or switch the System 2200 OFF and ON.

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

Try some of the following examples which illustrate the use of these keyboard functions.

1. Find the $\sqrt{114.6/53.47}$

   Touch keys PRINT SHIFT SQR(114.6/53.47) CR/LF-EXECUTE

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

:_
```

2. Find $\log_e 10$

   Touch keys PRINT SHIFT LOG(10) CR/LF-EXECUTE

```
READY
:PRINT LOG(10)
 2.302585092994

:_
```

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

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

:_
```

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

To reset the System 2200 to radian measure from either degrees or gradians, enter SELECT R CR/LF-EXECUTE, or switch the System 2200 to OFF then ON to reinitialize.

4. Find the COSINE 48°

   Touch keys SELECT SHIFT D CR/LF-EXECUTE PRINT SHIFT COS(48) CR/LF-EXECUTE

   Return the System 2200 to radian mode by touching keys

   SELECT SHIFT R CR/LF-EXECUTE

```
READY
:SELECT D

:PRINT COS(48)
 .6691306063585

:_
```

5. Find the absolute value of the expression

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

   Touch keys PRINT SHIFT ABS ((1.68↑2–46)/ 28.5) CR/LF-EXECUTE

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

:_
```

Section 3-3
Floating Point Numbers

Up to this point, entering numbers into the System 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 –.0095 requires the keystrokes – . 0 0 9 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 a maximum of thirteen digits, plus a decimal point and a sign, can be entered into the 2200. When entering a number greater than zero, a plus sign is implied, and need not be entered.

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 are 13 digits limiting in size 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*, can be used with System 2200 BASIC. When floating point format 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 format are;

6.02×10^{24} 5.1×10^{-5}

195×10^{18} $.016 \times 10^{5}$

Notice that in floating point format 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., 5.6E3 as opposed to 56.4E2 or .0564E5) it is said to be in *Scientific Notation*.

Using floating point format in the System 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 END^{E} keys are used.

PRINT the number 5.675 X 10^4

Touch keys PRINT 5.675E4 CR/LF-EXECUTE

```
READY
:PRINT 5.675E4
 56750

:_
```

PRINT the number 15.9 X 10^{-8}

Touch keys PRINT 15.9E–8 CR/LF-EXECUTE

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

:_
```

Other examples of the correct use of floating point format are shown to the right.

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 displays only the first nine digits in scientific notation, though the remaining digits are kept internally.

The largest exponent which System 2200 BASIC can 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 shown to the right.

CORRECT USE OF FLOATING POINT NOTATION

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

INVALID USE OF FLOATING POINT 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.

Section 3-4
Error Detectors

Basic rules must be followed by the user when entering numbers and formats of numbers into the calculator, as well as rules for the use of the BASIC language. The System 2200 is designed to make it easier for the user to detect when a number or format, etc., has been entered incorrectly. The system automatically tells you by displaying an error message on the screen at the approximate location in a line where the error is found.

Enter the following line.

Touch keys PRINT 3 * SQR(17 CR/LF-EXECUTE

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

ERR 05 means missing right parenthesis. The solution is to re-enter the problem and add the right hand parenthesis.

Appendix C contains a list of all the Error messages and their meanings.

Enter the following line.

Touch keys PRINT PRINT 3↑6.2 CR/LF-EXECUTE

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

ERR 15 means the system expected to find an expression following PRINT. There are two PRINT statements in the line. Re-enter the line with only one PRINT statement.

There are 94 error detectors to help you detect most problems in entering a calculation.

Section 3-5
Using Variables

WHAT IS A VARIABLE?

The use of variables[1] is a mathematical shorthand which allows you to assign a numeric value to a letter (variable) and use this letter in several different expressions where the variable has the same value in each expression. It is then an easy matter to change the value of the variable and recalculate each expression for different values. In the System 2200 there are 286 different variable names available. The names consist of a single letter (A-Z) or a letter and a digit (0-9). These variables are called numeric scalar variables.

ASSIGNING VARIABLES VALUES

When a numeric variable is given a value, the process is called assigning it a value. A numeric variable can have only one value at a time. The format for assigning variables is shown to the right. The variable is always on the left hand side of the equality sign and the expression or value assigned to the variable is always on the right hand side. The equality sign always must be used.

[1]There are several different types of variables available in the System 2200. The simple numeric scalar variable is discussed in this chapter. The other types are discussed in Part II.

LEGAL VARIABLE NAMES

$$\underbrace{\begin{array}{ccc}\text{Single Letters} & \qquad & \text{Letter and a Digit} \\ \text{A} - \text{Z} & & \text{A0} - \text{Z9}\end{array}}_{\text{256 Variable Names}}$$

ASSIGNING VARIABLES VALUES

X = 25

A variable can be assigned a simple numeric value or an expression. Some examples of variables are given to the right.

In the System 2200 a variable is assumed to have a value of zero, until assigned a value.

EXAMPLES OF VARIABLES

X = 5	The variable X is assigned the value of 5.
F = 4/3*#PI*7↑2	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.

Touch RESET

Touch keys PRINT X CR/LF-EXECUTE

```
READY
:PRINT X
 0

:_
```

Notice the value zero is printed for X. No value for X has been assigned. An undefined variable always has the value zero.

Touch RESET

Touch keys X = 5 CR/LF-EXECUTE

Now Touch keys PRINT X CR/LF-EXECUTE

```
READY
:X=5

:PRINT X
 5

:_
```

Notice now the value for X is printed. The PRINT statement had to be used in order to have the value for X printed.

Once a variable is assigned a value it keeps that value until it is assigned a new value, or until memory is cleared.

Touch RESET

Touch keys PRINT 5 * X + 3 CR/LF-EXECUTE

X retained the value of 5.

```
READY
:PRINT 5*X+3
 28

:_
```

Touch keys CLEAR CR/LF-EXECUTE

CLEAR CR/LF-EXECUTE clears *the display and also clears the entire memory.*

```
READY
:_
```

Touch keys PRINT X CR/LF-EXECUTE

X is now undefined and therefore has the default value of zero.

```
READY
:PRINT X
 0

:_
```

ASSIGNING A SINGLE VALUE TO MORE THAN ONE VARIABLE

A single value can be assigned to more than one variable in a single statement.

Touch RESET

Touch keys X, Y, Z, = 5 CR/LF-EXECUTE

```
READY
:X,Y,Z=5

:_
```

The three variables X, Y and Z now have the value of 5. The variables must be *separated by commas* generated by the lower case SQR(, key.

Touch keys PRINT X CR/LF-EXECUTE

Touch keys PRINT Y CR/LF-EXECUTE

Touch keys PRINT Z CR/LF-EXECUTE

```
READY
:X,Y,Z=5

:PRINT X
 5

:PRINT Y
 5

:PRINT Z
 5

:_
```

NOTE:

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

X = 5 + 9
could also be written

LET X = 5 + 19
with the SAME RESULTS. However, the verb LET is OPTIONAL. This keyword is not included in the 2215 BASIC keyword block. To generate the verb LET, key SHIFT LOCK L E T SHIFT.

Some illegal variable names are shown to the right.

Two letters or a digit and a letter (in that order) or more than two characters are illegal for variable names.

ILLEGAL VARIABLE NAMES

XX WL 5M C75

EXERCISES

Evaluate the following expressions on the System 2200.

Use the keyboard functions wherever possible.

		KEYSTROKES	ANSWERS
i	575^2	PRINT 575↑2 CR/LF-EXECUTE	330625
ii	$.00575^3$	PRINT .00575↑3 CR/LF-EXECUTE	1.90109375E–07
iii	$(-35)^2$	PRINT (–35)↑2 CR/LF-EXECUTE	1225
iv	$\frac{1}{35}^{4.2}$	PRINT (1/35)↑4.2 CR/LF-EXECUTE	3.27276041E–07
v	$3^3 \cdot 5^5$	PRINT 3↑3*5↑5	84375
vi	$7(8^3 + 5.9^2)$	PRINT 7*(8↑3 + 5*9↑2) CR/LF-EXECUTE	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 CR/LF-EXECUTE	4.999999999999
viii	$\frac{5 \cdot 6^2}{5 + 6^2}$	PRINT 5*6↑2/(5+6↑2) CR/LF-EXECUTE	4.390243902439
ix	$(2^5)^4$	PRINT 2↑5↑4 CR/LF-EXECUTE	1048576
x	$\sqrt{36}$	PRINT SQR(36) CR/LF-EXECUTE	6
xi	$\sqrt{\frac{1}{36}}$	PRINT SQR(1/36) CR/LF-EXECUTE	.1666666667
xii	$\sqrt{5 + \sqrt{8 * 9^2}}$	PRINT SQR(5 + SQR(8*9↑2)) CR/LF-EXECUTE	5.5186813754
xiii	Tangent $\pi/4$ radians	PRINT TAN #PI/4 CR/LF-EXECUTE	1
xiv	Arc Sine .5	PRINT ARC SIN (.5)	.5235987755982
xv	Log of $\sqrt{19.5}$	PRINT LOG(SQR(19.5)) CR/LF-EXECUTE	1.485207232792
xvi	Integer $3.8^2 - \frac{2}{.3^2}^{\,2}$	PRINT INT(3.8↑2– (2/.3↑2)↑2) CR/LF-EXECUTE	–480
xvii	Absolute value of $(18.2 - 16^2)$	ABS(18.2 – 16↑2)	237.8
xviii	Sign of the expression $\left(\frac{\sqrt{14.5 - 6}}{15.8 \cdot \text{cosine } 22 \text{ radians}}\right)$	SGN((SQR14.5–6)/(15.8*COS(22))) CR/LF-EXECUTE	–1

ixx Which of the following are *not* valid System 2200 BASIC scalar variables?

X, K2, B, Y, M12, PQ, ZZ, 2K

xx Which of the following *are* valid floating point numbers?

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

xxi 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. $Y = \sin \frac{X}{2}$

b. $A = \cos \frac{X}{4} \cdot \arctan (4X^{-3})$

ixx M12, PQ, ZZ, 2K

xx b, c, d, e

xxi

```
a. Y = SIN(X/2)
   PRINT Y
b. A = COS(X/4)*ATN(4*X↑(−3) )
   PRINT A
```

Chapter 4
Performing More Than One Calculation Per Line

In the previous chapter the examples used illustrated how to enter and execute calculations on the System 2200. Each of these calculations is called a BASIC STATEMENT.

However, only one statement was entered per line or per statement line. It is possible to enter and execute more than one statement per statement line. In doing this you can take advantage of the size of the CRT (i.e., 64 characters per line). The concept of a statement line is important as it is used frequently in the remainder of this text.

Notice each statement in the statement line is separated by a colon (:). The colon is generated with

```
               :
SHIFT and    STMT      Keys.
           NUMBER
```

The colon denotes the beginning of a new statement.

Enter the following statement line.

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

The results of each statement are obtained and printed, each on a separate line. A statement line can contain any number of statements. The statement line is only limited by the actual number of keystrokes allowed in a BASIC statement line. The maximum length is 192 keystrokes.

A BASIC STATEMENT = PRINT 527↑2

DEFINITION - A BASIC STATEMENT LINE

A BASIC statement line is a line composed of one or more statements each separated by a colon.

EXAMPLES:

```
PRINT 15:PRINT SQR(15):PRINT 15↑(1/3)
X = 2: PRINT X↑2: PRINT X
X = 2: Y = 3: PRINT X+Y: PRINT X↑2: Print Y↑2
```

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

:_
```

Chapter 4
Performing More Than One Calculation Per Line

EXERCISES

1. Write a single statement line to calculate and printout the following:

   5^2, 5^3, 5^4, 5^5, and 5^6.

2. Write a single statement line to calculate and printout the following:

   183.5 + 23.6, $(183.5 + 23.6)^2$ and $\sqrt{183.5 + 23.6}$.

3. Write a single statement line to calculate and printout the following:

   $3^3 \times 5^5$, Log of $\sqrt{19.5}$, ABS$(18.2-16^2)$.

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

:_
```

```
READY
:PRINT 183.5+23.6:PRINT (183.5+23.6)↑2:PRINT SQR(183.5+23.6)
 207.1
 42890.41
 14.390969391

:_
```

```
READY
:PRINT 3↑3*5↑5:PRINT LOG(SQR(19.5)):PRINT ABS(18.2-16↑2)
 84375
 1.485207232792
 237.8

:_
```

Chapter 5

Printing Out More Than One Value Per Line

Up to this point, every PRINT statement executed has resulted in the printing out of each result 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, System 2200 BASIC is capable of formatting output in several other ways. These include *zoned format*, *packed format*, *and tab format.*

Section 5–1
What is Zoned Format?

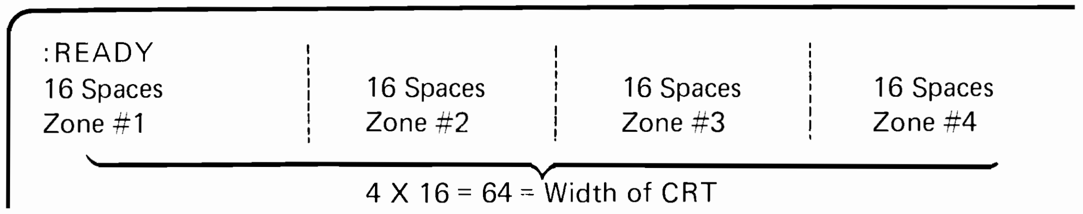
The CRT display is divided into four 16-space fields or zones.

Previously, all output has been printed one value to a line, generally in the first zone only.

To generate more than one output value per line, with each value in a separate zone, values are included in a single PRINT statement with COMMAS separating the values. This is known as ZONED format.

Touch keys PRINT 1,2,3,4 CR/LF-EXECUTE

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



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

:_
```

Chapter 5
Printing Out More Than One Value Per Line

Key in the following line which both generates four answers and prints out all the answers on a single line in zoned format.

Touch keys PRINT 5↑2, 5↑3, 5↑4, 5↑5, CR/LF-EXECUTE

```
READY
:PRINT 5↑2,5↑3,5↑4,5↑5       |                |
 25              |125         |625             |3125
                 |            |                |
:_               |            |                |
        Zone 1   |    Zone 2  |     Zone 3     |      Zone 4
```

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.[1]

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

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

:_
```

In the printing of literal strings in a zoned format, no space is left preceding the string, as left for an implied plus sign with numbers.

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.*

Touch keys 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      |     Zone 4
```

[1] Alphanumeric characters enclosed in quotes are usually referred to as literal strings.

Because the literal string[1] "THE PROBABILITY IS" is 18 characters in length, and extends into the second zone, the next value to be printed is printed in zone #3.

If more than four ouput 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.

Touch keys 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

:_
```

Similarly, multiple and/or leading commas can 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 shown to the right. (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

:_
```

[1] 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 space key.

Touch keys PRINT 5, :PRINT 10 CR/LF-EXECUTE

Touch keys PRINT 5 :PRINT 10 CR/LF-EXECUTE

```
READY
:PRINT 5,:PRINT 10
 5                10
```

```
:PRINT 5:PRINT 10
 5
 10

:_
```

Compare these two printouts. The difference in the output is caused by the comma following the first PRINT statement in first example, but which is not included in the second example. 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 at the end of the PRINT statement, a subsequent PRINT statement prints at the next new line.

Section 5–2
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* enables the user to print more than 4 values per line.

To generate packed format, SEMICOLONS are used between each of the values, instead of commas, in the PRINT statement.

Touch keys PRINT 1;2;3;4;5;6;7;8;9;10 CR/LF-EXECUTE

Touch keys PRINT –1;–2;–3;–4;–5;–6;–7;–8;–9;–10; CR/LF-EXECUTE

```
READY
: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

:_
```

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) and one blank space following the numeric when printing out numeric values.

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 variables themselves. However, the maximum number of characters that can be printed on the CRT on any one line is 64. Any output exceeding this is continued in the first space of the following line.

Touch keys PRINT "SQUARE ROOT="; SQR(729) CR/LF-EXECUTE

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

:_
```

Notice 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 left after the printout of a literal string as with a number. *The only spaces printed out with literal strings are those included within the quotation marks.*

Section 5–3
Mixing Zoned and Packed Format

Zoned and packed formats can be used together to achieve a wider range of format control.

Touch keys PRINT "VALUE="; 50, , "NEW VALUE="; 50↑2 CR/LF-EXECUTE

```
READY
:PRINT "VALUE=";50,,"NEW VALUE=";50↑2
VALUE= 50                               NEW VALUE= 2500

:_
```

Notice the use of the two commas caused the "NEW VALUE" literal string to be printed out starting in zone #3; the use of the semicolons caused the numbers to be printed in the same zone as the literal string.

Section 5–4
Using the TAB(Function for Format Control

In addition to using zoned and packed format in System 2200 BASIC, another formatting tool is available which is analogous to the "tab stop" found on a typewriter. This formatting tool is the TAB(function found in the keyword block on the left-hand side of the 2215 BASIC Keyboard.

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

Touch keys PRINT TAB(10); 25 CR/LF-EXECUTE

```
READY
:PRINT TAB(10);25
           25

:_
```

The System 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(10) actually spaces 11 spaces to column 10, because the 1st column is numbered column #0.

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

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

:_
```

In this case, the System 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(function 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(function 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 System 2200 is past the requested tab location at the time the TAB(function is encountered, the location of the CRT cursor does not change at all, and the TAB(function is ignored.

Touch keys PRINT TAB(20);20; TAB(10); 10 CR/LF-EXECUTE

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

:_
```

In this example, the System 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 continues to the next part of the PRINT statement, another TAB(function. However, the second TAB(function says to space to column 10. Since the CRT cursor is already past column 10, this TAB(function is ignored, and the System 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.

Touch keys PRINT TAB(10); 10; TAB(20):
20 CR/LF-EXECUTE

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

:_
```

NOTE:

When figuring the number to use within the parentheses of a TAB(function, remember that the number of spaces actually skipped is equal to the number in the TAB(function. As a result a TAB(11) function skips 11 spaces and prints (if something is to be printed) the output in column 11 which is actually at the 12th space on the CRT line. 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. Column 20 contains the blank for the plus sign.

EXERCISES

I Using Commas and Semicolons

1) The System 2200 BASIC Statement

   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 Exercises (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
```

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: PRINT,,B: PRINT,,C CR/LF-EXECUTE

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 System 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.

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 6

Executing a Line More Than Once (Looping)

In Chapter 5, you practiced entering several different calculations per line (a multi-statement line). However you are still unable to repeat the calculations of that line more than once. To obtain a repeat of a calculation means re-entering the line and executing it again. When using the System 2200 as a calculator, there is another way of repeating a line more than once without having to rekey the line each time. Repeating a calculation more than once is called looping. In System 2200 BASIC, the statements which allow you to loop are the FOR-TO and NEXT statements.

Section 6-1
Looping With The FOR-TO/NEXT Statements

Suppose you want to PRINT the integers 1 to 40. You would have to write a line as shown to the right.

```
READY
:PRINT 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40
```

This is both inefficient and time consuming. What happens when you want to print out the integers 1 to 100, or 1 to 1000?

Using the FOR-TO and NEXT statements, a single BASIC line does the job. The line is written as shown to the right.

```
FOR X=1TO 40:PRINT X,:NEXT X
```

Breaking this line down into individual parts:

FOR X = 1 to 40

This part of the line establishes a *Counter*. It tells the calculator how many times to repeat the line or to loop. A variable counter, here X, keeps track of the number of times the loop is executed.

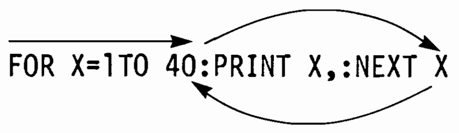
PRINT X,

Each time the loop is repeated X is printed. X takes on the values from 1 to 40.

NEXT X

The third part of the statement line increments X by one and then tests X to see if it has passed the outer limit of the FOR-TO statement (here 40). If it has, the calculator stops. If it hasn't, it processes the PRINT statement with X set at the current value and then continues incrementing.

The actual looping in these statements takes place between the PRINT and NEXT statement as shown to the right.



```
FOR X=1TO 40:PRINT X,:NEXT X
```

Enter this statement line.

Touch RESET

Touch keys FOR SHIFT X = 1 TO 40
:PRINT X, :NEXT SHIFT X CR/LF-EXECUTE

```
READY
:FOR X=1TO 40:PRINT X,:NEXT X
 1              2              3              4
 5              6              7              8
 9              10             11             12
 13             14             15             16
 17             18             19             20
 21             22             23             24
 25             26             27             28
 29             30             31             32
 33             34             35             36
 37             38             39             40

:_
```

Notice the keywords PRINT, FOR, TO, and NEXT were generated with a single keystroke.

The comma after PRINT X, is used to create a zoned output.

The variable in the FOR-TO statement must be the same variable in the NEXT statement.

The colon must be used to separate the individual statements in this line.

In this example, the counter was incremented by 1. The FOR-TO/NEXT statements have an added flexibility — you can increment the Counter by any amount by using the keyword STEP to specify the increment.

Redoing this same problem, but this time incrementing X by .5 and counting only from 1 to 10, the statement line would look like that shown to the right.

```
FOR X=1TO 10STEP .5:PRINT X,:NEXT X
```

Notice in the FOR-TO statement you have added the word STEP and .5 (the increment).

If the word STEP[1] is not included in the FOR-TO statement, the increment is assumed to be one; if STEP is added, a number must follow it.

Enter this statement line.

Touch RESET

Touch keys FOR SHIFT X = 1 TO 10 STEP .5: PRINT X, :NEXT X CR/LF-EXECUTE

This time the variable X was incremented by .5 and the loop stopped when X reached 10.

```
READY
:FOR X=1TO 10STEP .5:PRINT X,:NEXT X
 1                 1.5               2                 2.5
 3                 3.5               4                 4.5
 5                 5.5               6                 6.5
 7                 7.5               8                 8.5
 9                 9.5               10
:_
```

In general practice, the values which define the limits of the loop can be any legal expression. For example the limits of the loop can be calculated. See an example of such a FOR-TO statement to the right.

```
FOR X=SQR(2)TO SQR(100):
```

The general format of the FOR-TO/NEXT statement is shown to the right.

$$\text{FOR}\begin{bmatrix}\text{variable}\end{bmatrix} = \begin{bmatrix}\text{starting value}\\ \text{composed of}\\ \text{any legal}\\ \text{expression}\end{bmatrix}\text{TO}\begin{bmatrix}\text{loop limit}\\ \text{composed of}\\ \text{any legal}\\ \text{expression}\end{bmatrix}\text{STEP}\underbrace{\begin{bmatrix}\text{increment size}\\ \text{composed of}\\ \text{any legal}\\ \text{expression}\end{bmatrix}}_{\substack{\text{Optional}\\ \text{If not included Step is}\\ \text{automatically 1}}}\text{:NEXT}\begin{bmatrix}\text{variable}\end{bmatrix}$$

[1] STEP can have a negative value.

Some examples of FOR-TO/NEXT statements are shown to the right.

EXAMPLES OF FOR-TO/NEXT STATEMENTS

FOR X = 5 TO 500 NEXT X	X takes on 496 successive values 5,6,7 498,499,500
FOR X = 5 TO 500 STEP 5 NEXT X	X takes on 100 successive values 5,10,15....495,500
FOR A = SQR(2)E2 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 38 successive values 100, 98, 96,....28,26
FOR G = –10 TO + 10 STEP .4 NEXT G	G takes on 51 values –10, –9.6, –9.2,...–.4,0,+.4,...+9.6, +10

Some illegal FOR-TO/NEXT statements are shown to the right.

ILLEGAL FOR-TO/NEXT STATEMENTS

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 examples the loops are executed only once because of the increments defined in the STEP. In the first example, Y6 = 52.3 is executed, but when −1 is added to 52.3 it decreases the value of Y6 and the intended limit (100) can never be reached. As a final example, sum the integers from 1 to 25 printing out the sum at the end of each loop in a zoned format. The statement line to accomplish this is shown to the right.

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

Enter this line.

Touch keys FOR SHIFT X = 1 TO 25
:SHIFT Y = SHIFT Y + SHIFT X :PRINT
SHIFT Y, :NEXT SHIFT X CR/LF-EXECUTE

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

The format of the output of this statement line can be changed be either changing the comma to a semicolon or including no punctuation. For example,

Touch keys FOR SHIFT X = 1 TO 25 :SHIFT Y = SHIFT Y + SHIFT X :PRINT SHIFT Y :NEXT X CR/LF-EXECUTE

Because there was no punctuation in the PRINT statement, each answer is printed on a separate line.

When the line was executed the results were displayed with such rapidity that the output was difficult to follow. The System 2200 can be instructed to pause after each line of output. To instruct the System 2200 to pause for n/6 seconds after every line of output the SELECT P option can be used, where n can be any integer from 0 to 9. Each n represents a 1/6th second pause.

Touch keys SELECT SHIFT P 2 CR/LF-EXECUTE

Touch keys FOR SHIFT X = 1 TO 25 :SHIFT Y = SHIFT Y + SHIFT X :NEXT X CR/LF-EXECUTE

To turn a pause off,

Touch keys SELECT SHIFT P CR/LF

NOTE:

For the remainder of this text, the word SHIFT is no longer used to indicate the upper case character. It is assumed.

```
READY
:FOR X=1TO 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

:_
```

Section 6-2
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 System 2200 to plot a diagonal line of 11 pluses (+) down the CRT starting in the first column:

Touch keys FOR X=0 TO 10 :PRINT TAB(X);
"+" :NEXT X CR/LF-EXECUTE

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 on the CRT display.

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

Touch keys FOR X=0 TO 10 :PRINT TAB(X);
"+"; :NEXT X CR/LF-EXECUTE

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


:_
```

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

The result is different 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 System 2200 CRT in one statement line.

Touch keys FOR X=1 TO 8 :PRINT TAB (X↑2–1); "0" :NEXT X CR/LF-EXECUTE

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

:_
```

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.

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

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

```
READY
:FOR X=-6TO 6:PRINT TAB(X↑2);X↑2:NEXT X
                                     36
                          25
                 16
          9
     4
  1
 0
  1
     4
          9
                 16
                          25
                                     36

:_
```

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

Touch keys FOR X = 1 TO 5 :PRINT X :PRINT: NEXT X CR/LF-EXECUTE

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, as nothing is printed as the result of the second PRINT.

Touch keys FOR X = 1 TO 5 :PRINT X; :NEXT X :PRINT "DONE" CR/LF-EXECUTE

Touch keys FOR X = 1 TO 5 :PRINT X; :NEXT X :PRINT :PRINT "DONE" CR/LF-EXECUTE

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 line 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").

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

 2

 3

 4

 5


:_
```

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

:_
```

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

:_
```

SUMMARY

You have now completed the Introduction to the System 2200 as used in the Immediate mode. The problems used in Part I are only samples of what can be done in the Immediate mode and are only used to suggest a starting point for your own applications.

Continue now to Part II, which introduces the System 2200 as used in the Programming mode.

Part II

Using the System 2200 as a Programmable Calculator

Using the System 2200 in the Immediate Mode or in the Programming Mode depends upon the results or calculations you desire. If you wish to obtain one-time results, then the Immediate Mode should be used. If you wish to be able to repeat calculations, and be able to program the System 2200, you should use the Programming Mode. The remainder of this text deals with using the System 2200 in the Programming Mode and explains the differences between entering and executing a line in the Programming Mode vs. the Immediate Mode. The text then discusses in depth all the programming techniques available with the System 2200's BASIC language.

Chapter 7 The Basics of BASIC Programming

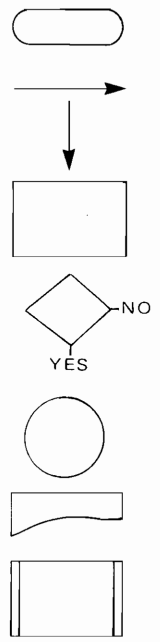
Chapter 7 introduces you to the basic concepts needed to write, enter and execute a BASIC program on the System 2200. In most cases, a single program is used to illustrate the various stages discussed in this chapter. What is discussed in this chapter is applicable to any program you write.

Section 7-1 Writing Programs

FLOW CHARTING

When a programmer decides to write a program, he (or she) does not sit down and immediately enter the program into the System 2200. Rather, a knowledgeable programmer begins by thoroughly analyzing the problem. If careful analysis is done in the beginning, fewer problems are encountered later. Part of the analysis process includes a flowchart. Thus the first step in writing a program is to make a flowchart.

A flowchart is a visual representation of all the steps required to solve a problem. It helps to crystalize the programmer's thoughts. Standard symbols are used in flowcharting. They are shown to the right.



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

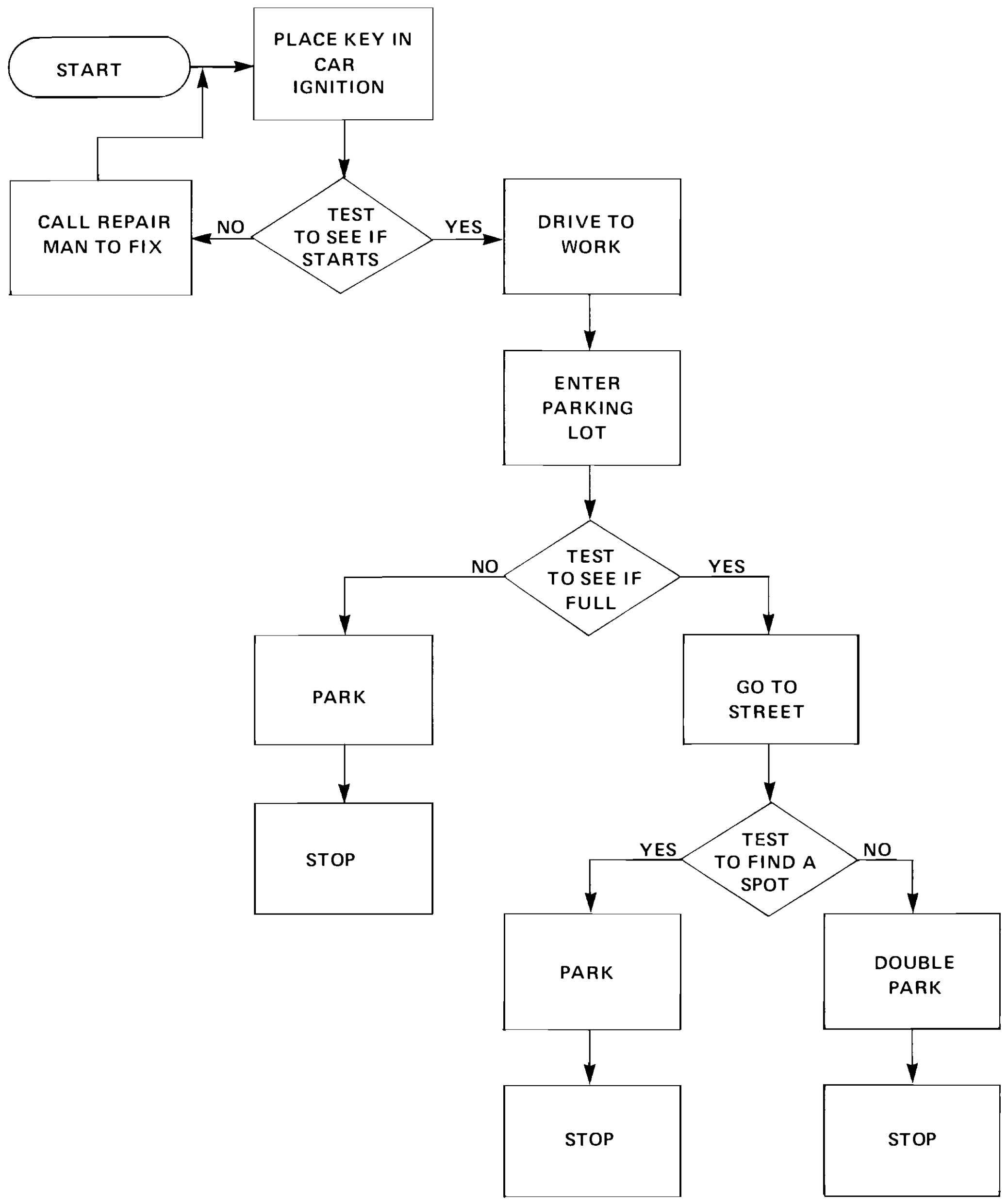
– A printout or display of any type (usually an answer).

– The predefined process symbol, generally used to represent a subroutine.

An example of a flowchart is shown to the right.

Notice each step is represented in the flowchart with the appropriate type of symbol. Also, the logic or flow is charted with the use of arrows.

EXAMPLE OF A FLOWCHART



Try writing a flowchart showing the solution to the following problem:

$C = \sqrt{A^2 + B^2}$

Check your results with the figure shown to the right.

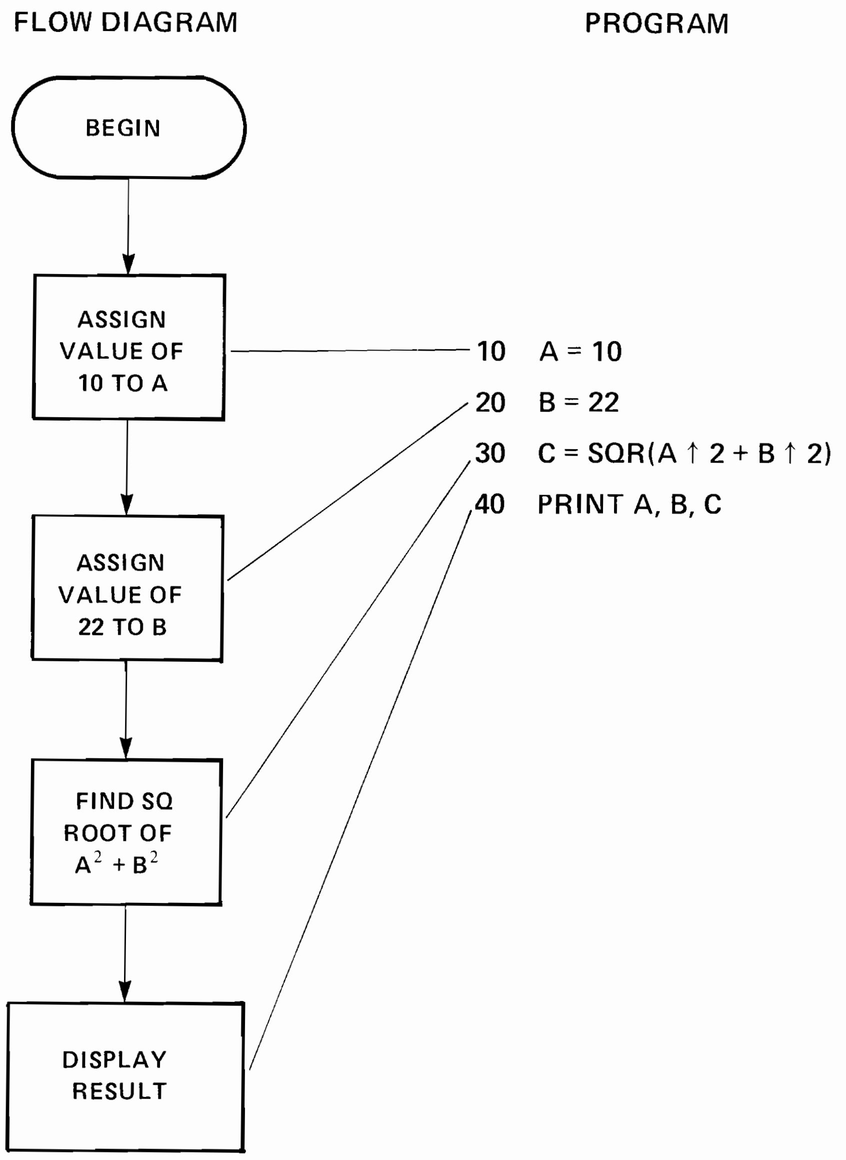
CONVERTING A FLOWCHART TO A PROGRAM

Once the flowchart is written and it represents the solution to the problem, it is then necessary to write the corresponding BASIC statement line(s) to accomplish the task indicated in each symbol of the flowchart.

In Chapters 1 through 6 you have written only a single line to solve a problem. A program consists of one or more lines of BASIC statements. In order to tell the calculator in what order to execute the lines, and to delineate the beginning of a new statement line, it is necessary to precede each line with a statement line number. The numbers can range from 1 to 9999. Common practice suggests the use of 10,20,30,40,... etc. as statement line numbers. There is a practical reason for this, which is explained later.

Write statement lines for your program ($C=\sqrt{A^2+B^2}$) and check your results with the figure to the right.

Notice each statement is preceded by a line number.



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

NOTE:

One of the major differences between using the System 2200 as a calculator or a programmable calculator is the use of statement line numbers. When a statement line number precedes a line, it immediately indicates to the System 2200 that it is in Programming Mode. The use of a line number enables you to execute a line again and again.

Section 7-2
Clearing Memory

Now that the program is written, the next step is to enter it into the memory of the System 2200. Before entering a program the memory should be cleared (erased) to assure it is free of other programs. As mentioned earlier in this text, the entire memory is cleared when the keys CLEAR CR/LF-EXECUTE are touched and only variables are cleared when keys CLEAR V CR/LF-EXECUTE are touched.

CLEAR EXECUTE	– Clears memory.
CLEAR V EXECUTE	– Clears only variables from memory.

If you wish to save variables already in memory, there is another CLEAR option which clears only program text from memory. This is CLEAR P CR/LF-EXECUTE.

CLEAR P EXECUTE	– Clears only program text from memory.

Any program recently entered would be cleared from memory, along with any other program entered earlier, if you used the CLEAR P option. What remains in memory is any variables that had been assigned by the program or any other program.

As may be evident at this time, the memory is divided into different storage areas. Program text is stored in a different area of memory than variables. Therefore, you can be selective as to what you want to clear from the memory. However, it is often necessary to clear the entire memory (CLEAR CR/LF-EXECUTE) before entering a new program.

Touch keys CLEAR CR/LF-EXECUTE

```
READY
:_
```

Section 7-3
Enterinq A Program

ENTERING A PROGRAM INTO MEMORY VIA THE MODEL 2215 KEYBOARD

1. First clear the memory.

   Touch CLEAR CR/LF-EXECUTE.

```
READY
:_
```

2. Enter a statement line number.

There are two ways to enter a statement line number:

A statement line number can be generated by using the numeric keyboard, or

A statement line number can be generated by touching the STMT NO. key, before entering each statement line.

Touch STMT NO. key

```
READY
:10 _
```

The statement line number 10 was generated automatically when the key was touched. 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 generated.

To generate a line number less than 10 more, you must use the numeric keyboard.

Touch keys SHIFT A=10 CR/LF-EXECUTE

CR/LF-EXECUTE causes statement line 10 to be entered in the memory of the System 2200.

```
READY
:10 A=10
:_
```

Enter the second line of the program.

Touch keys STMT NO. SHIFT B = 22 CR/LF-EXECUTE

```
READY
:10 A=10
:20 B=22
:_
```

Enter the third and fourth lines of the program.

Touch keys STMT NO. SHIFT C = SQR(SHIFT A ↑2 + SHIFT B ↑2) CR/LF-EXECUTE

STMT NO. PRINT SHIFT A, SHIFT B, SHIFT C CR/LF-EXECUTE

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

The entire program is now in memory as shown in the display. Although each statement was entered in the correct order (assured when using the STMT. NO. key) they need not be entered in that order. The System 2200 automatically reorders a program in the proper sequence according to statement numbers when the program is executed. That means you can enter a program in any sequence, but when entering in random order, you cannot use the STMT. NO. key. You must use the numeric keyboard to generate statement numbers. For example, the same program, if entered as shown to the right, would automatically be ordered correctly when executed. The numeric keys were used to enter the statement numbers.

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

Section 7-4
Executing A Program

EXECUTING A PROGRAM

After an entire program is entered into memory, the next step is to execute or run the program.

Touch keys RUN CR/LF-EXECUTE

When RUN is touched the System 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.
3. Checks to assure that no logic errors have been made in the program.
4. Once steps 1, 2, and 3 are completed, the program lines are executed sequentially and all instructions are carried out.

```
READY
:10 A=10
:20 B=22
:30 C=SQR(A↑2+B↑2)
:40 PRINT A,B,C
:RUN
 10              22              24.166091947
```

NOTE:

Another difference between the Immediate Mode and Programming Mode is the manner in which statements are executed. In the Immediate Mode a line is executed and the result(s) are obtained immediately when the CR/LF-EXECUTE key is touched at the end of a line. In the Programming Mode it is not until RUN CR/LF-EXECUTE is keyed that any results are obtained. All the CR/LF-EXECUTE key does at the end of a program line is to enter the line into the memory of the System 2200 and to verify that no syntax errors have been made.

Touch keys RUN CR/LF-EXECUTE again.

Notice the results of the execution of the program are again printed out. The results can be printed out as many times as desired in the Programming Mode by simply touching RUN CR/LF-EXECUTE.

There is an option available with the RUN key which allows you to execute part of a program. Touching the RUN key executes the entire program from beginning to end. If you wish to execute only statements 30 and 40, touch keys RUN 30 EXECUTE. This instructs the System 2200 to execute the program from line 30 to the end of the program. (In this example lines 30 and 40 are executed.) The general form of the RUN statement is shown to the right.

```
READY
:10 A=10
:20 B=22
:30 C=SQR(A↑2+B↑2)
:40 PRINT A,B,C
:RUN
 10               22                24.166091947

:RUN
 10               22                24.166091947

:_
```

GENERAL FORM

RUN [line number]

NOTE:

All BASIC statements have a general form which indicates what the statement can do. In the remainder of the text, as each statement is discussed, the general form is also given (the arguments enclosed in brackets are optional and arguments enclosed in braces are required for the statement to work).

Section 7-5
Listing A Program

USING THE LIST KEY

On the System 2200, a program can be listed by touching LIST CR/LF-EXECUTE.

Touch RESET

Touch keys LIST CR/LF-EXECUTE

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

The listing of the program is displayed. If a program was entered out of order, listing the program with the LIST key always displays the program with the correct sequence of statements.

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. LIST S causes the System 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. The 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 in the LIST statement which lines are desired. The general form of the LIST statement is shown to the right.

Touch RESET

Touch keys LIST 10,30 CR/LF-EXECUTE

Only lines 10 - 30 are listed

Touch RESET

Touch keys LIST 20 CR/LF-EXECUTE

Only line number 20 is listed.

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

GENERAL FORM

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

NO. OF FIRST STMT. LINE TO BE LISTED

NO. OF LAST STMT. LINE TO BE LISTED

```
READY
:LIST 10,30
10 A=10
20 B=22
30 C=SQR(A↑2+B↑2)
:_
```

```
READY
:LIST 20
20 B=22
:_
```

Section 7-6
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.

REDEFINING A STATEMENT LINE

To redefine a line already in memory, reenter the *same* line number followed by the new line and touch the CR/LF-EXECUTE key. This replaces the old line with the new line.

Touch RESET

Touch keys 10 SHIFT A = 5 CR/LF-EXECUTE

```
READY
:10 A=5
:_
```

The new line has replaced the old line in memory.

Touch RESET

Touch LIST CR/LF-EXECUTE

```
READY
:LIST
10 A=5
20 B=22
30 C=SQR(A↑2+B↑2)
40 PRINT A,B,C
:_
```

The listing illustrates the new line has replaced the old line.

DELETING A LINE FROM MEMORY

Enter the line number of the line to be deleted and touch CR/LF-EXECUTE.

Touch 20 CR/LF-EXECUTE

Touch LIST EXECUTE

```
READY
:20

:LIST
10 A=5
30 C=SQR(A↑2+B↑2)
40 PRINT A,B,C
:_
```

Notice line 20 has been deleted from memory.

INSERTING A NEW LINE

A new line can be entered into memory by entering an appropriate line number somewhere between the existing line numbers where the new line is to be inserted.

Touch RESET

Touch keys 22 SHIFT B = 22 CR/LF

```
READY
:22 B=22
:_
```

You added a line between lines 10 and 30. You also could have added several other lines between 10 and 30 as long as the line numbers were unused (i.e., 11, 12, 13, 14...21, 23...39).

Earlier in this chapter it was mentioned that spaced line numbers should be used (i.e., 10, 20, 30, etc). Spaced line numbers allow room to insert new lines at a later time. If you used line numbers such as 1, 2, 3, 4, etc., you could not add lines between them (line numbers can only be integers).

```
READY
:LIST
10 A=5
22 B=22
30 C=SQR(A↑2+B↑2)
40 PRINT A,B,C
:_
```

Section 7-7
Using The BASIC STOP Statement

Although the program you are currently working with is a complete program, an additional statement can be included anywhere in the program to signal the System 2200 to stop processing (i.e., the STOP statement). This STOP statement consists generally of a statement line number, followed by the BASIC keyword STOP.

Touch RESET

Touch keys LIST CR/LF-EXECUTE

Touch keys 50 STOP CR/LF-EXECUTE

```
READY
:LIST
10 A=5
22 B=22
30 C=SQR(A↑2+B↑2)
40 PRINT A,B,C
:50 STOP
:_
```

Statement number 50 is now added to the program.

When the System 2200 executes the STOP statement during the course of program execution, the word STOP is printed in the CRT display.

Touch RUN CR/LF-EXECUTE

```
READY
:LIST
10 A=5
20 B=22
30 C=SQR(A↑2+B↑2)
40 PRINT A,B,C
:50 STOP
:RUN
 5               22               22.561028345

STOP
:_
```

One use of the STOP statement is to delineate the end of one section of a program from another section. If a program has three sections and you wish to RUN only the first, you cannot unless they are separated in some way. The STOP statement can be used to do this.

The STOP statement is not required at the end of a program; the System 2200 automatically stops when it runs out of statement numbers.

STOP 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.

GENERAL FORM

STOP [''character string'']

Touch RESET

Touch keys 35 STOP SHIFT LOCK
''***** END OF CALCULATION ****''
SHIFT CR/LF-EXECUTE

Touch keys LIST CR/LF-EXECUTE

Touch keys RUN CR/LF-EXECUTE

```
READY
:35 STOP "****END OF CALCULATION****"
:LIST
10 A=5
22 B=22
30 C=SQR(A↑2+B↑2)
35 STOP "****END OF CALCULATION****"
40 PRINT A,B,C
50 STOP
:RUN

STOP ****END OF CALCULATION****
:_
```

The program is executed until the first STOP statement is encountered. This results in the program stopping and the word STOP and any literal string within the STOP statement being printed.

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 System 2200 as a calculator and immediately execute statement lines, without statement line numbers (Immediate Mode).

2. Print out a variable in the program for inspection.
3. Redefine a variable used in the program and re-execute it to see how this affects results.
4. Change the program flow, and instruct the System 2200 to continue execution at a different program line.

Continue execution of the program:

Touch keys CONTINUE CR/LF-EXECUTE

```
 RUN

STOP ****END OF CALCULATION****
:CONTINUE
 5                22                22.561028345

STOP
:_
```

Section 7-8
Using the BASIC END Statement In A Program

In addition to the STOP statement, another statement which terminates program execution is the END statement. The END statement line consists simply of a BASIC keyword END.

The END statement is optional in System 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 program statements are executed. Therefore END, as with STOP, need not be used to delineate the end of a program.

GENERAL FORM

END

Replace line 50 of the program with 50 END as follows:

Touch RESET

Touch keys 50 END CR/LF-EXECUTE

Touch keys LIST CR/LF-EXECUTE

Touch RUN CR/LF-EXECUTE

Touch CONTINUE CR/LF-EXECUTE

Executing a program with an END statement results in the following message being printed:

END PROGRAM
FREE SPACE = number

The Free Space number is an integer representing the approximate number of bytes[1] left in memory for storing additional program text or variables. The System 2200 requires approximately 700 bytes as a work area while executing statements. These 700 bytes of the 4096 bytes in a 4K machine are not available to the user (700 bytes are set aside for a work area in all System 2200 configurations, regardless of the amount of memory). Therefore, the above program in a 4K system requires 826 bytes of memory (i.e., 4096-3270=826; 826-700=126 for the program itself).

Every time a key is touched on the System 2200, a certain amount of area in memory is required for storage. See Appendix B for a detailed discussion of how to determine the approximate number of bytes required for a program.

[1] A byte is comparable to a programming step.

```
READY
:50 END
:LIST
10 A=5
22 B=22
30 C=SQR(A!2+B!2)
35 STOP "****END OF CALCULATION****"
40 PRINT A,B,C
50 END
:RUN
STOP ****END OF CALCULATION****
:CONTINUE
 5                22                22.561028345

END PROGRAM
FREE SPACE=3270

:_
```

OTHER USES OF THE END STATEMENT

Whenever the END statement is used, the System 2200 displays the FREE SPACE available at that time.

Touch keys CLEAR CR/LF-EXECUTE

Touch keys END CR/LF-EXECUTE

```
READY
:END

END PROGRAM
FREE SPACE=3398

:_
```

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

FREE SPACE is always 3398 in a 4K system, as long as:

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

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

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

:END

END PROGRAM
FREE SPACE=3386

:_
```

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

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

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

:END

END PROGRAM
FREE SPACE=3398

:_
```

With execution of these statements there is no loss of FREE SPACE, since no program text was used and no variables appeared in the statements.

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).

Section 7-9
The REM Statement

REM (Remark) statements are used to insert explanatory notes into a program. Unlike a PRINT or a FOR-TO/NEXT statement, the REM statement is *not executable*; that is, when the System 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; it does take up available memory space.

Consider the program to the right. It contains three REM statements. When the program is executed, lines 20, 40 and 60 are not printed and have no effect on the output; they appear only when the program is listed. REM statements do not require quotation marks.

PROGRAM USING REM STATEMENTS

```
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=5TO 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 PROG. COMPUTES THE AREA OF 3 CIRCLES
RADIUS          AREA
 5               78.53981633975
 10              314.159265359
 15              706.8583470578

END PROGRAM
FREE SPACE=3128

:_
```

SUMMARY: Chapter 7 has taken you through the stages of writing, entering, executing, listing and changing a program. All these steps are basic to understanding more complicated programming. You now should be able to write many programs. These basic programs are the basis for writing advanced programs.

Chapter 8

Branching In Programs

An important technique to master and use in programming is branching. Branching allows the flow of program to jump from one place to another within the program, disregarding the sequential order of statements.

There are two different types of branching, *unconditional branching* and *conditional branching.* An unconditional branch always causes a jump to another location, independent of any condition being met. A conditional branch causes a jump only when a certain condition is met, otherwise the regular program sequence is followed.

There are two types of conditional branches and two types of unconditional branches available in System 2200 BASIC. The two unconditional branches are: (1) the GOTO statement and (2) the GOSUB statement. The two condiitonal branches are: (1) the IF-THEN statement and (2) the FOR-TO/NEXT statement. Each is explained in this chapter.

Section 8-1
The GOTO Statement

Consider the program and associated flowchart shown to the right.

Notice in the program the GOTO statement always causes a branching back to statement number 20. No condition has to be met in order to have the branch take place. When this statement is executed by the System 2200, a branch to statement 20 always occurs.

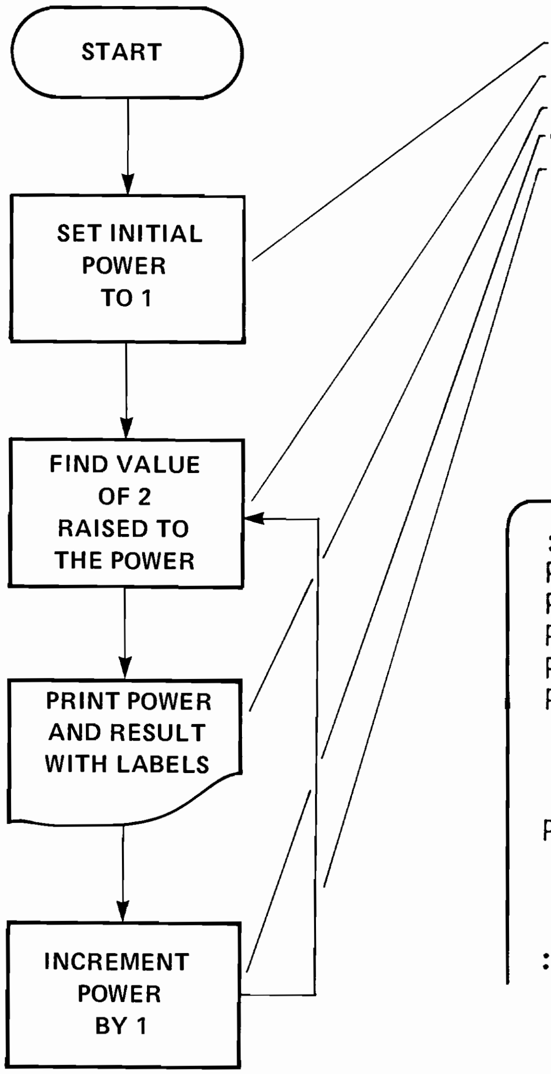
In the flowchart, the GOTO statement is represented by connecting lines with an arrow head.

Enter the program. Before running it, SELECT a half second pause (SELECT P3) then RUN. The results are shown to the right.

The ERR 03 means a math error. When (2^{332}) is evaluated, the resulting exponent causes an exponent overflow (> 10E+99). This stops the program.

Remember to deselect the pause by touching keys SELECT P, CR/LF-EXECUTE.

FLOWCHART


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

PROGRAM

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

RESULTS

```
:RUN
POWER= 1          Q= 2
POWER= 2          Q= 4
POWER= 3          Q= 8
POWER= 4          Q= 16
POWER= 5          Q= 32
  .
  .
  .
POWER= 332        Q= 8.74900289E+99

 20 Q=2↑P
        ↑ERR 03
:_
```

The general form of a GOTO statement is shown to the right.

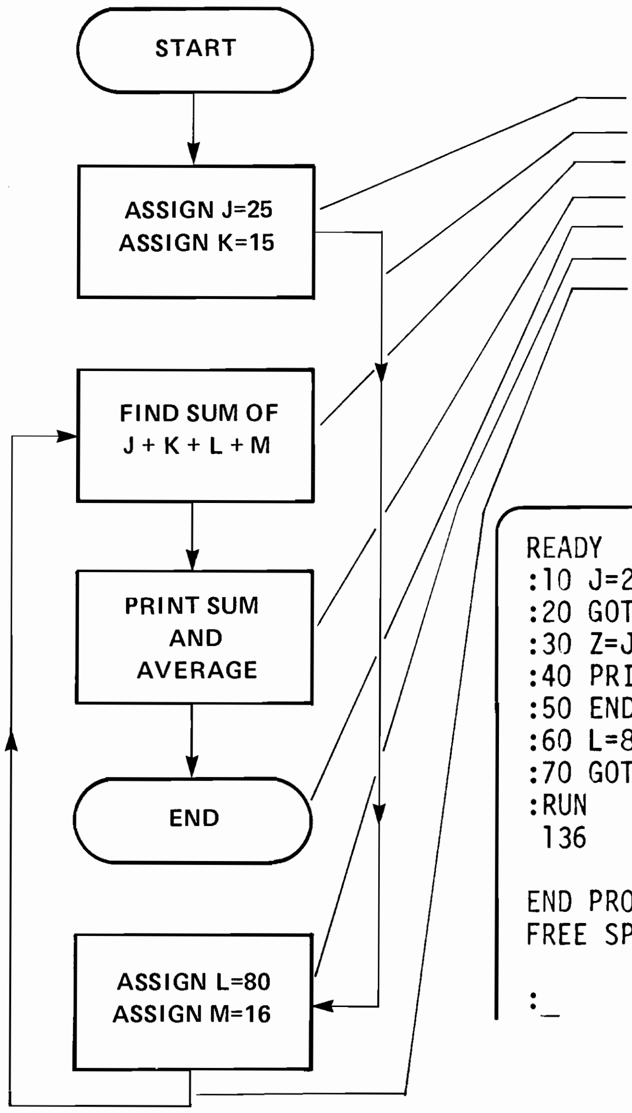
Consider a second program which uses two GOTO statements (see the program at the right).

What is the advantage of the GOTO statement? Consider what you would have to do to obtain the results of either of these programs if you did not use a GOTO statement. In the first program you would have to manually key in each different value of P and execute the program each time for each value. How would you do the second program?

GENERAL FORM

GOTO line number

FLOWCHART



PROGRAM

```
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
```

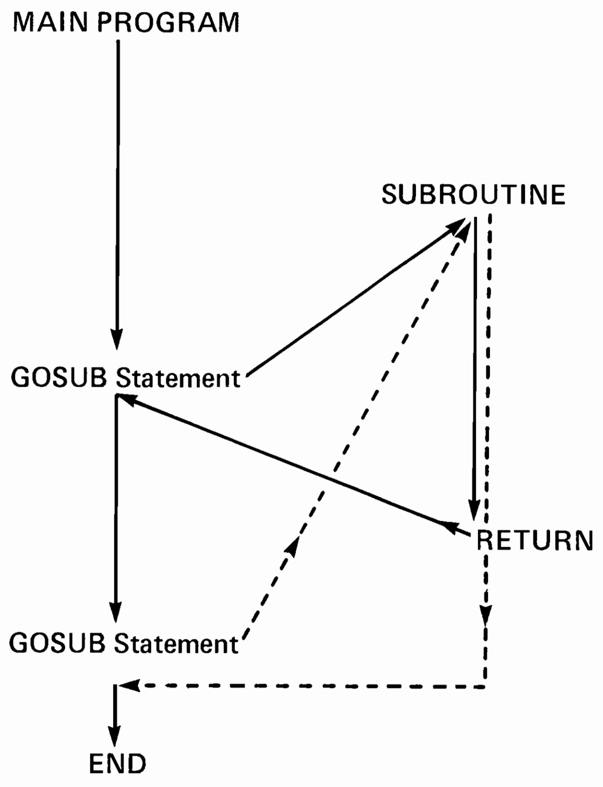
RESULTS

```
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

:_
```

Section 8-2
The GOSUB Statement (Subroutines)

The GOSUB statement causes a branch to a subroutine. No condition has to be met before the branch is made. 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 called upon each time they are needed with the GOSUB statement.

The GOSUB statement causes a branch from the main line of a program to the subroutine. After the subroutine is executed, the last line of the subroutine must be a RETURN statement which directs program flow back to the main program, namely the statement immediately following the GOSUB statement. If more than one GOSUB statement is used in a program, branching back is always to the line immediately following the last executed GOSUB statement. See the diaaram to the right.



The general form of the GOSUB statement is shown to the right. The line number is the number of the line beginning the subroutine.

GENERAL FORM

GOSUB line number

In the partial program shown to the right, statements 2000-2050 represent a subroutine. Notice the program branches to this subroutine three times by a GOSUB statement and that control is returned to the statement immediately following the last executed GOSUB in the main program. The GOSUB statement is not part of the subroutine, it just causes the branch to the subroutine.

In a 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) can be included on a line after a RETURN.

LEGAL

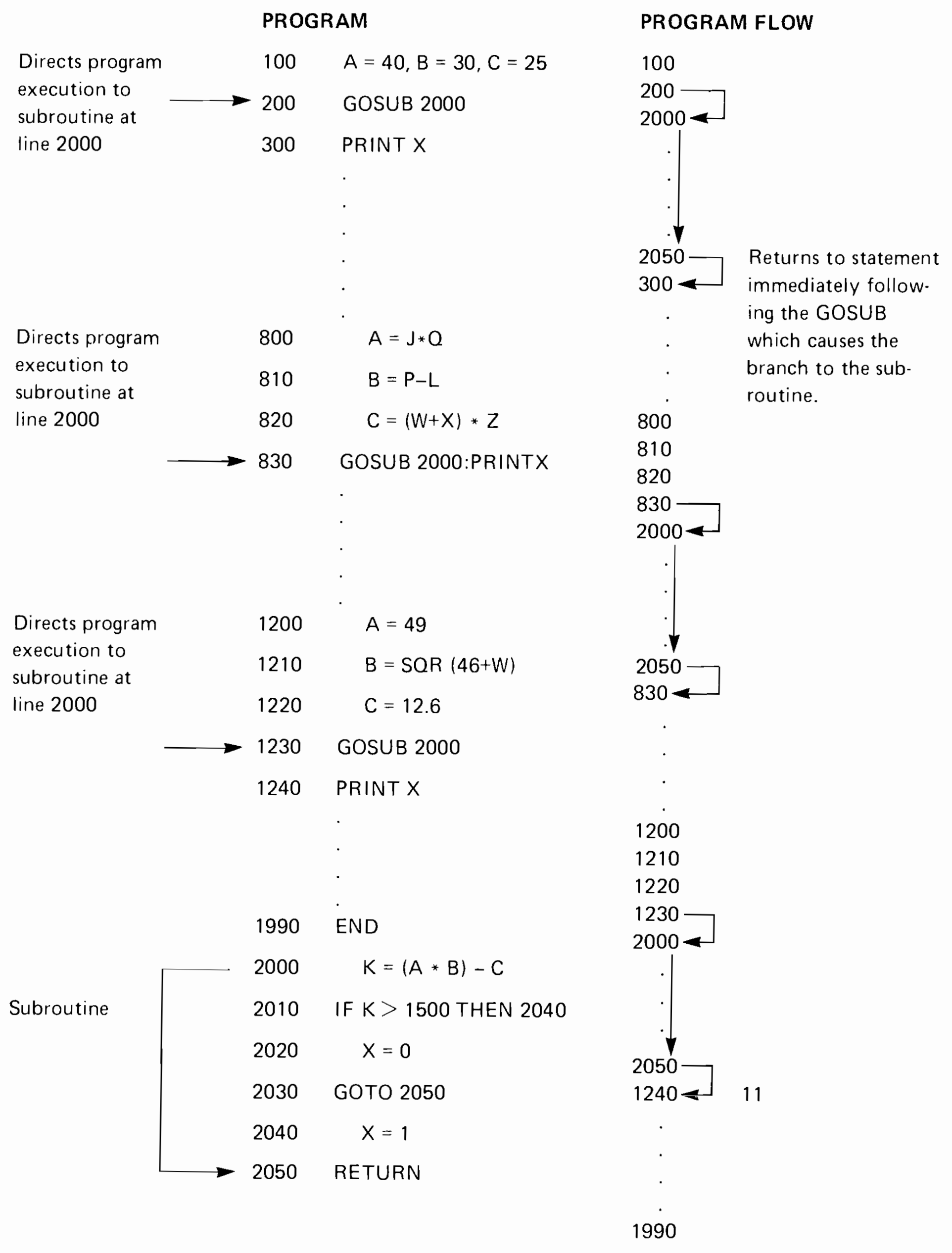
```
50 X = 10: RETURN: REM END
```

ILLEGAL

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

NOTE:

A program cannot have a GOSUB statement as its last statement. If it does, it causes a hang up in the System 2200.

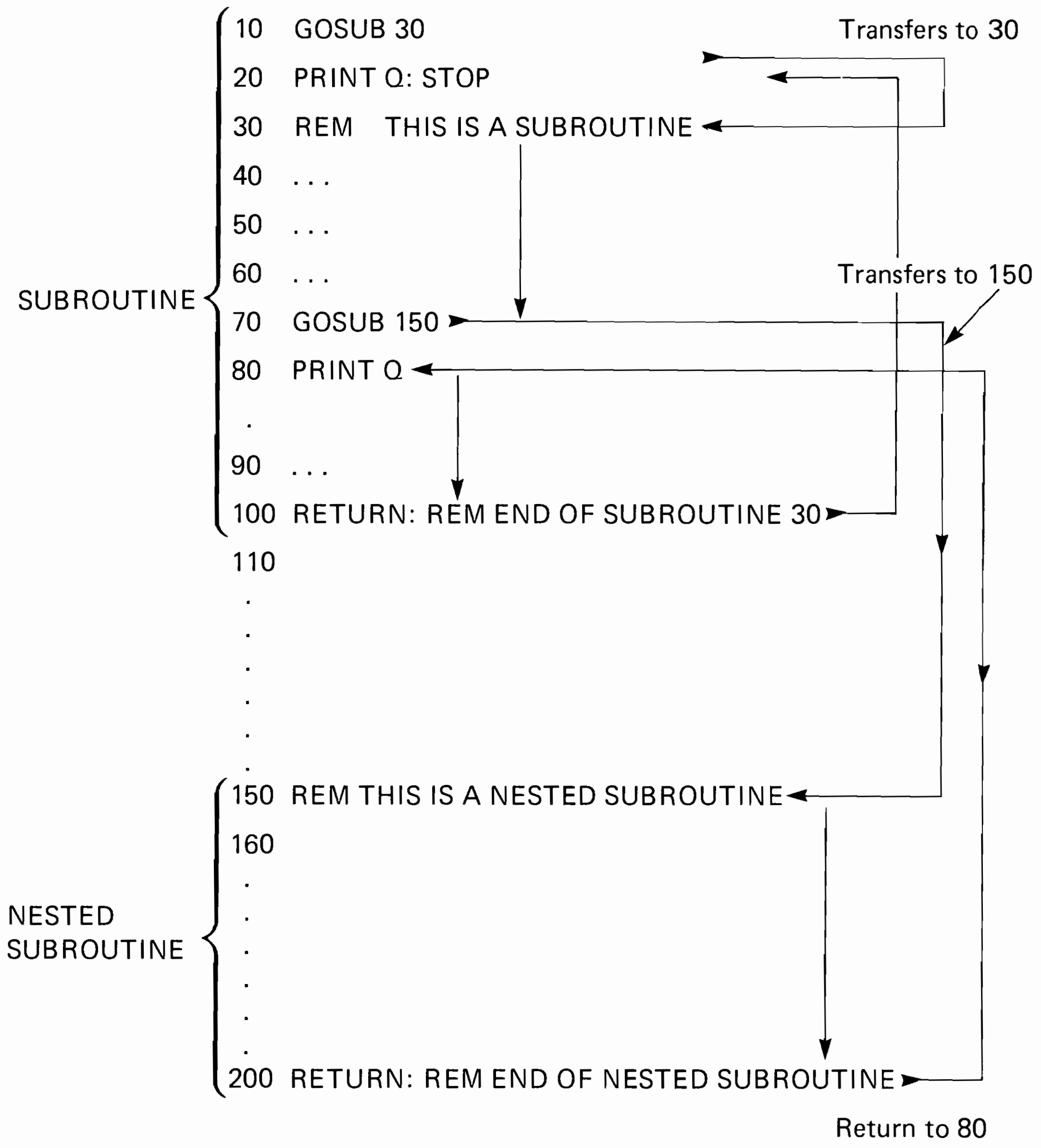

PROGRAM
PROGRAM FLOW
Directs program execution to subroutine at line 2000
Directs program execution to subroutine at line 2000
Directs program execution to subroutine at line 2000
Subroutine
Returns to statement immediately following the GOSUB which causes the branch to the subroutine.
100
200
2000
2050
300
800
810
820
830
2000
2050
830
1200
1210
1220
1230
2000
2050
1240
11
1990

```
100   A = 40, B = 30, C = 25
200   GOSUB 2000
300   PRINT X
      .
      .
      .
800     A = J*Q
810     B = P–L
820     C = (W+X) * Z
830   GOSUB 2000:PRINTX
      .
      .
      .
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  GOTO 2050
2040    X = 1
2050  RETURN
```

Branching can take place within 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.*

An example is shown to the right.

EXAMPLE OF A NESTED SUBROUTINE
SHOWING PROGRAM FLOW



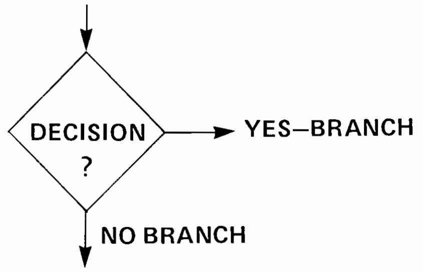
```
10   GOSUB 30
20   PRINT Q: STOP
30   REM   THIS IS A SUBROUTINE
40   . . .
50   . . .
60   . . .
70   GOSUB 150
80   PRINT Q
.
90   . . .
100  RETURN: REM END OF SUBROUTINE 30
110
.
.
.
.
.
.
150  REM THIS IS A NESTED SUBROUTINE
160
.
.
.
.
.
200  RETURN: REM END OF NESTED SUBROUTINE
```

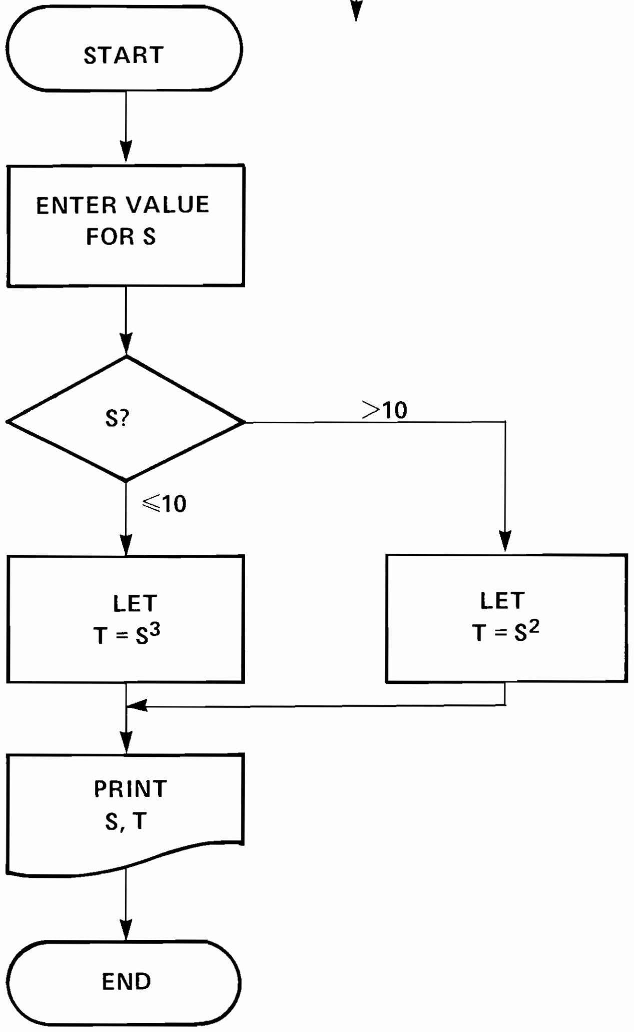
Section 8-3
The IF-THEN Statement

The IF-THEN statement is a conditional branch which has the ability to test values and branch if a condition is met, and not branch if the condition is not met.

In flowcharting, a decision or conditional branch is represented by a diamond with arrows.

Look at the flowchart to the right. Notice several different values are entered for the value of S. Each value is tested to see if it is $>$ 10; if it is, the program squares the value and prints out S and T. If not, the program cubes the value and prints out S and T. In either case, the program then stops.





The program corresponding to this flowchart is shown to the right. The step for entering the different values for S is shown as a STOP statement for manual entry of the values. Statement 10 is the statement which tests the value of S against 10. If the condition is true, the program branches to statement 40; if not true, it continues to the next statement (here 30).

Clear the memory, then enter this program into the System 2200.

Touch keys RUN CR/LF-EXECUTE

The program stops. Enter a value for S

Touch keys S = 2 CR/LF-EXECUTE

Touch CONTINUE CR/LF-EXECUTE

Touch keys RUN CR/LF-EXECUTE

The program stops. Enter a value for S

Touch keys S = 15 CR/LF-EXECUTE

Touch CONTINUE CR/LF-EXECUTE

THE PROGRAM

```
READY
:05 STOP
:10 IF S>10 THEN 40
:20 T=S↑3
:30 PRINT S,T
:35 END
:40 T=S↑2
:50 GOTO 30
```

```
READY
:RUN

STOP
:S=2

:CONTINUE
 2                 8

END PROGRAM
FREE SPACE=3299
```

```
:RUN

STOP
:S=15

:CONTINUE
 15                225

END PROGRAM
FREE SPACE=3299

:
```

Touch keys RUN CR/LF-EXECUTE

The program STOPS. Enter a value for S

Touch keys S = 11 CR/LF-EXECUTE

Touch CONTINUE EXECUTE

You can enter as many values of S as desired and with each entry the program makes a decision based on whether S > 10 and branches accordingly. (Later in this text you will be instructed on how to input data in a more efficient way than shown in this example.)

The general form of the IF-THEN statement is shown to the right.

```
:RUN

STOP
:S=11

:CONTINUE
 11               121

END PROGRAM
FREE SPACE=3299

:_
```

GENERAL FORM

$$\text{IF operand} \quad \left\{\begin{matrix} < \\ <= \\ = \\ >= \\ > \\ <> \end{matrix}\right\} \quad \text{operand THEN line number}$$

$$\text{where operand} = \left\{\begin{matrix} \text{literal string} \\ \text{alphanumeric variable} \\ \text{expression} \end{matrix}\right.$$

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

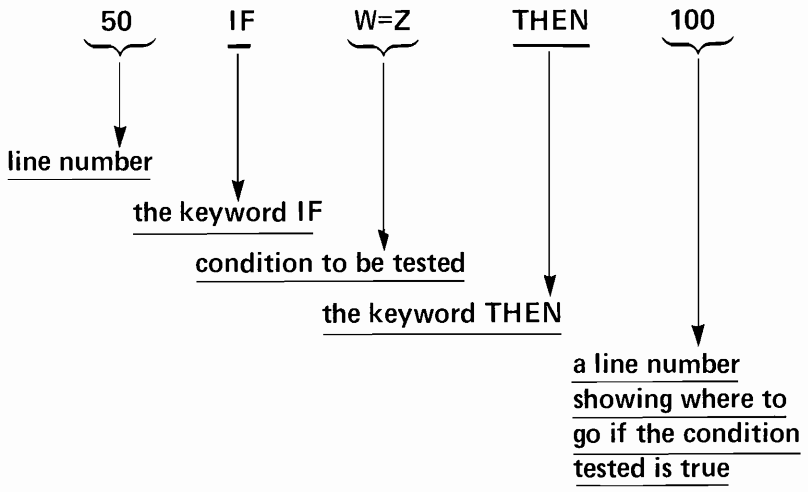
1. The "subject" – part to be tested.
2. The "object" – part the test is made against.
3. The "relation" – type of comparison to be made.

   The subject and object are expressions and must appear on opposite sides of the relation.

There are six different relationships that can be used with an IF-THEN statement (shown to the right).

Some legal and illegal uses of IF-THEN statements are shown for comparison.

EXAMPLE



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

LEGAL

```
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
```

ILLEGAL

```
25 IF W=X GOTO LINE 70

40 IF Y<7 THEN GOTO 50
```

Section 8-4
FOR-TO/NEXT Statement (Loop)

You have been introduced to the FOR-TO/NEXT statement in Part I of this manual. The FOR-TO/NEXT statement represents a conditoinal branch in that the branch depends 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 (loop). If the assigned limits are exceeded, branching stops and the next sequential statement is executed.

Simple uses of the FOR-TO/NEXT statement already has been explained in Chapter 6 (i.e., single loop). There are programming situations where it is necessary to execute one or more loops completely for each iteration of another loop. This is called *nested loops*.

Nested loops can be an efficient means of reducing unnecessary repetition in program text.

FOR-TO/NEXT STATEMENTS

```
FOR X=1 TO 25:PRINT X,X↑2:NEXT X
```

Branch or loop back until value of X is outside the assigned limit of 25.

The example to the right shows this situation applied to a mortgage payment calculation. Notice the interest rate is kept constant where the years of repayment vary from 20 to 25 to 30 years. Then the interest rate is changed and kept constant again for the years of repayment. The outer loop (statement 20) controls the interest rate while the inner loop (statement 30) controls the years of repayment.

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 I and N in the previous example), and that the loops do not overlap, that is the NEXT statement for the inner loop must come before the NEXT statement for the outer loop.

SAMPLE PROGRAM – NESTED LOOP

MORTGAGE PAYMENT PROBLEM, letting interest rate and load period vary

GIVEN:

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

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 0.06.

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

PROGRAM

```
:READY
:10 PRINT "AMOUNT BORROWED","INTEREST RATE","NO. OF YEARS",
    "MO. PYMT."
:20 FOR I=.075 TO .090 STEP .005:REM INTEREST RATE VARIES
:30 FOR N=20 TO 30 STEP 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
:_
```

Once the program is in memory

Touch RESET and RUN CR/LF-EXECUTE

The result is shown to the right.

```
:RUN
AMOUNT BORROWED INTEREST RATE   NO. OF YEARS    MO. PYMT.
$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
:_
```

Chapter 9

Customizing the System 2200

The System 2200 is unique in its class of calculators. In addition to offering a user the programming power of the BASIC language, with verbs such as DEFFN, it also enables a user to further customize the calculator to special needs with the DEFFN' verbs.

The BASIC verb DEFFN', uncommon to most BASIC languages, allows customization either under program control or directly from the keyboard via the 16 Special Function Keys located across the top of your keyboard.

The other statement (DEFFN), common to most BASIC languages, also is used to customize the calculator under program control.

In either case a user can add any number of special routines to an already powerful system.

Chapter 9
Customizing the System 2200

Section 9-1
Customizing the System 2200 Under Program Control
(DEFFN Statement)

You have several math functions on your keyboard (i.e., SIN, SQR, TAN, etc.). The DEFFN statement allows you to define other math function with one variable, accomplished only under program control.

The general form of the DEFFN statement is shown to the right. Examples are also shown.

EXPLANATION OF THE DEFFN STATEMENT:

1. When a DEFFN statement is defined it can appear anywhere in a program.
2. The function name can be any number (0-9) or letter (A-Z) - a total of 36 function names.
3. The dummy variable can be any numeric variable. It is solely a place holder, and has no effect on a variable of its same name used elsewhere in a program.
4. The expression is a special math function.
5. Once the function is defined in a program, it can be used *in that program* just like any keyboard function. The function is then referenced in the program by using the FNa (expression) format, where a is the name of the function and the expression is any numeric expression (i.e., 2*SIN (A+60)).

GENERAL FORM OF DEFFN STATEMENT

DEFFN	a	(v)	=	expression
↑	↑	↑	↑	↑
Keyboard DEFFN	Keyed Function Name	Dummy Variable	Equals Sign	Any expression Containing the Dummy Variable

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

EXAMPLES

```
10 DEFFNF(X)=SQR(X+9)-X
20 DEFFNE(G)=(4*G+6)/G
30 DEFFN8(L)=3*SIN(32)↑2-LOG(18)
40 DEFFN2(Y)=Y*TAN(Y/2)
50 DEFFN3(A)=A↑3.6-A↑2
```

FNa	(expression)
↑	↑
Function Name in DEFFN Statement	Any numeric expression whose value is given to the dummy variable in the DEFFN statement expression

An example of a DEFFN statement defined in a program is shown to the right.

In order to reference the DEFFN statement in a program, the FNa (expression) format is used (also shown to the right).

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

6. A function can be referenced as many times as needed in a program, similar to any keyboard function.

   A function can refer to an already defined function. (See example to the right.)

```
READY
: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

:_
```

7. A function cannot refer to itself nor can two functions refer back to each other (see examples).

ILLEGAL DEFFN STATEMENTS

Example 1

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

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

Example 2

```
10 DEFFNA(B)=FNA(C)
```

Function A refers to Function A

As a final example, a program is shown which makes use of the DEFFN function in a FOR-TO/NEXT loop.

```
READY
:10 REM LOOP PROGRAM FOR THE AREA OF CIRCLES
:20 DEFFNC(R)=#PI*R↑2
:30 PRINT "RADIUS","AREA"
:40 FOR I=1 TO 10
:50 PRINT I,FNC(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

:_
```

Section 9-2
Using The DEFFN′ Statement And Special Function Keys

In addition to the DEFFN verb used to customize the System 2200 *under program control*, another verb, the DEFFN′ verb, is available to customize the System 2200. But with DEFFN′, the special routines are accessible directly through the keyboard.

Located across the top of your keyboard are 16 Special Function Keys. These keys, when used in conjunction with the DEFFN′ verb, enable a programmer to: (1) 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) write and store "marked" subroutines which again can be accessed via a Special Function Key or under program control; and (3) provide argument passing capability in "marked" subroutines directly through the keyboard or under program control. Each use is explained in this section.

GENERAL FORM DEFFN′ VERB

The general form of the DEFFN′ verb is shown to the right.

The integer specified in the DEFFN′ statement is required and must be in the range $0 \leqslant \text{integer} < 256$. If the integer is less than or equal to 31, it represents the number of the Special Function Key. There are 256 possible special functions available on the System 2200, of which 32 are directly accessible via the keyboard (0-31).

The "character string" is optional and can be any string of characters within double quotes.

GENERAL FORM DEFFN′ VERB

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

The variable is optional and can be any legal variable name.

DEFFN' WITH COMMONLY USED CHARACTER STRINGS

Often certain strings are used repeatedly in the program text. If a specific string is special to your applications, it is convenient to have the string as a key on the keyboard. With the DEFFN' statement and the Special Function Keys, you 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 the characters over and over, 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 System 2200 keyboards (0-31), 0-15 functions are lowercase and 16-31 functions are uppercase. Therefore, you can add 32 special routines and access them through the 32 Special Function Keys on your keyboard.

Using the example just mentioned, assign the specified character strings to Special Function Keys 0, 1 and 2 respectively, as follows:

CLEAR MEMORY:

Touch keys STMT NO. DEFFN' 0 "CREDIT CARD" CR/LF-EXECUTE

Touch keys STMT NO. DEFFN 1 "INTEREST" CR/LF-EXECUTE

Touch keys STMT NO. DEFFN' 2 "PAYMENTS" CR/LF-EXECUTE

Each line is now in the memory.
Clear the display by touching RESET.

```
READY
:10 DEFFN'0"CREDIT CARD"
:20 DEFFN'1"INTEREST"
:30 DEFFN'2"PAYMENTS"
:_
```

Touch Special Function Key C (SF 0) and the SPACE key twice.

Touch Special Function Key 1 (SF 1) and the SPACE key twice.

Touch Special Function Key 2 (SF 2).

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

If a defined key is touched while a program line is being entered, the "character string" assigned to the Special Function Key appears on the display, as well as being entered into memory, just as if you touched a key like PRINT. Up to 32 separate "character strings" can be assigned this way.

Touch RESET

Touch keys 50 PRINT "THE MORTGAGE AND Touch SF1" CR/LF-EXECUTE

Notice when SF1 was touched the word INTEREST was automatically entered into the program text.

```
READY
:50 PRINT "THE MORTGAGE AND INTEREST"
```

DEFFN' USED WITH MARKED SUBROUTINES

You already have been introduced to the concept of subroutines. Subroutines used with the DEFFN' verb are called *marked subroutines*.

When writing a marked subroutine, the first statement of the subroutine always must be the DEFFN' integer statement and the last executable statement always must be a RETURN statement, as with any subroutine. There is no limit as to the number of lines within the subroutine.

An example of a marked subroutine is shown to the right. Enter this program into the memory of the System 2200.

```
READY
:10 DEFFN'1
:20 PRINT "THIS IS A MARKED SUBROUTINE"
:30 PRINT "5 SQUARED =";5↑2
:40 RETURN
:_
```

Notice line number 10 refers to Special Function Key 1 and the last line (number 40) is a RETURN statement.

Touch RESET

Touch Special Function Key 1

```
READY
:
THIS IS A MARKED SUBROUTINE
5 SQUARED = 25

:_
```

The Marked subroutine assigned to Special Function Key 1 is executed as if it were hardwired into the calculator.

As mentioned earlier, it is possible to define 256 special functions in a single program. Only the first 32 are directly accessible via the Special Function Keys. The remaining special functions (also the first 32) can be called under program control.

The program to the right is an example of a marked subroutine which cannot be called directly from the keyboard because the integer associated with it is > 31. Any marked subroutine number greater than 31 must be called under program control.

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

In order to call a marked subroutine in a program, the GOSUB' xxx command is used, where xxx is the same integer used in the DEFFN' integer statement.

The program to the right calls marked subroutine 100.

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

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

Enter the entire program (i.e., lines 10 to 60), then execute it.

The results are shown to the right.

```
READY
:LIST
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=3322

:_
```

Enter the next program shown to the right.

Touch RUN

Touch Special Function Key 5

When the program stops

Touch A=5: B=6: C=7 CR/LF-EXECUTE

CONTINUE CR/LF-EXECUTE

Touch Special Function Key 5 again.

Touch keys A=2: B=15: C=7 CR/LF-EXECUTE
CONTINUE

This program also could have been accessed with a GOSUB 5 statement, by adding one line to the program:

5 GOSUB' 5

```
10 DEFFN'5:REM GENERAL QUADRATIC EQUATION SOLUTION
20 STOP "ASSIGN VALUES TO COEFFICIENTS"
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"
```

```
:RUN

STOP ASSIGN VALUES TO COEFFICIENTS
:A=-5:B=6:C=7

:CONTINUE
X1=-.7266499161 X2= 1.9266499161

:

STOP ASSIGN VALUES TO COEFFICIENTS
:A=-2:B=15:C=7

:CONTINUE
X1=-.4407636535 X2= 7.9407636535
:_
```

Another example of a marked subroutine is shown to the right. Notice no RETURN statement is used. A RETURN is not required for text entry. Enter this program and touch Special Function Key 1.

If the word "HEX(" is frequently needed while entering a program, writing a marked subroutine and calling it with Special Function Key 1 can save a programmer much time since the word is generated with a single key.

```
100 DEFFN'1 "HEX("
```

```
READY
:HEX(
```

ARGUMENT PASSING CAPABILITY

The use of variables in the DEFFN' statement is reserved for argument passing; that is, the assigning of values to the variables (s) in the marked subroutine prior to execution of the marked subroutine. The variable names to be used in the subroutines are specified within the DEFFN' statement. The assigning of values to the variable is done either in the GOSUB' statement, calling the marked subroutine, or by entering the values separated by commas. The values are entered just before touching the Special Function Key which calls the marked subroutine from the keyboard. No where in the actual marked subroutine are the variables defined.

The example just used for the solution of a general quadratic equation can be changed so that the variables are assigned values before the subroutine is called.

First, the DEFFN' statement is rewritten as shown to the right, statement number 20 is taken out, and all other statements moved up.

```
READY
:10 DEFFN'5(A,B,C):REM GENERAL QUADRATIC EQUATION SOLUTION
: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":RETURN
```

Notice statement number 10 contains the variable names enclosed within parentheses used in the subroutine.

Enter this marked subroutine into the System 2200 memory, remembering to clear the memory before starting.

For Special Function Key entry to this subroutine, arguments (variable values) are passed by keying them in, separated by commas, just prior to touching the Special Function Key. For example,

Touch RESET

Touch keys 5, 6, 7 Special Function key 5

Touch keys 2, 15, 7 Special Function key 5

These arguments have been passed to the marked subroutine; the subroutine is executed and control is returned to the keyboard.

```
READY
:5,6,7
X1 AND X2 IMAGINARY

:2,15,7
X1=-.5          X2=-7

:_
```

For entry into this subroutine under program control, the GOSUB′ xxx statement, followed by the variable values in parentheses, is used. For example, this same DEFFN′ 5 subroutine is called under program control by adding statement 5 to the program shown to the right.

The GOSUB′ 5 statement assigns values to the variables.

Enter this statement, list the program then execute it.

Touch RESET

Touch key 05 GOSUB′ 5 (5,6,7):STOP

Touch LIST EXECUTE

Touch RUN EXECUTE

The number of the variables entered in the GOSUB′ statement, or keyed in prior to touching the Special Function Key, must be the same as the number of variables specified in the DEFFN′ statement. These values must be separated by commas. The value given can be any legal algebraic expression (see example one to the right). The order in which the values are assigned is the order in which they are represented in the DEFFN′ statement.

There is no limit as to the number of arguments that can be passed.

It is also possible to have nothing following the DEFFN′ integer. In this case you mark a subroutine but do not pass arguments. See example two to the right.

```
READY
:05 GOSUB'5(5,6,7):STOP
:LIST
5 GOSUB '5(5,6,7):STOP
10 DEFFN'5(A,B,C):REM GENERAL QUADRATIC EQUATION SOLUTION
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":RETURN
:RUN
X1 AND X2 IMAGINARY

STOP
:_
```

EXAMPLE 1

```
10 GOSUB' 20(SQR(10),5+6-SQR(6),SIN(.5))
:
:
:
:
90 DEFFN' 20 (A,B,C)
:
:
```

EXAMPLE 2

```
100 DEFFN' 16
110 PRINT "COL#1";TAB(15);"COL#2";TAB(35);"COL#3"
120 PRINT:PRINT
130 RETURN
```

Chapter 10

Additional Methods of Assigning Values to Variables

In Part I, the assignment statement was introduced. Thus far, it has been the only method used in this manual for assigning values to variables. The assignment statement is limited as it must be changed and re-entered if you want to change the value of the variable in the program. Two additional methods are available in System 2200 BASIC which make it possible to vary the value of variables without having to change BASIC statement lines. The two methods are (1) the READ and DATA statements and (2) the INPUT statement. Each is discussed with examples in this chapter.

Examples of Assignment Statements

```
1. 10 LET A=17.3
2. 20 B=23.9:C=-11.4:D=1.3E4
3. 30 LET E=SQR(A*2+B↑2+C↑2+D↑2)
4. 40 PRINT "E=";E
```

Chapter 10
Additional Methods Of Assigning Values To Variables

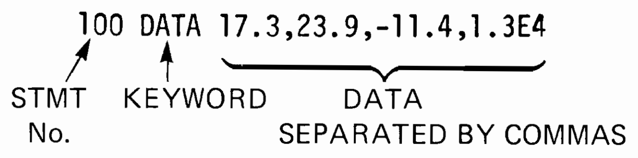
Section 10-1
DATA And READ Statements

The DATA statement (general form shown to the right) is used to store data in a program. The statement can be used only in the Program Mode, and consists of the BASIC keyword DATA, followed by one or more values separated by commas (see example to the right).

GENERAL FORM

DATA n [,n. . .]

where n = number or a character string enclosed in quotation marks.



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

The system automatically sets a data pointer to the location of the first value in the DATA statement. It uses the 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 100 to 120 to the right are equivalent to the single statement 100 in the previous example.

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

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 DATA 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 they are used. This is the function of the READ statement (general form shown to the right). The READ statement is composed of the BASIC keyword READ, followed by one or more variable names separated by commas.

GENERAL FORM

READ [,variable . . .]

The READ statement in line 20 (example at the right) 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 now may 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.

In the program to the right, only two data values are read by each READ statement (lines 30 and 50 respectively).

EXAMPLES OF READ AND DATA STATEMENTS

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

```
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

:_
```

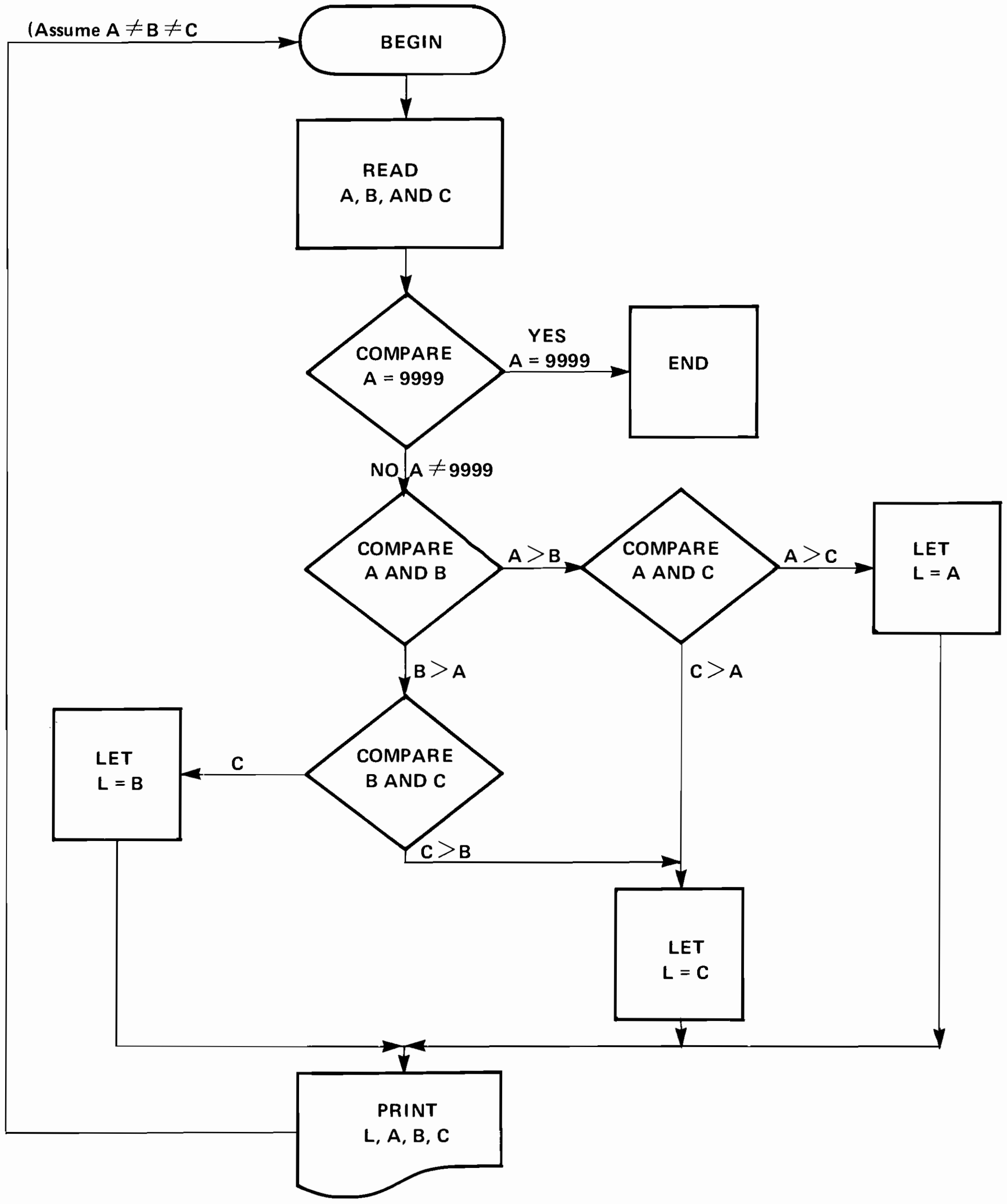
In this second example, exactly the same number of values stored are called for by the READ statement (line 10).

```
READY
: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

:_
```

If you have fewer values in a DATA statement (s) than are called for by the READ statement (s), the program stops when it runs out of data and prints out an error message (ERR 17) indicating that there was not enough data. You can request less data with a READ statement than there is in a DATA statement without obtaining an error. If you have only DATA statements without any READ statements, no error message is printed out, but there is no way of accessing the data.

DATA statements can appear anywhere in a program, before or after the READ statement(s). This is possible since the program is first scanned for all DATA statements, and any data is stored before the actual execution of the program is begun. A data pointer is set up to keep track of the data so that when a READ statement is executed, the proper data is used.

The flowchart and program on the next few pages represent the solution to the problem of determining which of three different given numbers is the largest. The values are inputted via READ and DATA statements. Notice in the program a common use of IF-THEN decision to test when the last piece of data is read. A value of 9999 is used as the last piece of data in the example and as each piece of data is read, it is tested to see if it equals 9999. When a data value of 9999 is read, the program ends; if not, the program continues.


(Assume A ≠ B ≠ C
BEGIN
READ
A, B, AND C
COMPARE
A = 9999
YES
A = 9999
END
NO
A ≠ 9999
COMPARE
A AND B
A > B
COMPARE
A AND C
A > C
LET
L = A
B > A
C > A
COMPARE
B AND C
C
LET
L = B
C > B
LET
L = C
PRINT
L, A, B, C

```
READY
:10 READ A,B,C
:15 IF A=9999 THEN 125
:20 IF A>B THEN 90
:30 IF B>C THEN 70
:40 L=C
:50 PRINT L,A,B,C
:60 GOTO 10
:70 L=B
:80 GOTO 50
:90 IF A>C THEN 110
:100 GOTO 40
:110 L=A
:120 GOTO 50
:125 END
:130 DATA 3,1,4,9,2,6,7,8
:140 DATA 2,7,6,5,1,2,3,9999,9999,9999
:RUN
 4               3               1               4
 9               9               2               6
 8               7               8               2
 7               7               6               5
 3               1               2               3

END PROGRAM
FREE SPACE=3214

:_
```

THE RESTORE STATEMENT

The previous examples show the READ statement(s) reading DATA values sequentially from DATA statements. Once the programs are executed, the data in the DATA statements cannot be re-used unless the programs are run again.

A method is available to allow the same data to be read more than once within a program. This is accomplished with the RESTORE statement.

The RESTORE statement resets the pointer, allowing the data in the DATA statements to be re-used without having to run the program again. The general form of the RESTORE statement is shown to the right.

The expression in the RESTORE statement is evaluated by the System 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 retrieves data beginning with the third data item stored. If the expression is omitted, the next READ statement retrieves data, starting with the first data item stored.

Consider the first program shown to the right. Statement line 10 reads the first four data values in statement 40 (i.e., 100, 200, 300, and 400). Statement line 20 restores the data pointer or sets it back to the first value (line 40). Statement 30 requests five values and the first five values in statement 40 are assigned to the variables (i.e., 100, 200, 300, 400 and 500). Then compare this program to the next program. Line 10 requests four values from data statement 40 (i.e., 100, 200, 300 and 400). Line 20 restores the data pointer to the third value (line 40). Line 30 requests five values and five values from statement 40, beginning with the third, are assigned to the variables (i.e., 300, 400, 500, 600, and 700).

GENERAL FORM

RESTORE [expression]

PROGRAM 1

```
READY
: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

:_
```

PROGRAM 2

```
READY
: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

:_
```

The RESTORE statement also can be used to skip over values in a DATA statement. Consider the program to the right. Notice the execution of statement line 20, RESTORE 5, causes the System 2200 to skip to the fifth data value for the subsequent READ statement in line 30.

```
READY
: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

:_
```

The RESTORE statement thus can 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 the last example program) results in an error message and termination of the program execution (see example).

```
READY
:10 READ M,N
:20 RESTORE 8
:30 READ O,P
:40 DATA 100, 200, 300, 400, 500, 600, 700
:50 PRINT M;N;O;P
:RUN

 30 READ O,P
         ↑ERR 27
:
```

Section 10-2
The INPUT Statement

With both the assignment and READ/DATA statements, all variable values are entered prior to running the program; that is, all variable values are contained within the program itself. Changing the values, in either case, means changing the program text. This is a valid approach for storing constants which may remain the same each time the program is executed. The INPUT statement allows a programmer to enter data after program execution has begun. The data inputted does not become a part of the program text.

Enter the program shown at the right into the System 2200 memory.

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

Line number 10 asks for two values (X and Y) to be inputted. Notice the variables names are separated by commas.

Touch RESET

Touch RUN CR/LF-EXECUTE

```
READY
:RUN
?
```

When a program with an INPUT statement is executed, the System 2200 continues executing the program line by line until program flow reaches the INPUT statement. The System 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.

Touch keys 3, 4 CR/LF-EXECUTE

When the CR/LF key is touched, program execution continues.

```
READY
:RUN
? 3,4
 3                9                4                16

END PROGRAM
FREE SPACE=3340

:_
```

If fewer than the required number of values (which in this case is two) are given before the CR/LF-EXECUTE key is touched, the System 2200 will continue printing "?"'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 shown to the right.

GENERAL FORM

INPUT ["character string",] variable [,variable . . .]

INPUT WITH AN INCLUDED TEXT STRING

The general form of the INPUT statement allows 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.

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 strokes.

Enter the program shown to the right.

```
10 INPUT "NEXT VALUE", S
20 REM TEST FOR NEG. VALUE
30 IF S<0 THEN 60
40 PRINT "SQUARE ROOTS OF";S;"ARE + AND-";SQR(S)
50 GOTO 10
60 PRINT "NO REAL ROOTS OF";S
70 GOTO 10
:
```

Line number 10 is an INPUT statement with a literal string. The program is set up as an infinite loop because the GOTO statements at lines 60 and 70 always direct program flow back to the first statement line.

Touch keys RUN CR/LF-EXECUTE

The program prints the literal string used to identify the input data.

Enter any value and continue execution of the program to see what happens.

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

Chapter 11

Arrays, and Array Variables

System 2200 BASIC allows for the definition of several types of variables. Up to now, you have been using only simple numeric variables called *numeric scalar variables*. Another type of variable that can be defined in the System 2200 is called the *array variable*.

Section 11-1
What Are Arrays?

An array is simply a set of numbers arranged in a table such that each number is uniquely identified by its position. In System 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.

The table to the right 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( ), where the value in the parentheses denotes the size of the array.

Notice each *element* of the R( ) array is associated with a *subscripted* letter in the table below it. The circled number 9 is in row 2 and also in column 2. Standard mathematical subscript notation indicates this by r_{22}.

In System 2200 BASIC a statement assigning the value of 9 to the element in the second row and second 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 System 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 can consist of either a single row or single column, as shown to the right.

$$\begin{array}{c|ccccc|}
\multicolumn{1}{c}{} & \multicolumn{5}{c}{\text{COLUMNS}} \\
\cline{2-6}
 & 7 & 6 & 5 & 4 & 9 \\
\text{R} & 1 & \textcircled{9} & 2 & 2 & 3 \\
\text{O} & 6 & 2 & 1 & 4 & 10 \\
\text{W} & 5 & 11 & 4 & 9 & 5 \\
\text{S} & 9 & \textcircled{10} & 6 & 1 & 5 \\
\cline{2-6}
\end{array}$$

R ( )
ARRAY

$$\begin{array}{|ccccc|}
\hline
r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\
r_{21} & \textcircled{r_{22}} & r_{23} & r_{24} & r_{25} \\
r_{31} & r_{32} & r_{33} & r_{34} & r_{35} \\
r_{41} & r_{42} & r_{43} & r_{44} & r_{45} \\
r_{51} & \textcircled{r_{52}} & r_{53} & r_{54} & r_{55} \\
\hline
\end{array}$$

$$\begin{bmatrix} r_1 \\ r_2 \\ \cdot \\ \cdot \\ r_i \end{bmatrix}$$

Column Array

$$[S_1 \quad S_2 . \; . \; . \; . \; . \; S_j]$$

Row Array

Section 11-2
Naming And Dimensioning Arrays

In System 2200 BASIC, *array variable* names are the same as simple (scalar) variable names except for the appropriate subscripts enclosed in parentheses. There are, therefore, 286 array names available (A-Z and A0-Z0). The subscripts contained in the parentheses describe a particular element position in the array (see examples to the right).

EXAMPLES OF LEGAL ARRAY NAMES

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

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 shown to the right.

GENERAL FORM

```
DIM  dim element   [ { ,dim element } . . . ]

                     ( numeric array variable            )
where  dim element = { alpha array variable¹   [integer] }   0 < integer < 65
                     ( alpha scalar variable¹  [integer] )
```

Examples of dimensioned numeric array variables are shown to the right.

DIM A(5,2)	Reserves space for a 2 dimensional array of 10 elements (5 X2)
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 respectively.
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.

[1] Alphanumeric variables are discussed in Chapter 12.

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 array 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, as shown in the program to the right.

```
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)
:_
```

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

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

Analysis of Problem:

Statement 10 - sets aside enough memory for the 100 values in array X( ).

Statement 20 - sets up a counter where I goes from 1 to 100. I is a variable used as a subscript in array X( ).

Statement 30 - assigns the value 5 to every element in the array X( ) where I goes from 1 to 100 (x_1, x_2, x_3, x_4, . . .x_{100}).

Statement 40 - increments the counter variable I, by one each time the program loops.

USING NESTED LOOPS TO DEFINE THE ELEMENTS OF A TWO-DIMENSIONAL ARRAY

Observe the program to the right. When the program is executed, the X( ) array is dimensioned as a 5 X 3 array and enough memory is set aside for the array.

Statement 20 sets up a FOR-TO/NEXT LOOP where the row position is given the name B, and a counter is set up from 1 to 5. Statement 30 sets up a FOR-TO/NEXT LOOP where the column position is given the name C and a counter is set up from 1 to 3. Execution of both these statements the first time results in the subscripts of array X( ) being assigned the values B=1 and C=1. Therefore, when statement 40 is executed, X_{BC} (or X_{11}) is set equal to 23. Statement 50 then is executed, which results in C being incremented by 1 and tested to see if the limit of the inner loop is satisfied. If not, statement 40 is executed and X_{BC} (now X_{12}) is set to 23. This continues until the inner loop is satisfied (i.e., $X_{BC} = X_{13}$) then statement 60 is executed. Since this is part of the outer loop, B is incremented by 1. The inner loop is processed completely each time the outer loop is processed once. In this way, each element, row by row (i.e., X_{11}, X_{12}, X_{13}, X_{21}, X_{22}, X_{23}, X_{31}, X_{32}, and X_{33}) is assigned the value of 23.

```
READY
:10 DIM X(5,3)
:20 FOR B=1 TO 5
:30 FOR C=1 TO 3
:40 X(B,C)=23
:50 NEXT C
:60 NEXT B
:70 END
:_
```

Chapter 12

Alphanumeric String Variables

A third type of variable can be defined in System 2200 BASIC, namely the alphanumeric string variable. An alphanumeric string variable is a variable with a value made up of a group of alphanumeric characters. An alphanumeric character in BASIC is any numeric digit, letter, and special character, such as +, –, ↑, (,), <, >, %, $, etc. (i.e., any printable character). In the System 2200 there are non-printable command codes associated with the various peripherals. These command codes can be a part of the value of alphanumeric string variables.

Section 12-1
String Variable - Names and Characteristics

NAMES

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 possible string variable names.

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

LEGAL STRING VARIABLE NAMES

A4$	W3$
X$	Z6$

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 in a DIM statement, the maximum number of alphanumeric characters a string variable can assume is 16; if specified, the size can range from 1 to 64. This compares to the maximum number of 13 digits a numeric variable can assume. If an attempt is made to assign a literal string of greater length than 16 to a string variable, without dimensioning it, the additional characters are simply ignored.

Section 12-2
Assigning Values To String Variables

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

Some examples of using string variables with assignment statements and READ/DATA statements are given to the right.

ASSIGNMENT STATEMENT

```
V$="JOE SMITH"

F$="MAPLE STREET"

X4$="G542H-16#"

W$="152,760"
```

READ AND DATA STATEMENTS

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

The situation with the INPUT statement is somewhat different. Alpha characters need not be included in quotation marks. However, if they are not, commas and carriage returns act as string terminators and leading spaces are ignored. Thus, if commas or leading spaces are to be included as part of the literal string in the INPUT statement, the string must be included in double quotes.

The example to the right shows the results of responding to an INPUT request with quotes and without quotes. Notice in the case without quotes, only the first two parts, as denoted by commas, are picked up. The rest of the data is ignored.

WITH QUOTATION MARKS

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

:_
```

WITHOUT QUOTATION MARKS

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

:_
```

Section 12-3 Using String Variables

Once a string variable is given a value, it may be used with all the relational operators (=, >, <, ⩾, ⩽, and <>).

EXAMPLES OF USING STRING VARIABLES WITH RELATIONAL OPERATORS

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

:_
```

However, string variables and strings *cannot* be used with arithmetic operators (i.e., +, –, *, /, ↑).

ILLEGAL USE OF STRINGS

PRINT C$, C$ ↑ 2	— Strings cannot 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 cannot be added numerically regardless of what characters they represent.

ALPHANUMERIC ORDERING

Two string variables are compared by the *relative* alphanumeric characters composing them. The ordering is given in the table to the right.

Notice the letters of the alphabet are ordered as expected (A < B <C . . . Z). Also notice the numerals 0 thru 9 are as expected, and numerous symbols fall throughout. A space has the lowest order and the ↑ has the highest order. When a comparison of strings is made, the strings are compared on a character-by-character basis.

Short strings are filled out with trailing spaces to allow for comparisons with longer strings. These trailing spaces have no effect otherwise in subsequent operations.

ALPHANUMERIC ORDERING TABLE

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

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 > –

"X–" > "X" because – > (space) ("X" is filled out with trailing spaces).

The program to the right shows a commonly used programming technique, in which string variables aid in the creation of conversational programming. Statement 10 is an INPUT statement with a literal string and is used to indicate to the user that a number is required. The inputted data is then assigned to a numeric variable. Statement line 20 asks whether the user desired the SQUARE or the CUBE of the number previously entered, and assigns the response to a string variable. Statement line 30 checks to see if the request is 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 goes 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 eventually to loop back and continue requesting.

Enter the program on the previous page:

Touch RUN CR/LF-EXECUTE

Touch 5 CR/LF-EXECUTE

Touch keys SHIFT LOCK SQUARED
SHIFT CR/LF-EXECUTE

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": GOTO 20
50   P = 2: GOTO 70
60   P = 3
70   PRINT X; N$; "="; X↑P: PRINT
80   GOTO 10
```

```
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? _
```

Section 12-4
Dimensioning String Variables

In Section 12-1, it was mentioned that the standard size of a System 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 unless specified in a DIM statement.

System 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 the line to the right 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
```

The use of the DIM statement with string variables differs from its use as described in chapter 11, in that the string variable still only represents a single "value". In this case, however, the "value" in terms of absolute size is changed.

The statements to the right show the results of using a DIM statement to alter the maximum length of a string variable.

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 specified number of characters.

```
READY
:A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"

:PRINT A$
ABCDEFGHIJKLMNOP

:DIM B$5,C$20

:B$,C$="1234567890ABCDEFGHIJKLMN"

:PRINT B$,C$
12345           1234567890ABCDEFGHIJ

:_
```

STRING ARRAYS

Arrays of string variables also can 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.

In the first DIM statement to the right, 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-dimensional 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 (X3$( )) of five rows and five columns, in which each element can assume up to fifty characters.

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

DIM X3$(5,5)50
```

Section 12-5
The STR(Function

Although string variables cannot be added,[1] 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 general form of the STR(function is shown to the right.

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.)

[1]System 2200B language features enable a user to add string variables with the ADD statement.

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

STR(A$,3,4) means

starting with the third character of A$, take four characters (i.e., the 3rd, 4th, 5th and 6th characters).

STR(A$,3) means

starting with the third 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.

The program shown to the right is a programming example using the STR(function. As the REM statements indicate, the credit card number entered via the INPUT statement in line 50 is checked for membership year and credit rating. Both pieces of information are "imbedded" in the credit card number.

ZZZZZ is entered as the last name processed. The ZZZZZ is test data, checked 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 INPUT 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 R
ATING"
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
:
```

Enter this program.

Touch RESET RUN CR/LF-EXECUTE

Touch keys SHIFT LOCK "DOE SHIFT, SHIFT LOCK JOHN" SHIFT CR/LF-EXECUTE

Touch keys 8 X 36-41-68379A-8 CR/LF-EXECUTE

Continue to enter the names and credit card numbers as shown in the output. Enter ZZZZZ when you wish to end the program.

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

```
READY
: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=2976

:_
```

Section 12-6
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 when working with alphanumeric computations.

The function is generated by touching keys L E N ( ). The general form is shown to the right.

GENERAL FORM

LEN (string variable)

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

Several examples of the LEN(function are shown to the right.

NOTE:

When a string is defined to contain all spaces, requesting the LEN( ) of the string always gives a value of one. You might expect the value to be zero, since the string is "empty," but by definition the LEN( ) function counts all characters until a trailing space is found. In this instance, the first space is considered a character and the next space is considered a trailing space, therefore the value of one.

```
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
:_
```

```
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 GOTO 10
:50 PRINT N$;" IS A LONG NAME."
:60 GOTO 10
:RUN
WHAT IS YOUR NAME? MICHAEL
MICHAEL IS A LONG NAME.
WHAT IS YOUR NAME?
```

Chapter 13

Use Of The COM (Common) Statement

Situations exist where programs too long for memory are segmented and stored on some external storage device (e.g., Tape Cassettes). They are then loaded into memory one segment at a time and executed. This is called program chaining and is a commonly used programming technique.

Ordinarily, as one segment is loaded any data or program text from the previous segment is lost. Variables which are common to two or more segments of the program must be retained from one segment of the program to the next. It would be extremely inefficient, for example, to generate data in segment one, needed in segments three and four, and have to manually re-enter this data as segments three and four are run. It is the COM (Common) statement which allows a programmer to designate data in one segment of a program as common which, in effect, keeps the data intact for subsequent program segments while program chaining occurs.

Chapter 13
Use Of The COM (Common) Statement

NOTE:

An in .depth discussion of Tape Cassette operations and the BASIC statements associated with the cassette can be found in the System 2200 Reference Manual and the Model 2217/2218 Tape Cassette Reference Manual. Statements are used in this chapter which have not been previously discussed in this text (e.g., LOAD). Please refer to the above mentioned manuals if you have questions.

Another programming technique associated with the use of the COM statement is the loading of subroutines from a cassette into a program resident in memory. Here data from the main program may be needed when the subroutine is executed, or data generated by the execution of a subroutine may be needed by the main program. In either case, the COM statement is used to keep the data intact as the subroutines are loaded and cleared from memory.

Both of these programming techniques, as used with the COM statement, are illustrated in this chapter.

Chapter 13
Use Of The COM (Common) Statement

Section 13-1
What Does The COM Statement Do?

The COM statement sets aside a segment of memory and stores in this section of memory any variables designated as common. As a result, the variables remain intact as programs and non-common variables are cleared from memory. Common variables are cleared from memory only when one of the following procedures are followed:

1) The operator executes a CLEAR CR/LF-EXECUTE command. This clears the *entire* memory.[1]
2) The operator executes a CLEAR V CR/LF-EXECUTE command. This clears *all* variables from memory (common and non-common variables).[1]
3) The operator Master Initializes the system. This clears the *entire* memory.[1]

[1]Common variables cannot be cleared under program control as none of these statements are programmable.

The introduction to this chapter described the COM statement being used in conjunction with Tape Cassette operations. In general, any information which is stored on a Tape Cassette is read back into memory via a LOAD[1] statement. When a LOAD statement is executed several things occur:

1) All program execution stops.
2) All program text and non-common variables (those variables not designated as common) are cleared from memory.
3) The previously recorded program, is loaded into memory.
4) The new program is executed.

Any data designated as Common remains intact at the end of this procedure.

Section 13-2 The COM Statement As Used In Program Chaining

An example is used to illustrate the use of the COM statement with program chaining. In this example, a single program is broken down into three segments and stored on a Tape Cassette. Each segment is uniquely identified with a name, e.g., "Part #1", "Part #2" and "Part #3". These names are used to locate the segment desired on the tape. The program segments are shown to the right.

EXPLANATION OF PROGRAM

Statement 10 of "Part #1" is a COM statement which designates the variables X, Y and A(10) as common. This statement serves to keep the specified variables in memory when statement 90 of "Part #1" is executed.

PART #1

```
10 COM X,Y,A(10),N$25
20 INPUT "X,Y=",X,Y
30 FOR I=1 TO 10
40 A(I)=X*I
50 NEXT I
60 INPUT "NAME=",N$
70 Y=LEN(N$)
80 PRINT : PRINT "LOADING NEXT SEGMENT"
90 LOAD "PART #2"
```

[1]The LOAD statement is described in the System 2200 Reference Manual and Model 2217/2218 Tape Cassette Reference Manual.

Statement 90, when executed, stops all program execution, clears all program text and non-common variables from memory, loads in "Part #2" and executes this second segment.

Statement 15 of segment two is another COM statement which adds the array A( ) to the common data already in memory.

When statement 75 is executed, all common data from segments 1 and 2 are available to segment 3.

Key this program into memory, store it on a cassette, load it back into memory and execute it. The necessary steps are as follows:

I. TO ENTER AND STORE THE PROGRAM:

a. Clear memory with a CLEAR CR/LF-EXECUTE.
b. Insert a Tape Cassette into the tape drive.
c. Rewind the tape. Touch the REWIND button.
d. Key in statements 10-90 of Part 1 exactly as shown.
e. Save this on tape. Touch keys SAVE "PART #1" CR/LF-EXECUTE.
f. The first segment is now on tape. DO NOT REWIND THE TAPE.

PART #2

```
15 COM A(10)
25 PRINT "DATA FROM FIRST SEGMENT":PRINT
35 PRINT "X=";X,"Y=";Y
45 PRINT "N$=";N$
55 FOR I=1 TO 10:PRINT A(I);:NEXT I:PRINT
65 REM A() PASSED TO NEXT SEGMENT
75 LOAD "PART #3"
```

PART #3

```
100 DIM B(10)
110 FOR I=1 TO 10
120 B(I)=A(I)+1
130 PRINT "A()=";A(I),"B()=";B(I)
140 NEXT I
150 END
```

```
 CLEAR

READY
:10 COM X,Y,A(10),N$25
:20 INPUT "X,Y=",X,Y
:30 FOR I=1 TO 10
:40 A(I)=X*I
:50 NEXT I
:60 INPUT "NAME=",N$
:70 Y=LEN(N$)
:80 PRINT: PRINT "LOADING NEXT SEGMENT"
:90 LOAD "PART #2"
:SAVE "PART #1"
```

g. Clear memory with a CLEAR CR/LF-EXECUTE.
h. Key in statements 15-75 of Part 2 exactly as shown.
i. Save this on tape. Touch keys SAVE "PART #2" CR/LF-EXECUTE.

   The second segment is now on tape. DO NOT REWIND THE TAPE.
j. Clear memory with a CLEAR CR/LF-EXECUTE.
k. Key in statements 100-150 of Part 3 exactly as shown.
l. Save this on tape. Touch keys SAVE "PART #3" CR/LF-EXECUTE.

   Segment 3 is now on tape.

m. REWIND THE TAPE. Touch the REWIND BUTTON.
n. CLEAR memory with a CLEAR CR/LF-EXECUTE.

```
 CLEAR

READY
:15 COM A(10)
:25 PRINT "DATA FROM FIRST SEGMENT":PRINT
:35 PRINT "X=";X,"Y=";Y
:45 PRINT "N$=";N$
:55 FOR I=1 TO 10:PRINT A(I);:NEXT I:PRINT
:65 REM A() PASSED TO NEXT SEGMENT
:75 LOAD "PART #3"
:SAVE "PART #2"
```

```
:CLEAR

READY
:100 DIM B(10)
:110 FOR I=1 TO 10
:120 B(I)=A(I)+1
:130 PRINT "A()=";A(I),"B()=";B(I)
:140 NEXT I
:150 END
:SAVE "PART #3"
:
```

II. TO EXECUTE THIS PROGRAM:

1) Load back into memory the first segment. Touch LOAD "PART #1" CR/LF-EXECUTE.

   The first segment is now in memory.

2) Execute this segment. Touch RUN CR/LF-EXECUTE.
3) Input the values for X and Y. Touch keys 5, 3 CR/LF-EXECUTE.
4) Input a name. Touch keys HARVEY SMITH CR/LF-EXECUTE.

Once the first segment is loaded and run, the remaining segments are automatically loaded by the program.

The results of the program are shown to the right. Data common to all three segments of the program is retained from segment to segment due to the COM statement.

```
CLEAR

READY
:LOAD "PART #1"
:RUN
X,Y=? 5,3
NAME=? HARVEY SMITH

LOADING NEXT SEGMENT
DATA FROM FIRST SEGMENT

X= 5            Y= 12
N$=HARVEY SMITH
 5  10  15  20  25  30  35  40  45  50
A()= 5          B()= 6
A()= 10         B()= 11
A()= 15         B()= 16
A()= 20         B()= 21
A()= 25         B()= 26
A()= 30         B()= 31
A()= 35         B()= 36
A()= 40         B()= 41
A()= 45         B()= 46
A()= 50         B()= 51

END PROGRAM
FREE SPACE=3070

:_
```

Section 13-3
Using The COM Statement With Chained Subroutines

An example also is used in this section to illustrate the second major use of the COM statement; the COM statement with subroutines. In this example, three subroutines are recorded on tape and are loaded from the cassette into memory in the midst of a program already resident in memory. In order to do this, the LOAD statements must specify exactly where in the resident program the subroutine is to be placed; that is, what part of the resident program text (if any) is to be cleared when the LOAD statement is executed. As mentioned earlier, the LOAD statement, when executed, clears memory of all program text. But an option of the LOAD statement can be used to specify only a portion of the program text is to be cleared. This is illustrated in the following example.

A listing of resident program is shown to the right.

Since the LOAD statement clears memory of all non-common variables, all variables to be used after the first LOAD statement is executed must be specified as common data (see statement 10). Statement 50 instructs the system to remove from memory all program text from lines 500 through 700. The remainder of the program remains intact. The system then locates the program named "VOLUME" on the cassette and loads it into memory.

Program Resident in Memory

```
10 COM L1,L2,L3,N
20 INPUT "LENGTH=",L1
30 INPUT "WIDTH=",L2
40 INPUT "HEIGHT=",L3
45 N=1
50 LOAD "VOLUME" 500,700
60 REWIND
65 N=2
70 LOAD "AREA" 500,700
80 REWIND
85 N=3
90 LOAD "DIAGONAL" 500,700
100 REWIND
110 END
500 REM DIAGONAL OF RECTANGULAR SOLID
550 D=SQR(L1↑2+L2↑2+L3↑2)
600 PRINT TAB(10);"DIAGONAL=",D
650 PRINT
700 REM END OF SUBROUTINE
1000 IF N=1 THEN 60
1010 IF N=2 THEN 80
1020 IF N=3 THEN 100
```

The first line number (500) specified in the LOAD statement must be present in the subroutine, since it is at this line number that execution of the subroutine begins. The second line number in the LOAD statement (700) is necessary to limit the amount of text that is removed from the main program prior to the loading of the subroutine. Statements 70 and 90 in the main program are used to load in the second and third subroutines and again clear memory of statements 500 through 700 inclusive.

The three subroutines to be stored on the cassette are shown to the right. Enter the first subroutine into memory and store it on a Tape Cassette.

1. Clear Memory with a CLEAR CR/LF-EXECUTE.
2. Place a cassette in the tape drive.
3. Rewind the cassette. Touch the REWIND button.
4. Key in the first subroutine.
5. Save this on tape. Use the name "VOLUME" to identify the subroutine.

   Touch SAVE "VOLUME" CR/LF-EXECUTE

   The first subroutine is now on tape. DO NOT REWIND THE TAPE.

Subroutine #1

```
CLEAR

READY
:500 REM VOLUME OF RECTANGULAR SOLID
:510 V=L1*L2*L3
:520 PRINT: PRINT "LENGTH=";L1,"WIDTH=";L2,"HEIGHT=";L3
:530 PRINT
:540 PRINT TAB (10);"VOLUME=",V
:700 REM END OF SUBROUTINE
:SAVE "VOLUME"
```

6. Clear Memory with a CLEAR CR/LF-EXECUTE.
7. Key in the second subroutine.
8. Save this on tape with the name "AREA"

   Touch SAVE "AREA" CR/LF-EXECUTE

   The second subroutine "AREA" is now on tape. DO NOT REWIND THE TAPE.

9. Clear Memory - CLEAR CR/LF-EXECUTE.
10. Key in the third subroutine.
11. Save this on tape with the name "DIAGONAL"
    Touch SAVE "DIAGONAL" CR/LF-EXECUTE

    The third subroutine is now on tape.

12. Rewind the tape. Touch the REWIND button.

Subroutine #2

```
CLEAR

READY
:500 REM SURFACE AREA OF RECTANGULAR SOLID
:515 A=2*(L1*L2+L2*L3+L1*L3)
:525 PRINT TAB(10);"SURFACE AREA=",A
:700 REM END OF SUBROUTINE
:SAVE "AREA"
```

Subroutine #3

```
CLEAR

READY
:500 REM DIAGONAL OF RECTANGULAR SOLID
:550 D=SQR(L1↑2+L2↑2+L3↑2)
:600 PRINT TAB(10);"DIAGONAL=",D
:650 PRINT
:700 REM END OF SUBROUTINE
:SAVE "DIAGONAL"
```

Now key in the resident program.

1. Clear Memory
2. Key in program

   This program, when run, will automatically load in the three subroutines from the cassette.

1. Touch RESET
2. Touch RUN CR/LF-EXECUTE
3. Enter a value for length. Touch keys 24 CR/LF-EXECUTE.
4. Enter a value for width. Touch keys 20 CR/LF-EXECUTE.
5. Enter a value for height. Touch keys 8 CR/LF-EXECUTE.

After each subroutine is loaded and executed, program statement 1000 in the resident program is the next statement to be executed. Statements 1000, 1010, and 1020 direct program flow to the correct statement in the resident program.

```
CLEAR

READY
:10 COM L1,L2,L3,N
:20 INPUT "LENGTH=",L1
:30 INPUT "WIDTH=",L2
:40 INPUT "HEIGHT=",L3
:45 N=1
:50 LOAD "VOLUME" 500,700
:60 REWIND
:65 N=2
:70 LOAD "AREA" 500,700
:80 REWIND
:85 N=3
:90 LOAD "DIAGONAL" 500,700
:100 REWIND
:110 END
:500 REM START OF SUBROUTINE AREA
:700 REM END OF SUBROUTINE AREA
:1000 IF N=1 THEN 60
:1010 IF N=2 THEN 80
:1020 IF N=3 THEN 100

READY
:RUN
LENGTH=? 24
WIDTH=? 20
HEIGHT=? 8

LENGTH= 24       WIDTH= 20        HEIGHT= 8

          VOLUME=                  3840
          SURFACE AREA=            1664
          DIAGONAL=                32.249030993

END PROGRAM
FREE SPACE=2962

:_
```

Section 13-4
The General Form Of The COM Statement

The general from of the COM statement is shown to the right.

The COM statement allows any type of variable to be defined as common, as well as any number of variables.

Some examples of the COM statement also are shown to the right.

GENERAL FORM

COM com element [,com element . . .]

where

$$\text{com element} = \left\{\begin{array}{l}\text{numeric scalar variable}\\ \text{numeric array variable}\\ \text{alpha scalar variable [integer]}\\ \text{alpha array variable [integer]}\end{array}\right\}\left[\left\{\begin{array}{l}\text{same}\\ \text{same}\\ \text{same}\\ \text{same}\end{array}\right\}\cdots\right]$$

$0 < \text{integer} < 65$

SOME EXAMPLES OF COM STATEMENTS:

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
```

Chapter 14

PRINTUSING And IMAGE Statements

The PRINT statement, in conjunction with commas, semicolons and the TAB(function, has been used to control the format (spacing) within a printed line. An additional two statements are available in System 2200 BASIC which allow a programmer more control over the spacing within a printed line, and also enable a programmer to pre-determine the exact image of the individual elements printed within a line. The statements are PRINTUSING and IMAGE. They always are used together.

Section 14-1
Printing Of Alpha Fields (Literal Strings)

In order to print strictly alpha fields with a PRINT statement, alpha fields must be enclosed with quotes. Spacing within quotes is followed exactly. Any spacing preceding or following the alpha fields is controlled with punctuation marks. An example is shown to the right.

```
READY
:10 PRINT, "ALPHA OUTPUT"
:RUN
                ALPHA OUTPUT

:_
```

The same output can be generated without using quotes or punctuation marks with the PRINTUSING/ IMAGE statements, as illustrated to the right.

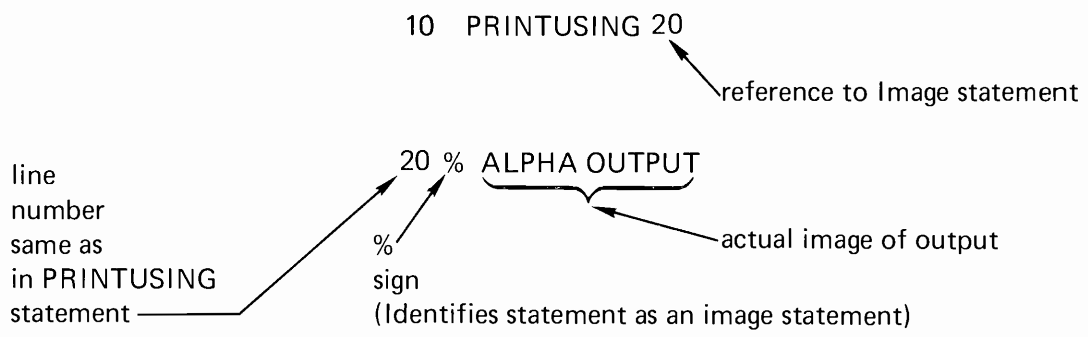
Notice the PRINTUSING statement works in conjunction with a referenced IMAGE statement. The IMAGE statement contains the exact image and format of what is to be printed. No quotes or punctuation marks are used.

The IMAGE statement is referenced in the PRINTUSING statement by placing the line number of the IMAGE statement directly after the PRINTUSING verb.

The IMAGE statement consists of a line number, (must be the same referenced in the PRINTUSING statement), a % sign, which marks it as an IMAGE statement, and the image of the elements and spacing to be followed. When alpha fields are included in the IMAGE statement, they are printed exactly as shown.

Generally, this method is used to label output or to display only alphanumerics. Some other examples are shown to the right.

```
READY
:10 PRINTUSING 20
:20%                 ALPHA OUTPUT
:RUN
                 ALPHA OUTPUT

:_
```


10 PRINTUSING 20
reference to Image statement
20 % ALPHA OUTPUT
line number same as in PRINTUSING statement
% sign (Identifies statement as an image statement)
actual image of output

```
READY
:100 PRINTUSING 110
:110% PROFIT AND LOSS STATEMENT
:RUN
 PROFIT AND LOSS STATEMENT

:_
```

```
READY
:20 PRINTUSING 30
:30%    INTEREST    YEAR    TOTAL TO DATE
:RUN
    INTEREST    YEAR    TOTAL TO DATE

:_
```

An alternative method exists for printing literal strings with the PRINTUSING/IMAGE combination. The method places the literal string in the same line as the PRINTUSING statement and uses symbols in the IMAGE statement to represent the characters in the literal string. The # symbol is primarily used to represent each character, but certain editing characters (i.e., $ – + , and .) also can be used in the image to represent a character. The symbols are mainly place holders and the system replaces each symbol in the image with an alpha character when a printout is generated.

The statements used to print the words *alpha output* by this second method are shown to the right.

```
READY
:20 PRINTUSING 30, "ALPHA OUTPUT"
:30 %############
:RUN
ALPHA OUTPUT

:_
```

Notice the literal string is enclosed in quotes in statement 20, and a comma is used to separate the elements to be printed from the beginning of the line. Quotes are required when the literal string is contained within the PRINTUSING statement. Statement 30 (the IMAGE statement) contains a symbolic representation of how the literal string should be printed out. Each symbol represents a character. There are 12 characters and 12 symbols, therefore all characters in the literal string are printed.

If fewer symbols are included in the IMAGE statement than are in the literal string, the string is truncated on the right and only the number of characters as specified by the image are printed. See the example to the right. In this example only five characters are printed as the image only calls for five characters.

```
READY
:20 PRINTUSING 30, "ALPHA OUTPUT"
:30 % #####
:RUN
 ALPHA

:_
```

If the image has more symbols than characters in the literal string, the printout is left justified and filled out with blanks. See the example to the right. In this example 12 characters plus two trailing spaces are printed as the image calls for 14 characters.

When the PRINT command is used to print a literal string, there is no way of truncating or lengthening the printout without actually changing the literal string. The PRINTUSING and IMAGE statements, however, can truncate or lengthen a printout by simply overformatting or under formatting the image.

```
READY
:20 PRINTUSING 30, "ALPHA OUTPUT"
:30% ##############
:RUN
 ALPHA OUTPUT__
                         2 Trailing Spaces
:_
```

Section 14-2
Printing Alphanumeric String Variables

Alphanumeric String Variables are printed with the PRINT statement as shown to the right. This string cannot be truncated or lengthened without actually defining a new value for A$.

The PRINTUSING/IMAGE statements, however, allow a programmer to shorten or lengthen (add spaces) the string variable by specifying the desired image in the IMAGE statement. See the example to the right.

Notice four characters are printed for A$ the first time, and five characters are printed the second time. The # symbol specifies how many characters are to be printed; any spacing entered into the image line is followed exactly. No punctuation is used.

```
READY
:10A$ = "ABCDEFG"
:20 PRINT A$,A$
:RUN
ABCDEFG          ABCDEFG

:_
```

```
READY
:10A$ = "ABCDEFG"
:20 PRINTUSING 30, A$,A$
:30% ####    #####
:RUN
 ABCD    ABCDE

:_
```

REUSING AN IMAGE

A single image in the IMAGE statement can be reused for each element in the PRINTUSING statement (example to the right).

The image in line 30 is used to print A$ three times. The comma used in the PRINTUSING statement causes the output to appear on a new line each time the IMAGE statement is reused.

If semicolons are used to separate the elements in the PRINTUSING statement, the output continues on the same line (see example).

```
READY
:10A$ = "ABCDEFG"
:20 PRINTUSING 30, A$,A$,A$
:30% ####
:RUN
 ABCD
 ABCD
 ABCD

:_
```

```
READY
:10A$ = "ABCDEFG"
:20 PRINTUSING 30, A$;A$;A$
:30% ####
:RUN
 ABCD ABCD ABCD

:_
```

An example is shown to the right which combines the printing of string variables and literal strings with the PRINTUSING/IMAGE statement.

Notice the second and third image results in the second and third elements being truncated when printed.

```
READY
:10 A$="ABCDEFG"
:20 PRINTUSING 30, A$,A$, "STUVWXYZ"
:30 %#######    #####    ####
:RUN
ABCDEFG    ABCDE    STUV

:_
```

This same example could be written differently by including the literal string in the IMAGE statement (see example).

Either way is correct, but by including the literal string in the IMAGE statement, you cannot edit the string as you did in the first example.

```
READY
:10A$ = "ABCDEFG"
:20 PRINTUSING 30, A$,A$
:30% #######  $#.##  STUV
:RUN
 ABCDEFG  ABCDE  STUV

:_
```

Section 14-3
Printing Numeric Fields

When printing numerics with the PRINTUSING/IMAGE statements, only the # symbol can be used to represent a digit. A period is used to represent a decimal point. In the example to the right, the number 123.45 is printed four times, each with a different image.

```
READY
:10 PRINTUSING 20, 123.45, 123.45, 123.45, 123.45
:20% ###  ###.#  ###.##  ###.###
:RUN
 123  123.4  123.45  123.450

:_
```

The first image (###) calls for three digits. The printout of 123.45 is therefore truncated to fit the image. Extra digits to the *right* of the decimal point not called for in the image are truncated.

123.45 in the image ### $\longrightarrow$ 123

The second image (###.#) calls for three digits, a decimal point and two more digits. The printout of 123.45 is exactly as shown.

The third image (###.##) calls for three digits, a decimal point and two more digits. The printout of 123.45 is exactly as shown.

The fourth image (###.###) calls for three digits, a decimal point and three more digits. Images which contain more # symbols to the *right* of the decimal than are present in the data in the PRINTUSING statement cause the output to be filled out with zeros.

As can be seen from this example, data formats in the IMAGE statement are read sequentailly and matched to the data supplied in the PRINTUSING statement.

OVERFORMATTING AN IMAGE

If an IMAGE contains more # symbols to the left of the decimal than digits to the *left* of the decimal in the data, the unnecessary leftmost # signs are ignored and leading blanks are inserted (see example).

Here the data (627.6) is formatted in the image ####.#, which calls for more digits to the left of the decimal than are in the data. As a result the leftmost # sign is ignored.

123.45 in the image ###.# $\longrightarrow$ 123.4

123.45 in the image ###.## $\longrightarrow$ 123.45

123.45 in the image ###.### $\longrightarrow$ 123.450 $\leftarrow$ added on

OVERFORMATTING

```
READY
:10 PRINTUSING 20, 627.6
:20% ####.#
:RUN
 627.6
```
one leading blank inserted
```
:_
```

627.6 in the image $\underset{\text{ignored}}{\boxed{\#}}$ ###.# $\longrightarrow$ 627.6

UNDERFORMATTING AN IMAGE

If an image contains fewer # symbols than there are digits to the *left* of the decimal in the data, the # signs are printed instead of the data (see example). This indicates you do not have a sufficiently large image to express the value of the numbers.

When the exact size of a number is not known, it is best to overformat the image.

UNDERFORMATTING

```
READY
:10 PRINTUSING 30, 958.2
:30% ##.#
:RUN
 ##.#

:_
```

Section 14-4
Including Editing Characters In Numeric Fields

When printing numeric fields, it is understood that # symbols are always used to represent digits and a period is used to represent the placement of a decimal point. Several other characters can be entered into numeric fields via the IMAGE statement. These characters are the comma, a plus or minus sign, a $ sign and an E for Scientific Notation. Certain rules apply when using these characters and are explained in this section.

COMMAS IN THE IMAGE STATEMENT

A comma is often edited into a printout to improve its readability. An example is shown to the right which includes commas in the IMAGE statement.

There is no way of editing a comma into a numeric field with a PRINT statement.

```
READY
:100 PRINTUSING 150, 1362594, 3726.59
:150 %#,###,###.##    #,###.##
:RUN
1,362,594.00    3,726.59

:_
```

USING PLUS OR MINUS SIGNS IN AN IMAGE STATEMENT

If a plus sign is placed at the beginning of an image, the correct sign (plus or minus) is always edited into the output preceding the first significant digit in the output (see example).

In this example the image is reused for each element in the PRINTUSING statement. The image calls for a plus or minus sign to be edited into the output. Notice the appropriate sign is edited into the output just preceding the first significant digit. Notice also, the decimal points are lined up one above the other, due to an image being used which overformats most of the numbers both to the left and right of the decimal point. Recalling the rules, any image which overformats to the left of the decimal results in the insertion of leading blanks; any which overformats to the right of decimal results in the filling out with zeros. The number –4.10 fits both cases: two leading blanks and one trailing zero are edited into the output.

If an image begins with a minus sign, the minus sign for a negative expression is edited into the output just preceding the first significant digit. No sign is included if the number is positive (see example).

```
READY
:10 PRINTUSING 20, 15.62, -158.936, 352, -4.1
:20% +###.##
:RUN
  +15.62
 -158.93
 +352.00
   -4.10

:_
```

```
READY
:30 PRINTUSING 40, 15.62, -158.936, 352, -4.1
:40% -###.##
:RUN
   15.62
 -158.93
  352.00
   -4.10

:_
```

If no sign is used in the image, no problem results if all the numbers are positive. However, if negative numbers are to be outputted the minus sign is entered into the printout and the entire number is shifted to the right (see example).

It is recommended if negative numbers are to be printed out using the PRINTUSING/IMAGE statements, a sign (plus or minus) must be used in the image line.

USING $ IN AN IMAGE STATEMENT

When a $ (dollar sign) is used at the beginning of an image, the dollar sign is printed in the output:

1) immediately preceding the first significant digit if the number is positive (see example), or
2) immediately preceding the minus sign if the number is negative (see example).

```
READY
:10 PRINTUSING 20, 57.25, -57.25, 326.1, -326.1, -859
:20% ###.##
:RUN
  57.25
 - 57.25
 326.10
 -326.10
 -859.00

:_
```

```
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

:_
```

SCIENTIFIC NOTATION IN THE IMAGE STATEMENT

Four ↑ 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 shown to the right.

$$\underset{\text{E}}{\uparrow} \qquad \underset{\text{Sign}}{\uparrow} \qquad \underbrace{\uparrow \qquad \uparrow}_{\text{Exponent}}$$

The example to the right illustrates the use of this field in printing modified Scientific Notation. Notice the automatic adjustment of the exponent according to the placement of the decimal in the image.

```
READY
:100 PRINTUSING 150, 5.376E8, 2.13E-5, 2.6E-9
:150% #.### ↑↑↑↑ .###↑↑↑↑  ##↑↑↑↑
:RUN
 5.376E+08 .213E-04  26E-10

:_
```

Section 14-5
The General Form Of The PRINTUSING And IMAGE Statements

Due to the complexity and the many operations available with the PRINTUSING/IMAGE statements, the discussion of the general form of these statements has been left to the end of the chapter.

At the right is the general form of the PRINTUSING statement. The PRINTUSING statement always consists of the PRINTUSING verbs plus the line number of the corresponding IMAGE statement. The remainder of the statement is optional and can consist of one or more or a combination of the following print elements:

1) alphanumeric variables,
2) literal string in quotes,
3) numeric variables,
4) numerics.

GENERAL FORM PRINTUSING STATEMENT

PRINTUSING 'line number' [, 'print element't . . .] [;]

where 'line number'	=	line number of the corresponding IMAGE statement
'print element'	=	alphanumeric variable
		literal string in quotes
t	=	comma or semicolon

A comma must be used to separate the print elements from the preceding part of the line, and a comma or semicolon must be used to separate each print element.

The general form of the IMAGE statement is shown to the right.

The IMAGE statement is designated by the % sign. It can contain a literal string without quotes (t), an alphanumeric image made of # symbols or any of the editing symbols, or the format image of a numeric variable or numeric (f).

GENERAL FORM IMAGE STATEMENT

% t [$\left\{ft\right\}$ . . . .]

where t = a literal string (not containing # character) or blank

f = format specification = $\left[\begin{matrix}+\\-\\\$\end{matrix}\right]$ [#[,] . . . [. # . . .]] [↑↑↑↑]

Section 14-6
Arrays With PRINTUSING

The PRINTUSING format can be used to generate the output in an array. The program at the right and resulting output is an example of a program which both defines and prints out a 4 x 4 array with each element in the array printed to a pre-set image (statements 60 and 70). Statement 90 causes just a carriage return without anything being printed.

SUMMARY

As you can see from this chapter, the PRINTUSING format is a powerful tool for producing output in almost any desired image. The examples in this chapter illustrate individual features of PRINTUSING, but any of the features can be combined in a single PRINTUSING/IMAGE statement.

```
READY
:10 DIM F2(4,4)
:20 N=.005
:30 FOR I=1TO 4
:40 FOR J=1TO 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
:_
```

Chapter 15

The Hexadecimal Function [HEX( )]

The HEX function allows the user to output hexadecimal codes to any peripheral on the System 2200. Each character or command related to a peripheral is expressed as a unique two-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 15-1
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. The table to the right shows how counting is done in the hexadecimal system as compared to the decimal system.

HEXADECIMAL		DECIMAL	
0	C	0	12
1	D	1	13
2	E	2	14
3	F	3	15
4	10	4	16
5	11	5	17
6	12	6	18
7	13	7	19
8	14	8	20
9	15	9	21
A	16	10	22
B		11	

Chapter 15
The Hexadecimal Function [HEX( )]

The hexadecimal system is used in the internal design of many computers. On the System 2200 all characters or commands outputted to a peripheral are represented by a two-digit HEX code. There are 256 such codes. The table to the right lists 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

Section 15-2
Format Of HEX Function In A BASIC Statement Line

The two-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.

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 (see example).

```
10 PRINT HEX(0909);"ABC"
10 PRINT HEX(09);HEX(09);"ABC"
```

The code HEX (09) causes the CRT cursor to move 1 space to the right. In the two examples given 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 15-3
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. The table to the right shows a list of the special characters and cursor movement codes.

The complete list of HEX codes for the CRT (Model 2216) is given in Appendix A.

Each peripheral available for the System 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.

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)		

Section 15-4
Plotting Example

The example to the right 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 (0C) is a CURSOR UP command.

```
READY
:10 I=0
:20 PRINT "COUNT=";I
:30 PRINT HEX(0C);:REM CURSOR UP
:40 I=I+1
:50 GOTO 20
:RUN
COUNT=329
```

Chapter 16

Debugging

As discussed earlier in this text, the System 2200 error diagnostics are capable of detecting syntax and execution errors in a program. Programming errors, those in which the program does not do what it should, are the responsibility of the programmer. A program error is commonly known as a bug. Several methods are available on the System 2200 to help a programmer debug a program. This chapter discusses the various techniques a programmer can draw upon as an aid to debugging programs.

Section 16-1
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:

1. Make sure you know how to solve the problem.
2. "Play computer" – go through a hand calculation first.
3. Trace through the flowchart before converting to a program.

Rule 2 Prevent problems before they happen:

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 logical as programming:

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, 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 the program uses subscripted varaibles, 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. The result is that the 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 STOP's in the program; when a problem is discovered, re-execute problem section with trace.

Section 16-2
Using HALT/STEP As A Debugging Aid

The HALT/STEP key enables a System 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 the execution of a program. This causes the System 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.

```
READY
:10 J=25
:20 K=15
:30 GOTO 60
:40 PRINT J+K+L
:50 END
:60 L=80
:70 GOTO 40
:_
```

Enter the program shown to the right into the System 2200 memory.

Touch RESET

Touch keys GOTO 10 CR/LF-EXECUTE

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

```
READY
:GOTO 10

:
10 J=25

:
20 K=15

:
30 GOTO 60

:
60 L=80

:
70 GOTO 40

:
40 PRINT J+K+L
 120

:
50 END

END PROGRAM
FREE SPACE=3386

:_
```

Notice in the above procedure as the HALT/STEP key 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 16-3
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.

Enter the program shown to the right into the System 2200 memory.

Touch RESET

Touch keys GOTO 10 CR/LF-EXECUTE

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

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 first statement. The next time the HALT/STEP key is touched, the first statement of the line is eliminated and the execution of the second statement is shown; this continues until all statements are shown and executed.

PROGRAM

```
10 X=5:Y=10:Z=15
20 PRINT X*Y-Z:STOP
```

```
READY
:GOTO 10

:
10 X=5:Y=10:Z=15

:
10:Y=10:Z=15

:
10::Z=15

:
20 PRINT X*Y-Z:STOP
 35

:
20:STOP

STOP
:_



READY
:100 TRACE
    .
    .
    .
:200 TRACE OFF
```

Section 16-4
Other Uses Of HALT/STEP Key

(1) After the HALT/STEP key is touched, the System 2200 can be used as a calculator to perform 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 touching

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 16-5
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 uppercase letters). In program text, TRACE is turned ON and OFF as shown to the right (i.e., TRACE and TRACE OFF can be used as ordinary program statements).

Once the TRACE is turned ON it remains ON throughout the 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 xxxx 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 statement.

Assume in the examples shown that each is from a separate program where the variables are already defined. The printout shown is therefore for only the individual statements.

```
READY
:100 TRACE
:
:
:
:200 TRACE OFF
```

TRACE EXAMPLES

EXAMPLE #1
```
30   X=52+SQR(81)
X = 61
```

EXAMPLE #2
```
70   READ A,B,X(22)
```

EXAMPLE #3
```
100  GOTO 200
```

EXAMPLE #4
```
30   GOSUB 80
110  FOR I=1TO 25
 .
 .
 .
190  RETURN
```

EXAMPLE #5
```
50   IF A > B THEN 90
```

TRACE can be used as part of a program. Enter the program shown to the right and RUN the program. Compare your results with the results at the right.

PROGRAM

```
10 TRACE
20 Y=21.5
30 IF X=86 THEN 60
40 X=4*Y
50 GOTO 30
60 TRACE OFF
70 STOP
```

RESULTING PRINTOUT

```
:RUN
Y= 21.5
X= 86
TRANSFER TO 30
TRANSFER TO 60

STOP
:_
```

Section 16-6
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.

Enter the program to the right.

```
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
:_
```

Touch RESET

Touch TRACE CR/LF

Touch keys GOTO 10 CR/LF

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

Touch HALT/STEP

```
READY
:TRACE

:GOTO 10

:
10 READ X,Y:Z=X*Y
X= 5
Y= 10

:
10:Z=X*Y
Z= 50
:
20 IF Z>100 THEN 40
:
30 GOTO 10
TRANSFER TO 10
:
10 READ X,Y:Z=X*Y
X= 15
Y= 20
:
10:Z=X*Y
Z= 300
:
20 IF Z>100 THEN 40
TRANSFER TO 40
:
40 PRINT Z
 300
:
50 STOP
STOP
:_
```

Section 16-7
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). However, when statements are numbered close together, there may be no room for inserting additional statements.

With the RENUMBER key, you can have the System 2200 automatically renumber program statements. 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 clear up a listing for appearance sake.

The RENUMBER statement has several options (see general form).

GENERAL FORM

RENUMBER [line number]	[,line number]	[integer]
where $0 < \text{integer} < 100$ 1st 'line number' specifies at which line to start renumbering. All lines $\geqslant$ 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.

Enter the program shown to the right in the System 2200 memory.

```
READY
:1 FOR X=1TO 10
:2 PRINT X,X↑2,X↑3
:3 NEXT X
:4 GOTO 1
:5 END
:_
```

This same program is used to illustrate the several different ways you can renumber a program, using the options mentioned above.

Touch RESET

Touch RENUMBER CR/LF-EXECUTE

Touch LIST CR/LF-EXECUTE

RESULT

```
READY
:RENUMBER
:LIST
10 FOR X=1TO 10
20 PRINT X,X↑2,X↑3
30 NEXT X
40 GOTO 10
50 END
:_
```

Comment: All program lines are renumbered in increments of 10. The first line number of the resulting program is 10. The line number referenced in the GOTO statement is altered to reflect renumbering.

Touch RESET

Touch RENUMBER 20, 12, 10 CR/LF-EXECUTE

RESULT

```
READY
:RENUMBER 20,12,10
:LIST
10 FOR X=1TO 10
12 PRINT X,X↑2,X↑3
22 NEXT X
32 GOTO 10
42 END
:_
```

Comment: All program lines beginning with line 20 are numbered in increments of 10. The new line number for 20 is 12.

Touch RESET

Touch RENUMBER, 5, 5, CR/LF-EXECUTE

Comment: All program line numbers are renumbered since the first parameter is omitted. The new starting line number is 5 and the increment is 5. The line number in the GOTO statement is altered to reflect the renumbering.

RESULT

```
READY
:RENUMBER ,5,5
:LIST
5 FOR X=1TO 10
10 PRINT X,X↑2,X↑3
15 NEXT X
20 GOTO 5
25 END
:_
```

Touch RESET

Touch RENUMBER, , 15 CR/LF-EXECUTE

Comment: All program line numbers are renumbered because the first parameter is omitted. The increment is 15 and the new starting line number equals the increment.

RESULT

```
READY
:RENUMBER ,,15
:LIST
15 FOR X=1TO 10
30 PRINT X,X↑2,X↑3
45 NEXT X
60 GOTO 15
75 END
:_
```

SUMMARY TO PART II

Chapter 16 completes Part II of this manual. You now have been instructed in the use of every statement and common programming technique executable on the System 2200A. If you own a System 2200A, the material covered in Part III *is not* applicable to a System 2200A. However, if you are interested in the features described in Part III, your System 2200A can be field upgraded to a System 2200B.

Introduction

Part III

Additional Programming Features Available on the System 2200B

The System 2200B provides an extensive set of built-in data manipulation operations that enable the System 2200 to easily scan, analyze, convert, reduce, and gather data in most any format. Essentially, the operations give the user much of the power and flexibility of an assembler language (lower level language) without requiring a user to learn difficult assemble language in programming. The operations are classified into the following categories:

1. Bit and Byte Maniulation
2. Position and Numeric Verification Statements
3. Data Conversion
4. Data Reduction
5. Data Gathering

All of the statements associated with the categories, plus others, are described in Part III of this manual.

Chapter 17

Computed Branches

You already have been introduced in Part II to both the GOTO and GOSUB statements which direct a program to branch to a specified line number (e.g., GOTO 20, GOSUB 200). These are unconditional branches. The ON statement allows a programmer to choose (compute) one of several line numbers to which program execution branches. The branch then becomes conditional upon the result of the computation.

Section 17-1
The General Form Of The ON Statement

The general form of the ON statement is shown to the right.

The expression in the ON statement is evaluated and truncated to produce an integer. The integer determines the line number to which program execution transfers. If the integer is 1, execution is transferred to the first line number in the ON statement. Likewise, if the integer is 2, 3, 4 etc., execution is transferred to the second, third, fourth, etc. line number in the statement. If the value of the expression is less than 1 or greater than the actual number of line numbers in the ON statement, no branch occurs. Instead, the next sequential statement in the program is executed.

GENERAL FORM

$$\text{ON expression} \left\{ \begin{matrix} \text{GOTO} \\ \text{GOSUB} \end{matrix} \right\} \text{line number [, line number. . . . .] . . . .}$$

Some examples of ON statements are shown to the right.

Result of Example 1: Execution of statement 25 causes a branch to statement 90, the second line number in the ON statement (because I = 2).

Result of Example 2: Statement 25 is not executed (i.e., no branch) because the value of I exceeds the number of line numbers in the ON statement. As a result the next statement in the program is executed.

Result of Example 3: Execution of statement 30 results in a branch to line number 200, the third line number in the ON statement (because the truncated integer value of the expression = 3).

EXAMPLE 1:

```
:20 I=2
:25 ON I GOTO  70, 90, 200
```

EXAMPLE 2:

```
:20 I=5
:25 ON I GOTO  70, 90, 200
```

EXAMPLE 3:

```
:20 T=1/7
:30 ON T*21 GOSUB  50, 90, 200, 75
```

Section 17-2
Using A Computed GOTO Or Computed GOSUB In A Program

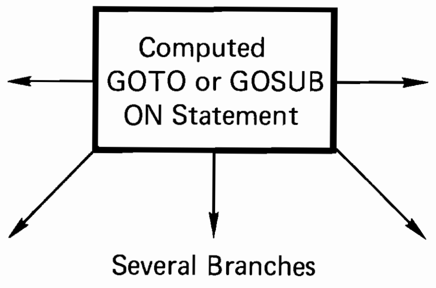
Enter and run the program shown to the right. (The BASIC word ON is entered with individual letters in uppercase.) This program uses the GOSUB parameter with the ON statement. Notice after each subroutine is executed, the program returns to the next statement in sequence after the last executed ON statement.

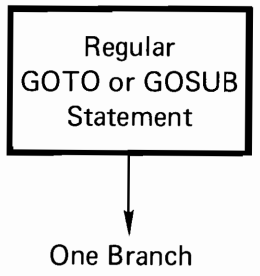
```
READY
:10 FOR  A=1 TO  3
:20 ON A GOSUB  100, 200, 300
:30 PRINT  "NEXT"
:40 NEXT  A
:50 STOP
:100 PRINT  "SUBROUTINE AT 100,A=";A
:110 RETURN
:200 PRINT  "SUBROUTINE AT 200,A=";A
:210 RETURN
:300 PRINT  "SUBROUTINE AT 300,A=";A
:310 RETURN
:RUN
SUBROUTINE AT 100,A= 1
NEXT
SUBROUTINE AT 200,A= 2
NEXT
SUBROUTINE AT 300,A= 3
NEXT

STOP
:_
```

The second program shown on this page is an example of the GOTO parameter with an ON statement. Enter this program and run it.

```
READY
:10 FOR  S=1 TO  4
:20 ON S GOTO  30, 40, 50, 60
:30 PRINT  S↑2:GOTO 70
:40 PRINT  S↑3:GOTO  70
:50 PRINT  S↑4:GOTO 70
:60 PRINT  S↑5:GOTO 70
:70 NEXT  S
:80 STOP
:RUN
 1
 8
 81
 1024

STOP
:_
```

In general, the ON statement used with the GOTO and GOSUB parameters allows a programmer much more flexibility in that one statement (ON) can cause the program flow to branch to any number of different places in a program, as compared to the regular GOTO or GOSUB statement, which allows a program to branch to one specified line (unconditional branch vs. conditional branch).





Chapter 18

Data Reduction

Many program applications require large amounts of data storage. The System 2200 stores each numeric value of data in 8 bytes of memory. When only a few significant digits are required, this method of storage becomes inefficient. The System 2200 allows a programmer to pack numeric values into an alphanumeric variable with the PACK statement. Once packed, the data cannot be used directly in calculations, as it is not is standard System 2200 form (i.e., 13 digits). It must therefore be "unpacked" before using, accomplished with the UNPACK statement.

Section 18-1
General Form Of The PACK Statement

The general form of the PACK statement is shown to the right.

In the general form of the PACK statement, the image specifies the decimal format of the numbers to be packed. The image is composed of # characters to signify digits and optionally +, –, ., and ↑ characters to specify sign, decimal point position and exponential form. The number of pound (#) signs that can be used is $0 < \#\text{'s} \leqslant 16$.

GENERAL FORM

$$\text{PACK (image)} \begin{Bmatrix} \text{Alpha Variable} \\ \text{Alpha Array Designator} \end{Bmatrix} \text{FROM} \begin{Bmatrix} \text{Numeric Array Designator} \\ \text{Expression} \end{Bmatrix}, \ldots$$

where: image = [+] [#. . . .] [.] [#. . . .] [   ]

Section 18-2
The General Form Of The UNPACK Statement

The UNPACK statement is used to UNPACK data packed by the PACK statement. The general form is shown to the right.

The same image used to pack the data must be used to UNPACK the data. An error results if more data is attempted to be unpacked than the amount already existing in the alphanumeric variable or array. However, less data can be unpacked than was packed.

The alpha array designator or alpha variable in the UNPACK statement refers to the name assigned to the packed data.

The numeric array designator or numeric variable designates the name which is assigned to the data as it is unpacked. This name does not have to be the same name used when the original data was packed.

Images can have two types of formats:

1) Format 1 - Fixed point e.g., ##.##
2) Format 2 - Exponential e.g., #.##↑↑↑↑

The alpha variable (A$) or alpha array designator (B$( )) designates the name assigned to the packed data. All data is stored sequentially into the specified alphanumeric variable or array. Arrays are filled starting with the first array element and ending with the last numeric data to be stored. An error results if the alphanumeric variable or array is not large enough to store all numeric values to be packed.

GENERAL FORM

$$\text{UNPACK (image)} \begin{Bmatrix}\text{Alpha Array Designator}\\ \text{Alpha Variable}\end{Bmatrix} \text{ TO } \begin{Bmatrix}\text{Numeric Array Designator}\\ \text{Numeric Variable}\end{Bmatrix}, \ldots.$$

where image: = [+] [#. . . .] [.] [#. . .] [↑↑↑↑]

The numeric array designator or expression in the PACK statement represents the data to be packed.

Some examples of PACK statements are shown to the right.

Explanation of Example 1: Scalar variable X is packed into the alphanumeric variable A$ in fixed point format.

EXAMPLE 1:
```
:10 PACK(####)A$ FROM X
```

Explanation of Example 2: Scalar varaibles X, Y and Z are packed, starting with X, into the alphanumeric variable A$ in fixed point format.

EXAMPLE 2:
```
:12 PACK(##.##)A$ FROM X,Y,Z
```

Explanation of Example 3: The scalar array N( ) is packed into the alphanumeric array A$( ) in exponential format.

EXAMPLE 3:
```
:15 PACK(+#.##↑↑↑↑)A$() FROM N()
```

Unpacking begins at the beginning of the specified alphanumeric variable or array. The unpacked data is converted to internal floating point values, and stored in the specified scalar array or variable. Some examples of UNPACK statements are shown to the right.

Explanation of Example 1: The packed data in alphanumeric variable (A$) is unpacked and stored in scalar variables X, Y and Z, respectively.

EXAMPLE 1:
```
:10 UNPACK(####)A$ TO  X,Y,Z
```

Explanation of Example 2: The packed data in alphanumeric array (A$( )) is unpacked and stored in the scalar array B( ), except for the last value which is stored in the scalar variable Y.

EXAMPLE 2:
```
:12 UNPACK(####)A$() TO  B(),Y
```

Chapter 18
Data Reduction

When considering how many values can be packed per alphanumeric variable or alphanumeric array, first determine (by the image) how many bytes are required for the number. As an example, values are to be packed in the image (##.##) which requires 2 bytes. This image is packed into the alphanumeric variable B$.

As already discussed in this manual (Chapter 12), an alpha variable is assumed to be 16 characters in length if not dimensioned otherwise. The length can vary from 1 to 64 characters if dimensioned. If the alpha variable is not dimensioned (16 characters), eight two-byte values can be packed into B$. If B$ is dimensioned (e.g., B$64) then 32 2-byte values can be packed.

What is true for alpha variables is also true of alphanumeric arrays. Each element of the array is assumed to be 16 characters in length if not dimensioned. When dimensioned, each element can range from 1 to 64 characters.

For example, assume you have an alpha array of A$(5,4)64. You can pack 5x4x64 = 1280 bytes into this array. Values having the packing image of ##.## (2 bytes) result in 640 data values being stored in this array.

The length of an alpha variable or alphanumeric array, if other than 16 characters, must be dimensioned in a DIM statement prior to use in a PACK statement.

Packing data in this fashion results in a great saving of storage space, either in memory, on cassette tapes, or on a disk.

Section 18-3
Rules For Packing Data

In order to estimate how many values can be packed per alphanumeric variable or array, it is necessary to understand how much storage space is taken up by packed data. The following rules are useful for determining storage space:

1) Every two digits in an image requires one byte of memory. A pound (#) sign represents one digit in the image.

2) If a sign (+ or –) is specified, it occupies 1/2 byte of memory. This byte contains the sign of the number and the sign of the exponent for exponential images.

3) If no sign is specified, the absolute value of the number is stored and the sign of the exponent is assumed to be plus (+).

4) The decimal point is not stored in memory. When unpacking data, the decimal point position is specified in the image.

5) The packed numeric value occupies a whole number of bytes of memory, e.g., the image (###) indicates 1-1/2 bytes are required for storage, however, 2 bytes are used, because only a whole number of bytes are used.

6) If the image has format 1, the value is edited as a fixed point number, truncating or extending with zeros any fraction, and inserting leading zeros for insignificant integer digits according to the image specification.

7) If the image has format 2, the value is edited as a floating number. The value is scaled as specified by the image (there are no leading zeros). The exponent occupies one byte.

The Table to the right indicates storage requirements for various images.

IMAGE	STORAGE
####	2 bytes
###	2 bytes
###.###	3 bytes
+#.##	3 bytes

Section 18-4
Programming Examples Of PACK And UNPACK

The first program at the right packs 100 pieces of data into alpha array A$(10)45. This data represents 10 purchases for each of 10 suppliers, therefore 100 purchases. The data from each supplier is packed into one element of alpha array A$(10)45. Each element is dimensioned to be 45 bytes (A$(10)45). Since the packing image is ######.##, 4 bytes are required for each piece of data. That means the data for each supplier requires at maximum 40 bytes. The elements in the alpha array are dimensioned slightly larger than the maximum 40, namely 45.

The data is inputted by the operator as seen in statement 70.

Enter this program into the System 2200 memory.

Touch RESET

Touch RUN CR/LF-EXECUTE

When the program stops, input the first purchase, i.e.,

Touch keys 5 6 3 3 2 1 . 5 9 CR/LF-EXECUTE

The program stops for additional values. Input 9 other values.

```
READY
:5 DIM  X(10),A$(10)45
:10 REM  ENTER AND STORE 10 PURCHASES FROM 10
:20 REM  SUPPLIERS IN A PACKED ALPHA-NUMERIC ARRAY
:30 REM  PACKED FORM IS ######.##
:40 FOR  I=1 TO  10
:50 PRINT  "SUPPLIER NO.";I
:60 FOR  J=1 TO  10
:70 INPUT  X(J)
:80 NEXT  J
:90 PACK(######.##)A$(I) FROMX()
:100 NEXT  I
:_
```

```
READY
:RUN
SUPPLIER NO. 1
? 563321.59
?
?
?
?
?
?
?
?
?
SUPPLIER NO. 2
?
```

After all the values are inputted for the first supplier, the program loops back and requests the data for the second supplier.

Continue entering any values until you have inputted 10 x 10 = 100 values. Once all the data is entered, the program then packs the data away in alpha array A$.

The second program to the right calculates the total purchase from the 10 suppliers. Before this calculation can take place, the data must be unpacked. Statement 80 unpacks the data in the same image that it was packed, and assigns the values to Array X.

Enter the program in the System 2200 memory.

RUN the program by touching keys RUN 110 CR/LF-EXECUTE.

The sum of the purchases is printed upon completion of the program.

```
READY
:110 DIM  X(10),A$(10)45
:120 T=0
:130 REM  COMPUTE THE TOTAL PURCHASES FROM
:140 REM  THE 10 SUPPLIERS WHOSE PURCHASES
:150 REM  WERE PREVIOUSLY PACKED IN A$() IN THE
:160 REM  FORM ######.##
:170 FOR  I=1 TO  10
:180 UNPACK(######.##)A$(I) TO  X()
:190 FOR  J=1 TO  10
:200 T=T+X(J)
:210 NEXT  J
:220 NEXT  I
:230 PRINT  T
:_
```

Chapter 19

Position and Numeric Verification Functions (POS and NUM Functions)

Section 19-1
General Form Of The POS Function

The POS (Position) function allows the programmer, with a single statement, to scan an entire alphanumeric variable and locate the numerical position of a certain character (i.e., whether the character is in the second, or tenth, twentieth, etc. position in the alphanumeric string). The POS function sets up a comparison and tests each character in the string (from the beginning) against another character.

The first character in the string to satisfy the comparison results in the System 2200 indicating the actual numeric position of that character in the string.

The general form of the POS function is shown to the right.

The alpha variable in the POS function specifies the name of the alpha variable to be scanned.

There are six relational operators that can be used for making comparisons ($<$, $\leqslant$, $=$, $\geqslant$, $>$, $<>$ (not equal)).

GENERAL FORM

$$\text{POS}\left(\text{Alpha Variable}\left\{\begin{matrix}<\\ \leqslant\\ =\\ \geqslant\\ >\\ <>\end{matrix}\right\}\left\{\begin{matrix}\text{"Character"}\\ \text{XX}\end{matrix}\right\}\right)$$

where: XX = hexadecimal digit (0-9 or A-F)

The character used for comparison is specified as:

(1) A single character in quotes (e.g., "9") or (2) the equivalent HEX code for that character (e.g., HEX (39)).

Some examples of POS functions are shown to the right.

Explanation of Example 1: The variable X is set equal to the number specifying the numeric position of the $ in the alpha variable A$.

Explanation of Example 2: This statement scans the alpha variable (A$) from the fourth to eighth character and tests each character to see if it equals HEX (20), which is a space. The numeric position of the first character to equal a space is printed out. If none of the characters equal a space, then POS is set equal to zero, and a zero is printed out.

Explanation of Example 3: This statement scans the alphanumeric variable A$ to find the first character that is < A. Once found, the position of the character is then tested to see if it is < 16. If the position of "A" < 16, then program flow goes to statement number 60; if the position of "A" ⩾ 16, program flow goes to the next sequential statement in the program.

As can be seen from these examples, the POS function can be used wherever numeric functions are normally used in BASIC. If no character in the alpha string satisfies the specified condition, then POS is set equal to zero (POS=0).

EXAMPLE 1:
```
10 X=POS(A$="$")
```

EXAMPLE 2:
```
20 PRINT POS(STR(A$,4,5)=20)
```

EXAMPLE 3:
```
30 IF POS(A$<"A")<16 THEN 60
```

Section 19-2
Programming Examples Using The POS Function

The program at the right illustrates the use of the POS function. In this example, a credit card company has a very extensive customer list, which requires that the list constantly be scanned for expired cards. The program is written to scan the data, and printout only the names and expiration date for cards no longer valid. The data in this example is written into the program; in actual practice, however, the data would be stored on some external storage device (e.g., tape cassette or disk) and "called" into the program as needed.

Enter the program and run it.

Explanation: statement 20 assigns the alphanumeric variable (A$) to the data in statement 100.

Statement 40 uses the STR(function in conjunction with the POS function, to locate the position of four characters preceded by an "*" (i.e., the year). The year is pulled out of the string and converted to a numeric and set equal to A (see Chapter 21) for a detailed discussion of the CONVERT statement).

If the date is < 1975, program flow jumps to statement 70 where only name and date are printed. If the date is ≥ 1975 program flow goes to statement 60 which transfers the program to statement 20. The entire process is then repeated. The program stops running when it reaches the text data "ZZZZ".

```
:10 DIM A$64
:20 READ A$
:30 IF A$="ZZZZ" THEN 150
:40 CONVERT STR(A$,POS(A$="*")+1,4) TO A
:50 IF A[1975 THEN 70
:60 GOTO 20
:70 PRINT "NAME","YR.EXP."
:80 PRINT STR(A$,1,POS(A$=" ")),STR(A$,POS(A$="*")+1,4)
:90 GOTO 20
:100 DATA "JOHN-SMITH 002544211 *1974 GEORGIA"
:110 DATA "BETSY-LOOKNER 116423596 *1975 MASS."
:120 DATA "JESSIE-BELL 211190421 *1973 ALABAMA"
:130 DATA "RICHARD-BATES 116423596 *1972 WISCONSIN"
:140 DATA "ZZZZ"
:150 STOP
:RUN
NAME            YR.EXP.
JOHN-SMITH      1974
NAME            YR.EXP.
JESSIE-BELL     1973
NAME            YR.EXP.
RICHARD-BATES   1972

STOP
:_
```

Section 19-3
The General Form Of The NUM Function

The NUM (numeric) function allows the programmer to scan an alphanumeric variable from the first character in a string, or from any character in a string if used in conjunction with the STR(function, and count the number of sequential valid numeric characters from that point until a non-numeric character is found or until the sequence of numeric characters fails to conform to standard BASIC number format. Valid numeric characters include the digits 0-9, spaces, +, −, ., and E. The NUM function is used to determine either the length of a numeric portion of an alphanumeric variable, or to verify whether an alphanumeric variable is a legitimate BASIC representation of a numeric value.

The general form of the NUM function is shown to the right.

GENERAL FORM

NUM (Alpha Variable)

The alpha variable in the NUM function specifies the name of the alphanumeric variable which is scanned by the NUM statement.

Some examples of the NUM function are shown to the right.

Explanation of Example 1: The variable X is assigned a value equal to the number of the first group of sequential numeric characters in the alpha string A$. In this example X = 12, because leading and trailing spaces are included in the count, as well as the plus sign. The count is terminated when a non-numeric character is encountered (in this case the letter "N").

EXAMPLE 1:

```
:5 A$="+ 265191211 NAME,ADDRESS"
:10 X=NUM(A$)
:15 PRINT X
:RUN
 12
```

Explanation of Example 2: The third through the 8th character in string A$ is scanned and the first sequential group of valid numeric characters is counted. If this number is not equal (<>) to 8, then the program branches to Statement 100. If the number is equal to 8, the next sequential statement in the program is executed. In this example the number equals 6, therefore the program branches to statement 100.

EXAMPLE 2:

```
:10 A$="+ 265191211 NAME,ADDRESS"
:20 IF NUM(STR(A$,3,6))<>8 THEN 100
```

Explanation of Example 3: Statement 20 defines M$ as being equal to the valid numeric characters in the string A$, which are 21.523.

In order to determine what characters are valid numerics, the STR(function is used to pick out part of the string. The POS statement is used to determine the starting position of the numeric characters (A$ ⩽ "9") and the NUM statement is used to determine how many sequential characters there are from the starting position so that the correct part of the string is assigned to M$.

EXAMPLE 3:

```
:10 A$="21.523 NAME"
:20 M$=STR(A$,POS(A$<="9"),NUM(A$))
:30 PRINT M$
:RUN
21.523
```

Explanation of Example 4: Statement 20 defines X as being equal to the number of valid numeric characters in A$. X is equal to 4 as the sequence of valid numeric characters fails to conform to a standard BASIC number when the + character is encountered.

EXAMPLE 4:

```
:10 A$="98.7+53.6"
:20 X=NUM(A$)
:30 PRINT X
:RUN
 4
```

Section 19-4
Programming Example Using The NUM Function

The program to the right shows a typical example of how the NUM function can be used.

An investment company has a record of all its clients and the amount of the money being handled by the firm per client. At the end of each month a tally of the number of clients and the total funds contributed by all clients is taken.

The key to this program is statement 50. Again the CONVERT statement is used (Chapter 21) to set the amount of each client's contribution equal to M. Both the POS and NUM functions are used in conjunction with the STR(function so that each DATA statement is scanned and the amount of each client's contribution is picked out of the string and assigned to variable M. The first POS function in the STR(function is used to determine the numeric position of the first character in the DATA statement representing the amount (i.e., POS(A$ = "S") +1). In DATA statement 110 this is 26(25 + 1). The NUM function is then used to determine how many numbers (characters) there are in the amount after the position of the amount is determined. NUM(STR(A$, POS(A$,= "$")+1).

In DATA statement 110 this is 8. Therefore, STR(A$, 26, 8) = 1000500 = M. Each DATA statement is unique due to the number of characters in the name as well as the different amounts contributed. Statement 50 is general enough to enable the program to pick out the required part of a string in each DATA statement.

```
READY
:5 DIM A$64
:10 A=0:M=0:X=0
:20 FOR S=1 TO 50
:30 READ A$
:40 IF A$="9999" THEN 90
:50 CONVERT STR(A$,POS(A$="$")+1,NUM(STR(A$,POS(A$="$")+1))) TO
M
:60 A=A+M
:70 X=X+1
:80 NEXT S
:90 PRINT "NUM. OF VALUES","SUM","MONTH"
:100 PRINT X,"$";A,"JUNE"
:110 DATA "BETTY WHITE 026-30-3191 $1000500 *1932"
:120 DATA "J.C.JONES 511-61-7236 $1005.00 *1947"
:130 DATA "ANDREA LAPOINT 001-42-6109 $2052.56 *1953"
:140 DATA "JOSHUA KING 781-91-4136 $633.92 *1948"
:150 DATA "KENNETH P. WILBUR 467-77-5841 $400000.25 *1909"
:160 DATA "9999"
:170 END
:RUN
NUM. OF VALUES  SUM             MONTH
 5              $ 1404191.73    JUNE

END PROGRAM
FREE SPACE=2801

:
```

Chapter 20

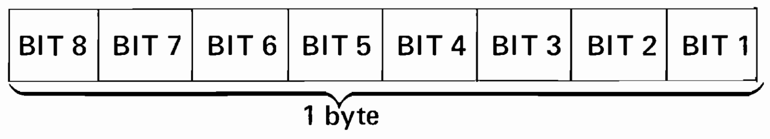
Bit and Byte Manipulation

There are several statements in the System 2200 which allow a programmer to look at an 8-bit word and change one or more bits. In effect, a programmer can manipulate any one bit in memory. This capability is extremely useful in many programming applications such as data conversion, code conversion, editing of BASIC programs by a program, program translation, and preparation of specially formatted output data.

In order to fully understand how this bit and byte manipulation occurs it is necessary to discuss what a bit is, what a byte is and what the binary numbering system is before a discussion is begun on what statements are involved.

Section 20-1
What Is A Bit? What Is A Byte?

Every character (or keyword) in an alphanumeric variable requires one byte of memory. A byte consists of 8 bits. Therefore every character in an alphanumeric variable is actually represented in memory as 8 bits (see diagram).



Chapter 20
Bit and Byte Manipulation

All characters or keywords in the System 2200 are represented in memory with different combinations of these 8-bits being used or more accurately being "turned on". If a bit is "turned on" this is represented with a 0. The diagram to the right illustrates how a byte looks in memory when representing a sample character. Several of the bits are "ON" (1) and several of the bits are "OFF" (0).

0	0	1	1	0	0	0	1

SAMPLE CHARACTER IN MEMORY

Section 20-2
What Does Binary Mean?

The use of 1 to represent an ON bit and a 0 to represent an OFF bit in the design of computer memories is more than coincidental. The numbering is used because 0 and 1 are the digits used in the Binary numbering system.

The literal definition of Binary is "consisting of two". The Binary numbering system thus consists of various combinations of two digits; 0 and 1. The decimal system ("consisting of ten") consists of various combinations of the digits 0-9. The table to the right shows several Binary numbers and their decimal equivalents. The table also shows the hexadecimal[1] equivalent of these same binary numbers. The reason hexadecimal is included in this table shall become apparent later.

BINARY	DECIMAL	HEXADECIMAL
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

[1]See Chapter 15 for a discussion of the hexadecimal numbering system.

HOW ARE BINARY NUMBERS GENERATED?

Each digit position in a Binary number represents an increasing power of two. If a zero is used you do not take the value of the position, if a one is used you do take the value of the position (see illustration).

In a byte of memory, the 8-bit configuration is two binary numbers. What do these two binary numbers represent? They represent the two digits of a Hexadecimal code. Recalling the discussion in Chapter 15, every key in the System 2200 generates a unique two digit HEX Code when it is used. This code is then converted to two 4 digit binary numbers and stored in memory as 8 bits or one byte (see example). Bits 1-4 represent the second digit of the hex code, bits 5-8 represent the first digit of the HEX Code.[1]

Because of the simplicity of the Binary numbering system (i.e., 0 and 1), many computer memories are designed this way.

Some other examples are shown to the right.

Each of the bit and byte manipulation statements performs a different mathematical or logical operation on the bit structure of one or more characters in an alphanumeric variable. The remainder of this Chapter discusses these individual statements.

Power	(2^5)	(2^4)	(2^3)	(2^2)	(2^1)	(2^0)
Value of Position	32	16	8	4	2	1
Binary Digit	0	1	0	0	1	0

$$\underbrace{0\quad 1\quad 0\quad 0\quad 1\quad 0}_{16 + 2 = 18}$$

BINARY DECIMAL
010010 = 18

$$\begin{tikzcd} \text{Touch } \boxed{A} \text{ Key} \arrow[d, "\text{generates}"] \\ \text{HEX(41)} \arrow[d, "\text{stored in memory as}"] \\ \text{} \end{tikzcd}$$

BIT 8	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	
0	1	0	0	0	0	0	1	A BYTE

$$\underbrace{0\quad 1\quad 0\quad 0}_{4}\ \underbrace{0\quad 0\quad 0\quad 1}_{1}$$

$$\begin{tikzcd} \text{KEY} \arrow[r, bend left, "\text{generates}"] & \text{HEX CODE} \arrow[r, bend left, "\text{stored as}"] & \text{8-BIT CONFIGURATION IN MEMORY (one byte)} \end{tikzcd}$$

KEY	HEX CODE	8-BIT CONFIGURATION IN MEMORY (one byte)
$	24	00100100
N	4E	01001100
?	3F	00111111

[1]All alphanumeric characters in the System 2200 are stored internally in ASCII. Hex codes are a subset of ASCII codes. For example, Hex(41) is the ASCII code for A. See Appendix A.

Section 20-3
Bit And Byte Manipulation Statements

The statements ADD, AND, OR, XOR, BOOL, ROTATE, and INIT are similar to instructions available on most mini-computers. They perform logical and arithmetic operations on the characters in the value of an alphanumeric variable. The entire value of the variable is operated on *including trailing spaces.* The alphanumeric variable can be considered a register or a collection of one byte registers that the operation is performed on. The length of this register may be set by defining the length of the alphanumeric variable in a DIM or COM statement (default length = 16 characters). Part of an alphanumeric variable may be operated on by using the STR(function to specify a portion of the variable.

ADD STATEMENT

The ADD statement is used to add a value to any alphanumeric variable. The addition is done in BINARY. The general form of the ADD statement is shown to the right.

In the ADD statement the [C] is optional. If not included the specified character (xx) is added to each character of the first alpha variable or the second alpha variable is added to the first alpha variable on a character by character basis.

If included the characters to be added are added to the *entire* string as if the string were a single character.

In the ADD statement the first *alpha variable* specifies the name of the alpha variable to be worked on.

GENERAL FORM

$$\text{ADD [C]} \left(\text{Alpha variable, } \begin{Bmatrix}\text{xx} \\ \text{Alpha variable}\end{Bmatrix}\right)$$

where: xx = hexadecimal digit (i.e., 0-9 or A-F)

c = add with carry

Two types of addition can be done with the ADD statement. They are:

(1) Immediate Addition
(2) String to String Addition

The type of addition to be done is indicated by the second term in the ADD statement.

If two hex digits (xx) are used this indicates immediate addition and the character specified by the hex digit is added to the first alpha variable.

If an alpha variable is used as the second term this indicates string to string addition and the value of the second string is added to the first.

The addition whether Immediate or String to String is done in binary.

See Example 1 to the right.

Explanation of Example 1: A$ is first defined as being two characters long (or 2 bytes long). This is done in order to operate on only two characters instead of two characters and 14 trailing spaces as would happen if A$ was undefined and had a default length of 16 characters. This is then an example of Immediate Addition where the hex digits (02) are added to each character (byte) of the alphanumeric string (A$). The addition is in binary as shown to the right. There is no carrier between bytes.

EXAMPLE 1:

```
:5 DIM A$2
:10 A$=HEX(0123)
:20 ADD(A$,02)
```

SETS A$ = HEX(0323)

$$\begin{array}{lllll}
A\$ & = & 00000001 & 00100011 & \text{(i.e., 01 23)}\\
 & + & & & \\
02 & = & 00000010 & 00000010 & \text{(i.e., 02 02)}\\
\hline
 & & \underbrace{00000011}_{03} & \underbrace{00100101}_{25} &
\end{array}$$

no carrier between bytes

Explanation of Example 2: A$ is defined as 2 characters long for the same reason used in Example 1. This is then an example of Immediate Addition when the hex digits (02) are added to the *entire* string as if the string were a single character (specified by the [C] option in the ADD statement). The addition is in binary as shown to the right and there is carrier between bytes.

NOTE:

The difference between ADD and ADD C is simply ADD does not have carrier between bytes and ADD C does. ADD C is used primarily for string to string binary addition.

Explanation of Example 3: A$ and B$ are both defined as two characters long. This is an example of String to String Addition where the addition is done on a character by character basis; i.e., the last character of the second value is added to the last character of the first value, the next to the last character of the second value is added to the next to the last character of the first value and so forth. The binary addition of these two alphanumerics is shown to the right. There is no carrier between bytes.

EXAMPLE 2:

```
:5 DIM A$2
:10 A$=HEX(0123)
:20 ADD C(A$,02)
```

SETS A$ = HEX(0125)

$$\begin{array}{rcrl} A\$ & = & 0000000100100011 & \text{(i.e., 01 23)} \\ 02 & = & \underline{00000010} & \text{(i.e., 02)} \\ & & \underbrace{00000001}_{01}\underbrace{00100101}_{25} & \end{array}$$

carrier between bytes

EXAMPLE 3:

```
:5 DIM A$2,B$2
:10 A$=HEX(0123):B$=HEX(00FF)
:20 ADD (A$,B$)
```

SETS A$ = HEX(0122)

$$\begin{array}{rclll} A\$ & = & 00000001 & 00100011 & \text{(i.e., 01 23)} \\ & + & & & \\ B\$ & = & \underline{00000000} & \underline{11111111} & \text{(i.e., 00 FF)} \\ & & \underbrace{00000001}_{01} & \underbrace{00100010}_{22} & \end{array}$$

no carrier between bytes

Explanation of Example 4: A$ and B$ are both defined as 2 characters long. This is an example of String to String Addition where both strings are treated as single characters and added since [C] is specified in the ADD statement. There is carrier between bytes.

Explanation of Example 5: This is an example of using the STR(function to work on part of an alphanumeric string with the ADD statement. The ADD statement is Immediate addition where the two hex digits (81) are added only to the third and fourth characters of the string A$ (STR(A$ 3,2)). The Binary addition is shown to the right.

EXAMPLE 4:

```
:5 DIM A$2,B$2
:10 A$=HEX(0123):B$=HEX(00FF)
:20 ADD C(A$,B$)
```

SETS A$ = HEX(0222)

$$\begin{array}{rll} \text{A\$} = & 0000000100100011 & \text{(i.e., 0123)}\\ + & 0000000011111111 & \text{(i.e., 00FF)}\\ \hline & \underbrace{0000}_{0}\underbrace{0010}_{2}\underbrace{0010}_{2}\underbrace{0010}_{2} & \end{array}$$

carrier between bytes

EXAMPLE 5:

```
:10 A$=HEX(01234567)
:20 ADD(STR(A$,3,2),81)
```

SETS A$ = HEX(01A44567)

only change in A$

$$\begin{array}{rll} \text{A\$ (. . . . 23. . . .)} = & 0010\quad 0011 & \text{(i.e., 23)}\\ + & & \\ 81 = & 1000\quad 0001 & \text{(i.e., 81)}\\ \hline & \underbrace{1010}_{A}\quad\underbrace{0100}_{4} & \end{array}$$

If an alphanumeric has a variable with a length less than the maximum length specified for that variable, the remaining characters are all set equal to space. (Normally trailing spaces are not considered part of the value.) Therefore, be sure to define the length of the variable with a DIM or COM statement otherwise the bytes for trailing spaces are operated on with the ADD statement.

When two alphanumeric variables are added and they are not of the same defined length the following rules apply:

1) The addition is right adjusted with lead characters of zero binary value being assumed for the variable of shorter length (see example 6).

```
:10 DIM A$10
:20 A$=HEX(0123)
```

therefore A\$ = $0123\underbrace{2020202020202020}$

filled out with trailing spaces
(HEX (20)) because A$ was
less than the defined length of 16

EXAMPLE 6:

```
:5 DIM A$3,B$2
:10 A$=HEX(012345):B$=HEX(00FF)
:20 ADD (A$,B$)
SETS A$ = HEX(012344)
```

```
lead character     A$  =  0000 0001    0010 0011    0100 0101    (i.e., 012345)
added to           B$  =  0000 0000    0000 0000    1111 1111    (i.e., 00 00 FF)
adjust length ──────────┘
                          ─────────────────────────────────────────────────────
                          0000 0001    0010 0011    0100 0100
                              01           23           44
```

2) The answer is stored right adjusted in the receiving variable. If the total answer is longer than the receiving variable the lower order portion of the answer is stored (see example 7).

> **NOTE:**
>
> *The INIT statement can be used to initialize all characters of an alphanumeric variable to any character code including zero. This can be done prior to moving a value into part of the variable with a STR function to eliminate trailing spaces.*

AND, OR, XOR STATEMENTS

The AND, OR and XOR statements perform the specified logical function (AND, OR, or EXCLUSIVE OR) on the characters of a specified alphanumeric variable. The general form of these statements are shown to the right.

EXAMPLE 7:

```
:5 DIM A$1,B$2
:10 A$=HEX(45):B$=HEX(00FF)
:20 ADD(A$,B$)
```

filled in with binary zeros to equalize the length

$$\begin{array}{lll} \text{A\$} & \boxed{0000\ 0000} & 0100\ 0101 \\ \text{B\$} & 0000\ 0000 & 1111\ 1111 \\ \hline & \underbrace{0000\ 0000}_{00} & \underbrace{0100\ 0100}_{44} \end{array}$$

dropped in final answer as length of A$ is defined only as one.

GENERAL FORM

$$\left\{\begin{array}{l}\text{AND}\\ \text{OR}\\ \text{XOR}\end{array}\right\}\left(\text{alpha variable, }\left\{\begin{array}{l}\text{XX}\\ \text{alpha variable}\end{array}\right\}\right)$$

where: XX = hexadecimal digit (i.e., 0-9 or A-F)

The AND statement performs the logical AND operation on alphanumeric variables. It compares bit for bit two alphanumeric variables. If both bits are ON (a one) the result is ON (a one), otherwise the result is a zero (OFF). (See Example 1 to the right.)

Explanation of Example 1: The logical AND operation is performed on two alphanumeric variables. After the comparison is made bit for bit, only those bits compared which were both "ON" (i.e., 1) stay "ON" (assume a value of 1), the others are OFF (0).

EXAMPLE 1:

```
:10 A$=HEX(0C):B$=HEX(08)
:20 AND(A$,B$)
```

SETS A$ = HEX(08)

$$
\begin{array}{lll}
 & \text{A\$} = 00001100 & \text{(i.e., 0C)} \\
\text{AND} & \text{B\$} = 00001000 & \text{(i.e., 08)} \\
\hline
 & \underbrace{0000}_{0}\underbrace{1000}_{8} & \\
\text{RESULT} & & = \text{HEX(08)}
\end{array}
$$

The OR statement performs the logical OR operation on alphanumeric values. It compares bit for bit two alphanumeric variables. If *both* bits are OFF, the result is OFF (a zero), otherwise the result is ON (a one). (See Example 2 to the right.)

Explanation of Example 2: The logical OR operation is performed on two alphanumeric variables after the comparison is made bit for bit, only those bits compared which were both OFF (a zero) stay OFF (a zero), all others are ON (a one).

EXAMPLE 2:

```
:10 A$=HEX(0C):B$=HEX(08)
:20 OR(A$,B$)
```

SETS A$ = HEX(0C)

$$
\begin{array}{lll}
 & \text{A\$} = 00001100 & \text{(i.e., 0C)} \\
\text{OR} & \text{B\$} = 00001000 & \text{(i.e., 08)} \\
\hline
 & \underbrace{0000}_{0}\underbrace{1100}_{C} & \text{HEX(0C)} \\
\text{RESULT} & &
\end{array}
$$

The XOR statement performs the logical EXCLUSIVE OR operation on alphanumeric values. It compares bit for bit two alphanumeric variables. If both bits are the same (both ON or both OFF) the result is OFF (a zero), if different the result is ON (a one). (See Example 3.)

Explanation of Example 3: Both variables are defined as one character long to eliminate any trailing spaces. The logical EXCLUSIVE OR operation is performed on two alphanumeric variables. After the comparison is made if both bits are ON or both OFF the result is OFF, otherwise the result is ON.

EXAMPLE 3:

```
:5 DIM A$1,B$1
:10 A$=HEX(0C):B$=HEX(08)
:20 XOR(A$,B$)
```

```
               SETS A$ = HEX(04)
               A$  = 00001100     (i.e., 0C)
               B$  = 00001000     (i.e., 08)
                     --------
                     00000100
                     \__/\__/
RESULT                0   4     HEX(04)
```

Programming examples of each of these statements are shown to the right.

Explanation of Program I: A$ has hexadecimal code of '25' and a byte configuration of 00100101. It is desired to change A$ to hexadecimal '20'. This is accomplished by using AND with hexadecimal 'F0' whose bit configuration is 11110000. The rule for AND is - if both bits ON then the resultant bit is ON, otherwise the resultant bit is OFF.

PROGRAM I   AND

```
READY
:1 DIM A$1
:5 A$=HEX(25)
:10 AND(A$,F0)
```

Explanation of Program II: It is desired to change A$ hexadecimal code '48' whose bit donfiguration is 01001000 to hexadecimal code '49' whose bit configuration is 01001001. This can be done by using OR and the hexadecimal code '01' whose bit configuration is 00000001. The rule for OR is - if at least one bit is ON then the resultant bit is ON, otherwise the resultant bit is OFF.

PROGRAM II   OR

```
READY
:10 DIM A$1
:20 A$=HEX(48)
:30 OR(A$,01)
```

Explanation of Program III: It is desired to change A$ hexadecimal '29' whose bit configuration is 00101001 to hexadecimal '5E' whose bit configuration is 01011110. This can be done by using XOR and the hexadecimal code '77' whose bit configuration is 01110111. The rule for XOR is - if both bits are ON or both OFF, the resultant bit is OFF, otherwise the resultant bit is ON.

PROGRAM III XOR

```
:10 DIM A$1
:20 A$=HEX(29)
:30 XOR(A$,77)
```

INIT (INITIAL) STATEMENT

The INIT statement initializes alphanumeric variables to any character the user specifies (sets the alpha variable to zero or spaces, etc.). It is very useful and often necessary to initialize variables to all zeros before performing Binary operations, i.e., ADD, XOR, etc.

The general form of the INIT statement is shown to the right.

GENERAL FORM

$$\text{INIT}\ \left(\left\{\begin{matrix}\text{XX}\\ \text{"character"}\\ \text{alpha variable}\end{matrix}\right\}\right)\left\{\begin{matrix}\text{alpha variable}\\ \text{alpha array designator}\end{matrix}\right\},\left\{\begin{matrix}\text{alpha variable}\\ \text{alpha array}\\ \text{designator}\end{matrix}\right\}\ \ldots$$

where: XX = hexadecimal digit (i.e., 0-9 or A-F)

alpha array designator = alpha array name( ) e.g., A$( )

Each character in the variable or array is set equal to the character specified inside the parentheses. This character can be represented by two hex digits. If an alphanumeric variable is enclosed in the parentheses, the 1st character of the value of the alphanumeric variable is used.

Some examples are shown to the right.

Explanation of Example 1: This statement sets every byte of the variable B$ and array X$( ) to zeros. Hex digits (00) are used to make this change.

EXAMPLE 1:
```
:10 INIT(00)B$,X$()
```

Explanation of Example 2: This statement sets every byte of array 15 A1$( ) and B$( ) to a space. The character "space" (i.e., HEX20) is used to make this change.

EXAMPLE 2:
```
:20 INIT(" ")A1$(),B$()
```

Explanation of Example 3: This statement sets every byte of array B$( ) to a one. The first character of alphanumeric variable A$ is used which is a one.

EXAMPLE 3:
```
:10 A$="102A7"
:20 INIT(A$),B$()
```

ROTATE STATEMENT

The ROTATE statement rotates the bits of each character (i.e., in each byte) in the value of the specified alphanumeric variable to the left from 1 to 7 places; the high order bits replace the low order bits. All characters are operated on including trailing spaces.

If an alphanumeric variable has a value with a length less than the maximum length of the variable, the remaining characters are all set equal to spaces.

The general form of the ROTATE statement is shown to the right.

GENERAL FORM

ROTATE (alpha variable, d)

where: d = digit from 1 - 7

See the example to the right.

EXAMPLE:
```
:10 A$=HEX(0132EF)
:20 ROTATE(A$,4)
```

SETS A$ = HEX(0132FE)

Explanation: Each byte is rotated 4 places to the left. The byte structure of A$ is shown to the right. Each of the bytes are rotated to the left 4 places as shown.

A$ requires 3 bytes

0000 0001	01	
0010 0011	23	
1111 1110	FE	

RESULT

0001 0000	10	
0011 0010	32	HEX(1032EF)
1110 1111	EF	

BOOL (BOOLIAN) STATEMENT

The BOOL statement allows the user to perform any of sixteen possible logical operations on the bit structure of two alphanumeric values. All characters of the alphanumerics are operated on including trailing spaces. If the alphanumerics are less than the specified maximum length trailing spaces are added. Parts of the alphanumerics can be operated on using the STR(function.

The general form of the BOOL statement is shown to the right.

The hex digit following BOOL specifies which of sixteen logical operations is to be used.

GENERAL FORM

$$\text{BOOL x} \left(\text{alpha variable, } \begin{Bmatrix}\text{xx} \\ \text{alpha variable}\end{Bmatrix}\right)$$

where: x = hexadecimal digit (i.e., 0-9 or A-F)

The characters of the first alpha variable are operated on and changed by the second value.

There are Immediate and String to String logical functions. Immediate Addition is specified if the second value in the BOOL statement is a hex digit. String to String Addition is specified if the second value in the BOOL statement is an alphanumeric value.

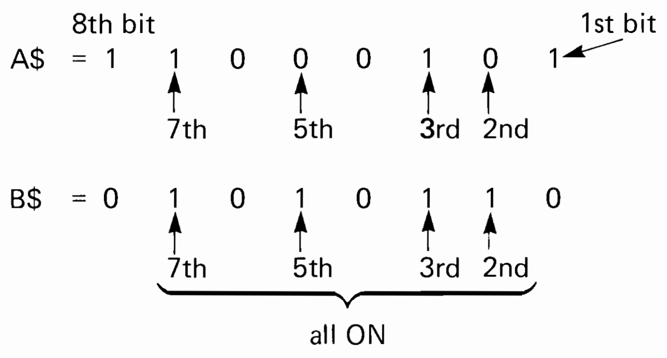
For those users who are familiar with symbolic logic and know in advance what type of logical operation is to be performed, the table to the right lists the hex digit (Column 2) to use in the BOOL statement to generate the specified logical function in Column 1. For example BOOL 5 (A$, B$) causes A$ to be set equal to the complement of B$. The hex digit 5 specifies which of the sixteen logical functions is to be used (i.e., complement of value #2).

For those users who are not familiar with symbolic logic, using one of the hex digits in the BOOL statement is valueless until what is happening is understood. Therefore an example is used here as a suggested approach to learning about symbolic logic and how it is used.

Suppose you have a one byte variable A$ with a bit configuration as shown to the right. You want to use the BOOL function to test the 7th, 5th, 3rd and 2nd bits to see if they are all ON.

A way to do this is to create a 1-byte mask (B$) with the 7th, 5th, 3rd and 2nd bits turned ON to mark the bits to be tested in A$.

LOGICAL FUNCTION	HEX DIGIT	IF BITS			
		1	1	0	0
		1	0	1	0
		RESULT			
null	0	0	0	0	0
not OR	1	0	0	0	1
	2	0	0	1	0
complement of value #1	3	0	0	1	1
	4	0	1	0	0
complement of value #2	5	0	1	0	1
exclusive OR	6	0	1	1	0
not AND	7	0	1	1	1
AND	8	1	0	0	0
equivalence	9	1	0	0	1
value #2	A	1	0	1	0
value #1 implies value #2	B	1	0	1	1
value #1	C	1	1	0	0
value #2 implies value #3	D	1	1	0	1
OR	E	1	1	1	0
identity	F	1	1	1	1



You must then formulate a rule(s) for comparing A$ with B$. For example:

1. A zero bit in B$ gives a zero bit no matter what the corresponding bit on A$ is as you are not concerned with testing these bits (i.e., 1st, 4th, 6th and 8th).
2. A one-bit in B$ gives a zero bit if the corresponding bit in A$ is OFF (0).
3. A one-bit in B$ gives a one bit if the corresponding bit in A$ is ON.

The result of this comparison based upon these rules is shown to the right.

```
A$     = 1 1 0 0 0 1 0 1
B$     = 0 1 0 1 0 1 1 0
         ---------------
Result = 0 1 0 0 0 1 0 0
```

If the bits to be tested in A$ were all ON (that which you are testing for) the result would exactly look like B$ (01010110). If the bits to be tested were not all ON (as in this example) the result would be different from B$.

To find the BOOL function which duplicates this rule, you must apply these same rules to two hypothetical values. These hypothetical values are given to you and are shown to the right. They are obtained as the result of comparing any two bits (one form each value). In this comparison four possible combinations can exist. Either both bits are ON (1) or both bits are OFF (2); or the first bit is ON and the second bit is OFF (3); or vice versa (4).

Bit from Value 1	1 1 0 1	—hypothetical value 1
Bit from Value 2	1 0 1 0	—hypothetical value 2

Now compare these two hypothetical values based upon your rules. The results are shown to the right.

```
Hypothetical
  Value 1        1 1   0 0
Hypothetical
  Value 2        1 0   1 0
                 ---------
Rules give ───►  1 0   0 0
```

If you look at Column 3 in the table (Page 213) you will find your results (1000). Match this result with a hex digit from Column 2. As you can see this is hex digit 8 and the logical operation AND.

The BOOL statement that would be used to perform the logical AND operation on your values is shown to the right.

```
:20 BOOL 8 (A$,B$)
```

Column 3 in the table was arrived at by applying the rules for each logical operation listed in Column 1 to the two hypothetical values shown at the top of Column 3. Therefore to determine which logical operation (hex digit) is to be used in the BOOL statement, you must apply any rules you use to these two hypothetical values, and match the results with those in Column 3. Then looking across the table you can determine the appropriate hex digit to use.

Some additional examples are used to help illustrate uses of other logical operations.

Explanation of Example 1: In Example 1 an alphanumeric value (A$) is compared to Hex (00) on a character by character basis. The BOOL statement asks that logical operation (3) be performed on these two values. This means that when comparing the two hypothetical values at the top of Column 3 in the table, the results are 0011 (the complement of Value #1). To develop a set of rules for this you would have:

1) If the first bit is ON, the result is OFF no matter what the second bit is.
2) If the first bit is OFF the result is ON no matter what the second bit is.

Applying these rules to the bit structure of the values in Example 1, you obtain the results as shown to the right.

EXAMPLE 1:

```
:10 A$=HEX(5432)
:20 BOOL 3(A$,00)
```

SETS A$ = HEX(ABCD)

```
Hypothetical Value 1      1 1 0 0
Hypothetical Value 2      1 0 1 0
Results of Logical  ───►  0 0 1 1
Operation 3
```

```
              A$  =  01010100    00110010    (i.e., 54   32)
              00  =  00000000    00000000    (i.e., 00   00)
Result ──────────►   10101011    11001101
                      A    B      C    D     HEX(ABCD)
```

Explanation of Example 2: In Example 2 Value 1 (A$) is compared to value 2 (B$) on a character by character basis.

The BOOL statement asks that logical operation (7) be performed on these two values. This means that when the two hypothetical values at the top of Column 3 are compared, the results are 0111 (a NOT AND operation).

To develop a set of rules for this you would have:

1) If both bits are ON, the result is OFF.
2) All other combinations result in an ON bit.

Applying these rules to the bit structure of the values in Example 2, you obtain the results as shown to the right.

If the second value is shorter in a String to String logical function the remaining characters in the first value are unchanged.

EXAMPLE 2:

```
:10 A$=HEX(4145):B$=HEX(2185)
:20 BOOL 7(A$,B$)
```

$$\begin{array}{ll}
\text{Hypothetical Value 1} & 1\ 1\ 0\ 0 \\
\text{Hypothetical Value 2} & \underline{1\ 0\ 1\ 0} \\
 & 0\ 1\ 1\ 1
\end{array}$$

$$\begin{array}{lllll}
 & A\$ = & 01000001 & 01000101 & (\text{i.e., } 41 \quad 45) \\
 & B\$ = & \underline{00100001} & \underline{10000101} & (\text{i.e., } 21 \quad 85) \\
\text{Result of NOT AND} \longrightarrow & & \underbrace{1111}_{F}\underbrace{1110}_{E} & \underbrace{1111}_{F}\underbrace{1010}_{A} & \text{HEX(FEFA)}
\end{array}$$

Chapter 21 Data Conversion

There are several statements in the System 2200B which are used for DATA conversion. These statements are CONVERT, VAL, BIN and HEXPRINT. Each is explained in this chapter.

Section 21-1 The BIN Statement

The BIN statement converts the integer portion of a numeric expression (must be < 256) to a binary number. The first character of an alphanumeric value is then set equal to the character with a bit configuration equivalent to this binary number.

The general form of the BIN statement is shown to the right (see example 1).

Explanation of Example 1: 64 is converted to its binary equivalent (01000000), which is equivalent to HEX Code 40, which represents the character @. The first character in A$ is therefore converted to @.

GENERAL FORM

BIN (alpha variable) = expression

where: $0 \leq$ value of expression < 256

EXAMPLE 1:

```
:10 A$="ABCDEFG"
:20 BIN(A$)=64
:30 PRINT A$
:RUN
@BCDEFG
```

Parts of strings can be operated on using the STR(function (see Example 2).

Explanation of Example 2: 64 is converted to its binary equivalent (01000000) which is equivalent to HEX Code 40 or the character @. The third character of A$ is converted to @.

An example of a program using the BIN statement is shown to the right.

EXAMPLE 2:

```
:10 A$="ABCDEFG"
:20 BIN(STR(A$,3,1))=64
:30 PRINT A$
:RUN
AB@DEFG
```

```
:10 REM TO PACK TODAYS DATE. DATE IS IN
:20 REM THE FORM MM/DD/YY WHERE C1=MM,
:30 REM C2=DD,C3=YY
:40 DIM D$3
:50 INPUT "MONTH, DAY, YEAR",C1,C2,C3
:60 BIN(STR(D$,1,1))=C1
:70 BIN(STR(D$,2,1))=C2
:80 BIN(STR(D$,3,1))=C3
```

Section 21-2
The VAL Function

The VAL function performs a binary conversion of the value of the first character of an alphanumeric or literal string (must be ⩽ 1111111) to a numeric. The VAL function can be used wherever numeric functions are normally used. It is the inverse of the BIN function.

The general form of the VAL statement is shown to the right (see examples).

GENERAL FORM

$$\text{VAL} \quad \left(\begin{Bmatrix}\text{alpha variable}\\ \text{literal string}\end{Bmatrix}\right)$$

Explanation of Example 1: A binary conversion is performed on the first character of A$. "A" equals 010000001 in binary or the numeric 65. Therefore, X is set equal to the numeric equivalent of character "A", which is 65.

Explanation of Example 2: The VAL statement is used with a literal string. The numeric equivalent of the character "A" is printed, namely 65.

The VAL function can also be used with the STR(function (see example).

Explanation of Example 3: A binary conversion is performed on the third character of A$ to a numeric equivalent (C=01000011 = 67). If this is less than 80, the value is printed.

The BIN and VAL functions are often used together. They are especially useful in code conversion. The program to the right illustrates a typical use of both functions.

In this program the character to be converted is input in line 10. The character is converted in line 20 to a numeric and this number is then used to set the data pointer to a particualr piece of data read by statement 30.

The new value (read in line 30) is then converted to a binary number and the first character in B$ is set equal to the character with a bit configuration equivalent to this binary number. The data in lines 100, 110, etc., would be selected so the number read into X and converted to B$ would be the equivalent of the input character in another code system.

EXAMPLE 1:

```
:10 A$="ABC"
:20 X=VAL(A$)
:30 PRINT "X=";X
:RUN
X= 65
```

EXAMPLE 2:

```
:10 PRINT VAL("A")
:RUN
 65
X= 32
```

EXAMPLE 3:

```
:10 A$="ABCDEF"
:20 IF VAL(STR(A$,3,1))<80 THEN 40
:
:
:40 PRINT VAL(STR(A$,3,1))
:
```

PROGRAM

```
:10 INPUT A$
:20 RESTORE VAL(A$)+1
:30 READ X
:40 BIN(B$)=X
:
:
:
:100 DATA 10,99,8,255,1,17,26,2
:110 DATA 31,59,48,62,112,7,213
```

Section 21-3
The HEXPRINT Statement

The HEXPRINT statement prints the hexadecimal codes of an alpha variable or alpha array. Trailing spaces in the variables *are not* ignored and are therefore printed (e.g., HEX(20)). Arrays are printed one element at a time in sequence with no separation. A carriage return is outputted after the printing of each alpha array or alpha variable unless a semicolon is used to separate the arguments.

The general form of the HEXPRINT statement is shown to the right (see examples).

Explanation of Example 1: A$ consists of the characters 1, 2 and 3. Using the HEXPRINT statement, these characters are converted to their equivalent hex codes, and then the codes are printed.

Explanation of Example 2: A$ equals three characters plus 13 trailing spaces. (Default dimension 16 characters.) The HEXPRINT statement converts all these characters to their equivalent hex codes and prints the codes.

Explanation of Example 3: Alpha variable (A$) and alpha array (B$) are both dimensioned as five characters. Each are assigned values in statement 20. The results of the conversion to hex codes are printed on two separate lines, due to the comma separating the arguments in the HEXPRINT statement.

GENERAL FORM

$$\text{HEXPRINT}\ \left\{\begin{matrix}\text{alpha variable}\\ \text{alpha array designator}\end{matrix}\right\}\left[\left\{\begin{matrix},\\ ;\end{matrix}\right\}\left\{\begin{matrix}\text{alpha variable}\\ \text{alpha array designator}\end{matrix}\right\}\ \ldots\ \left[\begin{matrix},\\ ;\end{matrix}\right]\right]$$

where: alpha array designator = alpha array name ( )   (i.e., A$( ))

EXAMPLE 1:

```
:5 DIM A$3
:10 A$="123"
:20 HEXPRINT A$
:RUN
313233        (the hex codes for
               characters 1, 2 and 3)
:
```

EXAMPLE 2:

```
:10 A$="ABC"
:20 PRINT "HEX VALUE OF A$=";
:30 HEXPRINT A$
:RUN
HEX VALUE OF A$=41424320202020202020202020202020
```

EXAMPLE 3:

```
:10 DIM A$5,B$(1,2)5
:20 A$="ABCDE":B$(1,1)="ABCDE":B$(1,2)="FGHIJ"
:30 HEXPRINT A$,B$()
:RUN
4142434445
4142434445464748494A
```

Section 21-4
The CONVERT Statement

The CONVERT statement is used to convert alphanumeric information to numeric form and vice versa. Therefore two formats are provided for the CONVERT statement. Each is explained in this section.

CONVERTING ALPHA TO NUMERICS

The CONVERT statement converts the value of an alphanumeric variable to a numeric value and then sets a numeric variable equal to that value.

Part of the general form of this statement is shown to the right.

The only restriction is that the original alphanumeric value must be a true representation of a valid BASIC number, otherwise an error results. For example A$ = "1234" or B$ = "5.5E07" are in legitimate numeric format, but C$ = "12A56" or L$ = "LM005" are not. However, the STR(function can be used with the CONVERT statement to isolate these portions of the alphanumeric which are in the correct format.

Some examples of the CONVERT statement are shown to the right.

Explanation of Example 1: The CONVERT statement converts the entire alpha variable A$ to its numeric equivalent and sets the variable X equal to that value.

GENERAL FORM

CONVERT alpha variable TO numeric variable

EXAMPLE 1:

```
:10 A$="1234"
:20 CONVERT A$ TO X
:30 PRINT "X=";X
:RUN
X= 1234
```

Explanation of Example 2: The CONVERT statement converts the eighth to the 16th characters of A$ to its numeric equivalent and sets the variable X equal to that value. The STR(function is used to isolate the portion of the string to be converted.

An example of the CONVERT function used in a program is shown to the right. This program is sufficiently documented with REM statements to explain its purpose.

NOTE:

One of the major values of being able to convert alpha to numeric is that it allows a program to be written (1) which gives the operator the ability to enter numeric data into an array (2) test if the data in the array is numeric with the NUM statement, (3) convert the array (i.e., alpha to numeric) and (4) use the numeric data in the program. In the process of doing this the data is validated as being correct before is is used (i.e., corrected).

EXAMPLE 2:

```
:10 A$="1234ABC4216.0543"
:20 CONVERT STR(A$,8,9) TO X
:30 PRINT "X=";X
:RUN
X= 4216.0543
```

```
:10 S=0
:15 DIM A$20
:20 A$="4.507.655.417.679.79"
:30 REM FIVE VALUES HAVE BEEN STORED IN A$ FOR
:40 REM SPACE SAVING PURPOSES. IT IS NOW DESIRED
:50 REM TO COMPUTE THE TOTAL OF THE FIVE VALUES.
:60 REM EACH OF THE FIVE VALUES IS MADE UP OF
:70 REM THREE DIGITS OF THE FORM #.##
:80 FOR I=1 TO 5
:90 CONVERT STR(A$,4*I-3,4) TO X
:100 S=S+X
:110 REM S IS THE TOTAL OF THE VALUES STORED IN A$
:120 NEXT I
:130 PRINT "THE TOTAL =";S
:RUN
THE TOTAL = 35.02

:_
```

CONVERTING NUMERICS TO ALPHA

The CONVERT statement also converts the numeric value of an expression to a character string and sets an alphanumeric variable equal to that character string. The image the numeric is to take on in the character string must be specified in the CONVERT statement.

Some of the major uses for this function are to form alphanumeric keys for data records (alphanumeric sorting) and for formatting output for plotting, etc.

The general form of the CONVERT statement is shown to the right.

GENERAL FORM

CONVERT expression TO alpha variable, (image)

where: image = [+] [# . . . .] [.] [# . . . .] [↑↑↑↑]

The image specifies the format of the converted expression. The image can have two possible formats:

FORMAT 1 = FIXED POINT (e.g., ##.##)
FORMAT 2 = FLOATING POINT (e.g., #.##↑↑↑↑)

Each character in the image specifies one byte in the resultant alphanumeric variable. These characters are # characters to specify digits, and +, –, ., and ↑ to specify sign, decimal point, and exponent characters.

Several rules are to be followed when formatting numerics:

1. If the image starts with a plus (+) sign, the sign of the value (+ or –) is edited into the character string (see example).

   The expression (X*2) is evaluated (–30.46) and the numeric is converted to a character string in the format of +##.## and set equal to A$. Notice the appropriate sign is edited into the character string.

```
:10 X=-15.23
:20 CONVERT X*2 TO A$,(+##.##)
:30 PRINT A$
:RUN
-30.46
```

2. If the image starts with a minus (–) sign a blank for positive values and a minus (–) for negative values is edited into the character string (see example).

   X is converted to a portion of a character string in the format of –##.## and inserted into the approriate place of the string A$. Notice a blank is left for the plus sign in the resultant string A$.

```
:10 X=15.23:A$="ABCDEFGHIJKLMN"
:20 CONVERT X TO STR(A$,3,6),(-##.##)
:30 PRINT A$
:RUN
AB 15.23IJKLMN
```

blank for the plus sign

3. If no sign is specified in the image, no sign is included in the character string (see example).

   X is converted to a character string in the format ### and set equal to A$. Notice the minus (–) sign is not edited into the character string as the image does not call for it.

```
:10 X=-450
:20 CONVERT X TO A$,(###)
:30 PRINT A$
:RUN
450
```

4. If the image has format 1, the value is edited into the character string as a fixed point number, truncating or extending with zeros any fraction and inserting leading zeros according to the image specification. The decimal point is edited in at the proper position. An error results if the numeric value exceeds the image specification (see example).

   The value for X is converted to a character string in the format of ####.##### and the character string is set equal to A$. Notice leading and trailing zeros are added to maintain the correct position of the decimal point.

```
:10 X=12.345
:20 CONVERT X TO A$,(####.####)
:30 PRINT A$
:RUN
0012.3450
```

5. If the image has format 2, the value is edited into the character string as a floating point number. The value is scaled as specified by the image (there are no leading zeros). The exponent is always edited in the form E±XX (see example).

   The value X is converted to a character string in the form (–#.#↑↑↑↑). This character string is then set equal to A$. Notice the decimal point is edited into the character string in the appropriate place and the exponent is determined correctly. Also, a leading blank is inserted into the image for the plus sign.

```
:10 X=12
:20 CONVERT X TO A$,(-#.#↑↑↑↑)
:30 PRINT A$
:RUN
 1.2E+01
```

blank for the plus sign

Chapter 22 Data Gathering

Several statements in System 2200 BASIC allow data to be loaded into memory from exterior peripheral devices. An in depth discussion of these statements can be found in the individual reference manuals for each peripheral. The only statement covered in this manual is the KEYIN statement.

Section 22-1 The KEYIN Statement

The KEYIN statement is used to receive information, *one character* at a time, from an input device. KEYIN assigns a value to the first character of an alphanumeric variable only, whereas the INPUT statement can assign a value(s) to either a numeric variable or alphanumeric variable. No CR/LF-EXECUTE is required to complete the data input with KEYIN, as is required with the INPUT statement.

The input can come from any one of several input devices (e.g., Model 2215 or 2222 Keyboards, Paper Tape Readers, Punched Card Readers, etc.). The input device is determined by which device has been selected for input. In all the examples in this manual, input is from the Model 2215 Keyboard, the default input device. (For users owning a Model 2222 Keyboard, the Model 2222 is the default device.) IF any other device is used for input, it must be selected with the SELECT statement. (See System 2200 Reference Manual and/or individual peripheral reference manuals for a further discussion of the SELECT statement.)

Another difference between the INPUT statement and the KEYIN statement is the INPUT statement requires the use of the CR/LF-EXECUTE key to process the input. The KEYIN statement, on the other hand, checks to see if a character is ready to come in from the input device. If a character is ready, it is received and put into the first character of the specified alphanumeric variable. (See general form of the KEYIN statement to the right.) Transfer then is made to the second line number in the KEYIN statement. If no character is ready to come in, no transfer is made and the next sequential statement in the program is executed. At no time is the CR/LF-EXECUTE key used to process the input; it is automatically processed once received.

GENERAL FORM

KEYIN alpha variable, line number, line number

Some examples of the KEYIN statement are shown to the right.

Explanation of Example 1: Execution of statement 10 causes the system to check if a character is ready to come in from the input device. If ready, the input is received and assigned to the first character of A$, then transfer is made to statement 100. If no character is ready, the next statement in the program is executed (i.e., GOTO 10). If a Special Function Key is touched, transfer goes to statement 200. Therefore, execution cycles through the KEYIN statement until a character or Special Function Key is touched.

EXAMPLE 1:

```
:10 KEYIN A$, 100, 200:GOTO 10
```

Explanation of Example 2: Execution of statement 10 results in the system transferring to line 100 after input is received or a Special Function Key is touched. If a character was not ready, the next statement in the program is executed.

EXAMPLE 2:

```
:10 KEYIN A$(1),100,100
:20 PRINT "CHARACTER NOT READY"
```

The program to the right illustrates the use of the KEYIN statement. The documentation within the program explains the purpose of the program.

```
:10 A$="BOB BONES"
:20 REM A$ WAS INCORRECTLY ENTERED AS
:30 REM "BOB BONES". THE 5TH CHARACTER
:40 REM SHOULD HAVE BEEN A "J" INSTEAD OF
:50 REM A "B".
:60 INPUT "ENTER POSITION OF INCORRECT CHARACTER",X
:80 KEYIN STR(A$,X,1),90,80
:85 GOTO 80
:90 PRINT A$
```

Enter this program.

Touch RESET.

Touch RUN CR/LF-EXECUTE

Touch 5 CR/LF-EXECUTE

Touch J

As soon as the requested input is received by the system, processing of the program continues and the new value of A$ is printed out.

Run the program, again substituting any letter you desire for the fifth character. You also can alter what character is to be changed by inputting another value for X in statement 60.

The KEYIN statement is executed so fast that statement 85 is used to loop back to statement 80 so that an operator can manually enter the correct character. Once the character is received, program execution continues automatically.

The major uses of the KEYIN statement are:

1) Editing capability.
2) Sampling the system with several low speed instruments and different keyboards.
3) In telecommunication systems where input comes in over telecommunication lines. The system can immediately process information as it is received over the lines without having to wait until the CR/LF-EXECUTE key is touched manually by an operator or continue processing if no character is ready.

```
:RUN
ENTER POSITION OF INCORRECT CHARACTER? 5
BOB JONES

:RUN
ENTER POSITION OF INCORRECT CHARACTER? 5
BOB TONES

:RUN
ENTER POSITION OF INCORRECT CHARACTER? 5
BOB AONES
```

Appendix A

WANG SYSTEM 2200 ASCII CHARACTER CODE SET

The following chart shows the ASCII codes (or Hex Codes) used by the System 2200. Each peripheral may not use all these codes. See the appropriate peripheral reference manual for the codes pertaining to a particular device. Codes not legal for certain devices may default to other characters.

HIGH ORDER HEXADECIMAL DIGIT OF CODE

Low Order Hexadecimal Digit Of Code

	0	1	2	3	4	5	6	7
0	NULL		SPACE	Ø	@	P		p
1	HOME (CRT)	X-ON	!	1 *	A	Q	a	q
2			"	2	B	R	b	r
3	CLEAR SCREEN (CRT)	X-OFF	#	3	C	S	c	s
4			$	4	D	T	d	t
5			%	5	E	U	e	u
6			&	6	F	V	f	v
7	BELL		' (apos)	7	G	W	g	w
8	BACKSPACE (CRT CURSOR ←)		(	8	H	X	h	x
9	HT (TAB) or (CRT CURSOR →)	CLEAR TAB	)	9	I	Y	i	y
A	LINE FEED (CRT CURSOR ↓)	SET TAB	*	:	J	Z	j	z
B	VT (VERTICAL TAB)		+	;	K	[	k	{
C	FORM FEED OR REV INDEX (CRT CURSOR ↑)		, (comma)	< or [	L	\	l	|
D	CR (CARRIAGE RETURN)		–	=	M	]	m	}
E	SO (SHIFT UP)	¢	.	> or]	N	↑ or or !	n	~
F	SI (SHIFT DOWN)	° (degree)	/	?	O	← or _	o	

*Example: 1 = ASCII 31 or HEX(31)

Appendix B

ESTIMATING PROGRAM MEMORY REQUIREMENTS

STEP 1: Estimate 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, TR(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$( )	=	7 X 10 elements X 4 bytes/element	=	380 bytes
T$( )	=	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 variables.
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 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 results 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

The Wang System 2200 BASIC checks for and displays syntax errors as each line is entered. The user may then correct the error before proceeding with his program. When any error is detected, the line being scanned by the system is displayed and on the next line, an "↑" symbol is placed at the point of the error followed by the error message number.

The example to the right shows the format of the System 2200 error pointer:

```
:10  DIM A(P)
            ↑ERR 13
```

The user may then refer to the listing of error messages to identify the error code number. The list contains a description of each and a suggested method for correcting the error.

> NOTE:
>
> *An error message can only indicate one possible type of error.*

Example 1:

```
:PXINT X
  ↑ERR 06 (expected equal sign)
```

In Example: 1 the system has interpreted 'P' as a variable and thus expects an equal sign following 'P'; whereas, the user may have meant:

```
:PRINT X
```

The system assumes the statement is correct until illegal syntax is discovered.

The error message, SYSTEM ERROR!, is displayed if certain hardware failures occur. The user should RESET or MASTER INITIALIZE (Power Off, Power On) the system and re-enter the sequence of events that produced this error.

> NOTE:
>
> *Certain combinations of illegal or meaningless operations may also result in a SYSTEM ERROR message.*

THREE TYPES OF ERRORS CAN OCCUR

A Syntax Error (Example: 2)

Results when the required format of a System 2200 BASIC statement is violated. Pressing a sequence of keys not recognized as an accepted combination results in this type of error. Syntax errors in a statement are recognized and noted, as soon as the execute key is touched to enter a statement. Examples of this type of error include misspelling verbs, illegal formats for numbers, operators, parentheses, and the improper use of punctuation.

An Error of Execution (Example: 3)

Results when an illegal arithmetic operation is performed, or the execution of an illegal statement or programming procedure is attempted when a program is executed. This type of error differs from a Syntax Error. The statement itself uses the proper syntax. However, the execution of the statement is impossible to perform and leads to an error condition. Typical errors of this type include illegal branches, arithmetic overflow or underflow, illegal "FOR" loops, etc.

A Programming Error

The System 2200 executes the statements entered properly, but the results obtained are not correct, because the wrong information or logic is used in writing a program. Although there is no way for the System 2200 to identify a programming error, debugging features such as TRACE, HALT/STEP, CONTINUE, can significantly speed up the process of debugging a program.

Example 2:

```
:10  DEFFN . (X) = 3*X↑2 - 2*X↑3
             ↑ERR 21
```

Example 3:

(Branch to non-existant statement number)

```
:100 GOTO 110
:105 PRINT "VALUES = " ;A, B, C
:120 END
:RUN
 100 GOTO 110
             ↑ERR 11
```

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 available space in memory for program text when line 820 was entered)

User must shorten or segment program.

CODE 02

Error:	Table Overflow
Cause:	All available space for internal operating system tables and variables has been used up (storage of variables, values, etc.) or a repetative program loop which illegally allows system tables to fill up was encountered. An example of the latter would be jumping out of FOR loops or subroutines without completing them.
Action:	Shorten or correct and/or chain the program by using COM statements and continue.

Example:

```
:10  DIM A(10), 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 06

Error:	Missing Equal Sign
Cause:	An equal sign (=) was expected.
Action:	Correct statement text.
Example:	

```
:10  DEFFNC(V) - V + 2
                 ↑ERR 06
:10  DEFFNC(V) = V+2                    (Possible Correction)
```

CODE 07

Error:	Missing Quotation Marks
Cause:	Quotation marks were expected.
Action:	Reenter the DATASAVE OPEN statement correctly.
Example:	

```
:DATASAVE OPEN TTTT"
                  ↑ERR 07
:DATASAVE OPEN "TTTT"                   (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 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 integer, producing an imaginary result, or X and Y were both zero. 6. ILLEGAL SIN, COS, OR TAN ARGUMENT. The function argument exceeds $2\pi \times 10^{11}$ 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:	

```
:10Y = INT(1.2↑5
               ↑ERR 05
:10Y = INT(1.2↑5)                              (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)
         ↑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 value exceeds the allowable limit. For example, a dimension is greater than 255 or 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 +
         ↑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 is referenced in the program which was not defined properly in a DIM or COM statement (i.e., an array variable was not defined in a DIM or COM statement or 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:	Reenter 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, or a branch was made into the middle of a subroutine.
Action:	Correct the program.

Example:

```
:10 FOR I=1 TO 20
:20 X=I*SIN(I*4)
:25 GOTO 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, or a branch was made into the middle of a FOR loop.
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 insufficient 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 Referenced 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     (Key in values and touch EXECUTE)
  ↑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:
```
:995 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
 AMOUNT =
: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
               ↑ERR 37
: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 A$ = "123456789ABCDEFG"
:20 B$ = STR(A$, 10, 8)
 RUN
:20 B$ = 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"
                ↑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
          ↑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
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 specification in a program statement is undefined.
Action:	Define the specified device numbers.

Example:

```
: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:	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 on bottom of cassette tape).
Action:	Mount another cassette or "unprotect" the tape cassette by covering the punched hole on the bottom of the cassette with the tab.
Example:	SAVE /103 ↑ERR 50

CODE 51

Error:	Illegal Statement
Cause:	The System 2200 does not have the capability 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 device was not positioned at a data record.
Action:	Make sure the correct device is 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:

```
:10 PRINTUSING 200, HEX(F4F5)
:200 % ###.##
:RUN
:10 PRINTUSING 200, HEX(F4F5)
                            ↑ERR 53
:10 A$ = HEX (F4F5)
:20 PRINTUSING 200,A$                     (Possible Correction)
```

CODE 54

Error: **Illegal Plot Argument**

Cause: An argument in the PLOT statement is illegal.

Action: Correct the PLOT statement.

Example:

```
100 PLOT < 5, , H >
                  ↑ERR 54
100 PLOT < 5, , C >                       (Possible Correction)
```

CODE 55

Error: **Illegal BT Argument**

Cause: An argument in a DATALOAD BT or DATASAVE BT statement is illegal.

Action: Correct the statement in error.

Example:

```
100 DATALOAD BT (M=50) A$
                   ↑ERR 55
100 DATALOAD BT (N=50) A$                 (Possible Correction)
```

CODE 56

Error: **Number Exceeds Image Format**

Cause: The value of the number being packed or converted is greater than the number integer digits provided for in the pack or convert image.

Action: Change the image specification.

Example:

```
100 PACK (##) A$ FROM 1234
                         ↑ERR 56
100 PACK (####) A$ FROM 1234                (Possible Correction)
```

CODE 57

Error: **Illegal Disk Sector Address**

Cause: Illegal disk sector address specified; value is negative or greater than 32767. (The System 2200 cannot store a sector address greater than 32767.)

Action: Correct the program statement in error.

Example:

```
100 DATASAVE DAF (42000 ,X) A,B,C
                        ↑ERR 57
100 DATASAVE DAF (4200 ,X) A,B,C            (Possible Correction)
```

CODE 58

Error: **Expected Data Record**

Cause: A program record or header record was read when a data record was expected.

Action: Correct the program.

Example:

```
100 DATALOAD DAF(0,X) A,B,C
                      ↑ERR 58
```

CODE 59

Error: **Illegal Alpha Variable For Sector Address**

Cause: Alphanumeric receiver for the next available address in the disk DA instruction is not at least 2 bytes long.

Action: Dimension the alpha variable to be at least two characters long.

Example:

```
10 DIM A$1
100 DATASAVE DAR( ) ,A$ ) X, Y, Z
                          ↑ERR 59
10 DIM A$2                                  (Possible Correction)
```

CODE 60

Error:	Array Too Small
Cause:	The alphanumeric array does not contain enough space to store the block of information being read from disk or tape or being packed into it. For cassette tape and disk records, the array must contain at least 256 bytes (100 bytes for 100 byte cassette blocks).
Action:	Increase the size of the array.

Example:

```
10 DIM A$(15)
20 DATALOAD BT A$( )
                         ↑ERR 60
10 DIM A$(16)                                          (Possible Correction)
```

CODE 61

Error:	Disk Hardware Error
Cause:	The disk did not recognize or properly respond back to the System 2200 during read or write operation in the proper amount of time.
Action:	Run program again. If error persists, re-initialize the disk; contact WANG Service Representative.

Example:

```
100 DATASAVE DCF X,Y,Z
                      ↑ERR 61
```

CODE 62

Error:	File Full
Cause:	The disk sector being addressed is not located within the catalogued specified file. When writing the file is full, for other operations, a SKIP or BACKSPACE has set the sector address beyond the limits of the file.
Action:	Correct the program.

Example:

```
100 DATASAVE DCT#2, A$( ), B$( ), C$( )
                                    ↑ERR 62
```

CODE 63

Error: Missing Alpha Array Designator

Cause: An alpha array designator (e.g., A$( )) was expected. (Block operations for cassette and disk require an alpha array argument.)

Action: Correct the statement in error.

Example:

```
100 DATALOAD BT A$
                 ↑ERR 63
100 DATALOAD BT A$( )                    (Possible Correction)
```

CODE 64

Error: Sector Not On Disk

Cause: The disk sector being addressed is not on the disk. (Maximum legal sector address depends upon the model of disk used.)

Action: Correct the program statement in error.

Example:

```
100 MOVEEND F = 10000
                   ↑ERR 64
100 MOVEEND F = 9791                     (Possible Correction)
```

CODE 65

Error: Disk Hardware Malfunction

Cause: A disk hardware error occurred (i.e., the disk is not in file ready position. This could occur, for example, if the disk is in LOAD mode or power is not turned on).

Action: Insure disk is turned on and properly setup for operation. Set the disk into LOAD mode and then back into RUN mode, with the RUN/LOAD selection switch. The check light should then go out. If error persists call your WANG Service Representative.
(Note, the disk should never be left in LOAD mode when running.)

Example:

```
100 DATALOAD DCF A$,B$
                  ↑ERR 65
```

CODE 66	
Error:	Format Key Engaged
Cause:	The disk format key is engaged. (The key is normally engaged only when formatting a disk pack.)
Action:	Turn off the format key.
Example:	100 DATASAVE DCF X,Y,Z ↑ERR 66

CODE 67	
Error:	Disk Format Error
Cause:	A disk format error was detected on disk read or write. The disk is not properly formatted such that sector addresses can be read.
Action:	Format the disk again.
Example:	100 DATALOAD DCF X,Y,Z ↑ERR 67

CODE 68	
Error:	LRC Error
Cause:	A disk longitudinal redundancy check error occurred when reading a sector. The data may have been written incorrectly, or the System 2200/Disk Controller could be malfunctioning.
Action:	Run program again. If error persists, re-write the bad sector. If error still persists, call WANG Service Representative.
Example:	100 DATALOAD DCF A$( ) ↑ERR 68

CODE 71	
Error:	**Cannot Find Sector**
Cause:	A disk seek error occurred; the specified sector could not be found on the disk.
Action:	Run program again. If error persists, re-initialize (reformat) the disk pack. If error still occurs call WANG Service Representative.
Example:	100 DATALOAD DCF A$( )

```
100 DATALOAD DCF A$( )
                     ↑ERR 71
```

CODE 72	
Error:	**Cyclic Read Error**
Cause:	A cyclic redundancy check disk read error occurred; the sector being addressed has never been written to or subsequently the sector was incorrectly written on disk (i.e., the disk pack was never initially formatted).
Action:	Format the disk if it was not done. If the disk was formatted, re-write the bad sector, or reformat the disk. If error persists call WANG Service Representative.
Example:	100 MOVEEND F =8000

```
100 MOVEEND F =8000
                ↑ERR 72
```

CODE 73	
Error:	**Illegal Altering Of A File**
Cause:	The user is attempting to rename or write over an existing scratched file, but is not using the proper syntax. The scratched file name must be referenced.
Action:	Use the proper form of the statement.
Example:	SAVE DCF "SAM1"

```
SAVE DCF "SAM1"
              ↑ERR 73
SAVE SCF ("SAM1") "SAM1"                (Possible Correction)
```

CODE 74

Error:	Catalog End Error
Cause:	The end of catalog area falls within the library index area or has been changed by MOVEEND to fall within the area already used by the catalog; or there is no room left in the catalog area to store more information.
Example:	SCRATCH DISK F LS=100, END=50

```
SCRATCH DISK F LS=100, END=50
                          ↑ERR 74
SCRATCH DISK F LS=100, END 500                (Possible Correction)
```

CODE 75

Error:	Command Only (Not Programmable)
Cause:	A command is being used within a BASIC program. Commands are not programmable.
Action:	Do not use commands as program statements.
Example:	10 LIST

```
10 LIST
   ↑ERR 75
```

CODE 76

Error:	Missing < or > (Plot Enclosures)
Cause:	The required PLOT enclosures are not in the PLOT statement.
Action:	Correct the statement in error.
Example:	100 PLOT A, B "*"

```
100 PLOT A, B "*"
         ↑ERR 76
100 PLOT <A, B, "*">                          (Possible Correction)
```

CODE 77

Error: Starting Sector Greater Than Ending Sector

Cause: The starting sector address specified is greater than the ending sector address specified.

Action: Correct the statement in error.

Example:

```
10 COPY FR(1000, 100)
                        ↑ERR 77
10 COPY FR(100, 1000)                              (Possible Correction)
```

CODE 78

Error: File Not Scratched

Cause: A file is being renamed that has not been scratched.

Action: Scratch the file before renaming it.

Example:

```
SAVE DCF ("LINREG") "LINREG2"
                        ↑ERR 78
SCRATCH F "LINREG"                                 (Possible Correction)
SAVE DCF ("LINREG") "LINREG2"
```

CODE 79

Error: File Already Catalogued

Cause: An attempt was made to catalogue a file with a name that already exists in the catalogue index.

Action: Use a different name.

Example:

```
SAVE DCF "MATLIB"
                      ↑ERR 79
SAVE DCF "MATLIB1"                                 (Possible Correction)
```

CODE 80

Error:	**File Not In Catalog**
Cause:	The error may occur if one attempts to address a non-existing file name or to load a data file as a program or open a program file as a data file.
Action:	Make sure you are using the correct file name; make sure the proper disk pack is mounted.
Example:	

```
LOAD DCR "PRES"
         ↑ERR 80
LOAD DCF "PRES"                    (Possible Correction)
```

CODE 81

Error:	**/XXX Device Specification Illegal**
Cause:	The /XXX device specification may not be used in this statement.
Action:	Correct the statement in error.
Example:	

```
100 DATASAVE DC /310, X
                ↑ERR 81
100 DATASAVE DC #1, X              (Possible Correction)
```

CODE 82

Error:	**No End Of File**
Cause:	No end of file record was recorded on file and therefore could not be found in a SKIP END operation.
Action:	Correct the file.
Example:	

```
100D SKIP END
       ↑ERR 82
```

CODE 83

Error: Disk Hardware Failure

Cause: A disk address cannot be properly transferred from the System 2200 to the disk when processing MOVE or COPY.

Action: Run program again. If error persists, call WANG Service Representative.

Example:

```
COPY FR(100,500)
               ↑ERR 83
```

CODE 84

Error: **Not Enough System 2200 Memory Available For MOVE or COPY**

Cause: A 1K buffer is required in memory for MOVE or COPY operation (i.e., 1000 bytes should be available and not occupied by program and variables).

Action: Clear out all or part of program or program variables before MOVE or COPY.

Example:

```
COPY FR(0, 9000)
               ↑ERR 84
```

CODE 85

Error: **Read After Write Error**

Cause: The comparison of read after write to a disk sector failed. The information was not written properly.

Action: Write the information again. If error persists, call WANG Service Representative.

Example:

```
100 DATASAVE DCF$ X, Y, Z
                   ↑ERR 85
```

CODE 86

Error: **File Not Open**

Cause: The file was not opened.

Action: Open the file before reading from it.

Example:

```
100 DATALOAD DC A$
                ↑ERR 86
10 DATALOAD DC OPEN F "DATFIL"             (Possible Correction)
```

CODE 87

Error:	**Common Variable Required**
Cause:	The variable in the LOAD DA statement, used to receive the sector address of the next available sector after the load, is not a common variable.
Action:	Define the variable to be common.
Example:	see below

```
10 LOAD DAR (100,L)
                  ↑ERR 87
5  COM L                                    (Possible Correction)
```

CODE 88

Error:	**Library Index Full**
Cause:	There is no more room in the index for a new name.
Action:	Scratch any unwanted files and compress the catalog using a MOVE statement or mount a new disk platter.
Example:	see below

```
SAVE DCF "PRGM"
                 ↑ERR 88
```

CODE 89

Error:	**Matrix Not Square**
Cause:	The dimensions of the operand in a MAT inversion or identity are not equal.
Action:	Correct the array dimensions.
Example:	see below

```
:10 MAT A=IDN(3,4)
:RUN
 10 MAT A=IDN(3,4)
                  ↑ERR 89
:10 MAT A=IDN(3,3)                          (Possible Correction)
```

CODE 90

Error:	Matrix Operands Not Compatible
Cause:	The dimensions of the operands in a MAT statement are not compatible; the operation cannot be performed.
Action:	Correct the dimensions of the arrays.

Example:

```
:10 MAT A=CON(2,6)
:20 MAT B=IDN(2,2)
:30 MAT C=A+B
:RUN

 30 MAT C=A+B
           ↑ERR 90
:10 MAT A=CON(2,2)                     (Possible Correction)
```

CODE 91

Error:	Illegal Matrix Operand
Cause:	The same array name appears on both sides of the equal sign in a MAT multiplication or transposition statement.
Action:	Correct the statement.

Example:

```
:10 MAT A=A*B
         ↑ERR 91
:10 MAT C=A*B                          (Possible Correction)
```

CODE 92

Error: **Illegal Redimensioning Of Array**

Cause: The space required to redimension the array is greater than the space initially reserved for the array.

Action: Reserve more space for array in DIM or COM statement.

Example:

```
:10 DIM(3,4)
:20 MAT A=CON(5,6)
:RUN
 20 MAT A=CON(5,6)
                  ↑ERR 92
:10 DIM A(5,6)                              (Possible Correction)
```

CODE 93

Error: **Singular Matrix**

Cause: The operand in a MAT inversion statement is singular and cannot be inverted.

Action: Correct the program.

Example:

```
:10 MAT A=ZER(3,3)
:20 MAT B = INV(A)
:RUN
 20 MAT B=INV(A)
                ↑ERR 93
```

CODE 94

Error: **Missing Asterisk**

Cause: An asterisk (*) was expected.

Action: Correct statement text.

Example:

```
:10 MAT C=(3)B
              ↑ERR 94
:10 MAT C=(3)*B                             (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 EQUAL 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 OF 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
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
CODE 54	ILLEGAL PLOT ARGUMENT
CODE 55	ILLEGAL BT ARGUMENT
CODE 56	NUMBER EXCEEDS IMAGE FORMAT
CODE 57	ILLEGAL SECTOR ADDRESS
CODE 58	EXPECTED DATA RECORD
CODE 59	ILLEGAL ALPHA VARIABLE FOR SECTOR ADDRESS
CODE 60	ARRAY TOO SMALL
CODE 61	DISK HARDWARE ERROR
CODE 62	FILE FULL
CODE 63	MISSING ALPHA ARRAY DESIGNATOR
CODE 64	SECTOR NOT ON DISK
CODE 65	DISK HARDWARE MALFUNCTION
CODE 66	FORMAT KEY ENGAGED
CODE 67	DISK FORMAT ERROR
CODE 68	LRC ERROR
CODE 71	CANNOT FIND SECTOR
CODE 72	CYCLIC READ ERROR
CODE 73	ILLEGAL ALTERING OF A FILE
CODE 74	CATALOG END ERROR
CODE 75	COMMAND ONLY (NOT PROGRAMMABLE)

CODE 76	MISSING < OR > (PLOT ENCLOSURES)
CODE 77	STARTING SECTOR > ENDING SECTOR
CODE 78	FILE NOT SCRATCHED
CODE 79	FILE ALREADY CATALOGED
CODE 80	FILE NOT IN CATALOG
CODE 81	/XXX DEVICE SPECIFICATION ILLEGAL
CODE 82	NO END OF FILE
CODE 83	DISK HARDWARE FAILURE
CODE 84	NOT ENOUGH MEMORY FOR MOVE OR COPY
CODE 85	READ AFTER WRITE ERROR
CODE 86	FILE NOT OPEN
CODE 87	COMMON VARIABLE REQUIRED
CODE 88	LIBRARY INDEX FULL
CODE 89	MATRIX NOT SQUARE
CODE 90	MATRIX OPERANDS NOT COMPATIBLE
CODE 91	ILLEGAL MATRIX OPERAND
CODE 92	ILLEGAL REDIMENSIONING OF ARRAY
CODE 93	SINGULAR MATRIX
CODE 94	MISSING ASTERISK

Appendix D

INDEX TO PROGRAMS (PROBLEMS) IN THIS MANUAL

INDEX TO PROGRAMS (PROBLEMS) IN THIS MANUAL (Continued)

INDEX TO PROGRAMS (PROBLEMS) IN THIS MANUAL (Continued)

Appendix E

AVAILABLE PERIPHERALS

Model Number	Product Name
2201	Output Writer
2202*	Plotting Output Writer
2203*	Punched Tape Reader
2207*	I/O Interface Controller (RS-232-C)
2212*	Analog Flatbed Plotter (10" x 15")
2214	Mark Sense Card Reader
2215	BASIC Keyword Keyboard
2216	CRT Executive Display
2217	Single Tape Cassette Drive
2216/2217	Combined CRT Executive Display/Single Tape Cassette Drive
2218	Dual Tape Cassette Drive
2219	I/O Extended Chassis
2221	Line Printer (132 Column)
2222	Alpha-Numeric Typewriter Keyboard
2227	Telecommunications Controller
2230-1*	Fixed/Removable Disk Drive (1,228,800 bytes)
2230-2*	Fixed/Removable Disk Drive (2,457,600 bytes)
2230-3*	Fixed/Removable Disk Drive (5,013,504 bytes)
2231	Line Printer (80 Column)
2232*	Digital Flatbed Plotter (31" x 42")
2234*	Hopper-Feed Punched Card Reader
2240-1*	Dual Removable Flexible Disk Drive
2240-2*	Dual Removable Flexible Disk Drive
2241	Thermal Printer (80 Column)
2243*	Triple Removable Flexible Disk Drive

AVAILABLE PERIPHERALS (Continued)

Model Number	Product Name
2244*	Hopper-Feed Mark Sense/Punched Card Reader
2250	I/O Interface Controller (8-Bit-Parallel)
2252	Input Interface Controller (BCD 10-Digit-Parallel)
2261	High Speed Printer (132 Column)
2290	CPU/Peripheral Stand

*Peripheral used with the System 2200B only. A System 2200A can be upgraded to a System 2200B upon request at a nominal charge.

Appendix F

DEVICE ADDRESSES FOR SYSTEM 2200 PERIPHERALS

A complete list of the System 2200 I/O devices and addresses is shown in the table to the right.

DEVICE ADDRESSES FOR SYSTEM 2200 PERIPHERALS
(For further detail, see the individual peripheral manuals.)

I/O DEVICE CATEGORIES	DEVICE ADDRESS (S)*	
KEYBOARDS** (2215, 2222)	001, 002, 003, 004	
CRT UNITS** (2216)	005, 006, 007, 008	
CASSETTE DRIVES (2217, 2218)	10A, 10B, 10C, 10D, 10E, 10F	
LINE PRINTERS (2221, 2218)	215, 216	
OUTPUT WRITER (2201)	211, 212	
THERMAL PRINTER (2241)	211, 212	
PLOTTERS (2202, 2212, 2232)	413, 414	
DISK DRIVES (2230-1, -2, -3) (2240-1, -2)	310, 320, 330	
CARD READERS (2214)	517	
PUNCHED PAPER TAPE READER (2203)	618	
TELETYPES (2207)	019, 01A, 01B	INPUT
	01D, 01E, 01F	OUTPUT
TELETYPE TAPE UNITS	41D, 41E, 41F	PUNCH/ READER
TELECOMMUNICATIONS (2227)	219, 21A, 21B	INPUT
	21D, 21E, 21F	OUTPUT

* 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. Therefore if a System 2200 has only one device of a particular category, such as a cassette, it is set up with the first device address listed (10A in the case of the cassette). If it has two cassettes, they are set up with device addresses 10A and 10B. Each device address is printed on the interface card which controls that device.

** All peripherals in this category are assigned to lowest device address shown. They may, however, be assigned unique addresses by customer request.

Index

Index

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

(Please tape. Postal regulations prohibit the use of staples.)

Cut along dotted line.

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

Printed in U.S.A.

WANG LABORATORIES (CANADA) LTD.
180 Duncan Mill Road
Don Mills, Ontario M3B 1Z6
TELEPHONE (416) 449-7890
TELEX 06-966546

WANG EUROPE, S.A.
Buurtweg 13
9412 Ottergem
Belgium
TELEPHONE: 053/74514
TELEX: 26077

WANG ELECTRONICS LTD.
1, Olympic Way, 4th Floor
Wembley Park,
Middlesex, England
TELEPHONE: 01/903/6755
TELEX: 923498

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 (0611) 252061-64

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

WANG 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.
700-3231
2-74-2.5M
Price $10.00