WANG

SYSTEM 2200

ALPHABETICAL INDEX

Wang BASIC Language Reference Manual

WANG LABORATORIES, INC.

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

Disclaimer of Warranties and Limitation of Liabilities

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

This manual has been written for the sole purpose of providing quick answers to questions concerning the operation of the System 2200. It is designed to be used in conjunction with the Introductory Manual.

The manual is divided into ten sections covering all the features of Wang 2200 BASIC. The non-programmable commands in Section VI and the BASIC statements in Section VII are arranged in alphabetical order for ease of locating a desired command or statement.

If you are seeing, reading, and hearing about the System 2200 and its BASIC language for the first time, we strongly recommend you first read the BASIC Programming Manual which discusses in detail the operational and programming features of the System 2200. Once you have completed the BASIC Programming Manual, then you can use this manual as a reference for specific questions concerning the operation of the System 2200.

PREFACE

This manual provides the user with a quick and easy reference guide to questions concerning the operation of the System 2200. The layout is designed to assist the user in the location of key information.

The manual is divided into ten sections as described below. The title page for each section contains a table of contents of the section. All BASIC statements and functions are included in the index provided inside the covers of the manual. For turn-on procedures and preliminary operating instructions, see the Introductory Manual.

Section I	Introduces you to Wang BASIC and details the various available systems.
Section II	The basic structure and components of the system are covered in this section, such as: syntax rules and terminology, line numbers, spacing, colons, Immediate Mode vs. Programming Mode, and the edit and debug features.
Section III	This section describes the elements of a numeric expression including Numeric Variables, Numeric Constants, System-defined Math Functions, Common Variables, Random Numbers and User-defined Functions.
Section IV	Alphanumeric capabilities are covered in this section, such as: Alpha Strings, Variables, Literal Strings, Alpha Functions, Hexadecimal and String Functions.
Section V	I/O Device Selection procedures are illustrated in this section; such things as Device Address for peripherals, Default Address, Input/Output Parameters.
Section VI	This section describes, in alphabetical order, the non-programmable commands necessary to communicate with the system.
Section VII	All the General BASIC statements available in the system are covered here, arranged in alphabetical order.
Section VIII	This section provides BASIC syntax for statements and commands used to operate peripherals. The user should also refer to the appropriate peripheral manuals.
Section IX	This section illustrates the various errors that can occur in both machine and programming techniques, and includes one of many ways in which an error can be corrected.
Appendices	This last section is divided into four subsections: Appendix A, ASCII and HEX codes with CRT Character Set; Appendix B, Cross Reference Chart for ASCII, HEX, Binary and VAL; Appendix C, Device Addresses; Appendix D, a description of the hexadecimal system; Appendix E, CPU Specifications; Appendix F, a list of the error messages, and Appendix G, a glossary.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Section I
Wang BASIC

Section I Wang BASIC

INTRODUCTION

Wang System 2200 BASIC is a computer language resembling English, modelled after the BASIC language first written at Dartmouth College. This manual describes 2200 BASIC and gives the programmer the rules for writing programs. It applies to all the following System 2200 computers:

System 2200A	System 2200S	WCS/20
System 2200B	System 2200T	WCS/30
System 2200C	WCS/10	

Each of these computers utilizes a slightly different subset of the available 2200 BASIC statements as follows:

System	Standard BASIC Statements			I/O BASIC (tape cassette statements)	
2200A	COM	GOTO	READ	BACKSPACE	
	DATA	IF END THEN	REM	DATALOAD	
	DEFFN	IF . . . THEN	RESTORE	DATARESAVE	
	DEFFN'	% (Image)	RETURN	DATASAVE	
	DIM	INPUT	SELECT	LOAD	
	END	LET	STOP	REWIND	
	FOR	NEXT	TRACE	SKIP	
	GOSUB	PRINT			
	GOSUB'	PRINTUSING			
		Functions		**Commands**	
	ABS	INT	STR	LOAD	
	ARCCOS	LEN	SQR	SAVE	
	ARCSIN	LOG	TAB		
	ARCTAN	RND	TAN		
	COS	SGN	X^n		
	EXP	SIN	# PI		
	HEX				
		Commands			
	CLEAR	LIST	RUN		
	CONTINUE	RENUMBER			
2200B	All 'A' statements, commands and functions plus the following:			All cassette statements and commands plus the following for all other peripherals:	
		Statements			
	ADD	INIT		PLOT	
	AND	KEYIN		DATALOAD BT	
	BIN	ON . . . GOSUB/GOTO		DATASAVE BT	
	BOOL	OR		COPY	DSKIP
	CONVERT	PACK		DATALOAD BA	LIMITS
	HEXPRINT	ROTATE		DATALOAD DA	LOAD DA (command)
		UNPACK		DATALOAD DC	LOAD DC
		XOR		DATALOAD DC OPEN	MOVE
		Functions		DATASAVE BA	MOVE END
	NUM	VAL		DATASAVE DA	SAVE DA (command)
	POS			DATASAVE DC	SAVE DC
				DATASAVE DC CLOSE	SCRATCH
				DATASAVE DC OPEN	SCRATCH DISK
				DBACKSPACE	VERIFY

System	Standard BASIC	I/O BASIC
2200C	All 'B' statements, commands and functions plus the following: COM CLEAR ON ERROR DEFFN' HEX RETURN CLEAR	All 'B' I/O statements and commands.
2200S WCS/10	All 'A' statements, commands and functions plus the following: CONVERT ON HEXPRINT RETURN CLEAR KEYIN VAL Function NUM Function	All 'A' I/O statements and commands.
2200T WCS/20 WCS/30	All 'C' statements, commands and functions p lus the following: MAT + MAT ZER $ GIO MAT CON MAT INPUT $IF ON MAT = MAT PRINT $PACK MAT IDN MAT REDIM $UNPACK MAT INV,d MAT COPY $TRAN MAT * MAT CONVERT MAT READ MAT MERGE MAT ( ) * MAT SEARCH MAT – MAT MOVE MAT TRN MAT SORT	All 'C' I/O statements and commands.

For a chart indicating the peripherals and options which can be used with each system, see the Introductory Manual.

In this manual all BASIC statements except for

- MAT . . . statements
- $ . . . statements
- disk statements

are described. Refer to Matrix, Sort, General I/O and Disk Manuals for information on these instructions. Refer to peripheral manuals for information on operation of peripherals attached to your system. A final section of this manual provides BASIC syntax for all peripheral statements except those used in disk operations.

For turn-on operating procedures and descriptions of the hardware components of your system, refer to the Introductory Manual.

DEBUGGING AND EDITING FEATURES

In addition to the features described in the Introductory manual, a number of features are available on your system to aid in editing and debugging your programs.

Character Erasing

Single keystroke entires in the current text line can be removed by touching the $\boxed{\text{BACK SPACE}}$ key.

Example:

```
                          :120 X=SQR (2+COS ( 17_
Key                       [BACK SPACE]   four times
                          :120 X=SQR (2_
correct remainder of line :120 X=SQR (2 - COS (17))_
                          :120 X = SQR (2 - COS (17))_
```

Removing the Current Line

The line currently being entered can be removed from the screen by touching the $\boxed{\text{LINE ERASE}}$ key.

Example:

```
          :300   PRINT "RESULT":   A(4 -
Key       [LINE ERASE]
          :_
```

Deleting a Line

A previously entered text line is deleted by entering its line number and touching the $\boxed{\text{EXEC}}$ key.

Example:

```
          10A = 14          display
          20 PRINT A
          .
          .
          .
Touch     20 [EXEC]
          10A = 14
          :_
```

When the program is listed again only line 10 remains.

Replacing a Line

An existing line is replaced by entering the same line number followed by the new line. When the $\boxed{\text{EXEC}}$ key is depressed, the new line is stored in memory.

Renumbering a Program

Line numbers in a program must always be integers. When lines of program text must be inserted between existing line numbers, and there are insufficient line numbers to accommodate the new lines, the existing line numbers can be automatically renumbered with the RENUMBER command. A suitable increment can be specified in the text of the RENUMBER command (see Section V, Non-Programmable Commands for syntax specifications).

Example:

```
READY
:100 IF I=4 THEN 102
:101 PRINT X, Y, I
:102 READ A, B$
:RENUMBER 101, 110
:LIST
100  IF I=4 THEN 120
110  PRINT X, Y, I
120  READ A, B$
```

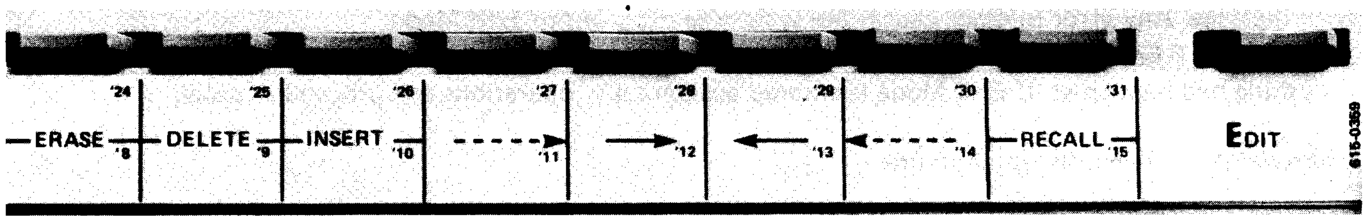
RENUMBER, starting at old line 101, using 110 as a starting statement line number, using an increment of 10

EDIT KEYS AND EDIT MODE

All system keyboards contain the Edit Mode keys which are convenient for rapid editing of program text or data being input.

Edit is a standard feature on all 2200 systems except the 2200A and B, on which it may be obtained as an option. Edit Mode operations are activated and controlled by the Edit Mode keys, the eight right-most Special Function keys (see Figure).

Each keyboard contains an Edit key, to the right of the sixteenth Special Function key, and is provided with the Edit Mode keys and an Edit Special Function strip. The Edit Mode keys and their operation are:



Key	Operation
EDIT	used to enter Edit Mode; when pressed, an asterisk replaces the usual colon at the beginning of the current line.
RECALL	used to recall a program line in memory to be edited.
◄- - -	moves the cursor five spaces to the left.
◄-	moves the cursor a single space to the left.
- - - -►	moves the cursor five spaces to the right.
-►	moves the cursor a single space to the right.
INSERT	expands a line for additional text or data entry by inserting a space character prior to the current cursor position.
DELETE	deletes the character at the current cursor position.
ERASE	erases a line from the current cursor position to the end of the line.

NOTE:
The Edit Mode keys operate as described as long as the CRT is the Console Output (CO) device and is selected with a line length of 64 (see SELECT). The CRT is the default device for CO set when the system is Master Initialized (see Turn-On Procedure and System Operation in the Introductory Manual).

NOTE:
These keys do not operate in the manner described on a System 2200A or 2200B unless Option 3, the Character Edit ROM is available. On such systems, the keys are Special Function keys only; the right-most EDIT key must not be used.

Edit Mode can be entered at almost any time; either just before or during keyboard entry of a BASIC command, BASIC statement or data line. When the BASIC or data line has been edited, it is stored in memory by touching the EXEC key. Once the EXEC (or LINE ERASE) key is touched, the system leaves Edit Mode and the Special Function keys revert to their normal use.

In a line of program or input data text being edited, any keyword (e.g., SIN(, PRINTUSING, etc.) entered by a single keystroke or recalled from memory acts as a single character when positioning the cursor. Some keywords (e.g., PRINT) contain a following space, while others [e.g., TAB(] do not. Experience with using Edit is the best guide. Edit Mode keys should be used to position the cursor when the system is in Edit Mode; the space bar and BACKSPACE keys overwrite characters in the line being edited with spaces. The STMT NUMBER and Special Function keys are inoperative when the system is in Edit Mode. The HALT/STEP key should not be used when the system is in Edit Mode; its use causes an error message. Any error message causes the system to drop out of Edit Mode.

If the EDIT key is pressed when the system is already in Edit Mode, the system continues to operate as if nothing had happened. If Edit Mode is entered accidentally, operations can proceed as usual.

Example 1: Given the program line

100 X = SIN (Y − 17.3) + (LOG(Z+4) +5)

in memory, delete the expression [SIN (Y − 17.3) +].

Operating Instructions	Display
1. (The line is in memory.)	:_
2. Depress the EDIT key.	
Note that an asterisk (*) is substituted for the usual colon (:).	*_
3. Key in the line number of the program statement.	*100_

4. Depress the RECALL Special Function key.

```
*100 X=SIN(Y-17.3)+(LOG(Z+4)+5)_
```

> *Note that the cursor is positioned at the end of the line.*

5. Position the cursor under the first character to be deleted, using the SPACE (←) and MULTISPACE (←- - - -) keys.

```
*100 X=SIN(Y-17.3)+(LOG(Z+4)+5)
```

> *Note that any keyword such as SIN(needs a single space only.*

6. Depress the DELETE key once. As each character is deleted, the line automatically contracts to eliminate spaces.

```
*100 X=Y-17.3)+(LOG(Z+4)+5)
```

7. Depress the DELETE key eight (8) more times to delete the rest of the expression and the plus sign.

```
*100 X=_(LOG(Z+4)+5)
```

8. Touch the RETURN/EXECUTE key to re-enter the line and drop the system out of Edit Mode. The new line automatically replaces the old line in memory since the statement numbers are identical.

> *Note that it does not matter where the cursor is positioned when EDIT operations are concluded. The entire new line is entered into memory.*

Example 2: The line

```
10 PRINT "THIS IS A
                          PROGRAM LINE"
```

is in memory. This offset line was created by accidentally touching the INDEX key (on the Model 2222 Keyboard only) while keying in the line. An invisible Index character thus was introduced in the line before the word PROGRAM. The line can be straightened by using EDIT. The line is represented in the buffer as indicated in the following keyword/character string diagram.

1	0	SPACE	PRINT	"	T	H	I	S	SPACE	I	S	SPACE	A	INDEX	P	R	O	G	R	A	M	SPACE	L	I	N	E	"	CR

Operating Instructions

1. Key in the line number, depress the EDIT key and the RECALL key.

```
*10 PRINT "THIS IS A
                      PROGRAM LINE"_
```

2. Position the cursor under the first character which follows the invisible Index character.

> NOTE:
> *SPACE Left until the cursor is under the P; depress the SPACE Left key* once more. *The cursor still remains under the P on the CRT but is positioned correctly to EDIT the Index character.*

```
*10 PRINT "THIS IS A
                      PROGRAM LINE"
```

3. Depress the DELETE key to delete the Index character.

```
*10 PRINT "THIS IS APROGRAM LINE"
```

4. Touch the RETURN/EXECUTE key to enter the line into memory and drop out of EDIT mode.

Example 3: The line of program text

40 GOTO 200: PRINTUSING 700

is in memory. You wish to have it read

40 GOTO 200: GOSUB 700

In this example using the Model 2222 Alphanumeric Keyboard, the new keyword must be inserted character-by-character rather than with a single keystroke.

1. The CRT displays:	:_
2. Depress the EDIT key.	*_
3. Key in the line number and depress the RECALL key.	*40 GOTO 200: PRINTUSING 700_
4. Depress the Single SPACE Left key four times.	*40 GOTO 200:PRINTUSING 700
5. Key in the 'G' of GOSUB.	*40 GOTO 200: G700

> *The entire word PRINTUSING is overwritten by the G since once in memory, PRINTUSING acts as a single character.*

6. To complete the line correctly, either enter "OSUB 700" or depress the INSERT key four times to allow room for the end of the word GOSUB.	*40 GOTO 200: GOSUB700

7. Touch the RETURN/EXECUTE key to enter the new line into memory and drop the system out of Edit Mode.

> **NOTE:**
> *If a statement has been partially edited, but not yet entered into memory, the original statement can be obtained by pressing the RECALL key again (before dropping out of Edit Mode).*

Example 4: The value

1.032E99

in exponential notation is to be keyed in as response to an INPUT statement, but is incorrectly entered as

1.03299

To correct this value, activate Edit Mode, insert a space and key in the necessary E.

Operating Instructions	**Display**
1. The CRT displays:	ENTER DATA ?_
2. Key in the required value.	ENTER DATA ? 1.03299_
3. Depress the EDIT key.	*1.03299_
4. Position the cursor at the position where the character is needed, using the Single SPACE Left key.	*1.03299
5. Depress the INSERT key.	*1.032_99
Note that the line is expanded to accommodate the insertion.	
6. Key in the new character.	*1.032E99
7. Touch the RETURN/EXECUTE key to enter the corrected value and resume program operation.	*1.032E99 :_

Example 5: To EDIT the line number of a program line recalled from memory. The line 110 GOTO 140 is in memory. You wish to change the line to: 125 GOTO 140. This can be done by editing the 110 and correcting it to 125 without deleting the line 110. However, the line 110 still remains in memory after the line 125 has been created and you must explicitly erase line 110 if you do not wish it to remain as part of your program.

	Operating Instructions	Display
1.	The CRT displays:	:_
2.	Key in the line number.	:110_
3.	Depress the EDIT key.	*110_
4.	Depress the RECALL key.	*110 GOTO 140_
5.	Depress the MULTISPACE Left key once.	*110_GOTO 140
6.	Depress the Single SPACE Left key twice.	*110 GOTO 140
7.	Key in the number 25.	*125_GOTO 140
8.	Touch the RETURN/EXECUTE key.	125 GOTO 140 :_
9.	Key in LIST and touch the RETURN/EXECUTE key. Both program lines are displayed.	110 GOTO 140 : 125 GOTO 140
10.	In order to delete the line 110, key in the line number and touch the RETURN/EXECUTE key.	
11.	The line 110 has been eliminated and is no longer in memory.	

Stepping Through a Program

Program execution can be halted at any time by touching the HALT/STEP key. Variables can be examined or modified by immediate execution statements; and execution can be continued by keying CONTINUE CR/LF-EXECUTE. If, after a program has been halted, the user wishes to step through the program, he continues touching the HALT/STEP key. Each time the key is touched, the next statement is executed; the executed statement and any normal PRINT class output (see I/O Device Selection) of that statement is displayed. Program stepping can be started at a particular statement line by entering a GOTO 'line number' statement, in the Immediate Mode, so long as the program has been resolved.

Example:

Enter the following program in memory:

```
10   FOR I = 1 TO 10
20   S = S + 1
30   PRINT S
40   NEXT I
```

OPERATING INSTRUCTIONS:	CRT DISPLAY
Key GOTO 10	READY :GOTO 10
Key HALT/STEP	: 10 FOR I =1 TO 10
Key HALT/STEP	: 20 S = S + I
Key HALT/STEP	: 30 PRINT S 1
Key HALT/STEP	: 40 NEXT I :_

The system can also be placed in TRACE mode and stepped. This provides both a display of each executed statement and the calculated results of each statement.

Executing a Program at any Given Line

Program execution can be started at any desired line by entering a RUN 'line number' command.

Example:

Key **RUN 130 CR/LF-EXECUTE**

NOTE:

The user should not begin execution in the middle of a FOR/NEXT loop or subroutine.

Programmable Trace

The TRACE statement provides for the tracing of the execution of a BASIC program. TRACE mode is turned on in a program when a TRACE statement is executed and turned off when a TRACE OFF statement is executed. TRACE is also turned off when a CLEAR command is executed, the system is RESET or Master Initialized. When in the TRACE mode, printouts will be produced when:

1. Any program variable receives a new value during execution; e.g., in LET, READ, FOR statements.
2. A program transfer is made to another sequence of statements; e.g., in GOTO, GOSUB, IF, NEXT statements.
3. A BASIC function is evaluated.

Example:

```
                READY
                :10  X = 1.2
                :20  TRACE
                :30  X = 2*X
                :40  IF X > 2  THEN 100
                :50  STOP
                :100   TRACE OFF
                :110   Y = X
                :120   STOP
                :RUN
. Trace         X = 2.4
  Outputs       TRANSFER TO 100
                STOP
                :_
```

Programmable Pause

The output of a program can be slowed down for easier visual inspection by selecting a pause of from zero to one-and-a-half seconds. The selected pause occurs whenever a CARRIAGE RETURN is output to the CRT display or a printer. The pause is turned on and off by executing the appropriate SELECT P 'digit' statement; the digit specifies the number of 6th's of a second to pause (e.g., P3 = 3 X 1/6 = 1/2 sec. pause). The pause feature is programmable, and can be turned on and off within a program.

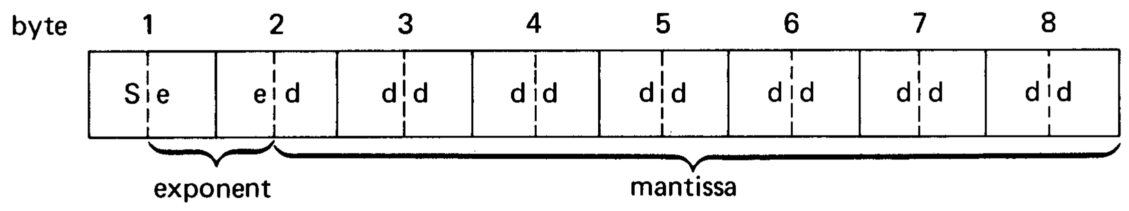
Example:

```
READY
:100   TRACE  :SELECT P6
:110   FOR I = 1 TO 20
:120   A(I) = I*COS (32.5)
:130   NEXT I
:132   TRACE OFF  :SELECT P0
:_
```

INTERNAL STORAGE

How Numeric Values are Stored

In your system, the fundamental unit of storage is the eight-bit byte (a single binary digit is called a bit). Letters, digits, characters and keywords used in a BASIC program are stored in memory in single bytes using the ASCII character code format (see Appendix A). Every numeric data value stored in a numeric variable or in the elements of a numeric array is stored in eight successive bytes as follows:



where S is the sign-indicator:

0 if mantissa	$\geqslant 0$ and exponent $\geqslant 0$
1 if mantissa	< 0 and exponent $\geqslant 0$
8 if mantissa	$\geqslant 0$ and exponent < 0
9 if mantissa	< 0 and exponent < 0

e are the digits of the exponent (low digit followed by high digit)
and d are the digits of the mantissa (with leading zeros removed).

The value is normalized when placed in this form. This is a floating point format which is optimum for numeric calculations. The six and one-half bytes of the mantissa allow for storage of mantissas of up to thirteen significant digits. Values between $10^{-99} <|\text{value}|< 10^{99}$ can thus be stored in your system, truncated to thirteen significant digits.

When displayed or printed with a PRINT statement, values whose absolute value is less than 10^{-1} or greater than 10^{13} are output in exponential format to nine significant digits; the PRINTUSING statement can be used to output values to full thirteen digit accuracy. Attempting to input numbers of more than thirteen digits produces the error message ERR 20.

How Alphanumeric Values are Stored

When a value specified as an alphanumeric value (e.g., the name "John Doe" or the code "PR145") is stored in memory it is stored in one character per byte. For example, the program line:

```
10 A$ = "JOHN DOE"
```

is stored as

byte	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	0 0	1 0	A	$	=	"	J	O	H	N	SPACE	D	O	E	"	CR

and the alphanumeric scalar A$ (set at resolution time [see the RUN command] as 16 bytes long) contains:

byte	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	J	O	H	N	SPACE	D	O	E	SPACE	SPACE	SPACE	SPACE	SPACE	SPACE	SPACE	SPACE

The size of the alpha scalar can be specified with a DIM or COM statement (see the DIM or COM statement).

How Programs are Stored

The following diagram represents how a program is stored in memory at execution time.

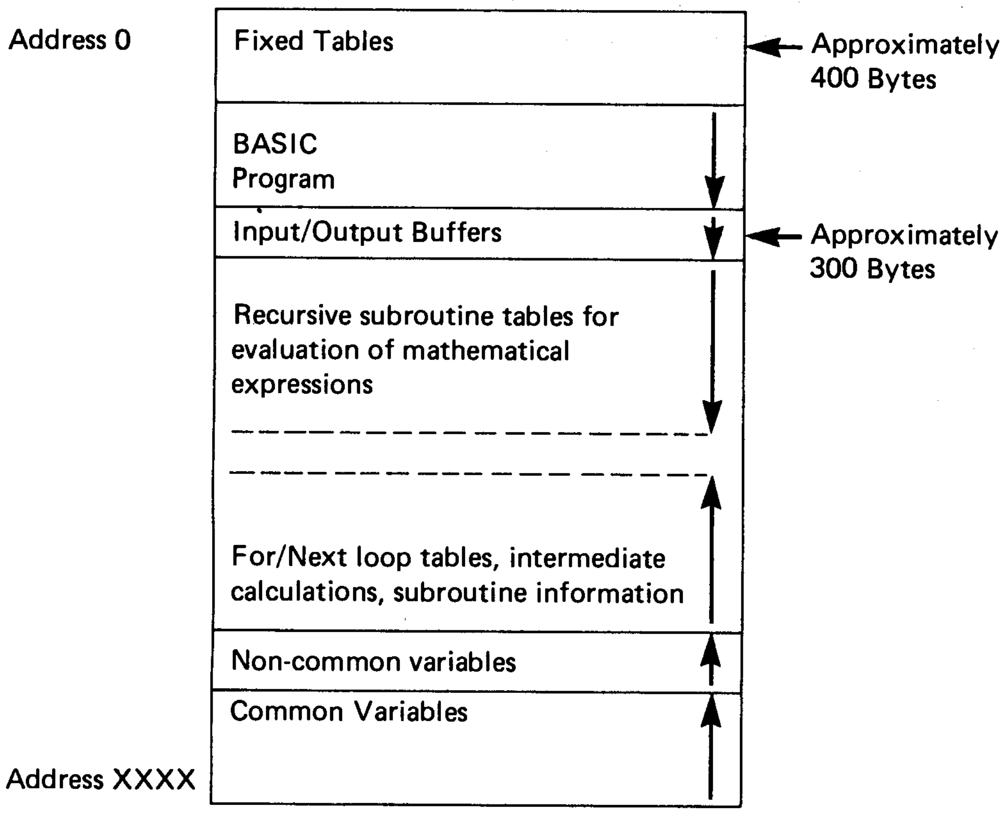


Figure 16. Memory Allocation

The BASIC program and the fixed tables and buffers are stored starting from address 0 (zero) in memory upward. Variables are stored starting at the highest memory address downward. When the program is run, these two areas are fixed in size. Between these two areas, are two dynamic tables. The table starting from the program upward stores recursive subroutine return information during evaluation and analysis of mathematical expressions; the other table, starting from the area for variables, stores subroutine and FOR/NEXT loop information downward. If at any time during program execution the end of one of these tables meets the end of the other, ERR 02 (Table overflow) occurs.

The amount of space used in memory for a given program can be approximately calculated with the END statement (see the END statement).

Section II BASIC Language Structure

Section II BASIC Language Structure

INTRODUCTION

Programs written in BASIC must adhere to certain rules of syntax. These rules and the terms used to define them are described below.

SYMBOL OPERATORS

In writing program lines in BASIC, only certain symbols and special characters can be used. They fall into three categories: arithmetic, relational and assignment symbols.

Arithmetic Symbols

The following arithmetic symbols are used to write formulas.
Operations in expressions are executed left to right in the following order:

Symbol	Sample Formula	Explanation
↑	A ↑ B	Raise A to the power B.
*	A * B	Multiply A by B.
/	A / B	Divide A by B.
+	A + B	Add B to A.
–	A – B	Subtract B from A.

1. operations within parentheses
2. exponentiation (↑)
3. multiplication and division (* and /)
4. addition and subtraction (+ and –).

Quantities within parentheses are evaluated before the parenthesized quantity is used in further computations. In the absence of parentheses, exponentiation is performed first, then multiplication and division, and finally addition and subtraction. For example, in the expression 1 + A/B, A is first divided by B and then 1 is added to the result. When there are no parentheses in the expression and the operators are at the same level in the hierarchy, the expression is evaluated from left to right. For example, in the expression A * B/C, A is multiplied by B and the result is divided by C.

NOTE:
*Where arithmetic operators occur side by side in an expression, correct evaluation and valid syntax require the use of parentheses. For example, the expression X = 2.4 * –A is not allowed; it must be written as X = 2.4 * (–A).*

Relational Symbols

Relational symbols are used in an IF . . . THEN statement when values are to be compared. For example, when the expression IF G < 10 THEN 60 is executed, if G is less than 10, processing continues at program line number 60.

The following relational symbols can be used in Wang BASIC:

Symbol	Sample Relation	Explanation
=	A = B	A is equal to B.
<	A < B	A is less than B.
<=	A <= B	A is less than or equal to B.
>	A > B	A is greater than B.
>=	A >= B	A is greater than or equal to B.
<>	A <> B	A is not equal to B.

These symbols are also used in the POS function (not available on the 2200A or S); and in MAT SEARCH.

Assignment Symbol

The equal sign = is used to indicate assignment of a value to a variable. For example, the formula A = 10, when written in BASIC, indicates 'assign the value 10 to the variable A'.

RULES OF SYNTAX

The following rules are used in this manual in the syntax specifications to describe BASIC program statements and system commands.

1. Uppercase letters (A through Z), digits (0 through 9) and special characters (*, /, +, etc.) must be written exactly as shown in the general form.
2. Lowercase words represent items which are supplied by the user.
3. Items in square brackets [ ] indicate that the enclosed information is optional. For example, the general form: RESTORE [expression] indicates that the RESTORE statement can be optionally followed by an expression. For example,

   ```
          RESTORE
       or RESTORE 2*X
   ```

   are both legal forms.
4. Braces { } enclosing vertically stacked items indicate alternatives; one of the items is required. For example,

   $$\text{operand} = \left\{\begin{array}{l}\text{literal}\\ \text{alpha variable}\\ \text{expression}\end{array}\right\}$$

   indicates that the operand can be either a literal, an alpha variable or an expression.
5. Ellipsis ..., indicates that the preceding item can be repeated as necessary. For example,

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

   indicates that additional variables as needed can be added to the INPUT statement.
6. Except in alphanumeric literals, BASIC ignores blanks. For example, the following statements are both valid and equivalent:

   ```
   10 LET A = 2*B+C
   10 LETA=2*B+C
   ```

7. The order of parameters shown in the general form must be followed.

TERMS USED IN SYNTAX SPECIFICATIONS

The following list defines the terms used in the syntax specifications for commands and statements.

Term	Meaning
alpha array designator	a parameter of the form: letter [digit] $( ).
alpha array name	a parameter of the form: A$ which refers to an alpha array.
alpha array variable	a parameter of the form: letter [digit] $ (expression [, expression]) in which the name of an alpha array is modified.
alpha function	STR and HEX are alpha functions; they can have alpha variables as arguments and, when used in a program line, can assign alpha values to other alpha variables.
alpha scalar variable	a parameter of the form: letter [digit] $.
alpha variable	{ alpha array variable / alpha scalar variable / STR function }
array	{ alpha array / numeric array } ; a one- or two-dimensional set of values
array name	{ alpha array name / numeric array name }
character string	any sequence of letters, digits or symbols in the ASCII character set not including control codes such as carriage return, backspace.
expression	a combination of one or more operators and numeric operands valid in BASIC.
line number	the number of a BASIC statement line in a program; it is of the form digit [digit] [digit] [digit].
literal	{ a character string within double quotes (") / a character string within single quotes (') / HEX function }
numeric array designator	a parameter of the form: letter [digit] ( )
numeric array name	a parameter of the form: letter [digit] which refers to a numeric array.
numeric array variable	a parameter of the form: letter [digit] (expression [,expression]) in which the name of a numeric array is modified.
numeric scalar variable	a parameter of the form: letter [digit]
numeric variable	{ numeric array variable / numeric scalar variable }
string variable	{ alpha array variable / alpha scalar variable }
variable	{ alpha scalar variable / numeric scalar variable. } a quantity that can assume any one of a given set of values.

Section III
Numerics

Section III Numerics

EXPRESSIONS

A numeric expression is any valid combination of numeric variables, functions, operators or constants connected by arithmetic symbols. An expression may be preceded by plus or minus and may be contained within parentheses. In the following examples valid BASIC expressions are boxed:

X = $\boxed{\text{A}}$
X = $\boxed{\text{5*Y+FNB(X) - LOG(Z)}}$
J($\boxed{\text{X2+5}}$, K)=9
FOR I = $\boxed{\text{3+K2}}$ TO $\boxed{\text{4*Y}}$ STEP $\boxed{\text{D(3+K) - 1}}$
PRINT $\boxed{\text{SIN(K) -4*J}}$

Operations in an expression are executed in sequence from highest priority level to lowest, as follows:
1. Operations within parentheses
2. Exponentiation (↑)
3. Multiplication or division (* or /)
4. Addition or subtraction (+or –)

Quantities within parentheses are evaluated before the parenthesized quantity is used in further computations. In the absence of parentheses, exponentiation is performed first, then multiplication and division, and finally addition and subtraction. For example, in the expression 1 + A/B, A is divided by B and then 1 is added to the result. When there are no parentheses in the expression and the operations have the same priority level, these operations are performed from left to right. For example, in the expression A*B/C; B is multiplied by A and the product is divided by C.

NUMERIC VARIABLES

A variable name is a string of characters that represents a data value. A variable can be given a new value in certain executable statements such as READ, LET, INPUT, NEXT, FOR. The value assigned to the variable in a program statement will not change until a second program statement is encountered which assigns a new value to the variable. Numeric variables are stored in memory in eight bytes (see Section I).

Numeric Scalar Variables

A numeric scalar variable is designated by a letter or a letter followed by a digit: there are 286 legal scalar variable names.

Examples:

A,A4

Numeric Array Variables

Array variables are used to operate on the elements of an array. Such variables can refer either to individual array elements, to parts of an array, or to the entire array.

A numeric array is referenced in BASIC in one of three ways:

1. by its numeric array name (letter [digit])
2. by its numeric array designator (letter [digit] ( ))
3. as a numeric array variable (letter [digit] (expression [, expression)])

The value of expressions must be 1 ⩽ value ⩽ 255.

Example of an array using the usual subscript notation of algebra:

$(a_1, a_2, \ldots, a_n)$

Example of an array element using algebraic subscripts:

b_{ij}

Examples of numeric array names:

A
A5

Examples of numeric array designators:

A ( )
C3 ( )
J2 ( )

Examples of one-dimensional numeric array variables:

A (10)
J (N)
Z3 (5)

Examples of two-dimensional numeric array variables:

C3 (1,5)
F (N,M+2)

The DIM or COM statement is used to define the amount of memory allocated to an array; when the array is defined, the parentheses contain constants giving the row and column dimensions of the array. The DIM or COM statement must precede the first reference to the elements of the array.

Examples using DIM and COM statements:

```
DIM A(10), C3(1,5), F7(2,4)
COM A(10), C3(1,5), F7(2,4)
```

NOTE:
Numeric scalar variables and all elements of numeric arrays are automatically initialized to zero when a program is first executed (RUN).

Example:

```
READY
:20  DIM Q(25) <---- defines the 1-dimensional array Q with 25 elements
:30  READ N    <---- reads value for N from first DATA statement
:40  FOR I = 1 to N )
:50  READ Q(I)      }  FOR/NEXT loop to read data,
:55  PRINT Q(I)     )  fill array and output it
:60  NEXT I
:70  DATA 5
:80  DATA 4, 5, 19, 37, 43
etc.
:_
```

For cases where an array variable is used as common data, it is specified in a COM (common) statement instead of a DIM statement to provide storage space.

1. The numeric value of the subscript for the first array element must be ≥ 1; zero is not allowed.
2. The dimension(s) of an array (rows, columns) cannot exceed 255.
3. The total number of elements of an array must not exceed 4,096.

An array variable and a scalar variable may have the same name; they are independent, unrelated variables. Singly subscripted and doubly subscripted arrays must not be defined with the same name.

COMMON DATA

The sharing of data common to several programs is possible by using the COM statement. Variables with data to be used in subsequent programs are defined to be common in a COM statement.

Example:

COM A(2, 4), B, C

defines the array A (of dimension 2 by 4) and the scalars B and C to be common data. When a RUN command is issued, all noncommon variables are removed from the system; common variables are not disturbed. In addition, common data can be retained when a new program is loaded or overlayed, and thus are passed on to the next program. Common variables are cleared from memory when a CLEAR or CLEAR V command is executed. COM CLEAR can be used to make common variables into non-common variables (see COM, COM CLEAR).

NUMERIC CONSTANTS

A numeric constant may be positive or negative and may consist of as many as 13 digits. Numbers with greater than 13 digits result in an illegal value diagnostic, ERR 18. Very large or very small numbers can be expressed in exponential form. In this case, the last four digits of the output value are reserved for the exponent and its sign. For example, 4.5×10^{7} is written as 4.5E+07 and 4.5×10^{-7} is expressed as 4.5 E–07. The magnitude of a numeric constant can be $10^{-99} \leq \text{constant} \leq 10^{99}$; exponents must be integers.

The following are examples of numeric constants in BASIC:

4, –10, 1432443, –.7865, 24.4563

Invalid Use of Scientific Notation

8.7E5.8	Not valid because of the illegal·decimal form of the exponent.
.87E–99	Not valid because it is equivalent to 8.7E–100, which is less than E–99.
–103.2E99	Not valid because it is equivalent to –1.032E101, an exponent greater than E99.

SYSTEM-DEFINED MATH FUNCTIONS

All the standard trigonometric functions (sin, cos, tan, arcsin, arccos, arctan), the logarithmic functions (log, exponential), exponentiation, taking square roots and assigning the value π are standard system-defined functions. Additionally, the absolute value, the greatest integer value, and the sign of an expression can be obtained and a random number generator is available. System-defined math functions are detailed on the next page. Functions are accessed by pressing the appropriate keyword key or entering the function name character by character.

Mathematical Functions

Keyboard Function	Meaning	Example
SIN(expression)	Find the sine* of the expression	SIN(π/3) = .866025403784
COS(expression)	Find the cosine* of the expression	COS(.693↑2) = .8868799122686
TAN(expression)	Find the tangent* of the expression	TAN(10) = .64836082745
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) = .9434480794
ARC TAN(expression)	Find the arctangent** of the expression	ARC TAN (3.2) = 1.2679114584
π Appears as #PI on CRT display	Assign the value 3.14159265359 (Displayed and printed as #PI)	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)	Find the largest integer ⩽ value of the expression	INT (8)=8, INT(3.6)=3 INT(-5.22)=-6
SGN(expression)	Assign the value 1 to any positive number, 0 to zero, and -1 to any negative number	SGN(9.15)=1 SGN(0)=0 SGN(-.124)=-1
LOG(expression)	Find the natural logarithm of the expression	LOG(3052)= 8.02355239240
EXP(expression)	Find the value of e raised to the value of the expression	EXP(.33*(5-6))= .71892373343
SQR(expression)	Find the square root of the expression	SQR(18+6)=SQR(24)= 4.8989794856

*Unless instructed otherwise, the argument is interpreted in radians. Degrees, grads (360° = 400 grads), or radians can be selected by entering the following statements:

SELECT	D	CR/LF–EXECUTE	–arguments in degrees for all subsequent calculations.
SELECT	R	CR/LF–EXECUTE	–arguments in radians for all subsequent calculations.
SELECT	G	CR/LF–EXECUTE	– arguments in grads for all subsequent calculations.

**The arctangent notation ATN(is also a recognized function notation.

† **RANDOM NUMBERS**

Each time the RND function is used, a random number is produced with a value between 0 and 1. If the argument of the RND function is not zero, the next number in the 'random number list' is produced. If the argument is zero, the first random number in the 'list' is produced. RND (0) is useful when debugging programs involving random numbers since the same results can be produced each time the program is executed.

The example below prints out the first 100 numbers in the 'random number list' each time the program is executed. Deletion of Line 10 produces a different set of random numbers each time the program is executed.

Example:

```
:10 X = RND (0)
:20 FOR I = 1 TO 100
:30 PRINT RND (3)
:40 NEXT I
:_
```

Whenever the system is Master Initialized (Power On), the random number generator is initialized; the next time RND is used, the first random number in the list will be produced. If the argument of the RND function is not otherwise defined, it is set to zero.

ADDITIONAL SYSTEM-DEFINED NUMERIC FUNCTIONS

The following additional functions can be used in expressions:

NUM	Test if a string of characters is a legal BASIC number.
POS	Locate first character in a string meeting specified relation.
VAL	Converts the binary value of a character to a numeric value.
LEN	Obtain the length of a string (an alpha scalar or alpha array element).

They are described in detail in Section VIII.

USER-DEFINED FUNCTIONS

A user-defined function is a mathematical function of a single variable; once defined, it can be used repeatedly within a program. A user-defined function is defined by a DEFFN statement; the form and use of DEFFN are described in Section VIII.

Section IV
Alphanumerics

Section IV Alphanumerics

ALPHANUMERIC STRING VARIABLES

The Wang 2200 provides for a non-numeric form of variable, the alphanumeric string variable. It is distinguished from a numeric variable by the manner in which it is named, a letter or a letter and a digit followed by a $. String variables permit the user to process alphanumeric strings of characters, (such as names, addresses and report titles).

Both alphanumeric scalar variables and alphanumeric array variables may be used. The dimensions of string arrays must be specified in a DIM or COM statement prior to their use in the program.

Formats for alphanumeric string variable names are given below; items enclosed in brackets are optional.

Alphanumeric scalar string variable 'letter' ['digit'] $ [length]	(i.e., A$, B$, C1$)
One-dimensional alphanumeric string array variable 'letter' ['digit'] $ (d_1) [length]	(i.e., A$ (3), B$ (N))
Two-dimensional alphanumeric string array variable 'letter' ['digit'] $(d_1 , d_2) [length]	(i.e., A$ (2,3), B$ (N,M))

where d_1 and d_2 are expressions defining the number of rows and columns in an array whose values are ≥ 1 and less than 256; and length is the number of bytes (characters) of the alpha scalar or alpha array element. The value of length must be $1 \leq$ length ≤ 64; its default value is 16 bytes.

The same variable name can be used as a scalar variable and as an array variable in the same program; the variables are unrelated. A one and a two dimensional alpha array may not have the same name in the same program. For example,

A A$ A(I) A$(I,J)

can all be used in the same program. However,

A$(I) and A$(I,J)

cannot.

Alpha Scalar Variables

An alpha scalar is designated by a letter or a letter followed by a digit and a dollar sign $.

Examples:
A$
A3$

Alpha Arrays

An alpha array can be either one- or two-dimensional and contains alphanumeric elements. An alpha array is referenced in BASIC statements in one of three ways:

1. by its alpha array name (letter [digit] $)
2. by its alpha array designator (letter [digit] $ ( ))
3. as an alpha array variable (letter [digit] $ (expression [, expression]))

The value of an expression must be $1 \leq$ value ≤ 255.

Example of alpha array names:

A$
Z2$

Examples of alpha array designators:

A$ ()
P1$ ()

Column

row

Examples of one-dimensional alpha array variables:

A$(5)
C3$(25)

Examples of two-dimensional alpha array variables:

B$(1,5)
C9$(N,3)

The DIM or COM statement is used to define the amount of memory allocated to an array; in the DIM or COM statement, the expressions following the array name contain constants giving the row and column dimensions of the array. The DIM or COM statement must precede the first reference to the array or any of its elements in the program.

Examples of DIM and COM statements:

```
DIM A$(10), C3$(1,5), F7$(2,4)
COM A$(12), C5$(2,2), F1$(1,2)
```

NOTE:
An alpha scalar or alpha array element is filled with blanks (HEX(20)) when the scalar or array is initially defined.

ALPHANUMERIC VARIABLE LENGTH

The default length of an alpha variable or alpha array element is 16 characters; however, the user may define the maximum length to be 1 to 64 by defining a different length in a DIM or COM statement (see DIM or COM). If an alpha variable receives a value of less than its maximum length, it reflects that shorter length in subsequent operations until it receives another value.

The end of the value of an alpha variable is normally its last nonblank character (except when the value is all blanks, in which case the value is treated as one blank). Hence, trailing blanks are generally not considered part of alpha variable values.

For example,

```
:10 A$="ABC   "
:20 PRINT A$;"DEF"
:RUN
ABCDEF          (note, trailing blanks of A$
                 were not output)
```

The length function, LEN, determines the number of characters in the current value of an alpha variable.

Example:

```
:10 A$="ABCD   "
:20 PRINT LEN(A$)
:RUN
4               (trailing blanks are not considered
                 to be part of the value)
```

ALPHANUMERIC LITERAL STRINGS

An alphanumeric literal string is a character string enclosed in double quotation marks ("). It can be used with alpha variables to provide an alpha variable with a string value or to output strings of alpha characters.

For example,

```
:10 A$="BOSTON, MASS."
:20 PRINT A$
:RUN
BOSTON, MASS.
```

Literal strings can be any length that can be expressed on one program line. However, when they are used to store values in alpha variables, they are truncated to the maximum length defined for the alpha variable.

For example,

```
:10 DIM A$5
:20 A$="123456789"
:30 PRINT A$
:RUN
12345          (note that the value was truncated to five
               characters since the maximum length of
               A$ was defined as 5)
```

The minimum length of a literal is 1; the null string (" ") is not allowed.

NOTE:
If the dimension of an alpha scalar or array element is larger than needed for storing a given value, the scalar or array element is filled out with trailing blanks HEX(20)) when the value is stored. Blanks are not evident in display or printing from a PRINT statement, but can be found with the HEX-PRINT statement.

NOTE:
The HEXPRINT statement is not available on a System 2200A.

Alphanumeric literals can in general be used wherever alpha variables are allowed, except where the statement requires a variable and does not permit the use of an alphanumeric constant.

Example:

INPUT A$, B$, C$(I)

is a legal BASIC statement using alpha variables.

INPUT A$, "B$"

is not a legal BASIC statement since "B$" is not an alpha variable (when enclosed in quotes it is a literal string).

USE OF ALPHA VALUES WITH THE INPUT STATEMENT

When using the INPUT statement to enter data into memory, it is not necessary to enclose data to be stored in an alpha scalar or array element in quotes. In the INPUT statement, commas and carriage returns (touching the EXEC key) act as terminators for variables and leading spaces are ignored. If commas or leading spaces are to be included in any alphanumeric value, the character string of the value must be enclosed in quotes.

Example:
INPUT A$, B$

To enter values for both A$ and B$, enter (for example):

TABLE $\boxed{\text{EXEC}}$
CHAIR $\boxed{\text{EXEC}}$
or TABLE, CHAIR $\boxed{\text{EXEC}}$

To enter any value containing a comma or leading space, enter (for example):

"BOSTON, MA" $\boxed{\text{EXEC}}$
NYC $\boxed{\text{EXEC}}$
or "BOSTON, MA", NYC $\boxed{\text{EXEC}}$

LOWERCASE LITERAL STRINGS

A special form of literal string is available for specifying lowercase characters. The literal string is entered with uppercase characters but enclosed in single quotes ('). The single quotes indicate that the uppercase letters are to be converted to lowercase by the system.

For example,

```
:10 PRINT "J";'OHN ';"D";'OE'
:RUN
John Doe      (if device is capable of printing lowercase)
   or
JOHN DOE      (if device only prints uppercase letters)
```

The following characters are valid in lowercase literals

letters:	**ABCDEFGHIJKLMNOPQRSTUVWXYZ**
digits:	**0123456789**
special characters:	**(space) !"#$%&()*+-/,.:;<=>?**

EXAMPLES OF STATEMENTS USING ALPHA VARIABLES AND LITERAL STRINGS

Alphanumeric string variables can be used in the BASIC statements listed below. Literal strings can generally be used in place of alpha string variables, except where a value is assigned to the string variable.

LET	LET A$=B$(2) A$="ABCD"

IF . . . THEN	IF A$=B$ THEN 100
	IF A$<"DR" THEN 200
	IF "ABCD">B$ THEN 300

INPUT	INPUT A$, B$(4)
READ	READ C$, D$, E$(1,2)
DATALOAD	DATALOAD #2,A$,B$ DATALOAD A$(I)
PRINT	PRINT A$,B$, "ABCD"
PRINTUSING	PRINTUSING 50,A$,B$, "LAST"
DATASAVE	DATASAVE A$, "GROUP1"

DATA	DATA "ABCD", "EFGH", 10
STR	A$=STR (B$(I),I)
HEXPRINT	HEXPRINT A$

NOTE:

When comparing alpha string variables with literal strings or other alpha string variables (e.g., IF A$ < "ABCD"), values are compared character by character. Trailing spaces are considered equivalent to hex(20) in determining where to place each value in the collating sequence. The variables fall at the same location in the collating sequence (i.e., they are equal) even if they do not have the same number of trailing spaces, so long as all their other characters are equal.

Example:

```
10 DIM A$4, B$5, C$5
20 A$="ABC"
30 B$=HEX(41424321)
40 C$="ABC  "
50 IF A$=B$ THEN 100
60 IF A$=C$ THEN 200
100 PRINT "A$=B$"; A$, B$
110 GOTO 300
200 PRINT "A$=C$"; A$, C$
300 HEXPRINT A$, B$, C$
A$=C$ABC            ABC
41424320
4142432120
4142432020
```

ALPHANUMERIC FUNCTIONS

Two functions which can operate on alphanumerics are provided by the system; they are the String Function STR and the Hexadecimal Function HEX. They are described in the next few pages.

HEX

FUNCTION

General Form:		HEX(hh [hh] . . .)
	where:	h = hexdigit (0 to 9 or A to F)

Purpose

The hexadecimal function, HEX, is a form of literal string that enables any 8-bit codes to be used in a BASIC program. It may be used wherever literal strings enclosed in double quotes are allowed. Each character in the literal string is represented by two hexadecimal digits. If the HEX function contains an odd number of hexdigits or any characters other than hexdigits, an error results.

The HEX function can be used to send control codes that do not appear on the keyboard to peripheral devices. For example,

```
:PRINT HEX(03)
```

clears the CRT and homes the cursor.

Any character can be represented by two hexdigits. A complete chart of HEX codes pertaining to the CRT is given in the ASCII Character Code Set Appendix. See the appropriate peripheral manual for codes available on other devices.

Examples:

```
:10 A$=HEX(0C0A0A)
:20 IF A$> HEX(7F) THEN 100
:40 PRINT HEX(0E);"TITLE"
```

STR

FUNCTION

General Form:	STR (alpha variable, s[,n])
where s	= starting character in sub-string (an expression)
n	= number of consecutive characters desired (an expression)
	s and n cannot be zero.

Purpose

The string function, STR, specifies a substring of an alpha variable. With it, a portion of an alpha value can be examined, extracted or changed. For example,

```
10 B$ = STR (A$,3,4)
```

sets B$ equal to the third, fourth, fifth and sixth characters of A$.

If 'n' is omitted, the remainder of the alpha variable is used, including trailing spaces. For example,

```
10 A$ = "ABCDE"
20 PRINT STR(A$,3)
```

produces CDE at execution time.

Examples of Syntax:

STR(A$,3,4)	Takes the third, fourth, fifth and sixth characters of A$.
STR(A$,3)	Starting with the third character in A$, takes the remaining characters of A$.

Example:

5 DIM A$20	
10 B$ = "ABCDEFGH"	Assigns the value ABCDEFGH to B$.
20A$ = STR (B$,2,4)	Assigns the value BCDE to A$.
30 STR (A$,4) = B$	Assigns the value ABCDEFGH to characters 4 through 11 of A$.
40 STR (A$,3,3) = STR (B$,5,3)	Assigns the value EFG to the third, fourth and fifth characters of A$.
50 IF STR(B$,3,2)="AB" THEN 100	Compares the third and fourth characters of B$ (CD) to the literal AB.
60 READ STR (A$,9,9)	Assigns the next data value read to characters 9 through 17 of A$.
70 DATA "A1B2C3D4E5F6G7H8I9"	

Example:

The following program assigns a value to B$, extracts characters 2 through 5 from B$ and assigns these characters to A$.

```
10 B$ = "ABCDEF"
20 A$ = STR(B$,2,4)
30 PRINT B$ : PRINT A$
```

Output produced at execution time:

```
ABCDEF
BCDE
```

The following program uses an alpha scalar with a literal and the STR function to riffle through the literal character by character.

```
10 B$ = "ABCDEF"
20 FOR I = 1 TO 5
30 A$ = STR(B$,I)
40 PRINT A$
50 HEXPRINT A$
60 NEXT I
```

Output at execution time:

```
ABCDEF
41424344454620202020202020202020
BCDEF
42434445462020202020202020202020
CDEF
43444546202020202020202020202020
DEF
44454620202020202020202020202020
EF
45462020202020202020202020202020
```

If the STR function is used on the left side of an assignment (LET) statement, and the value to be received is shorter than the specified substring, the substring is filled with trailing spaces on a 2200C, S or T. For example,

```
10 A$ = "123456789"
20 STR (A$,3,5) = "ABC"
30 PRINT A$
```

output at execution time: 12ABC..89

On a 2200A or B, the remainder of the substring remains unchanged.

output at execution time: 12ABC6789

The STR function can be used wherever alpha variables are allowed.

Examples:

```
10 A$ = STR (B$,2,4)
20 STR (D1$,I,J) = B$
30 IF STR (A$,3,5) > STR (B$,3,5) THEN 100
40 READ STR (A$,9,9)
50 PRINT STR (C$,3)
```

Section V
I/O Device Selection

Section V I/O Device Selection

INTRODUCTION

Every I/O device on your system, whether part of your console or a separate peripheral, has a unique *device address* with which it is accessed. Each I/O device address consists of three hexdigits.

Example:

$$\textbf{100 SELECT PRINT }\underbrace{\textbf{215,}}\textbf{ TAPE }\underbrace{\textbf{10B}}$$

device addresses

The device address has the form xyy, where x represents the *device type* (i.e., tape, disk, printer, etc.) and yy represents the *unit address* which selects the specific unit from the CPU. The three-hexdigit value is thus called a *device address.*

The *device type* signals the I/O control routines in the CPU to perform certain functions during the I/O operation. Functions are related to specific peripherals (a device type for a card reader is non-functional if used with a plotter or disk, etc.). The *unit address* represents the specific switches set at the factory on the I/O controller for each peripheral.

The following device types are recognized by the System 2200:

Type	Used With	Operation
0	Console Input (CI) devices; Nine-Track Tape Drive*; Printing or Display Output Devices; Teletype®	Supplies line feed to print or display devices which do not automatically perform a line feed after a carriage return.
1	Tape Cassette Drives	Activates tape accessing and formatting microcode.
2	Printers; I/O Interface Controllers; Output Writer; Digitizer; Telecommunications Controller	Does not provide line feed after carriage return on printer (printer provides its own CR/LF).
3	Disk Drives	Activates access routines for disk units.
4	Plotters, Teletype® Punched Tape Unit; Printers	Suppresses automatic carriage return when line length exceeded on printers.
5	Manual Mark Sense Card Reader	Activates access routines for mark sense cards.
6	Punch Tape, Mark Sense and Punched Card Readers	Activates punched tape and punched or mark sense card access routines.

Each unit on a system must have a unique device address, although in general all units of the same category (tape cassette drives, plotter, etc.) will use the same device type. The device addresses for all units are detailed on the Device Address Guide. A system with a single device of a particular category uses the first device address (on the Device Address Guide or in the Device Address Appendix) for the category; additional units of the same category have addresses sequentially assigned.

*The Nine-Track Tape Drive (Model 2209) is operated exclusively with $GIO statements which ignore the first hexdigit of the device address.

Example:

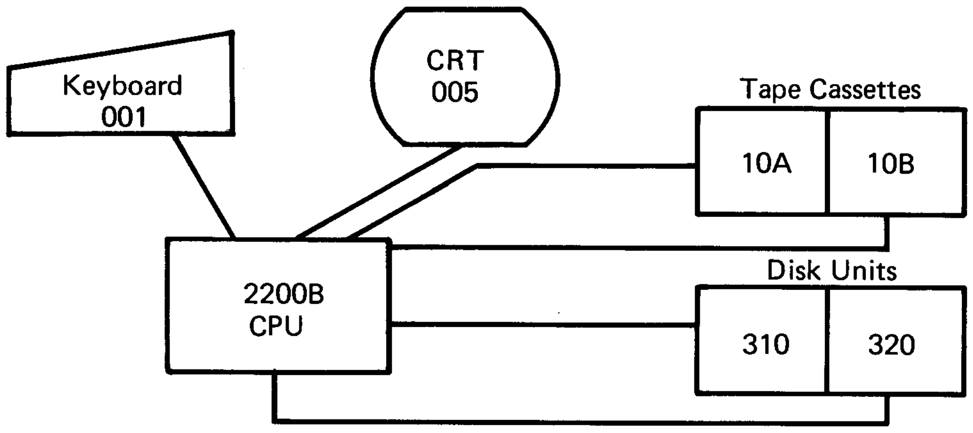


Figure V.1 Device Addresses in a Typical Configuration

The device address should be written on the Controller Board to which the unit is attached; it can also be written on the unit itself.

HOW IS A UNIT ADDRESSED?

When a BASIC command or statement which performs an I/O operation is executed, a specific unit is addressed in one of three ways:

1. with a SELECT statement or command

   Example:

   10 SELECT PRINT 005, LIST 215, CO 005

   Device Address

2. with certain BASIC words that can address units directly (such as LOAD, SAVE, etc.)

   Example:

   LOAD DC R / 320

   Device Address

   By using 'file numbers', such BASIC words can also address a unit indirectly; in this case, a SELECT statement must specify the device address of each 'file number'.

   Example:

   ```
   10 SELECT #2 10B,#3 10C
   20 DATASAVE #2,OPEN"DATAF"
   30 REWIND #3
   ```

3. by using the Default Value assigned when the system is Master Initialized.

USING SELECT

The SELECT word can be used either in the Immediate or the Program Mode. It permits the user to specify the device addresses (and line length) for entire classes of operations.

Example:

$$10\ \text{SELECT}\ \underbrace{\text{PRINT}}_{\text{class parameter}}\ \underbrace{215}_{\text{device address}}$$

This statement assigns the device address 215 for all operations of the PRINT class. PRINT class operations include all output from

- PRINT statements
- HEXPRINT statements
- PRINTUSING and % (Image) statements
- MAT PRINT statements

(see Figure V.2). Device address 215 specifies the printer as the output unit. Once a device address is specified for an I/O class parameter, all I/O operations of that class subsequently executed use that device address until it is changed (by another SELECT statement or by Master Initialization).

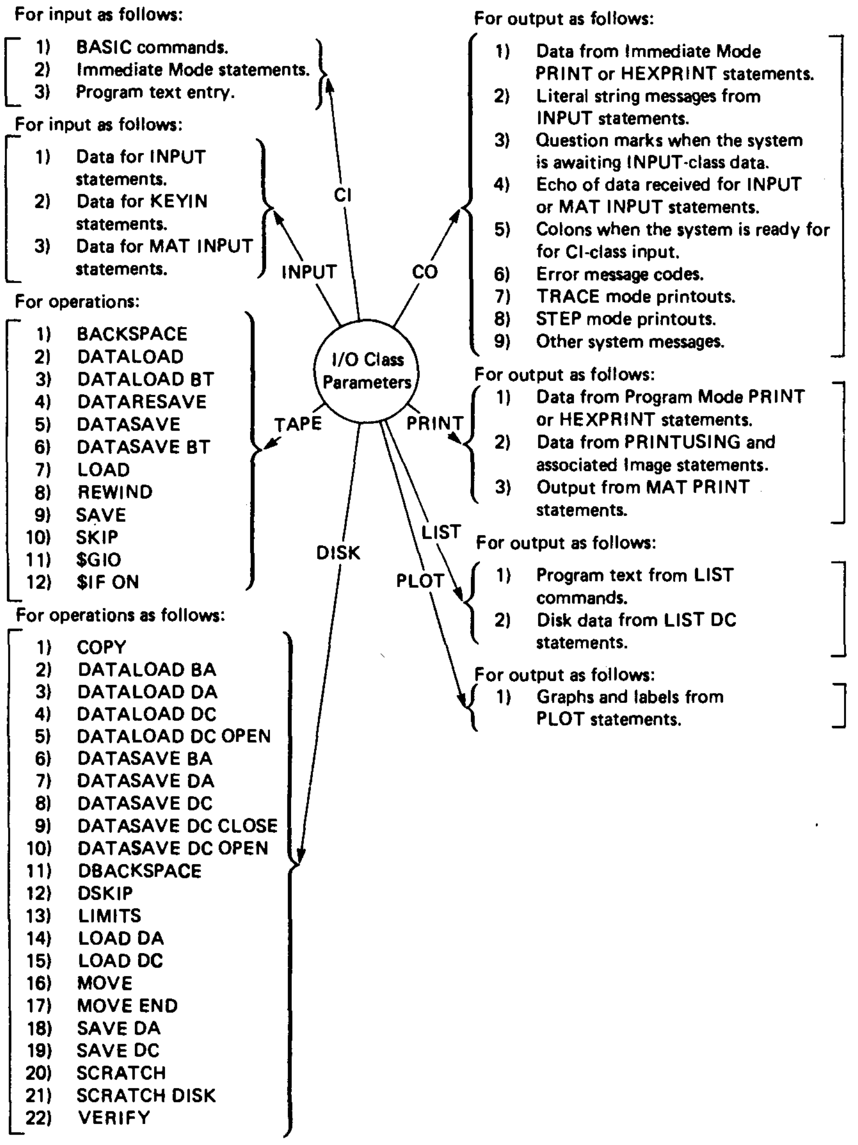

For input as follows:
1) BASIC commands.
2) Immediate Mode statements.
3) Program text entry.
For input as follows:
1) Data for INPUT statements.
2) Data for KEYIN statements.
3) Data for MAT INPUT statements.
For operations:
1) BACKSPACE
2) DATALOAD
3) DATALOAD BT
4) DATARESAVE
5) DATASAVE
6) DATASAVE BT
7) LOAD
8) REWIND
9) SAVE
10) SKIP
11) $GIO
12) $IF ON
For operations as follows:
1) COPY
2) DATALOAD BA
3) DATALOAD DA
4) DATALOAD DC
5) DATALOAD DC OPEN
6) DATASAVE BA
7) DATASAVE DA
8) DATASAVE DC
9) DATASAVE DC CLOSE
10) DATASAVE DC OPEN
11) DBACKSPACE
12) DSKIP
13) LIMITS
14) LOAD DA
15) LOAD DC
16) MOVE
17) MOVE END
18) SAVE DA
19) SAVE DC
20) SCRATCH
21) SCRATCH DISK
22) VERIFY
CI
INPUT
CO
I/O Class Parameters
TAPE
PRINT
DISK
LIST
PLOT
For output as follows:
1) Data from Immediate Mode PRINT or HEXPRINT statements.
2) Literal string messages from INPUT statements.
3) Question marks when the system is awaiting INPUT-class data.
4) Echo of data received for INPUT or MAT INPUT statements.
5) Colons when the system is ready for for CI-class input.
6) Error message codes.
7) TRACE mode printouts.
8) STEP mode printouts.
9) Other system messages.
For output as follows:
1) Data from Program Mode PRINT or HEXPRINT statements.
2) Data from PRINTUSING and associated Image statements.
3) Output from MAT PRINT statements.
For output as follows:
1) Program text from LIST commands.
2) Disk data from LIST DC statements.
For output as follows:
1) Graphs and labels from PLOT statements.

Figure V.2 I/O Class Parameters and Their Operations

SELECT

General Form: SELECT select parameter [, select parameter . . .]

where select parameter =

CI	device address
CO	device address [(length)]
DISK	device address
TAPE	device address
'file number'	device address
LIST	device address [(length)]
PRINT	device address [(length)]
INPUT	device address
PLOT	device address
P	[digit]
D	
R	
G	

device address = A three hexadecimal digit code specifying the desired device (see Device Address Guide).

length = An integer < 256 specifying the desired line length.

'file number' = One of the following: #1, #2, #3, #4, #5, #6

Purpose

The SELECT statement is used for three purposes:

1. To select device addresses for input/output statements or commands.
2. To specify a pause after every printed or displayed line of output (used mainly with CRT display), and
3. To specify degree, radian, or gradian measure for the trigonometric functions.

A given I/O class (select parameter) cannot occur more than once in a SELECT statement.

Examples:

To change the console device from the Primary (Output) Print Device to another, the following statement does the job.

```
SELECT CO 215 (80)
```

This selects the printer with device address 215 as the new Console Output device and sets the maximum line length at 80 columns.

THE INPUT PARAMETER

The INPUT select parameter specifies the device address to be used to enter in data for INPUT and KEYIN statements.

Example:

```
100 SELECT INPUT 002
110 INPUT "VALUE OF X, Y", X, Y
```

The message "VALUE OF X, Y?" appears on the console output device, while the values of X and Y are keyed in on the keyboard whose device address is 002.

THE PRINT PARAMETER

The PRINT parameter specifies the output device on which all program output from PRINT, HEXPRINT, and PRINTUSING statements are displayed.

Example:

```
100 SELECT PRINT 213(100)
110 PRINT"X=";X,"NAME=";N$
120 PRINTUSING 121, V
121 %TOTAL VALUE RECEIVED $#,###.##
```

The SELECT PRINT statement in line 100 directs all printed output to a Model 2201 Output Writer (device address (213); the line length is specified as 100 characters.

Example:

```
SELECT PRINT 005(64)
```

This statement reselects the CRT as the device to which all PRINT and PRINTUSING output is directed. The maximum line length is reset to 64 characters.

> **NOTE:**
>
> *The output from PRINT statements entered in the immediate mode always appears on the Console Output Device.*

THE LIST PARAMETER

The LIST select parameter specifies which output device is to be used for all program listings and disk catalog listings.

Example:

```
SELECT LIST 215(70)
```

This statement specifies that a line printer (device address = 215) is to be used for program listings. The maximum line length is specified as 70 columns.

NOTE:

All SELECT statement formats are legal in either program mode or immediate mode. Device selections remain in force until:

1. *They are changed by the execution of another SELECT statement, or*
2. *They are reset to the currently selected console devices by the execution of a CLEAR command with no parameter, or*
3. *They are reset to the Primary Console Devices by a Master Initialization.*

A CLEAR command with no parameters and Master Initialization (power on) clears all file number assignments. All file numbers then must be initialized by re-executing the SELECT statements. Reference to an unassigned or cleared file number causes an error output.

WARNING: Selecting an illegal device address for CI or CO causes the system to become locked out; it can be reset only by Master Initializing,i.e., by turning the power off then on again. All programs and variables will be lost.

SPECIFYING A PAUSE:

The 'P' select parameter causes the system to pause each time a carriage return character is output to a CRT so the user can scan the output rather than programming the system to halt execution whenever the CRT screen is full. The optional digit following the pause specifies the length of the pause in increments of 1/6 second. For example, the following statements generate the indicated pauses:

```
100 SELECT P1       pause  =  1/6 second
SELECT P6           pause  =  1 second
SELECT P (or P0)    pause  =  no pause
```

Selecting P or P0 removes the current pause.

SPECIFYING DEGREES, RADIANS, OR GRADS:

Degree, radian, or gradian measure may be selected for the trig function arguments by using the 'D', 'R' or 'G' parameters, respectively. For example:

```
SELECT D
```

causes the system to use degree measure for the trigonmetric functions. The unit of measure can be changed by executing another SELECT command or by Master Initialization, which automatically selects radians.

Example:

```
SELECT CO 005 (64)
```

This statement reselects the CRT as the Console Output Device. The line length is reset to 64 characters.

Example:

```
SELECT TAPE 10B
```

This statement selects the second cassette tape unit (device address = 10B) as the Console Tape Cassette unit. All statements involving cassette operations will access the second cassette drive unless the statements contain either of the two optional parameters, #n or /xxx which supply the device address.

INDIRECT SELECTION

The System 2200 provides two other methods for selecting tape cassette drives or other devices for input and output operations. The individual BASIC statements that execute I/O operations (LOAD, DATASAVE, SKIP, etc.) each contain two optional parameters designated #n and /xxx. The /xxx parameter allows the actual device address of a cassette drive to be placed directly in the statement. The xxx represents the three-character device address of the desired device. This method of selecting tape devices is independent of the SELECT statement.

Example:

```
DATASAVE /10B, OPEN "DATFILE"
```

This statement writes a data file header record on the cassette whose device address is 10B.

The #n parameter permits cassette or other device addresses to be assigned indirectly using the SELECT statement. #n is called a file number and must be one of the following: #1, #2, #3, #4, #5, #6. A particular device address can be assigned to a file number by a SELECT statement in a program. Thereafter in the program, BASIC Input/Output statements which contain that file number automatically use the previously assigned device address.

Example:

```
10 SELECT #2 10C, #3 10A
```

This statement assigns the cassette device address 10C to file #2, and the cassette device address 10A to file #3. In subsequent program statements which perform input/output operations, the file then can be used to supply the device address.

Example:

```
50  REWIND #2
60  DATALOAD #3, A( ), B$( )
```

The indirect assignment of device addresses in a program using file numbers offers several advantages. Subroutines can be written to perform a sequence of I/O operations for several devices. All device address assignments in a program can be changed by modifying a single statement. For instance, in the following example addresses can be assigned by changing statement 10.

Example:

```
10  SELECT #2 10C, #3 10A
20  SKIP #2, 2F
    .
    .
    .
100 REWIND #3
110 DATASAVE #2, OPEN "DATFILE"
```

DEFAULT VALUES

When the System 2200 is Master Initialized, the following values are automatically assigned to the SELECT parameters (see SELECT). The default value pertains to the whole I/O Class (see Figure V.2).

SELECT Parameter	Default Value
CI (Console Input)	001 (keyboard)
CO (Console Output)	005 (CRT)
INPUT	001 (keyboard)
LIST	005 (CRT)
PRINT	005 (CRT)
TAPE	10A (tape cassette drive)
DISK	310 (disk unit)
PLOT	413 (plotter)
line length	64 (for CO, LIST and PRINT)
'file number'	0 (zero)
D, R or G	R

Normally the CI and INPUT addresses are thus 001, the keyboard, the output device for LIST and PRINT class operations is 005, the CRT, etc. (see the Device Address Guide). Thus Master Initialization provides device addresses for:

- the keyboard
- the CRT
- the tape cassette drive (if any)
- the disk unit (if any)
- the plotter (if any).

If a System 2200 does not contain additional input/output devices, then device addresses need not be specified or selected in the BASIC commands and statements which perform input/output. If additional devices are present in the system, device address specification or selection is required.

To change the console output device from the CRT (device address = 005) to another output device, a statement having the following format can be used:

SELECT CO device address [(length)]

Example:

SELECT CO 215 (80)

This statement selects a line printer (device address = 215) as the new Console Output Device. The maximum line length to be used on the printer is set at 80 columns.

NOTE:

If a line length is not specified for console output, PRINT or LIST, the last line lengths selected for these operations are used. Master Initialization sets line length to 64 characters.

Section VI
Non-Programmable Commands

Section VI Non-Programmable Commands

INTRODUCTION

A BASIC command provides the user with a means for direct communication with the system. A BASIC command facilitates the running or modification of a program but is not part of the program itself.

For example, the RUN command initiates the execution of a program; the SAVE command instructs the system to record all program text on a cassette tape.

BASIC commands are entered one line at a time. They differ from BASIC statements in that they are not preceded by line numbers, and only one command can be entered on one line; multiple commands separated by colons on one line are not allowed. BASIC program statements are saved in memory for later execution; BASIC commands cause action and are not saved.

All the 2200 BASIC commands are described on the following pages. Any command that is executed by touching the EXEC key is described with a General Form; any command that causes immediate action is described as a Key.

NOTE:
All BASIC commands and statements can either be entered character-by-character or with the appropriate keyword key.

CLEAR

COMMAND

```
General Form:        CLEAR [ P [ line number [ , line number ] ] ]
                             [ V                                 ]
                             [ N                                 ]
```

Purpose

The CLEAR command clears the user program text and variable areas. CLEAR with no parameters removes all program text and variables from the system. The current console devices (CI and CO) are selected for all I/O operations and the file numbers previously selected are cleared (see SELECT). Also, pause and TRACE are turned off. A HEX(03) [clear CRT] is issued to the Console Output device; the 'READY' message is displayed and control is passed to the keyboard.

CLEAR V removes all variables (both common and noncommon) from memory.

CLEAR N removes all noncommon variables from the system; but names, attributes, and values of common variables are not changed.

CLEAR P removes program text from the system; variables are not disturbed. CLEAR P with no line numbers deletes all user program text from the system. CLEAR P with one line number deletes all user program lines from the indicated line through the highest numbered program line. If two line numbers are entered, all text from the first through the second line numbers, inclusive, is deleted.

Example:

```
CLEAR
CLEAR V
CLEAR N
CLEAR P 10, 20
CLEAR P 10
CLEAR P
```

CONTINUE
COMMAND

General Form:	CONTINUE

Purpose

This command continues program execution whenever the program has been stopped either by a STOP statement, an Immediate Mode GOTO, or the HALT/STEP key. The program continues with the program statement immediately following the last executed program statement, or at the program line specified by the GOTO.

Example:
CONTINUE

NOTE:
CONTINUE cannot be used if the system has dropped out of execution mode. This occurs whenever:
1. a text or table overflow error has occurred.
2. any CLEAR command is executed.
3. a RENUMBER command is executed.
4. any program text is modified.
5. the RESET switch is used.
6. Edit keys are used to recall program text.

HALT/STEP

Key:	HALT/STEP

Purpose

1. If a program is executing, the HALT/STEP key stops execution after the completion of the current statement. Program execution, beginning with the next statement, can be continued by entering the CONTINUE command.
2. If a program is being listed, the HALT/STEP key stops the listing after the current statement has been listed.
3. The HALT/STEP key can be used to step through the execution of a program. If program execution has terminated due to the execution of a STOP statement or the depressing of the HALT/STEP key, depressing the HALT/STEP key again causes the next program statement to be listed and executed; execution then terminates. In multiple statement lines, those statements which have already been executed are not listed; however the colons separating these statements are always displayed. The GOTO statement can be used in the immediate mode to begin stepping execution at a particular line number (see GOTO). However, protected programs may not be stepped.

NOTE:

An error message is printed out and execution does NOT continue if the user attempts to STEP program execution after the system has dropped out of execution mode. This occurs when:

1. a text or table overflow error has occurred.
2. any CLEAR command is executed.
3. program text is modified.
4. a RENUMBER command is executed.
5. the RESET key has been pressed.
6. Edit keys are used to recall program text.

Suppose the following program is in memory:

Example:

```
.
.
:90 Z = 5.5
:100 PRINT "CALCULATE X, Y"
:110 X=1.2: Y=5*Z+X: GOTO 100
.
.
```

and we wish to step through the program from line 90 on. TRACE is turned on so that variables receiving new values are displayed.

Turn TRACE mode on	:TRACE $\boxed{\text{EXEC}}$
Start stepping at line 90	:GOTO 90
Touch HALT/STEP key.	90 Z=5.5 Z= 5.5
Touch HALT/STEP key.	: 100 PRINT "CALCULATE X, Y" CALCULATE X,Y
Touch HALT/STEP key.	: 110 X=1.2: Y=5*Z+X: GOTO 100 X= 1.2
Touch HALT/STEP key.	: 110: Y=5*Z+X: GOTO 100 Y= 28.7
Touch HALT/STEP key.	: 110: :GOTO 100 TRANSFER TO 100
Touch HALT/STEP key.	: 100 PRINT "CALCULATE X,Y" CALCULATE X,Y
Touch HALT/STEP key.	: 110 X=1.2: Y=5*Z+X: GOTO 100 X= 1.2
Touch HALT/STEP key.	: 110: Y=5*Z+X: GOTO 100 Y= 28.7
Touch HALT/STEP key.	: 110: :GOTO 100 TRANSFER TO 100

LIST
COMMAND

General Form:	LIST [S] [line number [, line number]]

Purpose

The LIST command instructs the system to display the entire program text in line number sequence. If one line number follows the command, then one program line is listed. If two line numbers follow the command, all text from the first through the second line numbers inclusive are listed.

The 'S' parameter is a special feature for the CRT terminal. It permits the listing of the program in steps of 15 lines (the maximum capacity of the CRT screen). After 15 lines have been generated, the listing can be continued. To continue listing (up to the limit specified in the LIST command), the CR/LF-EXECUTE key is pressed.

Pressing HALT/STEP during the listing of a program stops the listing after the current statement line has been finished.

Alternatively, the user may slow down listing on the CRT by selecting a pause of from 1/6 to 1 1/2 seconds by executing a SELECT P statement. A pause will occur after each line is listed.

When the 2200 is Master Initialized (Power off, Power on), the CRT is initially selected for LIST operations. Other printing devices may be selected for listing by using a SELECT LIST command (see SELECT).

Examples:

```
       :LIST
       30 READ A, B, C, M
       .
       .
       .
       990 END

or     :LIST 30, 50
       30 READ A, B, C, M
       40 LET G=A*D-B*C
       50 IF G=0 THEN 60

or     :LIST 30
       30 READ A, B, C, M

       :SELECT P3 <---- Select a pause of 1/2 sec.
       :LIST

       :LIST S
       First 15 lines appear on the
       CRT; depressing the CR/LF-
       EXECUTE key lists the next
       15 lines, and so on until the
       entire program has been
       listed.
```

General Form:	RENUMBER [line number] [,line number] [,integer]
where	$0 <$ integer < 100

Purpose

The RENUMBER command renumbers the lines of the user program currently in memory. The first line number is the starting number and specifies the first line to be renumbered in the program. All program lines with line numbers greater than or equal to the starting line number are renumbered. If no starting line number is specified, the entire program is renumbered. The second line number in a RENUMBER command is the new line number which is assigned to the first line to be renumbered; note that the new line number must be greater than the highest line number preceding that line in the program. For example, if we are to renumber the following program starting with line 12, the new number assigned to line 12 must be >10 since line 10 precedes line 12 in the program.

Examples:

```
READY
:10  INPUT X
:12  FOR I = 1 TO 10
:14  PRINT X*I
:16  IF I > 100 THEN 20
:18  NEXT I
:20  STOP
:_

:RENUMBER 12, 20
:LIST
10   INPUT X
20   FOR I = 1 TO 10
30   PRINT X*I
40   IF I > 100 THEN 60
50   NEXT I
60   STOP
```

The integer specified in the RENUMBER command is the increment between line numbers; if no integer is specified, the increment is assumed to be 10. If no new starting line number is specified, the new starting line number equals the increment.

NOTE:

All references to line numbers within the program; e.g., in GOTO, GOSUB, or PRINTUSING statements are modified.

Examples:

```
RENUMBER
RENUMBER  100, 5
RENUMBER  100, 150, 5
RENUMBER  5
RENUMBER  , , 5
```

RESET

Key:	RESET

Purpose

The RESET switch immediately stops program listing or execution, clears the CRT screen, resets all I/O devices and returns control to the user. The program text is not lost; all program variables are maintained with their current values. If the TRACE mode was on, it is turned off.

Normally, program execution is terminated by the HALT/STEP command after which a program can be continued. RESET, on the other hand, terminates immediate execution statements or commands and restores the system after a temporary malfunction. RESET can be used to terminate program execution, but the program cannot be continued. The program can be rerun by touching the RUN key.

NOTE:

RESET should only be used to terminate program execution if HALT/STEP fails.

If the system has undergone a temporary malfunction which cannot be corrected by RESET, master initialize the system by turning the power switch on the Power Supply Unit off, then on again. This, however, erases programs and data previously in the system.

Example:

RESET

General Form:	RUN	[line number]

Purpose

The RUN command performs four operations:

1. Verifies syntax of the currently loaded program.
2. Resolves the program, setting all new numeric variables to zero and all new alpha variables to spaces. (Previously entered common variables are left intact.)
3. Resets the DATA value pointer (to be used in a READ statement), if needed, to the first data value in the program.
4. Executes the program line by line, starting with the lowest numbered line.

If a line number is specified, program execution begins at the specified line number without reinitializing program variables to zero; the variables are maintained at the last calculated values. Program execution must not be started in the middle of a FOR/NEXT loop or a subroutine.

NOTE:
Program execution should not be initiated directly with a Special Function key without using the RUN command; failure to issue a RUN command may bypass program resolution and cause a program error.

Examples:

```
RUN
RUN 30
```

Key:	Special Function Key

Purpose

There are 16 special function keys available on any system keyboard. Depressing them simultaneously with the SHIFT key provides up to 32 entry points for the currently loaded BASIC program, and allows the user to define his own special functions. The entry points are defined by the BASIC statement DEFFN' XX (where XX = 00 to 31). Thus, depressing special function key 2 causes an entry and execution of a line or subroutine beginning with a DEFFN' 2 statement. With this special entry, text strings can be entered or multi-argument subroutines can be executed.

If a special function key is defined for text entry, pressing the key causes the character string defined by the DEFFN' to be displayed and become part of the current text line (see DEFFN').

For example, if special function key 2 is defined by the following statement:

```
100 DEFFN' 2 "HEX("
```

pressing the special function key 2 after the following has been keyed in:

```
:20  PRINT
```

results in

```
:20  PRINT HEX(_
                 ↖cursor
```

on the CRT.

If a special function key is defined for marked subroutine entry (see DEFFN'), the subroutine can be executed either manually by touching the indicated special function key, or by using a GOSUB' statement (see GOSUB') within a program. Arguments are passed to the subroutine either by keying them in, separated by commas, immediately before the special function key is pressed, or by indicating them as parameters in the GOSUB' statement. The number of arguments passed must equal the number of variables in the DEFFN' statement marking the subroutine. When a RETURN statement is finally executed, control is passed back to the keyboard or to the program statement immediately following the GOSUB' statement.

Example:

```
:12.3, 3.24, "JOHN"
```
(Depress special function key 3.)

causes the following subroutine to be executed:

```
:100 DEFFN' 3 (A, B, C$)
:110 . . .
:120 . . .
       .
       .
:200 RETURN
```

where A is set to 12.3
B is set to 3.24
C$ is set to "JOHN"

Example:

Define the special function key 0 to evaluate the expression

$Z = 7X^2 + 14Y^2 - 7$

```
READY
:10 DEFFN' 0 (X, Y)
:20 Z = 7 * X ↑ 2 + 14 * Y ↑ 2 - 7
:30 PRINT "X=";X
:40 PRINT "Y=";Y
:50 PRINT "Z=";Z
:60 RETURN
:70 PRINT "END OF SR"
:_
```

Execute the subroutine for

X=.092 and Y=-.32

Solution:

(A) MANUAL ENTRY

Key .092, -.32

Touch special function key 0.

CRT Display:

```
:.092, -.32
X= 9.20000000E-02
Y=-.32
Z= 5.507152
:_
```

(B) PROGRAM ENTRY

```
READY
5 GOSUB' 0 (.092, -.32)
6 STOP "SUBROUTINE DONE"
:RUN
X = 9.20000000E-02
Y = -.32
Z = 5.507152

STOP SUBROUTINE DONE
:_
```

A RETURN CLEAR statement can be used at the end of a subroutine to continue execution without returning to the statement following the subroutine entry point. For example, if line 60 above = RETURN CLEAR, the output is:

(C) MANUAL ENTRY

Key .092, -.32

Touch special function key 0.

```
:.092, -.32
X= 9.20000000E-02
Y=-.32
Z=5.507152
END OF SR
```

(D) PROGRAM ENTRY

```
RUN
X = 9.20000000E-02
Y = -.32
Z = 5.507152
END OF SR
```

STATEMENT NUMBER
(NOT ON ALL KEYBOARDS)

Key:	STATEMENT NUMBER KEY

Purpose

This key automatically sets the line number of the next line to be entered. The line number generated is 10 more than the highest existing line number.

The statement number can also be keyed in manually, using the numeric entry keys. Statement numbers can be any integer from 1 to 4 digits.

Statements may be entered in any order; however, they are usually numbered in increments of five or ten so additional statements can be easily inserted. The system keeps them in numerical order regardless of how they are entered.

Example:

```
                                    READY
                                   (:10 X, Y, Z = 0
    Currently Entered Program      {:20 INPUT "ENTER VALUES", A, B
                                   (:30 Z = A*B + B↑2
    Depressing STMT NUMBER key      :40_
```

Section VII General BASIC Statements and Functions

Section VII General BASIC Statements and Functions

BASIC STATEMENTS

A BASIC statement is a word or operator defined in the BASIC language, which performs a specific operation or function, and is combined with an expression, a variable or some data values. BASIC statements are always parts of programs unlike BASIC commands such as RUN which cause immediate action and cannot be part of a program.

Examples:

READ A, B	A statement:	verb followed by variables
DATA 1, 4	A statement:	verb followed by values
LET A = 6*B	A statement:	verb followed by a variable (A), an equals sign, and an expression (6*B).

BASIC statement lines in a program must always begin with a line number; statement lines in the immediate mode do not require line numbers.

There are two types of BASIC statements: executable and non-executable. An executable statement specifies program action:

```
:10  READ A, B
:20  A = 6*B
:_
```

A non-executable statement provides information for program execution:

```
:10  DATA 1, 4
:_
```

or for the programmer:

```
:20  REM THIS IS PROGRAM 1
:_
```

A series of statements, separated by colons, may be entered on one line.

Example:

```
:20  FOR I = 1 TO 10 :PRINT I,X(I)*Y :NEXT I
:
```

or:

```
:FOR J = 1 TO 3 :PRINT J,J↑2, J↑3 :NEXT J
1     1     1
2     4     8
3     9    27
:
```

The remainder of this section defines the general BASIC statements available in the System 2200 for programming and their syntax.

ADD

(NOT ON 2200A)

General Form:	$\text{ADD}\begin{bmatrix}\text{C}\end{bmatrix}\left(\text{alpha variable,}\begin{Bmatrix}\text{hh}\\ \text{alpha variable}\end{Bmatrix}\right)$

where:	h = hexdigit
	C = add with carry

Purpose

The ADD statement is used to add the binary value specified by the second argument (an alphanumeric variable or two hexdigits) to the binary value specified by the first argument, an alphanumeric variable. The entire defined lengths of both alphanumeric variables are used in the addition, including trailing spaces. (Note: For most alphanumeric operations in the System 2200, if an alphanumeric variable receives a value with a length less than the maximum length of the variable, the remaining characters are all set equal to spaces. These trailing spaces normally are not considered to be part of the value.) Part of an alphanumeric variable can be operated on by using the STR function to specify a portion of the variable. For example,

```
ADD (STR(A$, 3, 2), 80)
```

Two types of adding may be done:

1. Immediate. Indicated by the second argument in the statement being two hex digits.
2. String-to-String. Specified by the second argument being a variable.

Immediate ADD

The immediate ADD statement adds (in binary) the character specified by the two hex digits to the entire value (each character in the define length) of the specified alphanumeric variable. If 'C' is not specified, the character is added independently to each character in the receiving alphanumeric variable with no carry propagation. If 'C' is specified, the character is added to the low order (last) character of the receiving alphanumeric variable and a carry, if present, is propagated to high order characters.

Example:

```
        If A$ = HEX (0123),
then    ADD (A$, 02)
        sets A$ = HEX (0325)

        If A$ = HEX (0123)
then    ADDC (A$, 02)
        sets A$ = HEX (0125)

        If A$ = HEX (02FFFE),
then    ADDC (A$, 02)
        sets A$ = HEX (030000)
```

String-to-String ADD

The String-to-String ADD statement adds (in binary) the entire value of the second alphanumeric variable to the entire value of the first alphanumeric variable. If 'C' is not specified, the add is on a character by character basis with no carry propagation. That is, the last character of the second value is added to the last character of the first value; then, the next to last character of the second value is added to the next to last character of the first value; and so forth. If 'C' is specified, the second value is treated as a single binary number and is added to the first value with carry propagation between characters.

If the two alphanumeric variables specified are not of the same defined length, the following rules apply:

1. The addition will be right adjusted, with lead characters of zero binary value being assumed for the variable of shorter length.
2. The answer will be stored right adjusted in the receiving variable. If the total answer is longer than the receiving variable the lower order portion of the answer will be stored.

Example:

```
If A$ = HEX (0123) and B$ = HEX (00FF),
ADD (A$, B$) sets A$ = HEX (0122)

If A$ = HEX (0123) and B$ = HEX (00FF)
ADDC (A$, B$) sets A$ = HEX (0222)
```

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.

The LEN function is also useful in determining the length of an alphanumeric variable value in conjunction with ADD operations.

Examples:

```
10  ADD (A$, FF)
20  ADDC (STR(A$, 3, 1), 81)
30  ADD (A$, B$)
40  ADDC (STR(A$, 3, 2), STR(B$, 4, 2) )
50  ADD (A$(I, J), I$)
```

AND, OR, XOR

(NOT ON 2200A)

General Form:

$$\left\{\begin{matrix}\text{AND}\\ \text{OR}\\ \text{XOR}\end{matrix}\right\}\left(\text{alpha variable, }\left\{\begin{matrix}\text{hh}\\ \text{alpha variable}\end{matrix}\right\}\right)$$

where: h = hexdigit

Purpose

These statements perform the specified logical operation (AND, OR or EXCLUSIVE OR) on the characters of the value of the first alphanumeric variable. All characters in this value are operated on including trailing spaces. (Note: for most alphanumeric operation in the System 2200, if an alphanumeric variable receives a value with a length less than the maximum defined length of the variable, the remaining characters are all set equal to spaces. The trailing spaces normally are not considered to be part of the value.) Part of an alphanumeric variable can be operated on by using the STR function to specify a portion of the variable. For example,

AND (STR(A$, 3, 2),80)

AND outputs a 1-bit if both bits are 1; or outputs a 1-bit unless both bits are 0; XOR outputs a 1-bit if both bits are different.

Two types of logical functions may be performed:

1. Immediate. Indicated by the second argument in the statement being two hex digits.
2. String-to-String. Specified by the second argument being a variable.

Immediate Logical Functions

The immediate logical functions form the logical AND, OR, or EXCLUSIVE OR of the characters specified by the two hex digits and each character in the defined length of the alphanumeric variable (or portion of alphanumeric variable if a STR function is used). The result becomes the new value of the alpha variable.

Example:

if A$ = HEX (41424320), OR(A$, 80)
OR's the character '80' with each character
in A$; thus, A$ would equal HEX(C1C2C3A0).

String-to-String Logical Functions

The String-to-String logical functions form the logical AND, OR, or EXCLUSIVE OR of the characters in the first alphanumeric variable with the characters in the second alphanumeric variable on a character by character basis starting with the first character of each variable. The first variable receives the result. If the second alphanumeric variable is shorter than the first, the remaining characters of the first alphanumeric variable are unchanged. If the second alphanumeric variable is longer than the first, the remaining characters are ignored.

Example:

if A$ = HEX (010203) and
B$ = HEX (4151), OR (A$, B$)
sets A$ = HEX (415303).

Examples:

```
10  AND (A$, 7F)
20  OR (A$(1), B$)
30  XOR (STR(A$, 2, 3), F0)
40  AND (A$, STR(B$, I))
```

General Form:	BIN (alpha variable) = expression
where:	0 ≤ value of expression < 256

Purpose

This statement converts the integer value of the expression to a character (i.e., to a one-byte binary number) and sets the first character of the value of the specified alphanumeric variable equal to the character. BIN is the inverse of the function VAL.

BIN can be especially useful for code conversion or for conversion of numbers from internal decimal to binary.

Examples:

```
10  BIN(A$) = 64                  sets A$ = HEX(40) (HEX(40) has
20  BIN(STR(A$, I, 1)) = X*T/2      a decimal value of 64)
```

BOOL

(NOT ON 2200A)

General Form:	BOOL h (alpha variable, {hh / alpha variable})	
where:	h = hexdigit	

The statement BOOL is a generalized logical operator that operates on the bits of the characters of the first alphanumeric variable. All characters are operated on including trailing spaces. (Note: For most System 2200 alphanumeric operations if an alphanumeric variable receives a value with a length less than the maximum defined length of the variable, the remaining characters are all set to spaces. These spaces normally are not considered to be part of the value.) Part of an alphanumeric variable can be operated on by using the STR function to specify a portion of the variable. For example,

BOOL 9 (STR(A$, 2, 3), A7)

The hexdigit following 'BOOL' defines which of the 16 available logical operations is to be performed (see chart below). Each byte of the first alpha variable, or its specified portion, is operated on, one bit at a time.

BOOL digit	Logical Operation	Result if arg 1 = 1100 and arg 2 = 1010
0	null (bits always = 0; logical inverse of BOOL F)	0000
1	not OR (1 iff *corresponding bits of both arg 1 and arg 2 = 0;)	0001
2	(1 iff *corresponding bits of arg 2 = 1 and arg 1 = 0;)	0010
3	binary complement of arg 1 (1 iff *bit of arg 1 = 0; otherwise 0)	0011
4	(1 iff *corresponding bits of arg 2 = 0 and arg 1 = 1;)	0100
5	binary complement of arg 2 (1 iff *bit of arg 2 = 0)	0101
6	exclusive OR (1 iff *corresponding bits of arg 1 and arg 2 are different)	0110
7	not AND (0 iff*corresponding bits of both arg 1 and arg 2 = 1)	0111
8	AND (1 iff *corresponding bits of both arg 1 and arg 2 = 1)	1000
9	equivalence (1 iff *corresponding bits are the same; i.e., both = 1 or both = 0)	1001
A	arg 2 (identical to bits of arg 2)	1010
B	arg 1→arg 2 (arg 1 implies arg 2; 1 unless arg 1=1 and arg 2=0)	1011
C	arg 1 (identical to bits of arg 1)	1100
D	arg 2→arg 1 (arg 2 implies arg 1; 1 unless arg 2=1 and arg 1=0)	1101
E	OR (1 unless both corresponding bits = 0)	1110
F	identity (bits always = 1; logical inverse of BOOL 0)	1111

*iff = if and only if

Two types of logical operations may be performed:
1. Immediate. Indicated by the second argument in the statement being two digits.
2. String-to-String. Specified by the second argument in the statement being a variable.

Immediate Logical Operations
The logical operation specified by the hex digit after 'BOOL' is performed using the character specified

Example:

BOOL 3 (A$, 00) complements each character in the value of A$.

String-to-String Logical Operations
The logical operation specified by the hex digit following 'BOOL' is performed on the characters in the first alphanumeric variable with the characters of the second alphanumeric variable on a character by character basis starting with the first character of each variable. The first variable receives the result. If the second variable is shorter than the first variable, the remaining characters in the first value are unchanged. If the second variable is longer than the first, the remaining characters are ignored.

Example:

if A$ = HEX (4145) and B$ = HEX (2185),
BOOL 7 (A$, B$) sets A$ = HEX (FEFA).

Examples:

```
10  BOOL1 (A$, F0)
20  BOOL7 (A$, B$)
30  BOOLE (STR(A$, I, 2), A5)
40  BOOL8 (A$, STR(B$, 2, 3)
```

General Form:	COM com element [, com element] . . .	
where	com element =	numeric scalar variable numeric array variable (integer [, integer]) alpha scalar variable [length integer] alpha array variable (integer [, integer]) [length integer]
		0 < length integer ≤ 64 0 < integer ≤ 255

Purpose

The COM statement defines scalar variables or arrays which are to be used in common by several program segments. Common variables are stored in an area of memory which is not cleared when subsequent programs are run.

When a program is run, previously existing common variables and their values are not disturbed. However, all non-common variables are cleared from memory. Common variables are only removed from the system when a CLEAR or CLEARV command is executed or the system is Master Initialized (i.e., turned on). The COM statement also provides array definition identical to the DIM statement for array variables; the syntax for one COM statement can be a combination of array variables (A(10), B(3,3)) and scalar variables (C2, D, X$). Integers must be used for array dimensions.

The common area variables must be defined before any other variable in the program is defined. Therefore, COM statements should be assigned the lowest executable line numbers in the program.

The following general rules apply to the COM statement:

1. A COM statement must not change the dimensions of a previously defined common variable.
2. Common variables must be defined before any noncommon variables are defined, or referred to in the program.
3. The number of array elements must not exceed 4096 in any one array.

The COM statement can be used to set the maximum length of alphanumeric variables (the maximum length is assumed to be 16 if not specified). The length integer (≤64) following the alpha scalar (or alpha array) variable specifies the maximum length of that alpha variable (or those array elements).

If a particular set of common variables are to be used in several sequentially run programs, the COM statements do not have to appear in any program other than the first. The variables will remain defined as common variables with the originally defined dimensions, lengths and values in subsequent programs. The COM statements may however, be included in subsequent programs (with identical dimensions and lengths) and new common variables may be defined. Some or all common variables may be defined as non-common with the COM CLEAR statement.

Examples:

```
10 COM A(10),B(3,3),C2
20 COM C, D(4,14), E3, F(6), F1(5)
30 COM M1$, M$(2,4), X,Y
40 COM A$10, B$(2,2) 32
```

```
General Form:
                    COM CLEAR      [ scalar variable   ]
                                   [ array designator  ]
```

Purpose

Either defines as non-common some or all previously defined common variables, or defines as common some or all previously defined non-common variables.

When a variable is not specified in a COM CLEAR statement, all currently defined common variables are redefined as non-common variables.

When a common scalar variable or array designator is listed in the COM CLEAR statement, all common variables which are defined in the program prior to the specified variable or defined in previous program segments are left as common variables; the specified common variable and all variables defined after it in the program are made non-common variables.

When a non-common variable or array designator is listed in the COM CLEAR statement, all occurrences of non-common variables defined earlier are made common.

COM CLEAR is extremely useful with program overlaying (chaining) to specify which variables are to be removed from the system when the overlay is performed.

All common variables are eliminated during overlaying. COM CLEAR also may be useful when debugging programs involving common variables, temporarily defining them as non-common for debugging. If the variable specified in the COM CLEAR statement is not defined, error 02 results.

Examples:

```
:10 COM CLEAR                          Redefines all previously defined
                                       common variables as non-common.

:110 COM A, B, X, Y (10)
:200 COM CLEAR X                       Redefines X and Y as non-common

:10 DIM X (10, 10)
:20 A = B + C
:30 COM CLEAR B                        Redefines X and A as common variables
```

CONVERT
(NOT ON 2200A)

General Forms:	1. CONVERT alpha variable TO numeric variable
	or
	2. CONVERT expression TO alpha variable, (image)
where: image	= [±] [#...] [.] [#...] [↑↑↑↑]
	0 < number of #'s < 14

Purpose

Alpha-to-Numeric Conversion

The CONVERT statement used with the first form converts the number represented by ASCII characters in the alphanumeric variable to a numeric value and sets the numeric variable equal to that value. For example, if A$ = "1234", CONVERT A$ TO X sets X = 1234. An error will result if the ASCII characters in the specified alphanumeric variable are not a legitimate BASIC representation of a number. Part of an alphanumeric value can be converted to numeric by using the STR function. For example,

CONVERT STR(A$, 1, 8) TO X

Alpha-to-numeric conversion is particularly useful when numeric data is read from a peripheral device in a record format that is not compatible with normal BASIC DATALOAD statements, or when a code conversion is first necessary. It also can be useful when it is desirable to validate keyed-in numeric data under program control. (Numeric data can be received in an alphanumeric variable, and tested with the NUM function before converting it to numeric.)

Numeric-to-Alpha Conversion

The CONVERT statement used with the second form converts the numeric value of the expression to an ASCII character string according to the image specified; the alphanumeric variable is set equal to that character string. The image specifies precisely how the numeric value is to be converted. Each character in the image specifies a character in the resultant character string. The image is composed of # characters to signify digits and optionally +, −, ., and ↑ characters to specify sign, decimal point, and exponent characters.

The image can be classified into two general formats:

Format	*Example:*
Fixed Point	##.##
Exponential	#.###↑↑↑↑

Numeric values are formatted according to the following rules:

1. If the image starts with a plus (+) sign, the sign of the value (+ or −) precedes the character string.
2. If the image starts with a minus (−) sign, a blank for positive values and a minus (−) for negative values precedes the character string.
3. If no sign is specified in the image, sign is omitted from the character string.
4. If the fixed point image is used, 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 will result if the numeric value exceeds the image specification.
5. If the exponential image is used, 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.

Numeric to Alpha conversion is particularly useful when numeric data must be formatted in character format in records (especially for alphanumeric sorting).

Examples:

```
10  CONVERT A$ TO X
20  CONVERT STR(A$, 1, NUM(A$)) TO X(1)
```

Examples:
(numeric to alpha)

```
10  CONVERT X TO A$, (###)
    (result: A$ = "012")          where: X = 12.195
20  CONVERT X*2 TO A$, (+##.##)
    (result: A$ = "+24.39")
30  CONVERT X TO STR(A$, 3, 8), (-#.#↑↑↑↑)
    (result: STR(A$, 3, 8) = "  1.2E+01")
40  CONVERT X TO A$, (####.#####)
    (result: A$ = "0012.19500")
```

General Form:	DATA n [,n] . . .
where	n = number or a character string enclosed in double quotation marks.

Purpose

The DATA statement provides the values to be used by the variables in a READ statement. The READ and DATA statements thus provide a means of storing tables of constants within a program.

Each time a READ statement is executed in a program the next sequential value(s) listed in the DATA statements of the program are obtained and stored in the variable(s) listed in the READ statement. The values entered with the DATA statement must be in the order in which they are to be used: items in the DATA list are separated by commas. If several DATA statements are entered, they are used in order of statement number. Numeric variables in READ statements must reference numeric values; alphanumeric variables must reference character strings enclosed in quotation marks (").

The RESTORE statement provides a means to reset the current DATA statement pointer (and reuse the DATA statement values (see RESTORE).

The DATA statement may not be used in the Immediate Mode.

Example:

```
:10  READ W
:20  PRINT W, W↑2
:30  GOTO 10
:40  DATA 5, 8.26, 14.8, -687, 22
:RUN
5      25
8.26   68.2276
14.8   219.04
-687   471969
22     484
10 READ W
          ↑ERR27    (insufficient data)
```

In the above example the 5 values listed in the DATA statement are sequentially used by the READ statement and printed. When a 6th value is requested, an error is displayed since all DATA statement values have been used.

Examples:

```
40  DATA 4, 3, 5, 6
50  DATA 6.56E + 45, -644.543
60  DATA "BOSTON, MASS", "SMITH", 12.2
```

NOTE:

On the 2200A, statements following DATA statements on multiple statement lines are not executed.

General Form:	DEFFN a(v) = expression
where a =	the identifier, a letter or digit which identifies the function
v =	a numeric scalar variable, the dummy variable

Purpose

The DEFFN statement is used to define a user's unique functions. Once defined, these functions can be used in expressions from any other part of the program. The function provides one dummy variable whose value is supplied when the function is referenced. Defined functions can be referenced up to five levels. The following program lines illustrate how DEFFN is used.

```
:10  X = 3
:20  DEFFN A(Z) = Z↑2 - Z
:30  PRINT X + FNA (2*X)
:40  END
:RUN
33
```

Processing

1. Evaluate the FN expression for the scalar variable (i.e., 2*X=6).
2. Find the DEFFN with the matching identifier (i.e., A).
3. Set the dummy variable (line 20) equal to the value of evaluated expression (i.e., Z=6).
4. Evaluate the DEFFN expression and return the calculated value (i.e., Z↑2 – Z).
   The above example prints the value 33, since 3 + (6↑2 – 6 = 33).

The DEFFN statement may be entered any place in a program, and the expression may be any formula which can be entered on one line. A function cannot refer to itself (e.g., the statement DEFFN A (X) = X + FNA(X) is illegal); it can refer to other functions. Up to five levels of function nesting are permitted. For example, the statements:

```
DEFFN1 (A)   = 3 * A
DEFFN2 (A)   = A+FN1 (A)
DEFFN3 (J)   = J+FN2 (J)
DEFFN4 (I)   = FN3 (I)/I
DEFFN5 (K)   = FN4 (K)
DEFFN6 (L)   = FN5 (L)
```

are nested to six levels and produce ERR 09 at execution time. Two functions cannot refer to each other (an endless loop); for example, the following combination of statements is illegal.

```
DEFFNA (A)  = FNB (A)
DEFFNB (A)  = FNA (A)
```

A reference to a DEFFN statement cannot be made from the Immediate Mode. The dummy scalar variable in the DEFFN statement can have a name identical to that of a variable used elsewhere in the program or in other DEFFN statements; current values of the variables are not affected during FN evaluation.

Examples:

```
60  DEFFN A (C) = (3*A) - 8C + FNB (2-A)
70  DEFFN B (A) = (3*A) - 9/C
80  DEFFN4(C) = FNB(C) * FNA(2)
```

General Form:

```
                 DEFFN' integer (variable [, variable] . . .)          )
          or                                                            > all systems
                 DEFFN' integer "character string"                     )

          or     ("character string" [ ;("character string")]  )) 2200C, 2200T
   DEFFN integer (HEX(hh[hh] . . .)  [ ;(HEX(hh[hh]. . .)  )] ..)) WCS/20, WCS/30
                                                                 )  systems only
          where  integer = ( 0 to 31 for keyboard special function key entries
                           ( 0 to 255 for internal program references
```

Purpose

The DEFFN' statement has two purposes:

1. To define a character string to be supplied when a special function key is used for keyboard text entry.
2. To define keyboard special function key or program entry points for subroutines with argument passing capability.

The DEFFN' statement must be the first statement on a line (i.e., it must immediately follow the line number). DEFFN' may not be used in immediate mode.

KEYBOARD TEXT ENTRY DEFINITION: To be used for keyboard entry, the integer in the DEFFN' statement must be a number from 0 to 31, representing the number of a special function key. When the corresponding special function key is pressed, the user's "character string" is displayed and becomes part of the currently entered text line. The character string is all characters included between the double quotation marks.

For example, statement 100 defines special function key number 12 as the character string "HEX(":

```
:100   DEFFN' 12 "HEX("
```

Pressing special function key number 12 after the following has been keyed in

```
:200   PRINT
```

results in the following line being displayed

```
:200   PRINT HEX(
```

Example:

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

DEFFN' HEX (not on 2200A or B; on 2200S and WCS/10 only with Advanced Programming Statements)

The character string may be represented by a character string in quotes, a HEX function or a combination of those elements. For example, line 20 defines Special Function key 01 as the character string "LOAD/10B carriage return:"

```
20 DEFFN' 01" LOAD/10B";HEX(0D)
```

NOTE:
The Special Function keys can be defined to output characters that do not appear on the keyboard by using the HEX function to specify the codes for these characters.

Examples:

```
10 DEFFN' 31 "PRINT HEX("
40 DEFFN' 12 HEX (1F)
50 DEFFN' 02 "PRINT"; HEX(22); "ANYONE CAN PLAY! "; HEX(220D)
```

NOTE:
When using Special Function keys for Immediate Mode execution, HEX(0D) must be the last character.

MARKED SUBROUTINE ENTRY DEFINITION

The DEFFN' statement, followed by an integer and an optional variable list enclosed in parentheses, indicates the beginning of a marked subroutine. The subroutine may be entered from the program via a GOSUB' statement (see GOSUB'), or from the keyboard by pressing the appropriate special function key. If subroutine entry is to be made via a GOSUB' statement, the integer in the DEFFN' statement can be any integer from 0 to 255; if the subroutine entry is to be made from a special function key, the integer can be from 0 to 31. When a special function key is depressed or a GOSUB' statement is executed, the BASIC program is scanned for a DEFFN' statement with an integer corresponding to the number of the special function key or the integer in the GOSUB' statement. Execution of the program then begins at that statement (i.e., if special function key 2 is pressed, execution begins at the DEFFN' 2 statement).

When a RETURN statement is encountered in the subroutine, control is passed to the program statement immediately following the last executed GOSUB' statement, or back to keyboard entry mode if entry was made by touching a special function key. The DEFFN' statement may optionally include a variable list. The variables in the variable list receive the values of arguments being passed to the subroutine; if the number of arguments to be passed is not equal to the number of variables in the list, an error results. In a GOSUB' subroutine call made internally from the program, arguments are listed (enclosed in parentheses and separated by commas) in the GOSUB' statement (see GOSUB').

Example:

```
:100    GOSUB' 2 (1.2, 3+2 * X, "JOHN")
 .
 .
:150    STOP
:200    DEFFN' 2 (A, B(3), C$)
 .
 .
:290    RETURN
```

For special function key entry to a subroutine, arguments are passed by keying them in, separated by commas, immediately before the special function key is depressed.

Example:

:1.2, 3.24, "JOHN" (now depress special function key 2)

The DEFFN' statement need not specify a variable list. In some cases it may be more convenient to request data from a keyboard in a prompted fashion.

Example:

```
100 DEFFN'4
110 INPUT "RATE", R
120 C = 100 * R - 50
130 PRINT "COST="; C
140 RETURN
```

When a DEFFN' subroutine is executed via keyboard special function keys while the system is awaiting data to be entered into an INPUT statement, the INPUT statement will be repeated in its entirety, upon return from the subroutine.

Example:

```
100 INPUT "ENTER AMOUNT",A
     .
     .
     .
200 DEFFN' 1
210 INPUT "ENTER NEW RATE",R
220 RETURN
```

```
DISPLAY:    ENTER AMOUNT?
            (Depress Special Function Key  1)
            ENTER NEW RATE? 7.5
            ENTER AMOUNT?
```

DEFFN' subroutines may be nested (i.e., call other subroutines from within a subroutine).

NOTE:

The DEFFN' statement may be used in conjunction with the special function keys to provide a number of entry points to run a program. Because, however, the system stores DEFFN' return information in a table, this should not be done repetitively unless:

1. *The RESET key is depressed prior to the special function key.*
2. *Program operation terminates with a RETURN statement (back to keyboard mode).*

Failure to do this will eventually cause a table overflow error (ERR 02). To eliminate this problem; use the RETURN CLEAR statement (not available on a 2200A or B).

General Form: DIM dim element [, dim element] . . .

where

$$\text{dim element} = \begin{Bmatrix} \text{numeric array variable (integer [, integer])} \\ \text{alpha array variable (integer [, integer]) [length integer]} \\ \text{alpha scalar variable [length integer]} \end{Bmatrix}$$

$0 <$ length integer $\leqslant 64$ $\quad 0 <$ integer $\leqslant 255$

Purpose

The DIM statement reserves space for one or two dimensional array variables which are referenced in the program. Space may be reserved for more than one array with a single DIM statement by separating the entries for array names with commas as shown in line 40 of the example below.

DIM statements must appear before any use of the variables in the program, and the space to be reserved must be explicitly indicated – expressions are not allowed.

The following rules apply to the use and assignment of array variables in a DIM statement.

1. The numeric value of the subscript of the first element must be 1; zero is not allowed.
2. The dimension(s) of an array cannot exceed 255; the dimensions must be integers.
3. The number of array elements must not exceed 4096 in any one array.

The DIM statement can also be used to set the maximum length of alphanumeric variables (the maximum length is assumed to be 16 if not specified). The integer ($\leqslant$ 64) following the alphanumeric variable or alpha array variable specifies the maximum length of that alpha variable (or those alpha array elements).

Examples:

20 DIM I(45)	Reserves space for a 1-dimensional array of 45 elements.
30 DIM J (8, 10)	Reserves space for a 2-dimensional array of 8 rows and 10 columns.
40 DIM K(35), L(3), M(8,7)	Reserves space for two 1-dimensional and one 2-dimensional array.
50 DIM A$32	Sets the maximum length of the variable A$ = 32 characters.
60 DIM B$(4,4) 10	Reserves space for the 2-dimensional alpha array with the maximum length of each array element = 10 characters.

General Form:	END

Purpose

This is an optional program statement used to compute the amount of free space remaining in memory. It need not be the last executable statement in a program. More than one END statement may be used in a program.

When the system executes an END statement, the following message is printed out.

```
END PROGRAM
FREE SPACE = xxxxx
```

and program execution terminates. "xxxxx" is the approximate amount of memory (in bytes) not used by this program.

In addition, when a program is being keyed into the system, an END statement may be entered without a line number (immediate mode) to obtain the FREE SPACE available at any particular time in the system.

Example:

```
:100   X=24 - 2*4
:110   PRINT Y,X
:END
 END PROGRAM
 FREE SPACE = 2379
:
```

The amount of free space displayed when END is executed is determined in two different ways:

1. When program is keyed in or loaded from a tape or other peripheral device following a CLEAR command, the free space displayed after entering an END statement in immediate mode reflects only the space occupied by the program.
2. After the program has been executed once, the free space displayed after either an immediate mode END or a program executed END reflects both the space taken up by the program and variables.

Example:

```
999 END
```

General Form:		FN a (expression)
	where	a is a function identifier previously defined in a DEFFN statement

Purpose

This function is used to refer to or call a function previously defined in a DEFFN statement which contains the same identifier. The variables in the FN expression need not be the same as the dummy variable in the associated DEFFN statement (see DEFFN).

Examples:

```
10 DEFFN A(A) = 3 * A
20 J = FNA (B) + K
```

General Form:	FOR v = expression TO expression [STEP expression]
	where v = a numeric scalar variable

Purpose

The FOR statement, and the NEXT statement, are used to specify a loop. The FOR statement is used at the beginning of the loop; the NEXT statement at the end. The program lines in the range of the FOR statement are executed repeatedly, beginning with v = '1st expression'; thereafter, v is incremented by the value specified in the STEP expression until the value of v passes the limit specified by the TO expression. The STEP portion of the statement may be positive or negative or may be omitted. If omitted, a step size of +1 is assumed. A STEP expression is evaluated only once, at the first entry to the loop. Loops may be nested with no limit.

If illegal values are assigned to the parameters in a loop (i.e., if the increment designated by STEP is in the wrong direction or 0), the loop is executed once only and program execution continues. Examples of invalid values are:

FOR R = 1 TO 10 STEP −1	Wrong Direction of STEP Expression.
FOR R = −1 TO −10 STEP 1	Wrong Direction of STEP Expression.
FOR R = 1 TO 10 STEP 0	STEP Expression equals 0.

A loop is executed to completion only if the values assigned the parameters are valid. The following restrictions apply to the use of FOR loops:

1. Branching into the range of a FOR loop from the loop is not permissible (GOTO, GOSUB, IF-THEN).
2. Branching out of range of a FOR loop is permissible; however, to conserve memory, it should not be done repeatedly unless a subsequent normal termination of an outer loop occurs or unless the loop is completely contained in a GOSUB routine. If repetative branches are made out of FOR loops, without terminating the loops, the FOR loop information is accumulated in an internal compiler table. This will eventually cause a table overflow condition (ERROR 02). See examples illustrating legal branches out of a loop .
3. Branching out of a FOR loop with a RETURN statement is legal but the loop is considered to be complete (i.e., branching back into the loop is illegal and an error message will be issued when the NEXT statement is encountered).

Example:

```
READY
:20 FOR Z3 = A(K) TO -COS(J) STEP -8 + INT(P(2))
:30 R(Z3) = A(K) + A(Z3)
:40 FOR Z4 = R(Z3) TO A(K) : Q(Z4) = 2*Z4*R(Z3)
:50 PRINT Q(Z4), "VALUE",FN6(Q(Z4))
:60 NEXT Z4: NEXT Z3
:_
```

Example:

```
:
:100 FOR I = 1 TO X
:110 IF A(I) > 100 THEN 130
:120 I=X :NEXT I : GOTO 200
:130 M = M + A(I) - B(I)
:140 NEXT I
          .....
:200 C = M*100/I
```

Legal branch out of FOR loop which properly terminates loop to avoid accumulation of FOR loop information in internal compiler stack.

Example:

```
READY
:20 FOR X = 1 TO 50
:30 PRINT X, SQR(X)
:40 NEXT X
:_
```

Example:

```
READY
:50 GOTO 70
:60 FOR I = 1 TO 10 STEP 2
:70 LET (ZI) = FNA(I)-LOG(I)
:90 NEXT I
```

Illegal branch into a FOR loop

```
:100 FOR J = 1 TO 4
:110 FOR K = 1 TO 6
:120 IF Z(K) > 10 THEN 160
:150 NEXT K
:160 NEXT J
:200 GOSUB 300
          .....
:300 FOR X = .1 TO Z STEP .05
:340 IF A(I) < 3.25 THEN 400
:390 NEXT X
:400 RETURN
```

Proper branches out of a FOR loop

FOR Loop within a GOSUB routine

GOSUB

General Form:	GOSUB line number

Purpose

The GOSUB statement is used to specify a transfer to the first program line of a subroutine. The program line may be any BASIC statement, including a REM statement. The logical end of the subroutine is a RETURN statement which directs execution of the system to the statement following the last executed GOSUB. The RETURN statement must be the last executable statement on a line, but may be followed by non-executable statements as shown below:

```
READY
:120 X = 20:GOSUB 200: PRINT X
:125
 .
 .
 .
:200 REM SUBROUTINE BEGINS
 .
 .
 .
:210 RETURN: REM SUBROUTINE ENDS
```

The GOSUB statement may be used to perform a subroutine within a subroutine (i.e., a nested GOSUB). This statement may not, however, be used to branch a program within a FOR loop where a NEXT statement will be encountered before a RETURN statement is encountered. Use of GOSUB is not permitted in the immediate mode; a GOSUB statement may not be the last statement in a program.

Repeated entries to subroutines without executing a RETURN or RETURN CLEAR should not be made. Failure to execute a RETURN or RETURN CLEAR causes information to be accumulated in a table which eventually causes a table overflow error, (ERR 02).

Example:

```
READY
:10 GOSUB 30
:20 PRINT X: STOP
:30 REM THIS IS A SUBROUTINE          The
:40 --                                subroutine
:50 --
   --
   --
   --
:90 RETURN: REM END OF SUBROUTINE
```

NESTED SUBROUTINES

```
READY
:10 GOSUB 30
:20 READ Q: STOP
:30 REM THIS IS A SUBROUTINE
:40
:50

:70 GOSUB 150
:80 PRINT Q
:90 --
:100 RETURN: REM END OF SUBROUTINE 30                 subroutine
:110
  .
  .
  .
:150 REM THIS IS A NESTED SUBROUTINE
  .
  .                               A nested-subroutine
  .
:200 RETURN: REM END OF NESTED SUBROUTINE
```

Illegal GOSUB Transfer into FOR Loop

```
                 READY
                 :500 GOSUB 750
                   .
                   .
                   .
FOR              :700 FOR I = 20 TO 50
Loop               .
                   .
                 :750 LET A(I) = LOG(12*A) - Z(I)
                 :760 NEXT I          Next statement occurs before RETURN
                 :770 RETURN
```

GOSUB'

```
General Form:          GOSUB' integer [( subroutine argument [ , subroutine argument] . . . ) ]
                 where 0 ≤ integer < 256
                                               ( character string in quotes )
                         subroutine argument = < alphanumeric variable        >
                                               ( expression                   )
```

Purpose

The GOSUB' statement specifies a transfer to a marked subroutine rather than to a particular program line as with the GOSUB statement; a subroutine is marked by a DEFFN' statement (see DEFFN'). When a GOSUB' statement is executed, program execution transfers to the DEFFN' statement having an integer identical to that of the GOSUB' statement (i.e., GOSUB' 6 would transfer execution to the DEFFN' 6 statement). Execution continues until a subroutine RETURN statement is executed. The rules applying to GOSUB usage also apply to the GOSUB' statement. Unlike a normal GOSUB, however, a GOSUB' statement can contain arguments whose values can be passed to variables in the marked subroutine.

The values of the expressions, literal strings, or alphanumeric variables are passed to the variables in the DEFFN' statement (see DEFFN'). Elements of arrays must be explicitly referenced.

Use of GOSUB' is not permitted in Immediate Mode; GOSUB' may not be the last statement in a program.

Repetitive entries to subroutines without executing a RETURN or RETURN CLEAR should not be made. Failure to execute a RETURN or RETURN CLEAR causes return information to accumulate in a table which could eventually cause table overflow error, (ERROR 02).

Example:

```
READY
:100 GOSUB' 7
:150 END
:200 DEFFN' 7 :SELECT PRINT 211 (80)
:210 RETURN
:_
```

Example:

```
READY
:25 GOSUB' 12 ("JOHN", 12.4, 3*X+Y)
:30 END
:100 DEFFN' 12 (A$,B,C(2) )
:110 PRINT A$,B,C(2)
:120 RETURN
:_
```

General Form: GOTO line number

Purpose

This statement transfers execution to the specified line number; execution continues at the specified line.

The GOTO statement can also be used in the immediate mode to permit the user to begin stepping through program execution from a particular line number. The GOTO statement sets the system at the specified line; execution does not take place until the user touches the HALT/STEP key or enters a CONTINUE command.

Example:

```
READY
:10 J=25
:20 K=15
:30 GOTO 70
:40 Z=J+K+L+M
:50 PRINT Z, Z/4
:60 END
:70 L=80
:80 M=16
:90 GOTO 40
:RUN
136          34

END PROGRAM                       } displayed
FREE SPACE = 3841                 }  output
```

HEXPRINT

(NOT ON 2200A)

General Form:

$$\text{HEXPRINT}\left\{\begin{array}{l}\text{alpha variable}\\ \text{alpha array designator}\end{array}\right\}\left[\begin{array}{c},\\ ;\end{array}\left[\begin{array}{l}\text{alpha variable}\\ \text{alpha array designator}\end{array}\right]\right]\ \ldots [\,;\,]$$

Purpose

This statement prints the value of the alpha variable or the values of the alpha array in hexadecimal notation. The printing or display is done on the device currently selected for PRINT operations (see SELECT). Trailing spaces, HEX(20), in the alpha values are printed. Arrays are printed one element after another with no separation characters. The carriage return is printed after the value(s) of each alpha variable (or array) in the argument list, unless the argument is followed by a semi-colon. If the printed value of the argument exceeds one line on the CRT display or printer, it will be continued on the next line or lines. Since the carriage width for PRINT operations can be set to any desired width by the SELECT statement, this could be used to format the output from arguments which are lengthy.

Example:

```
:10  A$ = "ABC"
:20  PRINT "HEX VALUE OF A$=";
:30  HEXPRINT A$
:RUN

HEX VALUE OF A$=41424320202020202020202020202020
```

Examples:

```
:100   HEXPRINT A$, B$(1), STR(C$, 3, 4)
:110   HEXPRINT A$; B$;
:120   HEXPRINT X$( )
```

General Form:	IF END THEN line number

Purpose

This statement is used to sense an end of file (i.e., trailer record) when reading data files. If an end of file (trailer record) has been encountered during the last data file read operation (DATALOAD), a transfer is made to the specified line number. The end-of-file condition is reset by the IF END statement, any subsequent DATALOAD operation, or when program execution is initiated. When a trailer record is read, during a DATALOAD statement, it causes the end-of-file indicator to be set and variables in the DATALOAD argument list to remain unchanged.

Example:

```
READY
:100 DATALOAD A, B, C$
:110 IF END THEN 130
:120 GOTO 100
:130 PRINT A, B, C$
:_
```

In this example, values are loaded continuously from tape until the end-of-file (EOF) is encountered; execution then continues at line 130.

General Form:

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

$$\text{where operand} = \left\{ \begin{array}{l} \text{literal string} \\ \text{alpha or numeric variable} \\ \text{expression} \end{array} \right\}$$

Purpose

The IF statement is a conditional GOTO; it causes the system to go to the line number following THEN, provided a specified condition is met.

If the value of the first item in the IF statement is in the specified relationship to the second item, program execution goes to the line number following THEN. If the specified relationship is not met, the program execution continues with the next statement.

If two alphanumeric values are being compared, the "<" operator is interpreted as "earlier in the collating sequence"; and the ">" operator, as "later in the collating sequence". The ASCII codes (see Appendix B) of the characters in the alpha values determine the collating sequence. For example, the character 1 falls earlier than A in the collating sequence since the ASCII hexcode for 1=31 and the ASCII hexcode for A=41. Trailing blanks (hexcode = 20) are ignored in any comparisons (e.g., "YES" = "YES ") and an error occurs if numeric values are compared with alpha values.

The IF statement cannot be used in the Immediate Mode.

Examples:

```
40 IF A < B THEN 35
50 IF A$ = "YES" THEN 100
60 IF A$=HEX(8082) THEN 200
70 IF X(1) <> 001 THEN 350
80 IF STR(A$,I,3) < B$(1) THEN 500
```

General Form:

```
              ⎧format specification⎫
            % ⎨                    ⎬    [ . . . ]
              ⎩character string    ⎭
character strings cannot contain #'s or colons (:)

                               ⎡+⎤  #[[,] # . . . ]    (integer)
where format specification =   ⎢-⎥  [ # . . . [,] # . . .] . [# . . . ]   (fixed point)
                               ⎣$⎦  [ # . . . [,] # . . .] [.  [ # . . .]] ↑↑↑↑  (exponential)
```

Purpose

This statement provides an image line for formatting output generated by literals and variables in a corresponding PRINTUSING statement. The Image (%) statement contains text characters and format specifications; the formats must be in the same order as the variables in the corresponding PRINTUSING statement. Text characters and formats can be interspersed as needed. The Image statement can be placed anywhere in a program; it should have a corresponding PRINTUSING statement. The Image statement must be the only statement on the program line.

Each format specification must have at least one #; it can begin with any of the allowed characters. A comma (,) can be placed in the integer portion of a format, so long as it is not the first character of a format and precedes any decimal point (.) or up-arrow (↑).

NOTE:
A format specification used for numeric output can contain no more than 16 #'s and, in exponential format must contain ↑'s as its last four characters.

An Image statement with no formats generates ERR 36; one with less than four ↑'s in exponential format generates ERR 38.

Examples:

```
10% CODE NO. = #### COMPOSITION = ####
600% #### UNITS AT $#,###.## PER UNIT
800% +#.##↑↑↑↑
920% ### ##### ###.#
```

There are three types of format specifications as follows:

Type of format	Examples of format				
integer	###	#,###	+###	-###	$####
fixed point	##.##	+#,###.##	-##.##	$##.##	.###
exponential	#.##↑↑↑↑	##,###.##↑↑↑↑		####↑↑↑↑	

Values passed to the Image statement from the corresponding PRINTUSING statement are output according to the format in the Image statement in two forms; there is one form for numeric values and another for alpha values and literals.

Numeric Values

1. The value is output according to the corresponding format as follows:
   integer format: The integer part of the value is output, fractions are truncated and preceding blanks inserted.
   fixed point format: The value is output as a fixed point number; fractions are truncated or filled with zeros and preceding blanks are inserted.
   exponential format: The value is output in floating point exponential format; it is scaled as required by the format and digits are output as integers or fractions as specified by the format. No preceding blanks are inserted. The exponent is output as E ± ee, where ee are the digits of the exponent.
2. If the format begins with a number sign (#) or a decimal point (.), and the value is positive, the value is output according to the format; if the value is negative, a minus (–) precedes the value and the length of the format is increased by one.
3. If the format begins with a plus (+), the sign of the value is placed before the value's first significant digit.
4. If the format begins with a minus (–), a positive value is preceded by a blank and a negative value, by a minus.
5. If the format begins with a dollar sign, $, the value is preceded by a $.
6. If the format contains commas (,) in its integer portion, the commas are output as specified, unless the value has too few significant digits for the commas to be used.
7. If the length of the value being formatted is less than the length of the format (i.e., it is over-formatted), the value is right justified when output. If the length of the integer portion of the value being formatted is greater than the length of the format (i.e., it is under-formatted), the format specification itself is output.

Alpha Values and Literals

1. The value of any alpha scalar, the element of any alpha array or any literal replaces the format in an Image statement character-for-character, from left to right.
2. If the value is shorter than the format, blanks are inserted at the right.
3. If the value is longer than the format, the value is truncated at the right.

Example 1:

```
10 PRINTUSING 20, 1242.3, 73694.23
20% TOTAL SALES= #### VALUE $###,###.##
```

Two fixed point values are output in integer and fixed point form. The first value is truncated; the second value is preceded by a $.

Output:

```
TOTAL SALES= 1242 VALUE $73,694.23
```

Example 2:

```
10 PRINTUSING 20,2. 13E–5, 2. 3E–9
20 % COEFF. =+. ###↑↑↑↑ ERROR=–##↑↑↑↑
```

Two exponential values are output in exponential form. The values are both scaled so that no significant digits are lost with the formats given.

Output:

```
COEFF =+ .213E–04 ERROR= 23E–10
```

Example 3:

```
10 PRINTUSING 20, 2. 13E-5,2. 34E-99
20 % COEFF =+. ###↑↑↑↑ ERROR=-##↑↑↑↑
```

Two exponential values are output in exponential form. The exponent of the second value has exceeded the legal range, so zeros are output.

Output:

```
COEFF =+.213E-04 ERROR= 00E+00
```

Example 4:

```
10 PRINTUSING 20,2. 13E-5, 22. 3412356
20 % COEFF =+.###↑↑↑↑ ERROR=-##.####
```

An exponential value and a fixed point value are output in exponential and fixed point formats. The fixed point value is longer than the corresponding format and is truncated at the right.

Output:

```
COEFF =+.213E-04 ERROR= 22. 3412
```

Example 5:

```
10 PRINTUSING 20,2. 13E-5, 422. 3412356
20 % COEFF =+. ###↑↑↑↑ ERROR=-##.####
```

An exponential value and a fixed point value are output in exponential and fixed point formats. The integer portion of the fixed point value is longer than the corresponding format and the format itself is output.

Output:

```
COEFF =+. 213E-04 ERROR =-##.####
```

Example 6:

```
10 FOR I=1 TO 10
20 B=45
30 A=B↑I
40 PRINTUSING 50,A
50 % ################
60 NEXT I
```

A FOR/NEXT loop illustrates the use of a 16-digit Image statement. At the tenth pass, the value output exceeds the format and the format itself is output.

Output:

```
              45
            2025
           91125
         4100625
       184528125
      8303765625
    373669453120
  16815125391000
  75668064256000
################
```

INIT

(NOT ON 2200A)

General Form:

$$\text{INIT}\left(\begin{Bmatrix}\text{hh}\\ \text{"character"}\\ \text{alpha variable}\end{Bmatrix}\right)\begin{Bmatrix}\text{alpha variable}\\ \text{alpha array designator}\end{Bmatrix}\left[,\begin{Bmatrix}\text{alpha variable}\\ \text{alpha array designator}\end{Bmatrix}\ldots\right]$$

where: h = hexdigit

Purpose

The INIT statement initializes the specified alphanumeric variable(s) and/or array(s). Each character in the second and subsequent variables or arrays is set equal to the character specified inside the parentheses. The character may be represented by two hexdigits, a single character literal or an alphanumeric variable. If an alphanumeric variable is enclosed in the parentheses, the first character of the value of the alphanumeric variable will be used.

The INIT statement is particularly useful when used in conjunction with other byte manipulation and conversion statements. It permits the user to initialize every character of the defined length of an alphanumeric variable to a known value such as zero.

Examples:

```
10  INIT (00) A$, B$( ), C$
20  INIT ("  ") A1$( ), B$( )
30  INIT (FF) X$, STR(B$, 3, 8)
40  INIT (A$) B$( )
```

General Form:	INPUT ["character string",] variable [, variable] . . .

Purpose

This statement allows the user to supply data during the execution of a program already stored in memory. If the user wants to supply the values for A and B while running the program, he enters, for example,

```
:40 INPUT A,B
    or
:40 INPUT "VALUE OF A,B",A,B
```

before the first program line which requires either of these values (A, B). When the system encounters this INPUT statement, it outputs the input message, VALUE OF A, B, followed by a question mark (?) and waits for the user to supply the two numbers. Once the values have been supplied, program execution continues. The question mark and input request message are always printed on the console output device. The device used for inputting data is the console input device unless another device has been specified by using the SELECT INPUT statement (see SELECT).

Each value must be entered in the order in which it is listed in the INPUT statement and values entered must be compatible with variables in the INPUT statement. If several values are entered on a line, they must be separated by commas or entered on separate lines. Several lines may be used to enter the required INPUT data. To include leading blanks or commas as part of an alpha value, enclose the value in double quotes ("); for example, "BOSTON, MASS.".

If there is a system-detected error in the entered data, the value must be reentered, beginning with the erroneous value. The values which precede the error are accepted.

A user may terminate an input sequence without supplying all the required input values by simply entering a carriage return (EXEC) with no other information preceding it on the line. This causes the system to immediately proceed to the next program statement. The INPUT list variables which have not received values remain unchanged.

Note: When inputting alphanumeric data, the literal string need not be enclosed in quotes. However, leading blanks are ignored and commas act as string terminators.

Example 1:

```
:10 INPUT X
:RUN
?12.2 CR/LF
```

Example 2:

```
:20 INPUT "X,Y", X,Y
:RUN
X,Y? 1.1, 2.3 CR/LF
```

Example 3:

```
:20 INPUT "MORE INFORMATION", A$
:30 IF A$="NO" THEN 50
:40 INPUT "ADDRESS", B$
:50 GOTO 20
:RUN
MORE INFORMATION? YES CR/LF
ADDRESS? "BOSTON, MASS" CR/LF
```

Example 4:

```
:10 INPUT "ENTER X",X
:RUN
ENTER X? 1.2734 CR/LF
```

SPECIAL FUNCTION KEYS IN INPUT MODE

Special function keys may be used in conjunction with INPUT. If the special function key has been defined for text entry (see DEFFN') and an INPUT statement is executed, pressing the special function key causes the character string in the DEFFN' statement to be displayed on the CRT. The displayed value is stored in the variable which occurs in the INPUT statement when the EXEC key is touched.

For example:

```
:10 DEFFN' 01 "COLOR T.V."
:20 INPUT A$
:RUN
?
                    Now, pressing special function key '01
                    will cause "COLOR T.V." to appear on the CRT.
? COLOR T.V._
                    ↑ CRT Cursor
```

If the special function key is defined to call a marked subroutine (see DEFFN') and the system is awaiting input, pressing the special function key will cause the specified subroutine to be executed. When the subroutine RETURN is encountered, a branch will be made back to the INPUT statement and the INPUT statement will be executed again. Repeated subroutine entries via special function keys should not be made unless a RETURN or RETURN CLEAR statement is executed; otherwise return information accumulates in a table and eventually causes a table overflow error (ERR 02).

For example

The program illustrated at the top of the next page enters and stores a series of numbers. Upon depressing special function key '02, they are totaled and printed.

```
:10 DIM A(30)
:20 N = 1
:30 INPUT "AMOUNT", A(N)
:40 N = N+1 :GOTO 30
:50 DEFFN' 02
:60 T = 0
:70 FOR I = 1 TO N
:80 T = T+A(I)
:90 NEXT I
:100 PRINT "TOTAL= " ; T
:110 N = 1
:120 RETURN
:[RUN] [EXEC]
AMOUNT? 7   [EXEC]
AMOUNT? 5   [EXEC]
AMOUNT? 11   [EXEC]
AMOUNT? (Depress special function key 2)
TOTAL = 23
AMOUNT?
```

KEYIN

(NOT ON 2200A)

General Form:	KEYIN alpha variable, line number, line number

Purpose

This statement checks if there is a character ready to come in from the input device buffer and, if one is ready, it reads the character into the system. For example, in the case of a keyboard, when a key is pressed, that character is stored in a buffer and the device is set to ready (i.e., a character is ready to come in). The following actions take place depending upon input conditions.

1. NOT READY – execution continues at the next statement.
2. READY WITH CHARACTER – the character is stored as the first character of the specified alphanumeric variable and execution continues at the first KEYIN line number.
3. READY WITH SPECIAL FUNCTION KEY – the hex code representing the special function key (hex 00 to 1F) is stored as the first character of the specified alphanumeric variable and execution continues at the second KEYIN line number.

The device used is that device currently selected for INPUT (Console Input device unless selected otherwise, see SELECT).

The KEYIN statement provides a convenient way to scan several input devices or to receive and edit keyed in information on a character by character basis. KEYIN may not be used in the Immediate Mode.

Example:

```
10 KEYIN A$, 100, 200
20 KEYIN A$(1), 100, 100
30 GOTO 20
40 KEYIN STR(A$,I,1), 100, 200
```

Example:

```
10 DIM A$1
20 KEYIN A$, 30, 100: GOTO 20
30 PRINT A$: GOTO 20
100 PRINT "SPECIAL FUNCTION";: HEXPRINT A$: GOTO 20
```

Line 20 waits for a character to be entered or a special function key to be pressed. When a character is entered it is displayed by line 30. When a special function key is pressed, it is displayed by line 100.

General Form:	LEN (alpha variable)

Purpose

Determines the number of bytes (characters) occupied by the current value of an alpha scalar, an alpha array element or the result of an alpha function. It can be used wherever an expression is permitted.

NOTE:
Trailing blanks are not considered part of alpha variables by the LEN function.

Example:
```
10 A$ = "ABCD"
20 PRINT LEN (A$)
```
These program lines give the value 4 at execution time.

Example:
```
30 X = LEN (A$) + 2
```
Combined with lines 10 and 20 above, this line assigns the value 6 to X at execution time.

Example:
```
110 IF LEN (A$(3)) < 8 THEN 150
```
This line tests the length of the value placed in the third element of the array A$, compares it with 8 and branches to line 150 if the relation is true.

Example:
```
10 A$ = "ABCD"
20 PRINT LEN (STR(A$,2))
```
These lines give the value 15 at execution time. A$ is not explicitly dimensioned so the default value for its length is 16 bytes. The STR function extracts the bytes from A$, starting at the second byte, to its end. The length of such a value is 15.

Example:
```
10 DIM A$64
20 A$ = "ABCD"
30 PRINT LEN (STR(A$,POS(A$=20)))
```
These lines give the value 60 at execution time. The length of the alpha scalar is initially 64; the value of the POS function is first determined, giving the position of the first blank character in A$. The STR function then extracts the number of bytes from the first blank character to the end of the scalar.

NOTE:
The POS function is not available on a System 2200A, nor on a 2200S or WCS/10 without the appropriate option.

```
         [ LET ] numeric variable [ , numeric variable ] . . . = expression
                                                              { alpha variable }
    or   [ LET ] alpha variable [, alpha variable] . . . =    { literal string }
```

Purpose

The LET statement directs the system to evaluate the expression following the equal sign and to assign the result to the variable or variables specified preceding the equal sign. If more than one variable appears before the equal sign, they must be separated by commas.

The word LET is, however, optional. If it is omitted, its purpose is assumed.

An error results if a numeric value is assigned to an alphanumeric variable or if an alphanumeric value is assigned to a numeric value.

Example 1:

```
40 LET X(3), Z, Y=P+15/2+SIN(P-2.0)
```

Example 2:

```
50 LET J = 3
```

Example 3:

```
(Here, LET is assumed)
10 X=A*E-Z*Y
:20 A$ = B$
:30 C$, D$(2) = "ABCDE"
:_
```

Example 4:

```
10 C$ = 'ABCDE'
20 A$ = "123456"
30 D$ = STR(A$,2)
40 E$ = HEX(41)
50 PRINT A$, C$, D$, E$
```

This routine produces the following output at execution time:

```
123456      ABCDE      23456      A
```

Example 4:
A, B, C, X = 1

> **NOTE:**
> *In any System 2200C, S, T and WCS/10, /20 and /30, if the receiving variable is a STR function and the value assigned is shorter than the length of the STR function, the receiving variable is padded with trailing spaces. To eliminate these spaces, put the value in a dummy variable and use the LEN function as illustrated below. On a 2200A and B, the receiver is not padded with trailing spaces.*

Example of routine and output from System 2200A or B:

```
10A$ = "12345"
20 STR (A$,2,3) = "A"
30 PRINT "A$ = "; A$
```

Output:

```
A$ = 1A345
```

Output from the same routine on a System 2200C, S, T or WCS/10, /20 or 30:

```
A$=1A   5
```

To obtain output without trailing spaces, the following program can do the job:

```
10 A$="12345"
20 V$="A"
30 STR(A$, 2, LEN(V$))=V$
40 PRINT "A$=";A$
```

Output:

```
A$=1A345
```

General Form:	NEXT numeric scalar variable

Purpose

The NEXT statement defines the end of a FOR/NEXT loop; it must contain the same index variable (a numeric scalar) as the corresponding FOR statement. In Immediate Mode, the NEXT and FOR statements must be in the same line.

When a FOR/NEXT loop is encountered, the index variable takes the value initially assigned. When the NEXT statement is executed, the STEP value is added to the value of the index. (If no STEP value is given, +1 is used.) If the result is within the range specified in the FOR statement, the result (index +STEP) is assigned to the index variable and execution continues at the statement following the FOR statement. If the result is outside the range specified in the FOR statement, the index variable is unaltered and execution passes to the statement following the NEXT statement.

Example:

```
10 M=1: N=20: J=3
20 FOR I=M TO N STEP J
30 PRINT "M=";M, "N="; N, "J="; J, "I="; I
40 IF I=7 THEN 60
50 GOTO 70
60 J=4: N=15
70 NEXT I:PRINT "INDEX=";I
```

Output:

```
M= 1          N= 20          J= 3          I=1
M=1           N= 20          J= 3          I= 4
M= 1          N= 20          J= 3          I= 7
M= 1          N= 15          J= 4          I= 10
M= 1          N= 15          J= 4          I= 13
M= 1          N= 15          J= 4          I= 16
M= 1          N= 15          J= 4          I= 19
INDEX= 19
```

Examples:

```
30 FOR M=2 TO N-1 STEP 30: J(M)=I(M)↑2
40 NEXT M
50 FOR X=8 TO 16 STEP 4 ┐
60 FOR A = 2 TO 6 STEP 2┐│
65 LET B(A,X) = B(X,A)  ││──►Nested Loops
70 NEXT A               ┘│
80 NEXT X                ┘
```

General Form:	NUM (alpha variable)

Purpose

The NUM function determines the number of sequential ASCII characters in the specified alphanumeric variable that represents a legal BASIC number. A numeric character is defined to be one of the following: digits 0 through 9, and special characters E, ., +, –, space. Numeric characters are counted starting with the first character of the specified variable or STR function. The count is ended either by the occurrence of a non-numeric character, or when the sequence of numeric characters fails to conform to standard BASIC number format. Leading and trailing spaces are included in the count. Thus, NUM can be used to verify that an alphanumeric value is a legitimate BASIC representation of a numeric value, or to determine the length of a numeric portion of an alphanumeric value. Note: the BASIC representation of a number cannot have more than 13 mantissa digits. NUM can be used wherever numeric functions are normally used. NUM is particularly useful in applications where it is desirable to numerically validate input data under program control.

Examples:

```
10  A$ = "+24.37#JK"
20  X = NUM(A$)
```

NOTE: X = 6 since there are six numeric characters before the first non-numeric character, #.

```
10  A$ = "98.7+53.6"
20  X = NUM(A$)
```

NOTE: X = 4 since the sequence of numeric characters fails to conform to standard BASIC number format when the '+' character is encountered.

```
10  INPUT A$
20  IF NUM(A$)=16 THEN 50
30  PRINT "NON-NUMERIC, ENTER AGAIN"
40  GOTO 10
50  CONVERT A$ TO X
60  PRINT "X="; X
:RUN
? 123A5
NON-NUMERIC, ENTER AGAIN
? 12345
X=12345
```

NOTE: The program illustrates how numeric information can be entered as a character string, numerically validated, and then converted to an internal number. In this example the variable A$ receives a keyed in value (alphanumeric ASCII characters). If the value represents a legal BASIC number, NUM(A$) equals 16, the number of characters in the string variable A$.

NOTE:
Do not use this function in a statement with PACK, UNPACK, CONVERT, SAVEDA, $..., MAT... if your system is a 2200B, C, S or WCS/10.

ON
(NOT ON 2200A)

General Form:	ON expression $\left\{\begin{matrix}\text{GOSUB}\\\text{GOTO}\end{matrix}\right\}$ line number [,line number] . . .

Purpose

The ON statement is a computed or conditional GOTO or GOSUB statement (see GOTO, GOSUB). Transfer is made to the Ith line specified in the list of line numbers if the truncated integer value of the expression is I. For example, if I = 2,

```
ON I GOTO 100, 200, 300
```

would cause a transfer to be made to line 200 in the program. If I is less than 1 or greater than the number of line numbers in the statement, no transfer is made; that is, the next sequential statement is executed. The ON statement may not be used in immediate mode.

Example:

```
10 ON I GOTO 10, 15, 100, 900
20 ON 3*J-1 GOSUB 100, 200, 300, 400
```

ON ERROR GO TO

(NOT ON 2200A OR B)*

General Form:

```
ON ERROR alpha variable, alpha variable GOTO line number
```

Purpose

Bypasses the normal System 2200 error display when an execution error occurs and branches to a specified statement number. The error code and line number at which the error occurred are stored in the variables provided.

The ON ERROR GOTO statement permits a BASIC program to undertake error recovery procedures when errors occur. When an error occurs during execution of a program, and if the ON ERROR GOTO statement exists somewhere in the current program, a branch is made to the line number specified in the ON ERROR GOTO statement. ON ERROR GOTO detects errors only during the execution of a program. When a program is RUN, the system scans the program, checking validity of line number references and setting up space for variables before beginning execution. Errors occurring during this scan are not processed by ON ERROR GOTO; the normal error message is displayed.

When an ON ERROR GOTO branch is made to the error recovery routine, the first alpha variable specified in the ON ERROR GOTO statement receives the two ASCII characters of the error code, and the second alpha variable receives the (four) ASCII characters of the line number.

If an error occurs in the Middle of a FOR/NEXT loop or subroutine and an ON ERROR GOTO statement is executed, the loop or subroutine information is cleared from the internal tables. Thus, transfer back into the loop or subroutine is illegal and an error (#25 or #26) results when the subsequent NEXT or RETURN statement is executed. If more than one ON ERROR GOTO statement occurs in a program, the one with the lowest line number is executed; a program should thus contain only one such statement.

When doing program overlays, the variables in the ON ERROR GOTO statement must be common (defined in a COM statement), since all noncommon variables are cleared during the overlay. If an error occurs during a program overlay, the line number is meaningless (since part of the program has been cleared).

Examples:

```
:100 ON ERROR E$, N$ GOTO 900
:900 REM ERROR RECOVERY ROUTINE
```

```
10 INPUT E$
20 ON ERROR E$, N$ GOTO 900
30 GOTO 1000
900 PRINT "E$=", E$, "N$=", :STOP "YOU INPUT THE WRONG VALUE"
1000 PRINT "CONTINUE INPUTTING E$", E$
1100 GOTO 10
```

If an illegal value (e.g., "A) is entered, the program branches to line 900 where the value of the error message (40) and the line number where the error occurred (10) are displayed with a message.

*Available on 2200S only with Advanced Programming Statements.

PACK

(NOT ON 2200A)

```
General Form:
                                          ┌ numeric variable              ┌ , numeric variable              ┐     ┐
              ┌ alpha variable          ┐ │                               │                                 │     │
              └ alpha array designator  ┘ FROM ┤ numeric array designator │ , numeric array designator      │ ... ┤
                                          └ expression                    └ , expression                    ┘     ┘

    where:    image = [±] [#...] [.] [#...] [↑↑↑↑]
              0 < number of #'s < 14
```

Purpose

The PACK statement packs numeric values into an alphanumeric variable or array, reducing the storage requirements for large amounts of numeric data where only a few significant digits are required. The specified numeric values are formatted into packed decimal form (two digits per byte) according to the format specified by the image, and stored sequentially into the specified alphanumeric variable or array. Arrays are filled from the beginning of the first array element until all numeric data has been stored. An entire numeric array can be packed by specifying the array with a numeric array designator (e.g., N()). An error will result if the alphanumeric variable or array is not large enough to store all the numeric values to be packed.

The image is composed of # characters to signify digits and, optionally, +, –, ., and ↑ characters to specify sign, decimal point position, and exponential format. The image can be classified into two general formats:

Format	Example
Fixed Point	##.##
Exponential	#.##↑↑↑↑

Numeric values are packed according to the following rules:

1. Two digits are packed per byte. A digit is stored for each # in the image.
2. If a sign (+ or –) is specified, it occupies 1/2 byte and 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. When unpacking the data (see UNPACK), the decimal point position is specified in the image.
5. The packed numeric value occupies a whole number of bytes. For example, the image ### indicates that 1-1/2 bytes are required for storage; however, 2 bytes will be used.
6. If a fixed point image is used, the value is packed as a fixed point number, truncating or extending with zeros any fraction and inserting leading zeros for nonsignificant integer digits according to the image specification.
7. If an exponential image is used, the value is packed as a floating point number. The value is scaled as specified by the image (without leading zeros). The exponent occupies one byte. .

Examples of storage requirements:

####	= 2 bytes
###	= 2 bytes
+##.###	= 3 bytes
+#.##↑↑↑↑	= 3 bytes

Examples of Syntax:

```
10  PACK(####)A$ FROM X
20  PACK(##.##)A$ FROM X, Y, Z
30  PACK(+#.##)STR(A$, 4, 2) FROM N(1)
40  PACK (+#.##↑↑↑↑)A$( ) FROM N( )
50  PACK (####.##) A$( ) FROM X, Y, N( ), M( )
60  PACK (###.#) A1$(I) FROM X( )
```

General Form:

$$\text{POS}\left(\text{alpha variable}\begin{Bmatrix} < \\ <= \\ = \\ >= \\ > \\ <> \end{Bmatrix}\begin{Bmatrix}\text{"character"} \\ \text{hh}\end{Bmatrix}\right)$$

where h = hexdigit

Purpose

The POS function finds the position of the first character in the specified alphanumeric value that is <, ⩽, =, ⩾, >, or <> the character specified following the relation operator. The character to be compared can be specified either by enclosing the character in quotes or by representing the character by two hex digits. If no character in the alphanumeric value satisfies the specified condition, POS = 0. POS can be used wherever numeric expressions normally are used.

Examples:

```
10  X = POS (A$ = "$")
20  PRINT POS(STR(A$, 4, 5)=0D)
30  IF POS (A$ < "A") < 16 THEN 100
```

NOTE:
Do not place this function in a statement with PACK, UNPACK, CONVERT, SAVE DA, $..., MAT... in any System 2200B or C.

PRINT

General Form:			PRINT [t] [print element] [t] [t print element] . . . [t] . . .
	where	t =	a comma or a semicolon
	print element	=	an expression, TAB (expression), an alpha or numeric variable, an array element, literal string.

Purpose

The PRINT statement causes the values of the listed variables, expressions, or literal strings to be printed on the output device currently selected for PRINT (see SELECT).

Printing may be done in zoned format which is signaled by a comma, or packed format, which is signaled by a semicolon separating each print element.

ZONED FORMAT: PRINT print element [, print element] ... [,]

The output line is divided into as many zones of 16 characters as possible; the four CRT zones are columns 0-15, 16-31, 32-47, and 48-63.

A comma signals that the next print element is to be printed starting in the next print zone, or if the final print zone is filled then the first print zone of the next line. For example

```
READY
:10 X=214.230 :Y=3564: Z=-.2379
:20 PRINT X, Y, Z
:RUN

214.23       3564        -.2379
```

```
10 X = 10 : Y = -140
20 PRINT , , X, Y
```

$$\underbrace{10}_{\text{Zone 3}} \qquad \underbrace{-140}_{\text{Zone 4}}$$

PACKED FORMAT: PRINT print element [; print element] ... [;]

A semicolon signals that the next print element is to be printed not by zones but by columns. For example, the statement

```
READY
:10 X=2 :Y=-3.4
:20 PRINT "X=";X;"Y=";Y
:RUN
```

in the following output:

```
X=  2 Y=-3.4
```

NOTE:
A positive value is preceded by a space when output; a negative value, by a minus sign.

A PRINT statement can contain both comma and semicolon element separators. Each separator explicitly determines the amount of space between elements. If a numeric value is output, it is followed by a single space. Alpha variables or literals are not followed by spaces.

```
READY
:10X=2 :Y=3 :Z=-4.2 :A=10 :B=-20 :A$="Q"
:20 PRINT "X=";X,"Y=";Y,"Z=";Z;A;B;A$;A$
:RUN
```

results in the following printout:

$\underbrace{\texttt{X= 2}}_{\text{Zone 1}} \qquad \underbrace{\texttt{Y= 3}}_{\text{Zone 2}} \qquad \underbrace{\texttt{Z =-4.2 \ \ 10 -20 QQ}}_{\text{Zone 3}}$

The end of a PRINT line signals a new line for output, unless the last symbol is a comma or semi-colon. A comma signals that the next print element encountered in the program is to be printed in the next zone of the current line. A semicolon signals that the next print element is to be printed in the next available space. For example, the statements

```
:10 PRINT "X=";
:20 PRINT 3.2970,
:30 PRINT "Y=";64
```

cause the following printout:

```
X= 3.297        Y= 64
```

A PRINT statement with no PRINT element advances the CRT cursor one line. If the number of characters to be output in a given PRINT element exceeds the line length (the default value or as specified in a SELECT statement, see SELECT), a carriage return/line feed is executed before the characters are output.

Values of expressions are printed in one of two forms depending on the value:

Exponential Form:	SM.MMMMMMMME±XX△	$\lvert VALUE \rvert \leqslant 10^{-1}$ or $> 10^{13}$
Fixed Point Form:	SZZZZZZ.FFFFFFF△	$10^{-1} > \lvert VALUE \rvert \leqslant 10^{13}$

where M = mantissa digits Z = integer digits
X = exponent digits S = minus sign if value < 0, or blank if value ⩾ 0.
F = fractional digits △ = space

In fixed point form, the decimal point is inserted at the proper position or omitted if the value is an integer. Leading and trailing zeros are omitted.

The following are examples of the printing of variables in the two forms:

Exponential	2.34762145E-09
	-1.64721000E+22
Fixed Point:	23.47954890123
	-.6374
	0
	-421

PRINTUSING

```
General Form:            PRINTUSING line number  [, print element] [t print element] . . . [;]
     where line number = Line number of the corresponding
                         Image (%) statement.
                         ⎧ expression                  ⎫
         print element = ⎨ alpha or numeric variable   ⎬
                         ⎩ literal string in double quotes ⎭
                     t = comma or semicolon
```

Purpose

The PRINTUSING statement permits numeric and alphanumeric values to be output as specified by formatting in a referenced Image (%) statement on the output device currently selected for PRINT (see SELECT). Formats are described under the Image statement (see Image (%)).

PRINTUSING operates in conjunction with a referenced IMAGE statement. Print elements in the PRINTUSING statement are edited into the print line as directed by the IMAGE statement. Each print element is edited, in the order in the PRINTUSING statement, into a corresponding format in the IMAGE statement. The IMAGE statement provides both alphanumeric text to be printed between the inserted print elements, and the format specifications for the inserted print element. The format for each numerical print element is composed of # characters to specify digits and optionally +, –, ., ↑, , and $ characters to specify sign, decimal point, exponent and edit characters. If the number of print elements exceeds the number of formats in the IMAGE statement, a carriage return/line-feed occurs, and the IMAGE statement is reused from the beginning for the remaining print elements. The carriage return/line-feed may be suppressed by replacing the comma, delimiting the print elements with a semicolon. A carriage return/line-feed normally occurs at the end of the execution of a PRINTUSING statement. This carriage return/line-feed can also be suppressed by placing a semicolon at the end of the PRINTUSING statement. PRINTUSING may not be used in the immediate mode.

Example 1:

```
          :10 X=2.3 : Y=27.123
          :20 PRINTUSING 30, X, Y
          :30 % ANGLE - ##.## LENGTH = +##.#
          :RUN
(output)  ANGLE =   2.30 LENGTH = +27.1
```

Example 2:

```
          :10 X=1: Y=2: Z=3
          :20 PRINTUSING 30, X, Y, Z
          :30 % #.#
          :RUN
(output)  1.0
          2.0
          3.0
```

Example 3:

```
:10 X=1: Y=2: Z=3
:20 PRINTUSING 30, X; Y; Z
:30 % #.#
```

(output)

```
1.0  2.0  3.0
```

Example 4:

```
:100 PRINTUSING 200
:200 % PROFIT AND LOSS STATEMENT
:RUN
```

(output)

```
PROFIT AND LOSS STATEMENT
```

Example 5:

```
:100 PRINTUSING 200, A$, T
:200 % SALESMAN ######## TOTAL SALES $##,###.##
:RUN
```

(output)

```
SALESMAN J. SMITH          TOTAL SALES $9,237.51
```

Example 6:

```
10 X=2.3: Y=27.123
20 PRINTUSING 30, X, Y, "ALPHA ZONE"
30 % ANGLE -#####.###### LENGTH=+##.#
```

(output)

```
ANGLE           2.300000  LENGTH=+27.1
ANGLE ALPHA ZONE          LENGTH=
```

Example 7:

```
10 X=2.3: Y=27.123
20 PRINTUSING 30,X,Y,"ALPHA ZONE";X+3
30 % ANGLE ##.##↑↑↑↑↑↑↑↑↑↑↑  LENGTH=+##.#
```

(output)

```
ANGLE 23.00E-01↑↑↑↑↑↑   LENGTH=+27.1
ANGLE ALPHA ZONE        LENGTH= +5.3
```

PRINT TAB

FUNCTION

General Form:	TAB (expression)

Purpose

This function permits the user to specify tabulated formatting. For example, TAB (17) would cause the typewriter carrier or the CRT cursor to move to column 17.

Positions are numbered 0 to 64 on the CRT, 0 to 155 on a typewriter, and 0 to 79 or 0 to 131 on a printer. The value of the expression in the TAB function is computed, and the integer part is taken. This becomes the column number to which the cursor, carrier or print head is moved. If the position has been passed, the TAB is ignored. If the value of the expression is greater than the maximum allowable, the output carrier moves to the beginning of the next line. Values of TAB expressions greater than 255 are illegal. For example:

```
READY
:10 FOR I=1 TO 5   \
:20 PRINT TAB(I);I  > causes the following output:
:30 NEXT I         /
:RUN
1
 2
  3
   4
    5
```

In the System 2200, a line length of 64 characters is the default value. If more than 64 characters are output without a carriage return, an automatic carriage return is generated. This line length can be changed to any value ($0 \leqslant$ value < 256) by a SELECT statement (see SELECT).

General Form:	READ variable [,variable] . . .

Purpose

A READ statement causes the next available elements in a DATA list (values listed in DATA statements in the program) to be assigned sequentially to the variables in the READ list. This process continues until all variables in the READ list have received values or until the elements in the DATA list have been used up. The variable list can include both numeric and alphanumeric variable names. However, each variable must reference the corresponding type of data or an error will result.

The READ statements and DATA statements must be used together. If a READ statement is referenced beyond the limit of values in a DATA statement, the system looks for another DATA statement in statement number sequence. If there are no more DATA statements in the program, an error message is written and the program is terminated. DATA statements may not be used in the immediate mode.

The RESTORE statement can be used to reset the DATA list pointer, thus allowing values in a DATA list to be re-used (see RESTORE).

NOTE:

DATA statements may be entered any place in the program as long as they provide values in the correct order for the READ statements.

Example:

```
:100 READ A, B, C
:200 DATA 4, 315, -3.98

:100 READ A$, N, B1$ (3)
:200 DATA "ABCDE", 27, "XYZ"

:100 FOR I = 1 TO 10
:110 READ A(I)
 120 NEXT I
     . . . . .
 200 DATA 7.2, 4.5, 6.921, 8, 4
 210 DATA 11.2, 9.1, 6.4, 8.52, 27
```

General Form:		REM [text string]
	where text string =	any characters or blanks (except colons; colons indicate the end of the statement)

Purpose

The REM statement is used at the discretion of the programmer to insert comments or explanatory remarks in his program. When the system encounters a REM statement, it ignores the remainder of the line.

Examples:

```
20 REM SUBROUTINE
210 REM FACTOR
220 REM THE NUMBER MUST BE LESS THAN 1
300 REM
```

General Form:	RESTORE [expression]
	where 1≤ value of expression < 256

Purpose

The RESTORE statement allows the repetitive use of DATA statement values by READ statements. When RESTORE is encountered, the system returns to the nth DATA value, where n is the truncated value of the expression if one is included in the RESTORE statement; otherwise, it is assumed to be the first DATA statement. Then, when a subsequent READ statement occurs, the data is read and used, beginning with the nth DATA element.

Example:

100 RESTORE

This statement causes the next READ statement to begin with the first data element.

The statement **100 RESTORE 11**

causes the next READ statement to begin with the 11th data element.

The statement **100 RESTORE X↑2+7**

causes the expression X↑2+7 to be evaluated and truncated to an integer. The next READ statement begins with the corresponding data element.

RETURN

General Form:	RETURN

Purpose

The RETURN statement is used in a subroutine to return processing of the program to the statement following the last executed GOSUB or GOSUB' statement.

If entry was made to a marked subroutine via a special function key on the keyboard, the RETURN statement will terminate program execution and return control back to the keyboard, or to an interrupted INPUT statement.

Repetative entries to subroutines without executing a RETURN should not be done. Failure to return from these entries causes return information to be accumulated in a table which will eventually cause the table overflow error (ERR 02) (also see RETURN CLEAR).

Example:

```
10 GOSUB 30
20 PRINT X :STOP
30 REM     THIS IS A SUBROUTINE
40  -
50  -
 -  -
 -  -
90 RETURN :REM    END OF SUBROUTINE

10 GOSUB' 03 (A,B$)
20 END
100 DEFFN' 03 (X,N$)
110 PRINTUSING 111, X, N$
111 % COST = $#,###,###.## CODE =   ####
120 RETURN
```

RETURN CLEAR
(NOT ON 2200A OR B)

General Form:

```
RETURN CLEAR
```

Purpose

Clears subroutine return-address information, generated by the last executed subroutine call, from memory.

The RETURN CLEAR statement is a dummy RETURN statement. With the RETURN CLEAR statement, subroutine return address information from the last previously executed subroutine call is removed from the internal tables; the program branches to the statement following the RETURN CLEAR.

The RETURN CLEAR statement is used to avoid table overflow when a program exits from a subroutine without executing a RETURN. This is particularly useful when using the Special Function keys to start program execution (when either in Console Input mode or using an INPUT statement). When a Special Function key is used in this manner, a subroutine branch is made to the appropriate DEFFN' statement to begin execution.

A subsequently executed RETURN statement either causes the system to return to the Console Input mode or causes the INPUT statement to be repeated automatically. However, the user may wish to start and continue a program without returning when a Special Function key is depressed. In this case, the RETURN CLEAR statement should be used to exit from the DEFFN' subroutine.

Examples:

```
100 DEFFN' 15: RETURN CLEAR
200 RETURN CLEAR
```

NOTE:

If a program repeatedly exits from a subroutine without executing a RETURN or RETURN CLEAR statement, error 02 results.

When a program is loaded into memory it must be initially executed by a RUN command, thereafter, it can be restarted at any point via special function keys.

ROTATE

(NOT ON 2200A)

General Form:	ROTATE (alpha variable, d)	
where:	d	= digit from 1 - 7

Purpose

This statement rotates the bits of each character in the value of the specified alphanumeric variable to the left from one to seven places; the high order bits replace the low order bits. All characters in the value are operated on including trailing spaces. (Note: for most alphanumeric operation in the System 2200, if an alphanumeric variable receives a value with a length less than the maximum length of the variable, the remaining characters are all set equal to spaces. The trailing spaces normally are not considered to be part of the value.)

Example:

```
if A$ = HEX(0123FE), ROTATE (A$, 4)
sets A$ = HEX (1032EF)
```

Part of an alphanumeric variable can be operated on by using the STR function to specify a portion of the variable. For example,

```
ROTATE (STR(A$, 2, 3), 3)
```

Examples:

```
10  ROTATE(A$, 4)
20  ROTATE(STR(A$,I), 7)
```

General Form:	STOP ["character string"]

Purpose

The STOP statement terminates program execution. A program can have several STOP statements in it.

When a STOP statement is encountered, the word STOP (followed by the optional specified character string) appears on the Console Output device.

To continue program execution at the statement immediately following the STOP statement, a CONTINUE command must be entered.

Example:

```
100 STOP
100 STOP "MOUNT DATA CASSETTE"
```

TRACE

General Form:	TRACE [OFF]

Purpose

The TRACE statement provides for the tracing of the execution of a BASIC program. TRACE mode is turned on in a program when a TRACE statement is executed and turned off when a TRACE OFF statement is executed. TRACE also is turned off when a CLEAR command is entered, the system is RESET, or the system is turned on. To trace an entire program, TRACE may be turned on by entering a TRACE immediate mode statement prior to execution, and similarly turned off by entering an immediate mode TRACE OFF after execution. When the TRACE mode is on, output on the Console Output device is produced when:

1. Any program variable receives a new value during execution (LET, READ, FOR statements, etc.).

   format: variable = received value

2. A program transfer is made to another sequence of statements (GOTO, GOSUB, IF, NEXT).

   format: TRANSFER TO 'line number'

3. A BASIC function is executed.

   format: function name

Example 1:

```
5 DIM Z(5)
10 A, B, C=2
30 LET X, Y, Z(5)=A+SIN(B)/C
A=
B=
C=2
X=
Y=
Z ( ) = 2.454648713413
```

Example 2:

```
:40 READ A, B, C, D
:50 DATA 9.4, 64.27, 137492.1E8, 99.4
A = 9.4
B = 64.27
C = 1.37492100E+13
D = 99.4
```

Example 3:

```
:100 GOTO 200
```

produces

```
TRANSFER TO 200
```

Example 4:

```
30 GOSUB 10
```

produces

```
TRANSFER TO 10
```

Example 5:

```
:5 DIM X(3)
:10 FOR I=1 TO 3
:15 PRINT X(I);
:20 NEXT I
```

produces

```
I=1
0
I=2
TRANSFER TO 15
0
I=3
TRANSFER TO 15
0
I=>  (end-of-loop indicator)
```

Example 6:

```
:10 A$=HEX (414243)
```

produces

```
A$=HEX(414243
```

Example 7:

```
:10 STR(A$,I,4)= "ABCD"
```

produces

```
STR(
A$=ABCD
```

Example 8:

```
10 AND (A$, 00)
```

produces

```
A$=HEX (00000000000000000000000000000000
```

Example 9:

```
10 DIM A (4)
20 A(1)= 24.2: A(2)= 25.36: A(3)= 48.001: A(4)= 14.759
:100 FOR I = 1 TO 4
:110 TRACE
:120 X = X+A(I)
:130 TRACE OFF
:140 NEXT I
RUN
```

produces

```
X = 24.2
X = 49.56
X = 97.561
X = 112.32
```

UNPACK

(NOT ON 2200A)

General Form:

$$\text{UNPACK (image)}\left\{\begin{array}{l}\text{alpha array designator}\\ \text{alpha variable}\end{array}\right\}\ \text{TO}\left\{\begin{array}{l}\text{numeric array designator}\\ \text{numeric variable}\end{array}\right.\ \left[\begin{array}{l}\text{, numeric array designator}\\ \text{, numeric variable}\end{array}\right]\ \ldots\Bigg\}$$

where: image = [±] [#. . .] [.] [#. . .] [↑↑↑↑]
0 < number of #'s < 14

Purpose

The UNPACK statement is used to unpack numeric data that was packed by a PACK statement. Starting at the beginning of the specified alphanumeric variable or array, packed numeric data is unpacked and converted to internal floating point values, and stored into the specified numeric variables or arrays. The format of the packed data is specified by the image (see PACK); thus, the same image that was used to pack the data should be used in the UNPACK statement. An error results if more numeric values are attempted to be unpacked than can exist in the alphanumeric variable or array.

Examples of Syntax:

```
10   UNPACK (####)A$ TO X, Y, Z
20   UNPACK (+#.##) STR(A$, 4, 2) TO X
30   UNPACK (+#.##↑↑↑↑) A$( ) TO N( )
40   UNPACK (######) A$( ) TO X, Y, N( ), M( )
```

Example:

```
10 X=24: DIM A$3
20 PACK (####) A$ FROM X
30 PRINT X
40 HEXPRINT A$
50 UNPACK (####) A$ TO Y
60 PRINT A$,Y
```

output

```
 24
002420
$                 24
```

General Form:	VAL $\left(\begin{Bmatrix}\text{alpha variable}\\ \text{literal string}\end{Bmatrix}\right)$

Purpose

This function converts the binary value of the first character of the specified alphanumeric value to a floating point number. The VAL function is the inverse of the BIN statement. VAL can be used wherever numeric functions normally are used (also see VAL decimal equivalents in Appendix B).

VAL is particularly useful for code conversion and table lookups, since the converted number then can be used either as a subscript to retrieve an equivalent code or data from an array, or with the RESTORE statement to retrieve codes or information from DATA statements.

Examples of Syntax:

```
10  X = VAL(A$)
20  PRINT VAL("A")
30  IF VAL(STR(A$, 3, 1) ) < 80 THEN 100
40  Z = VAL(A$)*10 - Y
```

NOTE:
Do not use this function on the same program line as PACK, UNPACK, CONVERT, SAVEDA, $ . . ., MAT . . . on any 2200B or C.

Example:

A = hex 41

NOTE: $41_{16} = (4 \times 16^1) + (1 \times 16^0)$

$41_{16} = 64 + 1$

$41_{16} = 65$

first digit $\times 16^1$ + 2nd digit $\times 16^0$

Section VIII Peripheral Commands and Statements

Section VIII Peripheral Commands and Statements

INTRODUCTION

Commands and statements which are used to operate peripherals (I/O devices) are described in this section of this manual for easy reference. Only the minimum syntax and explanatory information is provided here; for further information, see the peripheral manual for the unit in question. All disk statements and explanatory information are contained in the System 2200 Disk Memory Reference Manual. For device addresses, see the Device Address Guide or Device Address appendix in this manual.

The statements and commands used to operate peripherals have been grouped by type of unit.

- tape cassette drive
- card readers (both mark sense and punched card)
- punched tape reader
- plotter
- Teletype®

Keyboards, CRT's and printing output devices are operated by INPUT, PRINT, PRINTUSING and HEXPRINT statements described in Section VII, and by MAT INPUT and MAT PRINT statements described in the Matrix Statements manual. All other units such as disk drives, the nine-track tape unit, the Digitizer, Interface Controllers and the Telecommunications Controller are operated by statements and commands described in their respective manuals.

TAPE CASSETTE DRIVES

Tape cassette drives provide low-cost sequential storage for Wang systems. All material is dual-recorded to prevent loss of data or programs and to permit certain kinds of recovery from such loss. The system automatically formats both data and programs for the user. A single (150 ft) tape cassette can contain up to 76,000 bytes of information. Any program to be stored on a cassette can be named for later recall; its "name" is recorded on the cassette.

BASIC logic to support tape cassette drives is available on all systems (except for BT statements, which are not available on a 2200A and must be obtained for a 2200S and WCS/10 with Option 22).

CASSETTE STATEMENT

General Form:

$$\text{BACKSPACE} \begin{bmatrix} \#f, \\ \\ /xyy, \end{bmatrix} \begin{Bmatrix} \text{BEG} \\ n \\ nF \end{Bmatrix}$$

where #f	=	File number assigned to cassette drive by SELECT statement (#f = #1, #2, #3, #4, #5, or #6) (see SELECT).
xyy	=	Device address of cassette drive. If neither of the above is specified, the default device address (the device address currently assigned to TAPE [see SELECT]) is used.
BEG	=	Backspace to beginning of file. (After header record.)
n	=	Backspace n logical records.
nF	=	Backspace n files (Note, if n=1 backspace to beginning of current file before header record.)
n	=	Expression (the integer portion of the value of the expression is used and must always be ⩾ 1).

Purpose

The BACKSPACE statement allows the user to reposition the cassette in the indicated unit backwards to the start of any program or data file, or backward a specified number of logical records within a data file. The 'BEG' parameter positions the tape at the beginning of the current file immediately after the header record. The 'n' parameter is for data files only; it allows the user to backspace the tape over n logical records to the start of any desired logical record. The 'nF' parameter backspaces the tape n files; the tape is positioned before the header record.

Example:

```
100 BACKSPACE /10A, BEG
220 BACKSPACE #2, 4F
150 BACKSPACE (5-3*X)
```

DATALOAD

CASSETTE STATEMENT

General Form:

$$\text{DATALOAD} \begin{bmatrix} \#f, \\ /xyy, \end{bmatrix} \left\{ \begin{array}{l} \text{"name"} \\ \text{argument [,argument]...} \end{array} \right\}$$

#f	= File number assigned to cassette drive by SELECT statement (f is an integer from 1-6)
xyy	= Device address of device to load from. If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.
"name"	= The name of the data file to be searched. "name" is from 1 to 8 characters.
argument	= {alphanumeric variable / numeric variable / alpha or numeric array designator}

Purpose

The DATALOAD statement reads a logical record from the designated tape and assigns the data values read to the variables and/or arrays in the argument list, sequentially. Arrays are filled row by row. If the variable list is not complete, another logical record is read. Data in the logical record, not used by the DATA-LOAD statement, is ignored. If the end of file (trailer record) is encountered while executing a DATA LOAD statement, the tape remains positioned at the end of file trailer record and the values of remaining variables in the argument list remain at their current values. An IF END THEN statement will then cause a valid transfer.

The "name" parameter permits a data file to be searched out. Upon execution of a DATALOAD "name" statement, the tape is positioned just after the header record of the specified file.

Example:

```
DATALOAD "PROGRAM1"
DATALOAD A, B, C(10)
DATALOAD #1, A, B( ), C$
DATALOAD /10B, A, B, X1 , STR(A$, 3, 5)
```

DATALOAD BT

CASSETTE STATEMENT

General Form:

```
                                      ┌ #f,  ┐
     DATALOAD BT [(N=expression)]     │      │  alpha array designator
                                      └ /xyy,┘
```

where:

expression = 100 or 256 (size of block to read)

#f = File number assigned to cassette drive by SELECT statement (f is an integer from 1 to 6).

xyy = Device address of device to load from.

If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.

Purpose

This statement reads the next block of 100 or 256 bytes from cassette tape and stores the information in the specified alphanumeric array. If the N parameter is not specified, the block is assumed to be 256 bytes. An error will result if the array is not large enough to hold the entire block to be read.

The DATALOAD BT statement permits 2200 programs to be read as data. Thus, tape duplication, program conversion, and program packing programs can be written. In addition, Wang 1200 cassettes which have a block size of 100 characters can be read.

Example:

```
DATALOAD BT A$( )
DATALOAD BT (N=100) A$( )
DATALOAD BT /10B, B1$( )
DATALOAD BT (N=100) #5, Q$( )
```

NOTE:

This statement is not available on a 2200A; it is available on a 2200S or WCS/10 only if Advanced Programming Statements are available.

DATARESAVE

CASSETTE STATEMENT

General Form:

```
DATA RESAVE [#f,  ] { OPEN "name"                         }
            [/xyy,] { argument        [, argument] . . .  }
```

where: #f	=	File number assigned to cassette drive by SELECT statement (f is an integer from 1 to 6).
xyy	=	Device address of device to save on.
		If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) will be used.
OPEN	=	Rewrite a data file header record with the name "name". Name is from 1 to 8 characters.
argument	=	{ literal string / alpha or numeric variable / expression / array designator }

Purpose

The DATA RESAVE statement allows the user to rewrite (i.e. update) any record, including the header record, of an existing data file. Rewriting the header record permits the user to rename a file. Rewriting is done in place on the original tape.

Rewriting (updating) a logical record within a file involves three steps:

1. Locating the beginning of the file with a DATALOAD "name" statement (see DATALOAD).
2. Locating the particular logical record to be updated using the DATALOAD, SKIP or BACKSPACE statements.
3. Re-recording the logical record using the DATARESAVE statement.

When executing the DATARESAVE statement, the tape must be positioned just before the record to be updated. The DATARESAVE statement must be used for updating; if an update is performed using a DATASAVE statement, there is no assurance that the new record will be written in the proper place – extraneous information may be left over from the old record. The user must be sure that the number of physical records in the logical record created by the DATARESAVE statement is the same as the number of physical records in the logical record being updated. This situation is assured if the 'argument list' in the DATARESAVE statement is identical to the 'argument list' in the original DATASAVE statement.

NOTE:

Extensive use of the DATARESAVE command with the same cassette (e.g., file maintenance) is not *advised. Using the DATARESAVE command repeatedly with the same tape is not recommended since after a period of time, normal tape wear, loss of magnetic flux, and accumulation of dust particles can cause the data integrity to be less reliable.*

Example:

```
DATARESAVE /10B, A, B$, C
DATARESAVE #1, OPEN "DATAFILE"
DATARESAVE A$( )
DATARESAVE STR(A$, 5, 1), HEX (010203), "WANG LABS."
DATARESAVE R*SIN(X)
```

CASSETTE STATEMENT

General Form:

```
DATASAVE [ #f,  ] { OPEN "name"                   }
         [ /xyy,] { END                           }
                  { argument [ , argument] . . .  }
```

where:

#f	=	File number assigned to cassette drive by SELECT statement (f is an integer from 1 to 6).
xyy	=	Device Address of cassette drive on which data is written.
		If neither of the above is used, the default device address (the device address currently assigned to TAPE (see SELECT)) will be used.
OPEN	=	Write a data file header record with the name "name". The name is from 1 to 8 characters.
END	=	Write a data file trailer record.
argument	=	{ literal string / alpha or numeric variable / expression / array designator }

Purpose

The DATASAVE statement causes the values of variables, expressions, and array elements to be written sequentially onto the specified tape. Arrays are written row by row. Each DATASAVE statement produces one logical record. Each numeric value occupies 9 characters in a record; each literal occupies the number of characters in the value +1; each value of an alpha variable string occupies the maximum defined length of the variable +1.

The OPEN and END parameters are used to write header and trailer records at the beginning and end of a data file. However, data files can be created without the need for header and trailer records. If a single data file is to be written on a cassette, it can be done simply by using one or more DATASAVE statements with argument lists. The data in the file can be retrieved using DATALOAD statements with argument lists. If more than one data file is to be written on a cassette, it is common practice to place a header record at the start of each file and a trailer record at the end of each file. In this way the user can search out any file by using the assigned 'name' in the header record (see DATALOAD) and can test for the end of a file using the trailer record (see IF END THEN). The header and trailer records can also be used in backspacing over and skipping records and files (see BACKSPACE, SKIP).

Example:

```
DATASAVE A, B, C, D(4,2)
DATASAVE #2, A, B, C( )
DATASAVE /10A, A$, B, C, D( )
DATASAVE OPEN "PROGRAM 1"
DATASAVE #5, END
DATASAVE STR(A$,3,5),.HEX(0102), "WANG LABS."
DATASAVE Y*SIN(R)
```

CASSETTE STATEMENT

General Form:

```
DATASAVE BT [R] [([N=expression] [,] [H] ) ]  [#f,  ]  alpha array designator
                                              [/xyy,]
```

where:

R	=	Resave
expression	=	100 or 256 (size of block to record)
H	=	Record header block (0's timing mark)
#f	=	File number assigned to cassette drive by SELECT statement (f is an integer from 1 to 6).
xyy	=	Device Address of cassette drive on which data is written.

If neither of the above is used, the default device address (the device address currently assigned to TAPE (see SELECT)) will be used.

NOTE:
A comma must separate the N and H parameters if both are specified.

Purpose

This statement records a block of data (100 or 256 bytes) on cassette tape with no control information. If the array is greater than 100 (or 256) bytes, the first 100 (or 256) bytes of the array are recorded. If the array is smaller than the specified block size, the block is filled with unpredictable characters. If the 'N' parameter is not specified, the block is assumed to be 256 bytes.

If a header record is being recorded, the 'H' parameter is used; this causes a special timing mark to be written on the cassette indicating that this block is a header block. This timing mark is used by the system when backspacing files.

The 'R' parameter is used to rewrite a block on cassette using DATASAVE BT. Before the record is written, the tape is automatically backspaced one block.

The DATASAVE BT statement permits tapes containing a number of Program and/or Data Files to be copied and BASIC programs to be generated by conversion programs.

Example:

```
DATASAVE BT A$( )
DATASAVE BT (N = 100) A1$( )
DATASAVE BT (N=100,H) /10C, A$( )
DATASAVE BT (H) #6, Q$( )
```

NOTE:
This statement is not available on a 2200A; it is available on a 2200S or WCS/10 only if Advanced Programming Statements are available.

CASSETTE COMMAND

General Form:

```
LOAD [#f,  ]  ["name"]
     [/xyy,]
```

where	#f	=	File number assigned to cassette drive by SELECT statement (f = an integer from 1 to 6)
	xyy	=	Device address of cassette drive to load from. If neither of the above is specified, the default device address (the device address currently assigned to TAPE, see SELECT) is used.
	"name"	=	Is the name assigned to the program on tape. "name" is from one to eight characters.

Purpose

When the LOAD command is entered, the specified program on the selected tape will be appended to the current program in memory. If no program name is specified, the next program file on the selected tape is loaded. This command permits an additional program to be loaded and appended to a program currently in the 2200, or if entered after a CLEAR command, the entry of a new program.

LOAD can also be used as a program statement (see LOAD — cassette statement).

Example:

```
LOAD
LOAD "LINREGR"
LOAD#1, "PROGRAM1"
LOAD/10B
LOAD#4
```

CASSETTE STATEMENT

General Form:	LOAD [#f, /xyy,] ["name"] [line number 1] [, line number 2]		
where:	#f	=	file number assigned to cassette device by SELECT statement. (f is an integer from 1 to 6)
	xyy	=	device address of cassette drive. If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.
	"name"	=	Is the name of the program to be searched and loaded; it is from 1 to 8 characters. Searching is always forward.
	line number 1	=	The line number of the first line to be deleted from a currently loaded program prior to loading the new program. After loading, execution continues at the line whose number is equal to line number 1. An error results if there is no line number = 'line number 1' in the new program.
	line number 2	=	The line number of the last line to be deleted from the program currently in memory, before loading the new program.

Purpose

This is a BASIC program statement which in effect produces an automatic combination of the following:

STOP	(stop current program execution)
CLEAR P	[line number 1 [, line number 2]] (delete current program text)
CLEAR N	(remove noncommon variables only)
LOAD	["name"] (load new program)
RUN	[line number 1] (run new program)

If only 'line number 1' is specified, the remainder of the current program is deleted starting with that line number. If no line numbers are specified, the entire current program is deleted, and the newly loaded program is executed from the lowest line number.

This permits segmented jobs to be run automatically without normal user intervention. Common variables are passed between program segments. LOAD must be the last statement on a statement line. The LOAD statement must not be within a FOR/NEXT Loop or subroutine; an error results when the NEXT or RETURN statement is encountered.

In the immediate execution mode, LOAD is interpreted as a command (see LOAD command).

Example:

```
100 LOAD
100 LOAD #2
100 LOAD "SAM"
100 LOAD /10A
100 LOAD /10B, "PROG#7", 500
100 LOAD #2, "SAM" 400, 1000
```

CASSETTE STATEMENT

General Form:

$$\text{REWIND} \begin{bmatrix} \#f \\ /xyy \end{bmatrix}$$

where:	#f	=	File number assigned to cassette drive by SELECT statement (f is integer from 1 to 6).
	xyy	=	Device address of cassette drive.
			If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.

Purpose

The REWIND statement causes the cassette in the specified tape drive to be rewound.

Example:

```
REWIND
100 SELECT #2 10B
110 REWIND #2
30 REWIND
40 REWIND /10C
```

CASSETTE COMMAND

General Form:

```
SAVE  [#f,   ]  [P]  ["name"] [line number  [, line number] ]
      [/xyy, ]
```

where	#f	= File number to which device address is assigned (#1 to #6).
	xyy	= Device address of cassette drive on which output is recorded. If neither of the above is specified, the default device address (the device address currently assigned to TAPE, see SELECT) is used.
	P	= Sets the protection bit on the program file to be saved.
	"name"	= Is the name assigned to the program on tape. "name" is from one to eight characters.
	1st 'line number'	= Starting line number to be saved.
	2nd 'line number'	= Ending line number to be saved.

Purpose

The SAVE command causes BASIC programs (or portions of BASIC programs) to be written onto the selected tape. The program may be named by using the "name" parameter so the user can address this program file in subsequent LOAD commands.

If no line numbers are specified, the entire user program text is written onto the specified tape. SAVE with one line number causes all user program lines from the indicated line through the highest numbered program line to be written onto tape. If two line numbers are entered, all text from the first through the second line number, inclusive, is written.

The 'P' parameter permits the user to protect saved programs. That is, if a program that has been saved by a SAVE P command is loaded, it may not be listed or saved again. Note, in order to list or save ANY program after a protected program has been loaded, the user must enter a CLEAR command (with no parameters) or MASTER INITIALIZE the system, (i.e., turn power off and then on).

SAVE is a command and may not be used within a BASIC program.

Examples:

```
SAVE
SAVE #3
SAVE/10B
SAVE "MAT INV"
SAVE/10B, 100, 200
SAVE #5, "SUBR1" 400, 500
```

CASSETTE STATEMENT

General Form:

$$\text{SKIP} \begin{bmatrix} \#f, \\ /xyy, \end{bmatrix} \begin{Bmatrix} \text{END} \\ n \\ n\text{F} \end{Bmatrix}$$

where	#f	=	File number assigned to cassette drive by SELECT statement (f is an integer from 1 to 6).
	xyy	=	Device address of cassette drive. If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.
	END	=	Skip to the end of current data file.
	n	=	Skip n logical data records.
	nF	=	Skip n files.
	n	=	expression (the integer portion of the value of the expression is used, must be $\geqslant 1$)

Purpose

The SKIP statement allows the user to skip over any number of program or data files, or any number of data records. The END parameter is used with data files only. It causes the indicated cassette tape to skip to the end of the current data file; the tape is positioned before the trailer record. The n parameter is also used exclusively with data files. It causes the indicated cassette tape to skip n logical data records. If the trailer record is encountered, the tape backspaces so that it is positioned before the trailer record. The nF parameter causes the tape to skip n complete program or data files; the tape is positioned at the beginning of the next file.

Example:

```
350 SKIP END
270 SKIP #1, 2F
SKIP 10
SKIP/10B, (X+2)F
```

CARD READERS (Mark Sense and Punched Card)

Card readers can be attached to any Wang system, but their use on a 2200A is limited and on a 2200S or WCS/10 is greatly improved with the addition of Option 22. Cards to be used in Wang card readers can either be prepared manually or on an IBM keypunch.

MARK SENSE CARD READER

Procedure:

(1) Select the Mark Sense Card Reader for Console Input by entering:

SELECT CI 517 $\boxed{\text{EXEC}}$

(2) Read one or more mark sense cards containing commands or program text line. If the lines contain a statement number, they will be loaded and saved, if they do not, they will be executed immediately and not saved. Each card will be displayed on the CRT as it is entered. If a read error occurs, reread the card.

(3) When loading is complete, reselect the keyboard for Console Input by loading a card which contains:

SELECT CI 001

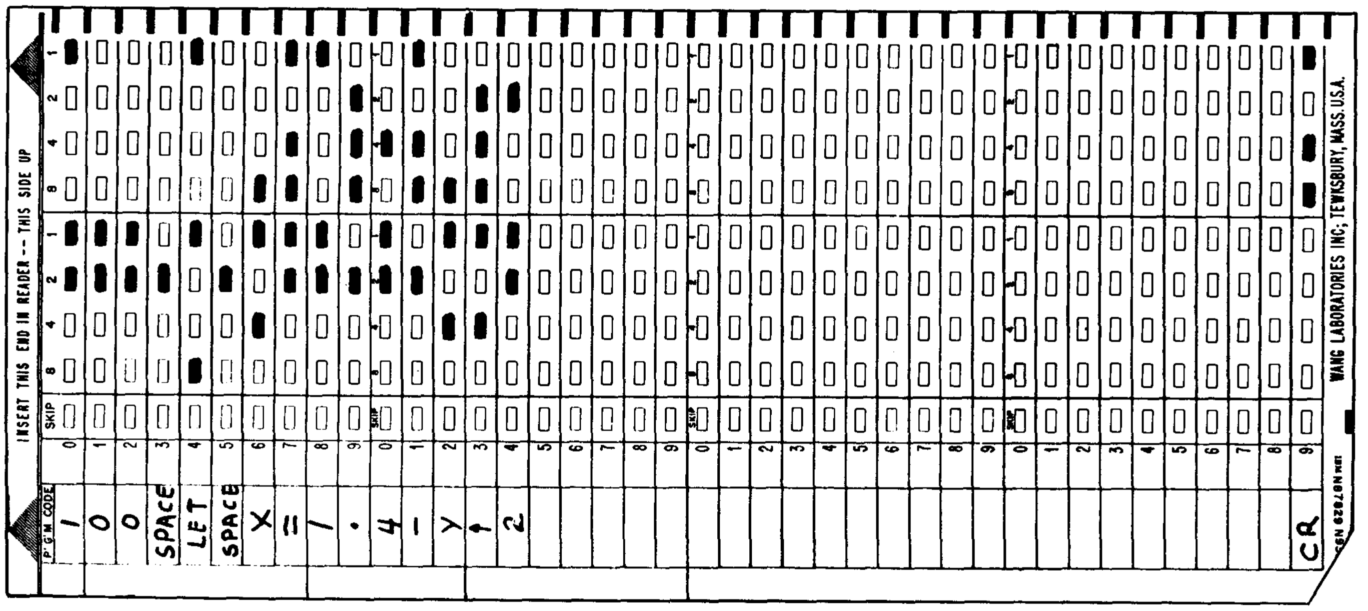
This procedure allows programs to be loaded from the 2214 Mark Sense Card Reader and displayed, or commands and immediate execution mode statements to be read and executed.

Card Format:

Program and command lines are marked on each card in ASCII character code format. Special single column codes (TEXT ATOMS, see Table at the end of this section) can be marked for BASIC statement verbs, functions, etc. Unmarked columns are ignored. A carriage return character should be marked at the end of each program line in the last column on the card. Multi-statement lines separated by colons are permissible. Program lines may overlap from 1 card to the next; the carriage return must only appear on the last card of the program line. If the SKIP position is marked in any column, this column is ignored.

Example:

100 LET X = 1.4 – Y↑2


INSERT THIS END IN READER -- THIS SIDE UP
SKIP
P'G'M CODE
1
0
0
SPACE
LET
SPACE
X
=
–
.
4
–
Y
↑
2
CR
WANG LABORATORIES INC; TEWKSBURY, MASS. U.S.A.

DATALOAD

(NOT ON 2200A)

MARK SENSE CARD READER STATEMENT

General Form:

DATALOAD $\begin{bmatrix}\#f, \\ /517,\end{bmatrix}$ argument [, argument] . . .

where: #f	=	logical file number assigned to unit by SELECT statement (f is integer from 1 to 6).
xyy	=	device address of card reader (517). If neither of the above is specified the default device address for TAPE is used (see SELECT).
argument	=	$\left\{\begin{array}{l}\text{variable} \\ \text{array designator}\end{array}\right\}$

Purpose

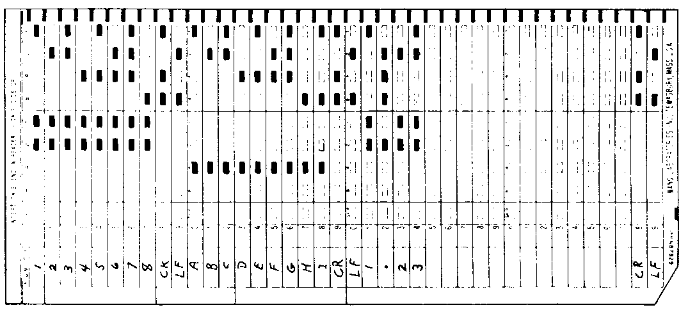
This statement reads values from a mark sense card reader and sequentially assigns those values to the variables in the argument list. A maximum of 40 characters per card can be entered. The values are marked on the card in ASCII code. Blank (unmarked) columns are ignored. Each value must be followed by a CR (carriage return) and LF (line feed) characters. The carriage return and line feed characters for the last value on the card should always be marked in the last two columns of the card. Alphanumeric or numeric values may be assigned to alphanumeric variables; values assigned to numeric variables must be legitimate BASIC numbers. Arrays are filled row by row.

Numeric and alphanumeric values are marked in ASCII in any legal BASIC form (i.e., 4.2, −732.71, 21.2+E07). Quotes are not required with alpha values. Leading space characters (HEX (20)) of alphanumeric values will be ignored. All values, whether numeric or alphanumeric, must be separated by a carriage return and a line feed.

Values are successively read from one or more cards until all arguments are satisfied or until the end-of-file is encountered. For each card read, a CR and LF character must be the last two characters on the card (i.e., occupy the last two columns on the card). End of file is indicated by marking an X-OFF character on a card, followed by a carriage return and line feed. When an end-of-file is encountered, the remaining variables in the list are left with their current values; an IF END THEN statement then causes a valid transfer. If the SKIP position is marked in any card column, that column will be ignored. If a read error occurs which produces an illegal number format, an error message will be displayed, and program execution terminates. The program can be restarted at the DATALOAD statement and all cards reread.

Examples:

```
SELECT TAPE 517
DATALOAD X, Y, A$, B$
DATALOAD #3, N( ), A$, B$
DATALOAD /517, A1$( ), X, A$
```



MARK SENSE/PUNCHED CARD READER STATEMENT

General Form:

$$\text{DATALOAD}\ \begin{bmatrix}\#f,\\ /628,\end{bmatrix}\ \text{argument [, argument] ...}$$

where	/628	= The device address which designates the card reader as the device from which data is to be read, and also determines the type of code conversion routine which is to be performed, and the data format to be expected. Address 628 causes the reader to expect discrete data values, and to perform an automatic Hollerith-to-ASCII conversion for each data character read.
	#f	= A file number to which the device address has been assigned in a SELECT statement ('f' is an integer from 1 to 6). If neither a device address nor a file number is specified, the address of the default TAPE device (normally device address 10A) is used.
	argument	$= \begin{Bmatrix}\text{variable}\\ \text{array designator}\end{Bmatrix}$

Purpose

The DATALOAD statement with a device address of 628 initiates the reading of discrete data values in Hollerith code from one or more data cards, converts each character to ASCII, and assigns the values read sequentially to receiving variables in the DATALOAD argument list. (Arrays are filled row by row.) Multiple values on a single card must be separated by commas. If the argument list is not filled by a single card, additional cards are read until all receiving variables are filled. Unread data on the last card is lost. Both alphanumeric and numeric values may be stored in alphanumeric variables, but only legitimate BASIC numbers can be stored in numeric variables (otherwise, an error results and program execution terminates).

Examples:

```
DATALOAD /628 A$, N( )
DATALOAD A, B, C, N$
```

DATALOAD BT

(NOT ON 2200A)

PUNCHED CARD READER STATEMENT

General Form:

DATALOAD BT (N=82) $\begin{bmatrix}\#f, \\ /629,\end{bmatrix}$ alpha array designator

where:	(N=82)	= The number of characters to be received for each card read. For Hollerith card images (address 629), a total of 82 characters are received for each card (80 data characters and two control characters). If the card has fewer than 80 columns, the reader automatically "pads" the remaining unread characters up to 80 with HEX(FF) characters.
	/629	= The device address of the card reader which also determines the type of code conversion which is to be performed and the data format which is to be expected. Address 629 initiates the reading of a complete 80-character card image, and causes an automatic Hollerith-to-ASCII conversion of all data read.
	#f	= A file number assigned to unit in a SELECT statement (f is an integer from 1 to 6).

If neither a device address nor a file number is specified, the address of the default TAPE device is used.

Purpose

The DATALOAD BT statement with a device address of 629 initiates the reading and conversion of a complete 80-character Hollerith card image. Each column is translated from Hollerith code into its ASCII equivalent, and stored in the receiving alphanumeric array. In this mode, the system always expects to receive exactly 80 data characters for each card read (blank columns on a card are read as ASCII space characters). A card which contains timing marks may have fewer than 80 columns, however. In this case, the card reader itself generates HEX(FF) characters for all unread characters up to 80, so that exactly 80 data characters are transmitted. In addition, the card reader generates a LENGTH code and an ERROR code, and transmits them as the 81st and 82nd characters for each card read. The receiving alpha array must therefore be dimensioned to hold at least 82 characters, and the number of characters read in every case must be 82 (N=82). If an entire card or any portion of a card cannot be read (due to a card jam or other reader malfunction), 80 HEX(FF) codes are transmitted, and the error code identifies the source of the difficulty.

Example:

```
50 DIM A$(3)40
500 DATALOAD BT (N=82)/629,A$( )
```

The receiving array of three forty-byte elements segregates 80 bytes of data in A$(1) and A$(2) and the two control bytes in A(3).

DATALOAD BT

(NOT ON 2200A)

MARK SENSE CARD READER STATEMENT

General Form:

```
                   ┌┌                              ┌hh            ┐            ┌hh            ┐   ┐┐┌#f,  ┐┌alpha variable         ┐
DATALOAD BT ││([N=expression) [,] [L= ┤               ├] [,] [S= ┤               ├ ]│││      ││                        │
                   └└                              └alpha variable┘            └alpha variable┘   ┘┘└/xyy,┘└alpha array designator ┘
```

where:

N	=	Number of characters to read (generally 40).
L	=	Leader code character (ignored when reading until a different character is encountered). If an alpha variable is specified, the first 8-bits are used. If L is not specified, no leader code is assumed (optional).
S	=	Stop character (optional). If an alpha variable is specified, the first cahracter is used.
hh	=	Hexdigits.
#f	=	File number assigned to unit by SELECT statement (#1 to #6).
xyy	=	Device address of card reader. If neither of the last two above parameters is specified, the default device address for TAPE is used (see SELECT).

NOTE:
Commas must separate the N, L, and S parameters when more than one is specified.

Purpose

This statement allows 8-bit characters in any code format to be read from a mark sense card (up to 40 characters) and stores the characters read in the alpha variable or alpha array designated. The card is read and characters stored until the specified alpha variable or array is filled or until the specified number of characters are read, or until the specified STOP character is read.

The 'L' parameter specifies the leader code on the card; when a card is read, leader code is ignored (i.e., all characters equal to the specified leader code character are ignored until a character is read that is not equal to the leader code character).

This statement is generally used when specially coded information, which does not conform to a specific character code format such as ASCII, must be read from a mark sense card. The data is read into alphanumeric variables or alphanumeric arrays; from there it can be converted and processed. Data manipulation and conversion features available in the 2200B are particularly useful for this.

Reading can be terminated for each card by specifying the number of characters to be read (N parameter), or termination character code (S parameter), or both. The recommended procedure is specifying N = 40, since there are 40 columns on each mark sense card. If termination does not occur with the last character of the card, another mark sense card operation should not be requested for at least 20 milliseconds times the number of remaining characters on the card, since some of these remaining characters on the card may be read if another read operation is initiated rapidly.

If the SKIP position is marked in any column, that column is ignored.

If a device is not specified the device currently selected for TAPE will be used. This should previously be selected to 517.

Example:

```
DATALOAD BT (N = 40) /517, A$
SELECT TAPE 517
DATALOAD BT (N = 40) A$( )
SELECT #1 517
DATALOAD BT (N = 40, L = FF, S = 99) #1, B$
```

NOTE:
This Statement is available on a 2200S or WCS/10 only with Advanced Programming Statements.

DATASAVE BT

PUNCHED CARD READER STATEMENT

General Form: DATASAVE BT $\begin{bmatrix}\text{/42E,}\\ \text{\#f,}\end{bmatrix}$ alpha variable

where:

/42E	=	The special card reader device address which specifies Hollerith Look-Ahead operations. Address '42E' causes the card reader to feed in the next card from the input hopper, convert the data from Hollerith to ASCII, and hold the converted data in the card reader output buffer awaiting transmission to the system.
#f	=	A file number to which address 42E has been assigned in a SELECT statement ('f' must be an integer from 1 to 6). If neither a device address nor a file number is specified, the address of the device currently assigned to TAPE device is used.
alpha variable	=	A "dummy" alphanumeric variable included to satisfy general format requirements for the DATASAVE BT statement, but not used in the Look-Ahead operation. (Note that the dummy alpha variable is most efficiently dimensioned to the minimum length of one byte.)

Purpose

The DATASAVE BT statement with a device address of 42E initiates the reading of one card into the card reader buffer, and converts the data from Hollerith to ASCII. Illegal characters are translated as ASCII '!' characters. LENGTH and ERROR codes are also generated, and can be obtained if a Hollerith card image (DATALOAD BT, address 629) is subsequently read. The Look-Ahead operation in effect constitutes the first stage of a reading operation in certain card reader modes. The data cannot actually be transmitted from the card reader buffer into memory, however, until a DATALOAD, address 628, or DATALOAD BT, address 629, statement is executed. There are no timing restrictions governing when a DATALOAD or DATALOAD BT statement may be executed following a Look-Ahead operation.

Example:

```
10 DIM F$1
100 DATALOAD /628, A$, B$, N
200 DATASAVE BT /42E, F$      (initiates reading next card)
```

MARK SENSE CARD READER STATEMENT

General Form:	INPUT ["character string",] variable [,variable] . . .

Purpose

This statement allows the user to supply data from the Mark Sense Card Reader during execution of a program already in memory. The Mark Sense Card Reader is specified with a SELECT statement as the INPUT device and cards containing the data are read instead of keying in the data from the keyboard. To reset the keyboard as the input device, another SELECT statement is executed. For example, a program sequence which allows a user to enter values for the variables A and B via a Mark Sense Card is shown below:

```
30 SELECT INPUT 517
40 INPUT "ENTER VALUES OF A, B", A, B
50 SELECT INPUT 001
```

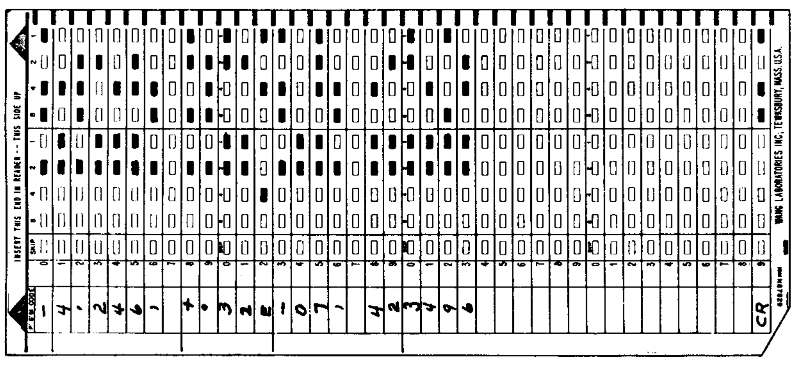
Line 30 selects the card reader as the INPUT device. The system then displays the input request messaoe ENTER VALUE OF A, B?, and waits for the values to be entered. A mark sense card containing the values can then be read. As the card is read, the information is displayed on the CRT, just as in keyboard entry. When the values have been received and assigned to the variables A and B, statement 50 is executed and INPUT operations are selected back to the keyboard.

Each value must be entered on the card or cards in the order in which variables are listed in the INPUT statement. If more than one value is entered on a card, they must be separated by commas. A carriage return character must be marked in the last column of the card. Several cards may be used to enter the required input data. If the SKIP position in any column is marked, that column is ignored.

If there is a system detected error in the entered data, an error message is displayed and the value must be re-entered beginning with the erroneous value. The values which precede the error are accepted. A user may terminate any INPUT statement sequence without supplying all required input values by using a card containing only a carriage return character. This would cause the system to proceed to the next program statement. The variables which have not received data will remain unchanged.

Card Format:

Data values are marked on the card in ASCII code. A carriage return character, HEX (0D), must be marked in the last column of the card (column 40). If more than one value is entered on a card, they must be separated by commas. Numeric data is marked in free-form (i.e., 4.2, –7.24 E+05, 2714.132). Space characters and unmarked columns are ignored. When marking alphanumeric data, the literal string need not be enclosed in quotes. However, leading blanks are ignored and commas act as string terminators. If leading blanks or commas are to be included, enclose the string in double quotes. Space characters must be marked as ASCII space codes, HEX(20). Unmarked columns are ignored. If the SKIP position in any column is marked, that column is ignored.


INSERT THIS END IN READER -- THIS SIDE UP
SKIP
CR
WANG LABORATORIES INC., TEWKSBURY, MASS. U.S.A.

Values on card: –4.246, +.32E–07, 423496

INPUT

MARK SENSE/PUNCHED CARD READER STATEMENT

General Form:	INPUT ["character string",] variable [,variable] . . .

Purpose

Once the card reader has been selected for INPUT operations with a SELECT INPUT 62B statement, the reader functions like a keyboard, reading one or more data cards in response to each INPUT request from the controlling program. As each input request is executed, the system displays a question mark ('?'), preceded by the optional character string. Data values are read from cards and sequentially assigned to the receiving variables in the INPUT argument list. Both numeric and alphanumeric values can be stored in alphanumeric variables. However, only legitimate BASIC numbers in free-format may be read into numeric variables. Otherwise, an ERROR 29 (Illegal Data Format) is generated, and the erroneous value is skipped. Each value may be marked or punched on a separate card, or multiple values may be placed on a single card, provided the values are separated by commas. If the argument list is not satisfied by a single card, additional cards are read until all receiving variables in the argument list have been assigned values. Note, however, that each time the INPUT statement is executed, it automatically begins reading with the next card in the input hopper, even if there are remaining unread data values on the previous card. It is not therefore possible to use two or more INPUT statements to read several data values from a single card. Unread data values on the last card read by an INPUT statement are lost.

Examples:

```
30 SELECT INPUT 62B
40 INPUT A, B
```

PUNCHED CARD READER STATEMENT

General Form:

$$\text{LOAD} \begin{bmatrix} \text{/62B,} \\ \text{\#f,} \end{bmatrix} \text{[line number 1 , line number 2]}$$

where: /62B	=	The device address which designates the card reader as the device from which programs are to be loaded, and also determines the type of code conversion which is to be performed. Address 62B causes program text to be converted from Hollerith into ASCII.
#f	=	A file number to which the device address 62B has been assigned in a SELECT statement ('f' is an integer from 1 to 6). If neither a device address nor a file number is specified, the address of the device currently selected for TAPE is used. Address 62B can be selected with a SELECT TAPE 62B statement.
line number 1	=	The line number of the first line of program text to be cleared from memory prior to loading in the new program, and the first line to be executed in the overlaid program.
line number 2	=	The line number of the last line of program text to be cleared from memory prior to loading the new program.

Purpose

The LOAD statement with address 62B initiates the reading and conversion of BASIC programs from Hollerith cards. The LOAD statement must be executed on a numbered statement line (otherwise, it is interpreted as a LOAD command). When the LOAD statement is executed, it produces an automatic combination of the following operations:

STOP	Stop current program execution.
CLEAR P	[line number 1] [line number 2] Remove program text.
CLEAR N	Remove non-common variables.
LOAD	Load new program.
RUN	[line number 1] Run new program.

Examples:

```
10 LOAD /62B, 100, 200
500 LOAD /62B
```

MARK SENSE/PUNCHED CARD READER COMMAND

General Form:

```
LOAD [/62B]
     [#f  ]
```

where:

/62B = The device address which designates the card reader as the device from which programs are to be loaded, and also determines the type of code conversion to be performed. Address 62B causes program text to be converted from Hollerith to ASCII.

#f = A file number to which the device address 62B has been assigned in a SELECT statement ('f' is an integer from 1 to 6).

If neither a device address nor a file number is specified, the address of the Console Tape device (device address 10A) is used. Address 62B can be designated as the TAPE address with a SELECT TAPE 62B statement.

Purpose

The LOAD command with a device address of 62B initiates the reading of BASIC program cards, and automatically converts program text from Hollerith to ASCII. Newly-loaded program text is appended to the current program in memory, with new program lines which have the same line numbers as existing lines replacing the old lines in memory. Otherwise, existing program text in memory is unaffected by the LOAD operation. For example, if the old program in memory has line numbers 10,20,30, etc., and the newly loaded program has line numbers 15,25,35, etc., the resultant program in memory following the LOAD is numbered, 10,15,20,25,30, etc. Lines which contain syntax errors are loaded and displayed with an appropriate error code. The LOAD operation is not terminated by syntax errors, but the program cannot be run until all syntax errors are corrected. The last card in the program deck must be an END card (with an 'E' in column 80); otherwise, the system continues attempting to load program lines.

Loading a Hollerith Program Deck (LOAD Command, Address 62B)

```
CLEAR    EXEC
LOAD/62B    EXEC
```

Old program text and variables are cleared from memory, and a new program is loaded from Hollerith cards. After the new program is loaded, the operator must enter RUN, EXEC to run the program. The program must be terminated by an 'E' in the last (80th) column of the last program card, or by the inclusion of an END card as the last card in the deck. If a termination card is not encountered, the system hangs up when the hopper is empty.

PUNCHED TAPE READER

The punched tape reader provides the facility for Wang systems to read punched tapes in both forward and reverse directions. Reading is done optically. The punched tape reader can accommodate both standard ASCII and non-standard punched tape sizes and codes.

This unit cannot be run on a 2200A, and Option 22 must be available to use it on a 2200S or WCS/10.

DATALOAD

(NOT ON 2200A)

PUNCHED TAPE READER STATEMENT

General Form: DATALOAD $\begin{bmatrix}\#f, \\ /xyy,\end{bmatrix}$ argument [, argument] . . .

where #f = Logical file number assigned to unit by SELECT statement (f is integer from 1 to 6).

xyy = Device address of punched tape reader.

If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.

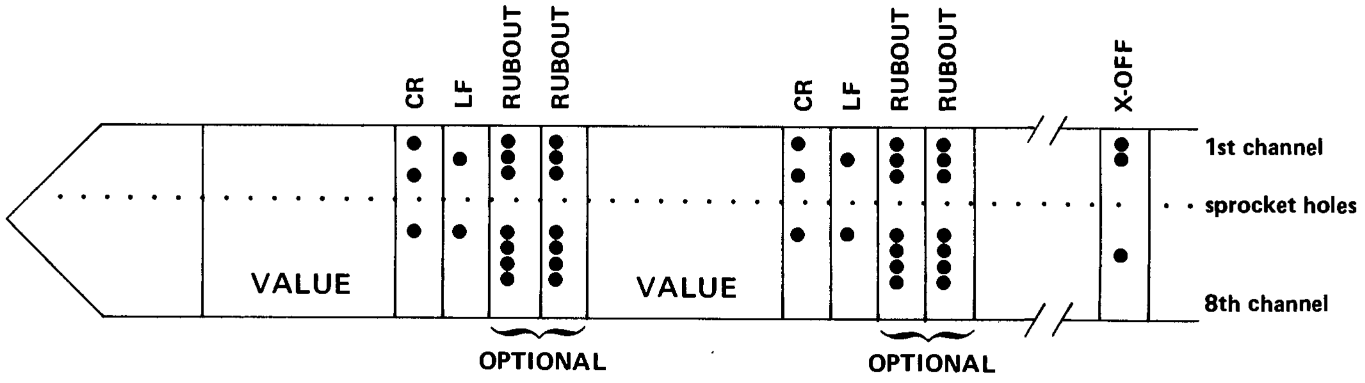
argument = $\left\{\begin{array}{l}\text{variable} \\ \text{array designator}\end{array}\right\}$

Purpose

This statement reads values from paper tape and sequentially assigns those values to the variables in the argument list. Numeric values may be assigned to alphanumeric variables; values assigned to numeric variables must be legitimate BASIC numbers. Arrays are filled row by row.

Values are successively read from the tape until all variables in the list are satisfied or until the end-of-file is encountered (i.e., an X-OFF character is read). When an end-of-file is encountered, the remaining variables in the list are left with their current values; an IF END THEN statement will then cause a transfer to the specified line number.

To be read, the paper tape must conform to the following format:



Values are punched in ASCII character code and are separated by CR LF RUBOUT RUBOUT; the rubouts are, however, optional. DATALOAD reads only the first seven channels of the tape; the 8th bit is always read as 0. Nonpunched frames and RUBOUTS are ignored when reading the tape.

Paper tapes punched on a Teletype via DATASAVE statements conform to this format. To read tape not in this format, use the DATALOAD BT statement.

Example:

```
DATALOAD X, Y, A$, B$
DATALOAD #3, N( ), A$
DATALOAD /618, A1$( ), X, Y
DATALOAD STR(A$, I, J)
```

DATALOAD BT

(NOT ON 2200A)

PUNCHED TAPE READER STATEMENT

General Form:

$$\text{DATALOAD BT [R]}\left[\left(\left[\text{N=expression}\right][,]\left[\text{L=}\left\{\begin{matrix}\text{hh}\\ \text{alpha variable}\end{matrix}\right\}\right][,]\left[\text{S=}\left\{\begin{matrix}\text{hh}\\ \text{alpha variable}\end{matrix}\right\}\right]\right)\right]\left[\begin{matrix}\#\text{f,}\\ \text{/xyy}\end{matrix}\right]\left\{\begin{matrix}\text{alpha variable}\\ \text{alpha array designator}\end{matrix}\right\}$$

where:

R	=	Reverse (read in reverse direction).
N	=	Number of characters to read.
L	=	Leader code character (ignored when reading until a different character code is read). If alpha variable is specified, the first character is used to specify the leader code. If L is not specified, no leader code is assumed.
S	=	Stop character. If alpha variable is specified, the first character is used to specify the stop character code.
hh	=	Hexdigits.
#f	=	Logical file number assigned to unit by SELECT statement (f is integer from 1 to 6).
xyy	=	Device address of punched tape reader. If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.

NOTE:
Commas must separate N, L, S arguments if more than one is present.

Purpose

This statement reads a punched tape forwards or backwards and stores the characters that are read in the alpha variable or alpha array designator specified. The tape is read until the stop character is encountered, the alpha variable or array is full, or the number of characters specified by N are read, whichever occurs first. All eight channels of the paper tape are read. The tape is read in the reverse direction if the 'R' parameter is included in the DATALOAD BT statement. The 'L' parameter specifies the leader code on the paper tape; when a tape is read, leader code is ignored (i.e., all characters read which are equal to the specified leader code character are ignored until a character is read that is not equal to the leader code).

DATALOAD BT permits punched tapes in any format to be read by the System 2200. Conversions can be done with bit manipulation statements such as BOOL.

Examples:

```
DATALOAD BT /618, A$
DATALOAD BTR (L=FF, S=0D) #1, A$( )
DATALOAD BT (N=100) A$( )
DATALOAD BT (N=200, L=00, S=A$) /618, B$( )
```

LOAD

(NOT ON 2200A)

PUNCHED TAPE READER COMMAND

General Form:

```
LOAD [#f,  ]
     [/xyy ]
```

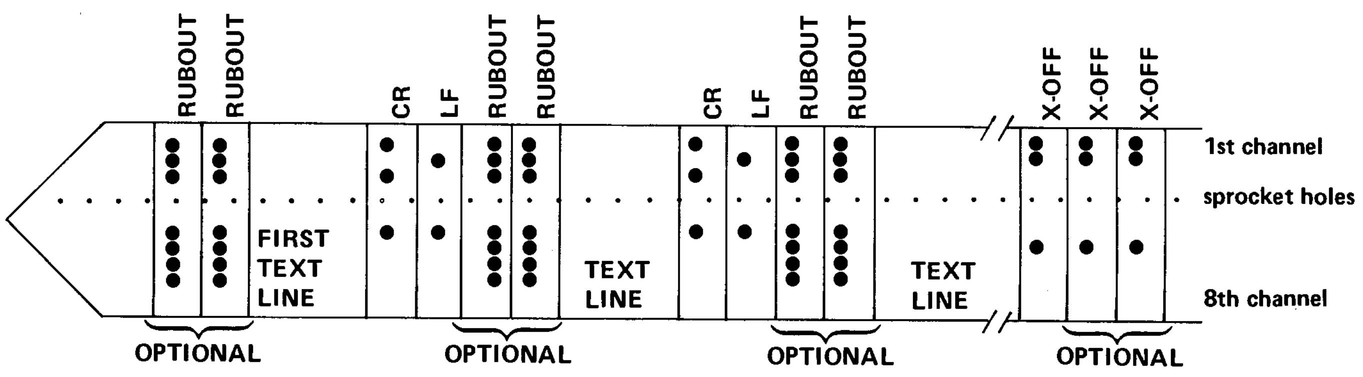
where:

#f	=	File number assigned to punched tape reader by SELECT statement (f is an integer from 1 to 6).
xyy	=	Device address of punched tape reader.
		If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.

Purpose

When the LOAD command is entered, the program punched on the tape is loaded and appended to the current program in memory. This command permits additions to a current program, or if entered after a CLEAR command, entry of a new program.

To be read, the paper tape must conform to the following format:



Text lines are punched in ASCII character code and are separated by CR LF RUBOUT RUBOUT; the rubouts are optional but are punched when a program is saved on Teletype. The program is terminated by an X-OFF character. LOAD reads only the first seven channels of the paper tape; the 8th bit is always read as 0. Nonpunched frames and RUBOUTS are ignored when reading the tape.

LOAD also can be used as a program statement for program chaining, as is described on the next page.

Examples:

```
LOAD
LOAD #1
LOAD /618
```

(NOT ON 2200A)

PUNCHED TAPE READER STATEMENT

General Form: LOAD [#f, /xyy,] [line number 1 [,line number 2]]

where:	#f	= File number assigned to unit by SELECT statement (f is an integer from 1 to 6).
	xyy	= Device address of punched tape reader. If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.
	line number 1	= The line number of the first line to be deleted from the current program. After loading, execution continues at the line 'line number 1'. An error results if 'line number 1' does not exist in the new program.
	line number 2	= The line number of the last line to be deleted from the program currently in memory, before loading tne new program.

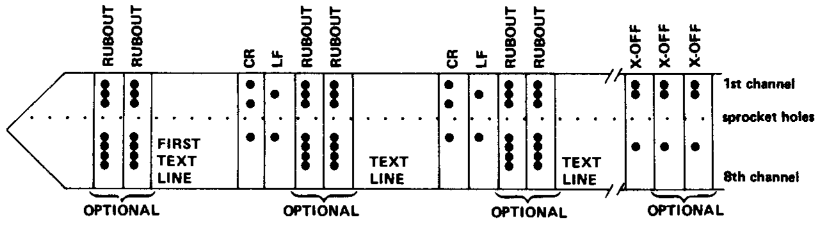
Purpose

This is a BASIC program statement which, in effect, produces an automatic combination of the following:

STOP	(stop current program execution)
CLEAR P	[line number 1 [line number 2]] (remove program text)
CLEAR N	(remove noncommon variables only)
LOAD	(load new program)
RUN	[line number 1] (run new program)

If only 'line number 1' is specified, the remainder of the current program is deleted, starting with that line number. If no line numbers are specified, the entire current program is deleted, and the newly loaded program is executed from the lowest line number. This permits segmented jobs to be run automatically without user intervention. Common variables are passed between program segments. LOAD must be the last statement on a statement line.

To be read, the punched tape must conform to the following format:



The LOAD statement must not be within a FOR/NEXT Loop; an error results when the NEXT or RETURN statement is encountered.

Text lines are punched in ASCII character code and are separated by CR LF RUBOUT RUBOUT; the rubouts are optional but are punched when a program is saved on Teletype. The program is terminated by an X-OFF character. LOAD reads only the first seven channels of the paper tape; the 8th bit is always read as 0. Nonpunched frames and RUBOUTS are ignored when reading the tape.

In Immediate Mode, LOAD is interpreted as a command (see LOAD command).

Examples:

```
100 LOAD
100 LOAD #2
100 LOAD /618
100 LOAD /618, 100
100 LOAD #2, 400, 1000
```

PLOTTERS

The PLOT statement can be used to operate any Wang plotter, the Plotting Output Writer, the Analog or the Digital Plotter.

This statement is not available on a 2200A and must be obtained on a 2200S or WCS/10 with Option 22.

PLOTTER STATEMENT

General Form:

```
                                                      ┌ literal        ┐
                                                      │ alpha variable │
                                                      │ D              │
PLOT [expression 0] <[expression 1] , [expression 2], │ R              │ >[, <[exp 1], [exp 2] , [parameter] >] . . .
                                                      │ U              │
                                                      │ S              │
                                                      └ C              ┘
```

where: expression 0 represents the replication factor, or the number of times the values enclosed in <> are plotted (1 ≤ expression 0 < 1000). If omitted, expression 0 = 1.

expression 1 represents Δ x in increments of .015"* (.01" for Model 2202) (–1000 < expression 1 < 1000). If omitted, expression 1 = 0.

expression 2 represents Δ y in increments of .01"* (–1000 < expression 2 < 1000). If omitted, expression 2 = 0.

All three expressions are truncated to integer values.

literal, alpha variable represent character or characters to be plotted or printed.

Parameters for Plotting:

No parameter and U imply 'move the distance (Δ x, Δ y) specified in expressions 1 and 2 with pen up (without plotting)'.

D implies 'draw a line while moving the distance (Δ x, Δ y) specified in expressions 1 and 2'.

R (RESET) moves the pen to the zero position as manually set on the plotter.

Parameters for Setting Plot Conditions:

C sets the character size (expression 1) for character plotting. Character size is an integer from 1 to 15.

S sets the horizontal (expression 1) and vertical (expression 2) spacing between characters for character plotting.

NOTE:
Parameters U, D, C, S, R cannot be used with the Model 2202. S and C cannot be used with the Model 2232A.

*These values assume full-scale plotting.

Purpose

This statement moves the plot pen (Model 2212 or Model 2232A) or typing element (Model 2202) from its current position to a position a distance x (expression 1; to the right if positive, to the left if negative) and y (expression 2; up if positive, down if negative) from the current position. The movement can be made with the pen up (U, or no parameter) or down (D). When a literal string or alpha variable is supplied as the parameter, the movement to the new position is made with the pen up and then the characters are plotted.

Example 1:

```
10 N = 10
20 PLOT N < 10, , "ABC" >
```

Advance Δx = 10 increments and print ABC. Do this 10 times.

Example 2:

```
10 PLOT < X, Y, "VALUE" >, < 40, 60, "-" >, < A + 10, B, C$ >
```

The multiple arguments in the same PLOT statement are processed sequentially from left to right.

Example 3:

```
10 PLOT < 10, 20, HEX (FB) >
```

This statement prints the normal plotting character (the centered dot) of the Model 2202. For more information on the available plotter, see the plotter manual.

TELETYPE

A Teletype® unit appropriately attached to a Wang system can be activated by a number of BASIC statements and commands. Tapes can be both read and punched.

Teletype-activating statements are not available on a 2200A and must be obtained for a 2200S and WCS/10 with Option 22.

DATALOAD

(NOT ON 2200A)

TELETYPE STATEMENT

General Form: DATALOAD [#f, /4yy,] argument [, argument] . . .

where #f	=	Logical file number assigned to unit by SELECT statement (f is integer from 1 to 6).
4yy	=	Device address of Teletype. Output (41D, 41E or 41F) If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.
argument	=	{variable, array designator}

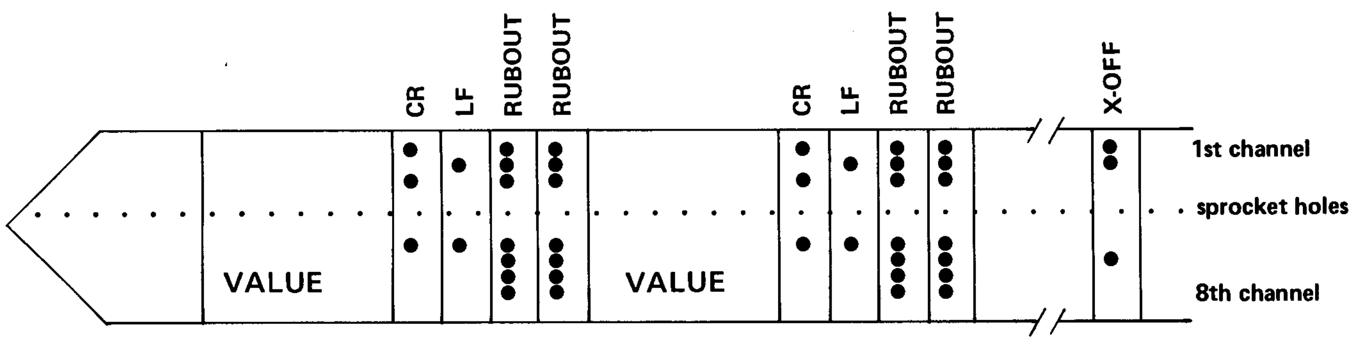
Purpose

This statement reads values from the Teletype® punched tape and sequentially assigns those values to the variables in the argument list. Numeric values can be assigned to alphanumeric variables; values assigned to numeric variables must be legitimate BASIC numbers. Arrays are filled row by row.

Values are successively read from the tape until all variables in the list are satisified or until the end-of-file is encountered (i.e., an X-OFF character is read). When an end-of-file is encountered, the remaining variables in the list are left with their current values; an IF END THEN statement then causes a transfer to the specified line number.

The System 2200 will automatically transmit a X-ON character to the Teletype to start the tape reader, and a X-OFF character to stop it when reading is completed.

To be read, the punched tape must conform to the following format:



Values are punched in ASCII character code and are separated by CR LF RUBOUT RUBOUT. All other RUBOUTS and nonpunched frames on the tape are ignored when the tape is read. DATALOAD reads only the first seven channels of the tape; the 8th bit is always read as 0.

Paper tapes punched on a Teletype via DATASAVE statements conform to this format. To read tape not in this format, use the DATALOAD BT statement.

Example:

```
DATALOAD X, Y, A$, B$
DATALOAD #3, N( ), A$
DATALOAD /41D, A1$( ), X, Y
DATALOAD STR (A$, I, J)
```

DATALOAD BT

(NOT ON 2200A)

TELETYPE STATEMENT

General Form:

$$\text{DATALOAD BT}\left[\left(\left[\text{N=expression}\right][,]\left[\text{L=}\left\{\begin{matrix}\text{hh}\\\text{alpha variable}\end{matrix}\right\}\right][,]\left[\text{S=}\left\{\begin{matrix}\text{hh}\\\text{alpha variable}\end{matrix}\right\}\right]\right)\right]\left[\begin{matrix}\#\text{f,}\\\text{/4yy,}\end{matrix}\right]\left\{\begin{matrix}\text{alpha variable}\\\text{alpha array designator}\end{matrix}\right\}$$

where:

N = Number of characters to read.

L = Leader code character (ignored when reading until a different character code is read). FF

If alpha variable is specified, the first character is used to specify the leader code.
If L is not specified, no leader code is assumed.

S = Stop character. 80 OR OD

If alpha variable is specified, the first character is used to specify the stop code.

hh = Hexdigits.

#f = Logical file number assigned to unit by SELECT statement (f is integer from 1 to 6).

4yy = Device address of Teletype. Output (41D, 41E, or 41F).

If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.

NOTE:
Commas must separate N, L, S arguments if more than one is present.

Purpose

This statement reads a punched tape on a Teletype® and stores the characters read in the alpha variable or alpha array designator specified. The tape is read until the stop character is encountered, the alpha variable or array is full, or the number of characters specified by N are read, whichever occurs first. All eight channels of the tape are read.

The System 2200 automatically sends out an X-ON character to start the tape reader and an X-OFF character to stop it. Because two additional characters are read after the X-OFF is sent, the following considerations should be observed. For termination by count (N parameter), the system normally sends out the X-OFF character after N-2 characters have been read. Therefore, if the number of characters to be read is specified by N, N should be ⩾ 3. If N = 1 (or 2), the next 2 or (1) characters may be lost. Similarly, if reading is terminated by filling the variable or array, the number of characters in the variable or array should be ⩾ 3. If a stop character is encountered, the stop character and the next 2 characters are read; the tape then stops. The 'L' parameter specifies the leader code on the punched tape; when a tape is read, leader code is ignored (i.e., all characters read which are equal to the specified leader code character are ignored until a character not equal to the leader code is recognized).

DATALOAD BT permits punched tapes in any format to be read by the System 2200. Conversion to ASCII code can be effected with BOOL or other bit manipulation statements.

Examples: DATALOAD BT (L=FF, S=80)/41D OR 7)

```
DATALOAD BT /41D, A$
DATALOAD BT (L = FF, S = 0D) #1, A$( )
DATALOAD BT (N = 100) A$( )
DATALOAD BT (N = 20, L = 00, S = A$ A1$( )
```

DATASAVE

(NOT ON 2200A)

TELETYPE STATEMENT

General Form:

```
DATASAVE [ #f,  ]  { OPEN "name"                     }
         [ /4yy,]  { END                             }
                   { argument [, argument ] . . .    }
```

where:

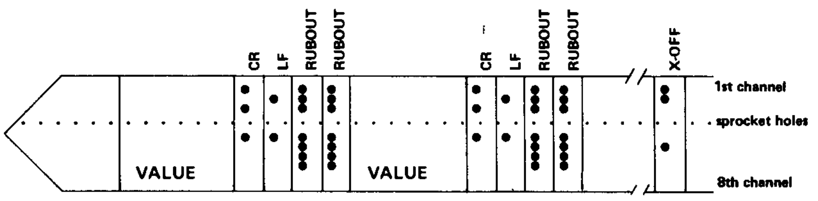
#f	=	File number assigned to unit by SELECT statement (f is integer from 1 to 6).
4yy	=	Device address of Teletype. Output (41D, 41E, or 41F) If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.
argument	=	{ literal string / alpha variable / expression / array designator }
name	=	1 to 8 characters (note, the name is required but is not used).
OPEN	=	Punch leader code (50 null characters).
END	=	Punch X-OFF character and trailer code (50 null characters).

Purpose

This statement causes the values specified in the argument list to be punched on tape. Numeric values are punched in a form identical to that resulting from a PRINT statement (i.e., exponential or fixed point form).

Alphanumeric values are punched in ASCII character code and separated by CR LF RUBOUT RUBOUT; trailing spaces in values of alphanumeric variables are not written. Alphanumeric values must not contain any of the following characters; CR, RUBOUT, X-OFF; trailing spaces are ignored. The OPEN parameter writes leader code of 50 null characters. The END parameter terminates the data file by punching an X-OFF character and trailer code of 50 null characters.

The paper tape is punched in the following format:



If the Teletype has the facility for turning the tape punch on and off with TAPE-ON and TAPE-OFF codes these can be utilized under program control by transmitting the codes to the Teletype by a PRINT statement prior to and after punching.

Example:

```
DATASAVE X, Y, A$
DATASAVE OPEN "TTY"
DATASAVE END
DATASAVE #1, A$( )
DATASAVE /41D, N( ), A$, X, Y, Z
DATASAVE STR(A$, I, J), HEX(FAFB)
```

DATASAVE BT

(NOT ON 2200A)

TELETYPE STATEMENT

General Form: DATASAVE BT $\begin{bmatrix}\#f,\\ /4yy,\end{bmatrix} \begin{Bmatrix}\text{alpha variable}\\ \text{alpha array designator}\end{Bmatrix}$

where:	#f	= Logical file number assigned to unit by SELECT statement (f is integer from 1 to 6).
	4yy	= Device address of Teletype. Output (41D, 41E, or 41F)
		If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.

Purpose

This statement punches the values of an alpha variable or alpha array onto a paper tape with no control information (i.e., no CR LF RUBOUT RUBOUT separating values). Trailing spaces in alpha values are punched (see other Teletype statements).

DATASAVE BT permits paper tapes to be punched in any format. Any 8-bit codes may be punched. If the Teletype has the facility for turning the tape punch on and off with TAPE-ON and TAPE-OFF codes these can be utilized under program control by transmitting the codes to the Teletype by a PRINT statement prior to and after punching.

Example:

```
DATASAVE BT #2, A$( )
DATASAVE BT /41D, B1$
DATASAVE BT Q$( )
```

[illegible]

TELETYPE STATEMENT

General Form:	LOAD [#f, / /4yy,]	[line number 1 [,line number 2]]
where:	#f	= File number assigned to teletype by SELECT statement (f is an integer from 1 to 6).
	4yy	= Device address of Teletype. If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.
	line number 1	= The line number of the first line to be deleted from the current program. After loading, execution continues at the line = line number 1. An error occurs if 'line number 1' does not exist in the new program.
	line number 2	= The line number of the last line to be deleted from the program currently in memory, before loading the new program.

Purpose

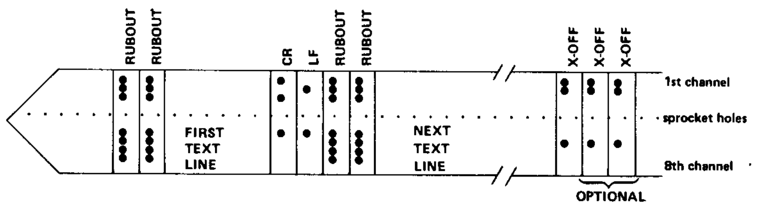
This is a BASIC program statement which produces an automatic combination of the following:

STOP	(stop current program execution)
CLEAR P	[line number 1 [,line number 2]] (remove program text)
CLEAR N	(remove noncommon variables only)
LOAD	(load new program)
RUN	[line number 1] (run new program)

If only 'line number 1' is specified, the remainder of the current program is deleted starting with that line number. If no line numbers are specified, the entire current program is deleted, and the newly loaded program is executed from the lowest line number. This permits segmented jobs to be run automatically without user intervention. Common variables are passed between program segments. LOAD must be the last statement on a statement line.

The LOAD statement must not be within a FOR/NEXT loop or subroutine; an error results when the NEXT or RETURN statement is encountered.

To be read, the punched tape must conform to the following format:



Text lines are punched in ASCII character code and are separated by CR LF RUBOUT RUBOUT. The program is terminated by three X-OFF characters. LOAD reads only the first seven channels of the paper tape; the 8th bit is always read as 0. Nonpunched frames and RUBOUTS are ignored when reading the tape.

In Immediate Mode, LOAD is interpreted as a command (see LOAD command).

Example:

```
100 LOAD
100 LOAD #2
100 LOAD /41D
100 LOAD #2, 400, 1000
100 LOAD /41D, 100
```

LOAD

(NOT ON 2200A)

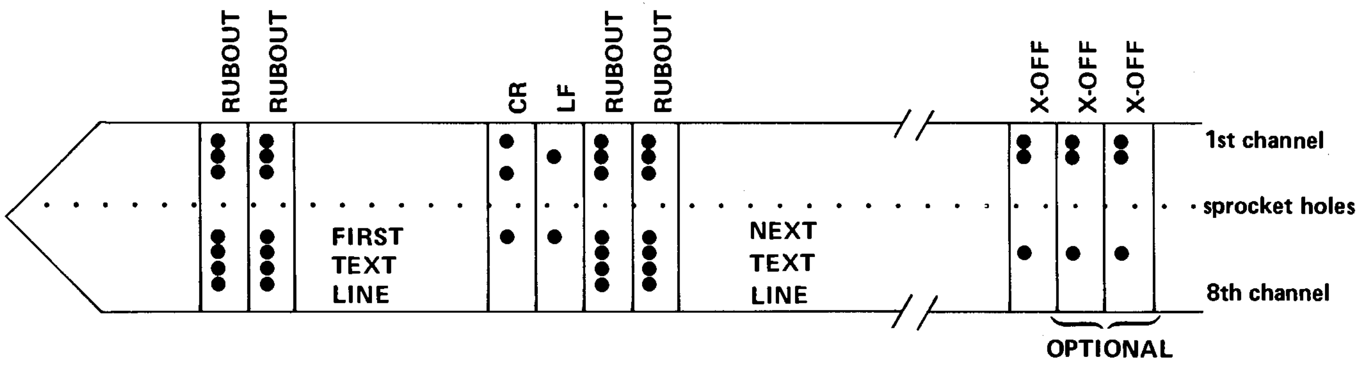
TELETYPE COMMAND

General Form:			LOAD [#f / /4xx]
where:	#f	=	File number assigned to unit by SELECT statement (f is an integer from 1 to 6).
	4yy	=	Device address of device to load from. (41D, 41E, or 41F).
			If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.

Purpose

When the LOAD command is entered, the program punched on the paper tape is loaded and appended to the current program in memory. This command permits additions to a current program, or if entered after a CLEAR command, entry of a new program.

To be read, the paper tape must conform to the following format:



Text lines are punched in ASCII character code and are separated by CR LF RUBOUT RUBOUT. The program is terminated by 3 X-OFF characters. LOAD reads only the first seven channels of the paper tape; the 8th bit is always read as 0. Nonpunched frames and RUBOUTS are ignored when reading the tape.

LOAD also can be used as a program statement, as described on the next page.

Examples:

```
LOAD
LOAD #1
LOAD /41D
```

SAVE

(NOT ON 2200A)

TELETYPE COMMAND

General Form:

SAVE [#f, /4yy,] [line number [,line number]]

where #f = File number assigned to unit by SELECT statement (#1 to #6).

4yy = Device address of Teletype. (41D, 41E, or 41F).

If neither of the above is specified, the default device address (the device address currrently assigned to TAPE, (see SELECT)) is used.

1st line number = Starting line number to be saved.

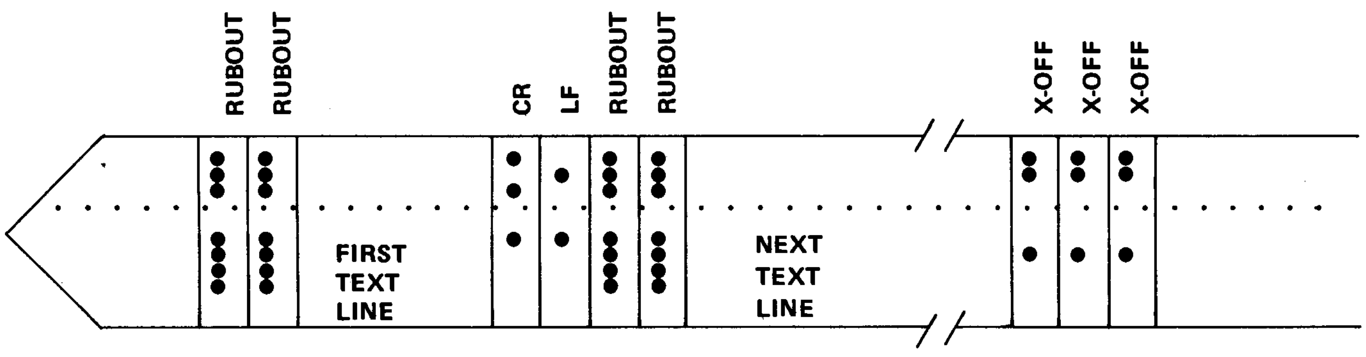
2nd line number = Ending line number to be saved.

Purpose

The SAVE command causes BASIC programs (or portions of BASIC programs) to be punched on paper tape.

If no line numbers are specified, the entire user program text is saved. SAVE with one line number causes all user program lines from the indicated line through the highest numbered program line to be punched on tape. If two line numbers are entered, all text from the first through the second line number, inclusive, is punched.

The paper tape format is:



Text lines are punched in ASCII character code and are separated by CR LF RUBOUT RUBOUT. The program is terminated by 3 X-OFF's.

Examples:

```
SAVE
SAVE #3
SAVE /41D
SAVE /41D, 100, 200
SAVE #5, 400
```

Section IX
Error Codes

INTRODUCTION

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 with an "↑" symbol just below the point of the error. The arrow is followed by the error message number (e.g., ↑ ERR 02).

The following example shows the format of the System 2200 error pointer:

```
:10 DIM A(P)
          ↑ ERR 13
```

The user can refer to the list of error messages to identify the error by 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:

```
:PXINT  X
  ↑ ERR 06 (expected equal sign)
```

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 On, Power Off) 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.

TYPES OF ERRORS

A Syntax Error occurs when the syntax of a System 2200 BASIC statement is violated. Pressing a sequence of keys not recognized as an accepted combination produces this type of error. Syntax errors in a statement are recognized and noted as soon as the EXEC key is touched to enter a statement or program line. Examples of this type of error include mispelled verbs, illegal forms for numbers, operators, or parentheses, and the improper use of punctuation.

Example:

```
:10 DEFFN . (X) = 3*X↑2 - 2*X↑3
            ↑ ERR 21
```

An **Error of Execution** occurs 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 since 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, division by zero, etc.

Example:
(Branch to non existent statement number)

```
:100 GOTO 110
:105 PRINT "VALUES=" ;A, B, C
:120 END
:RUN
100 GOTO 110
         ↑ ERR 11
```

A **Programming Error** occurs when 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 2200 to identify a programming error, debugging features such as TRACE, HALT/STEP, CONTINUE, can significantly speed up the process of debugging a program.

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 the program, internal tables and variables has been filled (see "Internal Storage," Section I). When ERR02 occurs, all non-common variables are cleared.

Action: Examine program for:

1) excessive DIM, COM statements.
2) subroutines not terminated by RETURN or RETURN CLEAR, improper exits from FOR/NEXT loops.

Suggestion: Insert an END statement as the first line in the program and execute the routine. If END = value appears, the error is probably case 2); otherwise, case 1).

Example:

```
:10 DIM A(19), B(10,10), C(10,10)
RUN
↑ERR 02
```

(the space available for variable tables was exceeded) user must reduce program and variable storage requirements or change program logic.

CODE 03

Error: **Math Error**

Cause:

1. EXPONENT OVERFLOW. The exponent of the calculated value was < –99 or > 99. (+, – *, /, ↑, 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)
                              ↑ERR 03          (exponent overflow)
```

CODE 04

Error:	**Missing Left Parenthesis**
Cause:	A left parenthesis (( ) was expected.
Action:	Correct statement text.
Example:	See below

```
: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:	See below

```
:10 Y = INT(1.2↑5
                  ↑ERR 05
:10 Y = INT(1.2↑5)                 (Possible Correction)
```

CODE 06

Error:	**Missing Equals Sign**
Cause:	An equals sign (=) was expected.
Action:	Correct statement text.
Example:	See below

```
: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:	See below

```
: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:	See below

```
:10 X=FNC(2)
:20 PRINT "X";X
:30 END
:RUN
10 X=FNC(2)
      ↑ERR 08
:05 DEFFNC(V)=COS(2*V)             (Possible Correction)
```

CODE 09

Error: **Illegal FN Usage**

Cause: More than five levels of nesting were encountered when evaluating an FN function.

Action: Reduce the number of nested functions.

Example:

```
:10 DEF FN1(X)=1+X       :DEF FN2(X)=1+FN1(X)
:20 DEF FN3(X)=1+FN2(X)     :DEF FN4(X)=1+FN3(X)
:30 DEF FN5(X)=1+FN4(X)     :DEF FN6(X)=1+FN5(X)
:40 PRINT FN6(2)
:RUN
10 DEF FN1(X)=1+X        :DEF FN2(X)=1+FN1(X)
              ↑ERR 09
:40 PRINT 1+FN5(2)                (Possible Correction)
```

CODE 10

Error: **Incomplete Statement**

Cause: The end of the statement was expected.

Action: Complete the statement text.

Example:

```
:10 PRINT X"
            ↑ERR 10
:10 PRINT "X"
     OR
:10 PRINT X                       (Possible Correction)
```

CODE 11

Error: **Missing Line Number or Continue Illegal**

Cause: The line number is missing or a referenced line number is undefined; or the user is attempting to continue program execution after one of the following conditions: A text or table overflow error, a new variable has been entered, a CLEAR command has been entered, the user program text has been modified, or the RESET key has been pressed.

Action: Correct statement text.

Example:

```
:10 GOSUB 200
           ↑ERR 11
:10 GOSUB 100                     (Possible Correction)
```

CODE 12

Error: **Missing Statement Text**

Cause: The required statement text is missing (THEN, STEP, etc.).

Action: Correct statement text.

Example:

```
:10 IF I+12*X,45
            ↑ERR 12
:10 IF I=12*X THEN 45             (Possible Correction)
```

CODE 13

Error:	**Missing or Illegal Integer**
Cause:	A positive integer was expected or an integer was found which exceeded the allowed limit.
Action:	Correct statement text.
Example:	:10 COM D(P)

```
:10 COM D(P)
         ↑ERR 13
:10 COM D(8)                    (Possible Correction)
```

CODE 14

Error:	**Missing Relation Operator**
Cause:	A relational operator (<, =, >, <=, >=, <>) was expected.
Action:	Correct statement text.

Example:

```
:10 IF A-B THEN 100
        ↑ERR 14
:10 IF A=B THEN 100             (Possible Correction)
```

CODE 15

Error:	**Missing Expression**
Cause:	A variable, or number, or a function was expected.
Action:	Correct statement text.

Example:

```
:10 FOR I=, TO 2
        ↑ERR 15
:10 FOR I=1 TO 2                (Possible Correction)
```

CODE 16

Error:	**Missing Scalar Variable**
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 Element or 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, or an array contains more than 4,096 elements.

Action: Correct the program.

Example:

```
:10 DIM A(2,3)
:20 A(1,4) = 1
:RUN
 20 A(1,4) = 1
          ↑ERR 18
:10 DIM A(2,4)                    (Possible Correction)
```

CODE 19

Error: **Missing Number**

Cause: A number was expected.

Action: Correct statement text.

Example:

```
:10 DATA L
          ↑ERR 19
:10 DATA +                        (Possible Correction)
```

CODE 20

Error: **Illegal Number Format**

Cause: The form of a number 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 or Array Element**

Cause: An array variable which was not defined properly in a DIM or COM statement is referenced in the program. (An array variable was either not defined in a DIM or COM statement or has been referenced as both a one-dimensional and a two-dimensional array, or has been changed during execution (CLEAR V to correct the latter).)

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 Mode statement was encountered. Re-enter

Action: a corrected Immediate Mode 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 GO TO 100
:30 NEXT I: END
:100 PRINT "X=";X
:110 RETURN
:RUN
X=-.7568025

110 RETURN
          ↑ ERR 25
:25 GOSUB 100                    (Possible Correction)
```

CODE 26

Error: Illegal FOR/NEXT Usage

Cause: There is no companion FOR statement for a NEXT statement, or a branch was made into the middle of a FOR/NEXT 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 are not enough data values to satisfy READ statement requirements.

Action: Correct program to supply additional data.

Example:
```
:10 DATA 2
:20 READ X,Y
:30 END
:RUN

 20 READ X,Y
           ↑ERR 27
:11 DATA 3                       (Possible Correction)
```

CODE 28

Error:	**Data Reference Beyond Limits**
Cause:	The data reference in a RESTORE statement is beyond the existing data limits.
Action:	Correct the RESTORE statement.

Example:

```
:10 DATA 1,2,3
:20 READ X,Y,Z
:30 RESTORE 5
    ....
    ....
    ....
:90 END
:RUN
 30 RESTORE 5
                ↑ERR 28
:30 RESTORE 2                (Possible Correction)
```

CODE 29

Error:	**Illegal Data Format**
Cause:	The value entered as requested by an INPUT statement is in an illegal format.
Action:	Reenter data in the correct format starting with erroneous number or terminate run with the RESET key and run again.

Example:

```
:10 INPUT X,Y
    ....
    ....
    ....
:90 END
:RUN
:INPUT
?1A,2E-30
  ↑ERR 29
?12,2E-30                    (Possible Correction)
```

CODE 30

Error:	**Illegal Common Assignment**
Cause:	A COM statement was preceded by a non-common variable definition.
Action:	Correct program, making all COM statements the first numbered lines.

Example:

```
:10 A=1 :B=2
:20 COM A,B
:99 END
:RUN

 20 COM A,B
          ↑ERR 30
:10[CR/LF-EXECUTE]           (Possible Correction)
:30 A=1 :B=2
```

CODE 31

Error: **Illegal Line Number**

Cause: The 'statement number' key was pressed producing a line number greater than 9999; or in renumbering a program with the RENUMBER command a line number was generated which was greater than 9999.

Action: Correct the program.

Example:

```
:9995 PRINT X,Y
:[STMT NUMBER Key]
 ↑ERR 31
```

CODE 33

Error: **Missing HEX Digit**

Cause: A digit or a letter from A to 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 after attempting to read the bad record ten times.

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 Image Statement.

Example:

```
:10 PRINTUSING 20, 1.23
:20% AMOUNT =
:RUN
:10 PRINTUSING 20,1.23
                  ↑ERR 36
:20% AMOUNT = #####           (Possible Correction)
```

CODE 37

Error: **Statement Not an Image Statement**

Cause: The statement referenced by the PRINTUSING statement is not an Image statement.

Action: Correct either the PRINTUSING or the Image statement.

Example:

```
:10 PRINTUSING 20,X
:20 PRINT X
:RUN
:10 PRINTUSING 20,X
                  ↑ERR37
:20% AMOUNT = $#,###.##       (Possible Correction)
```

CODE 38

Error: **Illegal Floating Point Format**

Cause: Fewer than 4 up arrows were specified in the floating point format in an image statement.

Action: Correct the Image statement.

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 alpha variable.

Example:

```
:10 B$ = STR(A$, 10, 8)
                     ↑ERR 41
:10 B$ = STR(A$, 10, 6)       (Possible Correction)
```

CODE 42

Error: File Name Too Long

Cause: The program name specified is too long (a maximum of 8 characters is allowed).

Action: Correct the program text.

Example:
```
:SAVE "PROGRAM#1"
                  ↑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; any program in memory is cleared.

CODE 45

Error: Program Line Too Long

Cause: A statement line may not exceed 192 keystrokes.

Action: Shorten the 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 #f file specification in a program statement is undefined.

Action: Define the specified file number with a SELECT statement.

Example:
```
:SAVE #2
      ↑ERR 47
:SELECT #2 10A
:SAVE #2                  (Possible Correction)
```

CODE 48	
Error:	**Undefined Keyboard Function**
Cause:	There is no DEFFN' in a user's program corresponding to the Special Function key pressed.
Action:	Correct the program.
Example:	:[Special 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, make sure the tape is correctly positioned or, if loading a program or datafile by name, be sure you have mounted the correct tape.
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 opening tab on bottom of cassette).
Action:	Mount another cassette or "unprotect" the tape cassette by covering the hole on the bottom of the cassette with the tab or tape.
Example:	SAVE /103 ↑ERR 50

CODE 51	
Error:	**Illegal Statement**
Cause:	The statement input is not a legal 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 incorrectly. 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) ↑ 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 of 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:	**Value Not Between 0 and 32761**
Cause:	Illegal value specified; value is negative or greater than 32767. (The System 2200 cannot store a sector address greater than 32767 and cannot handle certain MAT arrays with addresses outside this range.)
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 or MAT locator array too small.
Action:	Dimension the alpha variable to be at least two bytes (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:	Transient 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; if error still persists contact Wang service personnel.
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$( )

```
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 or Disk Not Scratched**
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 personnel. (Note, the disk must 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 should be engaged only when formatting a disk.)
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. The error can be either in the medium or the hardware.
Action:	Format the disk again; if error persists, call for Wang service.
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 personnel.
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. If error still occurs call Wang Service personnel.
Example:	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 was incorrectly written. This usually means the disk was never initially formatted.
Action:	Format the disk. If the disk was formatted, re-write the bad sector, or reformat the disk. If error persists call Wang Service personnel.
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 > (in PLOT statement)**
Cause:	The required PLOT angle brackets are not in the PLOT statement.
Action:	Correct the statement in error.

Example:

```
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, to load a data file as a program or open a program file as a data file.
Action:	Make sure the correct file name is being used; make sure the proper disk is mounted.
Example:	LOAD DCR "PRES"

```
LOAD DCR "PRES"
                 ↑ERR 80
LOAD DCF "PRES"               (Possible Correction)
```

CODE 81

Error:	**/XYY Device Specification Illegal**
Cause:	The /XYY 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 Error**
Cause:	A disk address was not properly transferred from the CPU to the disk when executing MOVE or COPY.
Action:	Run program again. If error persists, call Wang Field Service Personnel.

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)

```
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. This is usually an error in the medium.
Action:	Write the information again. If error persists, call Wang Field Service personnel.
Example:	100 DATASAVE DCF$ X, Y, Z

```
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$

```
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:	10 LOAD DAR (100,L)

```
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:	SAVE DCF "PRGM"

```
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:

```
: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 equation 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 CON 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)
```

CODE 95

Error: **Illegal Microcommand or Field/Delimiter Specification**

Cause: The microcommand or field/delimiter specification used is invalid.

Action: Use only the microcommands and field/delimiter specifications provided.

Example:

```
:RUN
:10 $GIO (1023 , A$)
              ↑ ERR 95
:10 $GIO (0123 , A$)                              (Possible Correction)
```

CODE 96

Error: **Missing Arg 3 Buffer**

Cause: The buffer (Arg 3) of the $GIO statement was either omitted or already used by another data input, data output, or data verify microcommand.

Action: Define the buffer if it was omitted, or separate the two data commands into separate $GIO statements.

Example:

```
10 $GIO /03B (A000 C640 , A$) B$
                          ↑ ERR 96
10 $GIO /03B (A000, A1$) B1$
20 $GIO /03B (C640, A2$, B2$                      (Possible Correction)
```

CODE 97

Error: **Variable or Array Too Small**

Cause: Not enough space reserved for the variable or array.

Action: Change dimensioning statement.

Example:

```
:10 DIM A$6
:20 $GIO (0123, A$)
:RUN
:20 $GIO (0123, A$)
                ↑ ERR 97
:10 DIM A$10                                      (Possible Correction)
```

CODE 98

Error: **Illegal Array Delimiters**

Cause: The number of bytes specified by the delimiters exceeds the number of bytes in the array.

Action:

Example:

```
:10 DIM A$(3) 10, B$(4) 64
:20 $TRAN (A$() < 10, 23 > ,B$() )
:RUN
:20 $TRAN (A$() < 10, 23 > ,B$() )
                           ↑ ERR 98
:20 $TRAN (A$() < 10, 13 > ,B$() )                (Possible Correction)
```

CODE =1

Error:	**Missing Numeric Array Name**
Cause:	A numeric array name [e.g., N( )] was expected.
Action:	Correct the statement in error.

Example:

```
100 MAT CONVERT A$( ) TO N( )
                       ↑ ERR=1
100 MAT CONVERT N( ) TO A$( )                 (Possible Correction)
```

CODE =2

Error:	**Array Too Large**
Cause:	The specified array contains too many elements. For example, the number of elements cannot exceed 4096.
Action:	Correct the program.

Example:

```
10 DIM A$(100,50)2, B$(100,50)2, W$(100,50)2
.
.
.
100 MAT SORT A$() TO S$(), B$()
                          ↑ ERR=2
10 DIM A$(100,40)2, B$(100,40)2, W$(100,40)2     (Possible Correction)
```

CODE =3

Error:	**Illegal Dimensions**
Cause:	The dimensions defined for the array or its element length are illegal.
Action:	Dimension the array properly.

Example:

```
10 DIM A$(63), B$(63)1, W$(63)2
.
.
.
100 MAT SORT A$() TO W$(), B$()
                          ↑ ERR=3
10 DIM A$(63), B$(63)2, W$(63)2                 (Possible Correction)
```

CODE SYSTEM ERROR!

Error:	Either CPU hardware malfunction or operator faux pas (no execution mode).
Cause:	Program must be resolved before any Special Function Key can be used or other operations performed. Certain meaningless operations can cause this error. A true hardware malfunction can also cause this error.
Action:	This error is unrecoverable. Master Initialize to remove error and run program again. If error persists, call for Wang service.

Appendices

0

APPENDIX A

ASCII AND HEX CODES WITH CRT CHARACTER SET

The following chart shows the ASCII 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	prime	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 ↑)		,	< 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	■

NOTE:

Codes 60 to 7F are available only on an upper/lowercase CRT and on a Model 2221W printer.

APPENDIX B

ASCII, Hex and Binary Codes with VAL Decimal Equivalents

The character set and control codes are those for the CRT; other devices such as printers have slightly different character sets (see the appropriate peripheral manuals). Codes between hex (80) and hex (FF) are Text Atoms and are not normally used as character data.

LOW ORDER

HIGH ORDER

HEX →		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
↓	BINARY	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	0000	NUL 0	SOH (Cursor home) 1	2	(Clear screen, Cursor home) 3	4	5	6	(Alarm) 7	(Cursor Back-Space) 8	HT or Cursor right 9	LF or Cursor down 10	VT 11	FF (Rev-Index) 12	CR 13	(Elon-gated Char.) 14	SI (Shift Down) 15
1	0001	DLE 16	X-ON 17	18	X-OFF 19	20	21	22	23	24	Clear Tab 25	Set Tab 26	27	28	29	30	31
2	0010	SP (Space) 32	! 33	" 34	# 35	$ 36	% 37	& 38	' (Quote) 39	(40	) 41	* 42	+ 43	, (Comma) 44	— (Dash) 45	. (Period) 46	/ 47
3	0011	0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
4	0100	@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
5	0101	P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	↑ 94	← 95
6	0110	` (Prime) 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
7	0111	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	¦ 124	} 125	~ 126	■ 127
8	1000	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	1001	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	1010	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	1011	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	1100	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	1101	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	1110	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	1111	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
↑ HEX →		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

0 = Zero	Numbers at lower right corner of
I = Letter i	each box are decimal (VAL) equivalents.

APPENDIX C

DEVICE ADDRESSES

Categories are in alphabetical order; boxed values are Default Device Addresses (also see SELECT).

I/O Categories	Device Addresses
BCD Input Interface	25A, 25B, 25C, 25D, 25E, 25F
CRT Units	$\boxed{005}$, 006, 007, 008
Digitizer	25A, 25B, 25C, 25D, 25E, 25F
Disk Units	$\boxed{310}$, 320, 330 (for the Model 2243, third device address is 350, 360 or 370; for the WCS/30, the diskette has device address 310, and the hard disk has device address 320).
Hopper-Feed Card Readers	628
Keyboards	$\boxed{001}$, 002, 003, 004
Manual Card Reader	517
Nine-Track Tape Unit	07B, 07D, 07F
Output Writer	211, 212
Parallel I/O Interface	23A, 23C, 23E Input 23B, 23D, 23F Output
Plotters	$\boxed{413}$, 414
Printers	215, 216
Punched Tape Reader	618
Tape Cassette Units	$\boxed{10A}$, 10B, 10C, 10D, 10E, 10F
Telecommunications Controller	219, 21A, 21B Input 21D, 21E, 21F Output
Teletype® Units	019, 01A, 01B Input 01D, 01E, 01F Output
Teletype Punched Tape Units	41D, 41E, 41F
WCS/10 Triple Controller	$\boxed{001}$ (keyboard), 215 (printer), $\boxed{10A}$ (tape cassette)
WCS/20 Triple Controller	$\boxed{001}$ (keyboard), 215 (printer), $\boxed{310}$ (diskette)
WCS/30 Triple Controller	$\boxed{001}$ (keyboard), 215 (printer), $\boxed{310}$ (diskette), 320 (hard disk)

*The Nine-Track Tape Unit is activated exclusively with $GIO statements which ignore the first hexdigit of the Device Address.

APPENDIX D

THE HEXADECIMAL SYSTEM

The binary form used for coding data stored in memory, as its name suggests, is built on a base of two digits, zero and one. This binary form is used in most computers because the two binary digits can represent the 'on-off' condition of any electronic switch. Binary forms, however, are extremely cumbersome because it is necessary to use many places to write down even very small numbers. For example, the number 7 requires only one place when written as a decimal number, but in binary it requires three (111).

There are many times when it is useful to know the binary form of a number or character stored in your system. For this purpose, the hexadecimal system of representing binary values is provided. In the hexadecimal system, each byte is separated into two groups of four bits each and a single hexdigit is used to express the value of a single group. The hexadecimal system is built on a base of sixteen digits, the numbers 0 through 9 and the letters A through F. A comparison of the binary, decimal and hexadecimal forms of some numbers is shown in the Table.

Decimal, Binary and Hexadecimal Numbers

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Example:

The binary value 0101 1101 can be expressed as 5D in the hexadecimal system.

In your Wang system, the ASCII eight-bit character codes are used for the representation of all letters, characters and symbols. Each byte is represented by two hexdigits.

For example, the character 1 in ASCII format is 0011 0001; its hexcode is 31. Letters, characters and control codes can be decoded in a similar fashion. The Wang HEXPRINT statement provides access to the binary form of any byte by outputting its value in hexadecimal notation. Appendix A gives the complete CRT character set with equivalent hex codes; Appendix B provides a chart of the relationships between ASCII binary, hexadecimal and decimal VAL forms.

APPENDIX E

CPU SPECIFICATIONS

Average Execution Times (at full 13-digit precision)

Operation	Time
add	0.8 ms
subtract	0.8 ms
multiply	3.9 ms
divide	7.4 ms
square root	46.4 ms
X^y	45.4 ms
e^x	25.3 ms
$\log_e x$	23.2 ms
integer	0.24 ms
absolute value	0.25 ms
sign	0.25 ms
sine	28.3 ms
cosine	38.9 ms
tangent	78.5 ms
arctangent	72.5 ms
read/write cycle	1.6 μs

(ms = millisecond = 1/1000 second,
μ sec = microsecond = 1/millionth second)

Memory Size
4,096 bytes to 32,768 bytes

Peripheral Capacity
1 to 6 (expandable to 11 maximum) for 2200A, B, C.
1 to 3 (expandable to 9 max) for 2200S, T.

Largest Line Number
9999

Magnitude of Largest Number
10^{99}

Magnitude of Smallest Number
10^{-99}

Precision
13 significant digits

Maximum Nesting of FOR/NEXT Loops and Subroutines
50 levels

Maximum Length of Alpha Variable
64 characters

Storage Code
ASCII

Size of Byte
8 Bits

Type of Arithmetic
Decimal

System Language
BASIC

APPENDIX F

ABBREVIATED ERROR MESSAGES

ERR 01	TEXT OVERFLOW
ERR 02	TABLE OVERFLOW
ERR 03	MATH ERROR
ERR 04	MISSING LEFT PARENTHESIS
ERR 05	MISSING RIGHT PARENTHESIS
ERR 06	MISSING EQUALS SIGN
ERR 07	MISSING QUOTATION MARKS
ERR 08	UNDEFINED FN FUNCTION
ERR 09	ILLEGAL FN USAGE
ERR 10	INCOMPLETE STATEMENT
ERR 11	MISSING LINE NUMBER OR CONTINUE ILLEGAL
ERR 12	MISSING STATEMENT TEXT
ERR 13	MISSING OR ILLEGAL INTEGER
ERR 14	MISSING RELATION OPERATOR
ERR 15	MISSING EXPRESSION
ERR 16	MISSING SCALAR
ERR 17	MISSING ARRAY
ERR 18	ILLEGAL VALUE
ERR 19	MISSING NUMBER
ERR 20	ILLEGAL NUMBER FORMAT
ERR 21	MISSING LETTER OR DIGIT
ERR 22	UNDEFINED ARRAY VARIABLE
ERR 23	NO PROGRAM STATEMENTS
ERR 24	ILLEGAL IMMEDIATE MODE STATEMENT
ERR 25	ILLEGAL GOSUB/RETURN USAGE
ERR 26	ILLEGAL FOR/NEXT USAGE
ERR 27	INSUFFICIENT DATA
ERR 28	DATA REFERENCE BEYOND LIMITS
ERR 29	ILLEGAL DATA FORMAT
ERR 30	ILLEGAL COMMON ASSIGNMENT
ERR 31	ILLEGAL LINE NUMBER
ERR 33	MISSING HEX DIGIT
ERR 34	TAPE READ ERROR
ERR 35	MISSING COMMA OR SEMICOLON
ERR 36	ILLEGAL IMAGE STATEMENT
ERR 37	STATEMENT NOT IMAGE STATEMENT
ERR 38	ILLEGAL FLOATING POINT FORMAT
ERR 39	MISSING LITERAL STRING
ERR 40	MISSING ALPHANUMERIC VARIABLE
ERR 41	ILLEGAL STR(ARGUMENTS
ERR 42	FILE NAME TOO LONG
ERR 43	WRONG VARIABLE TYPE
ERR 44	PROGRAM PROTECTED
ERR 45	PROGRAM LINE TOO LONG
ERR 46	NEW STARTING STATEMENT NUMBER TOO LOW
ERR 47	ILLEGAL OR UNDEFINED DEVICE SPECIFICATION
ERR 48	UNDEFINED SPECIAL FUNCTION KEY
ERR 49	END OF TAPE
ERR 50	PROTECTED TAPE
ERR 51	ILLEGAL STATEMENT
ERR 52	EXPECTED DATA (NONHEADER) RECORD
ERR 53	ILLEGAL USE OF HEX FUNCTION
ERR 54	ILLEGAL PLOT ARGUMENT
ERR 55	ILLEGAL BT ARGUMENT
ERR 56	NUMBER EXCEEDS IMAGE FORMAT
ERR 57	ILLEGAL VALUE
ERR 58	EXPECTED DATA RECORD
ERR 59	ILLEGAL ALPHA VARIABLE
ERR 60	ARRAY TOO SMALL
ERR 61	TRANSIENT DISK HARDWARE ERROR
ERR 62	FILE FULL
ERR 63	MISSING ALPHA ARRAY DESIGNATOR
ERR 64	SECTOR NOT ON DISK OR DISK NOT SCRATCHED
ERR 65	DISK HARDWARE MALFUNCTION
ERR 66	FORMAT KEY ENGAGED
ERR 67	DISK FORMAT ERROR
ERR 68	LRC ERROR
ERR 71	CANNOT FIND SECTOR
ERR 72	CYCLIC READ ERROR
ERR 73	ILLEGAL ALTERING OF A FILE
ERR 74	CATALOG END ERROR
ERR 75	COMMAND ONLY (NOT PROGRAMMABLE)
ERR 76	MISSING < OR > (PLOT STATEMENT)
ERR 77	STARTING SECTOR > ENDING SECTOR
ERR 78	FILE NOT SCRATCHED
ERR 79	FILE ALREADY CATALOGED
ERR 80	FILE NOT IN CATALOG
ERR 81	/XYY DEVICE SPECIFICATION ILLEGAL
ERR 82	NO END OF FILE
ERR 83	DISK HARDWARE ERROR
ERR 84	NOT ENOUGH MEMORY FOR MOVE OR COPY
ERR 85	READ AFTER WRITE ERROR
ERR 86	FILE NOT OPEN
ERR 87	COMMON VARIABLE REQUIRED
ERR 88	LIBRARY INDEX FULL
ERR 89	MATRIX NOT SQUARE
ERR 90	MATRIX OPERANDS NOT COMPATIBLE
ERR 91	ILLEGAL MATRIX OPERAND
ERR 92	ILLEGAL REDIMENSIONING OF ARRAY
ERR 93	SINGULAR MATRIX
ERR 94	MISSING ASTERISK
ERR 95	ILLEGAL MICROCOMMAND OR FIELD/ DELIMITER SPECIFICATION
ERR 96	MISSING ARG 3 BUFFER
ERR 97	VARIABLE OR ARRAY TOO SMALL
ERR 98	ILLEGAL ARRAY DELIMITERS
ERR=1	MISSING NUMERIC ARRAY NAME
ERR=2	ARRAY TOO LARGE
ERR=3	ILLEGAL DIMENSIONS

GLOSSARY

argument	a syntax entity whose value is used to determine the value of some function.
collating sequence	any logical sequence used to order items of data.
exponent	in the floating-point representation of a number, those digits following 'E' which represent the power of 10 to which the base is raised; exponents are of the form: E digit [digit] or E $\{\pm\}$ digit [digit].
device address	hhh where h = hexdigit.
digit	any integer from 0 through 9.
function	an entity valid in BASIC that assigns a value or values to a variable (or to the elements of an array) on the basis of a definition provided by the system microcode.
hexdigit	a digit in the hexadecimal numbering system; the integers 0 through 9 and the letters A through F.
image	the representation of the form in which data are to be output.
letter	any letter of the alphabet from A through Z.
mantissa	in the floating point representation of a number, all characters except E and the digits of the exponent.
operand	that which is operated upon.
operator	that which indicates action to be performed.
parameter	a variable given a constant value for a specific purpose.

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.

700-3038G

TITLE OF MANUAL: **WANG BASIC LANGUAGE REFERENCE MANUAL**

COMMENTS:

Fold

Fold

(Please tape. Postal regulations prohibit the use of staples.)

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: Technical Writing Department

Fold

Cut along dotted line.

Printed in U.S.A.

ALPHABETICAL INDEX

WANG LABORATORIES (CANADA) LTD.
49 Valleybrook Drive
Don Mills, Ontario M3B 2S6
TELEPHONE (416) 449-2175
Telex: 069-66546

WANG EUROPE, S.A.
Buurtweg 13
9412 Ottergem
Belgium
TELEPHONE 053/704514
Telex: 26077

WANG DO BRASIL COMPUTADORES LTDA.
Rua Barao de Lucena No. 32
Botefogo ZC-01 20,000
Rio de Janeiro RJ
Brasil
TELEPHONE 226-4326
Telex: 2123296

WANG COMPUTERS (SO. AFRICA) PTY. LTD.
Corner of Allen Rd. & Garden St.
Bordeaux, Transvaal
Republic of South Africa
TELEPHONE (011) 48-6123

WANG INTERNATIONAL TRADE, INC.
836 North Street
Tewksbury, Massachusetts 01876
TELEPHONE (617) 851-4111
TWX 710-343-6769
Telex: 94-7421

WANG SKANDINAVISKA AB
Pyramidvaegen 9A
S-171 36 Solna,
Sweden
TELEPHONE 08-826814
Telex: 11498

WANG NEDERLAND B.V.
Damstraat 2
Utrecht, Netherlands
(030) 93-09-47
Telex: 47579

WANG PACIFIC LTD.
902-3 Wong House
26-30, Des Voeux Road, West
Hong Kong
TELEPHONE 5-435229
Telex: 74879

WANG INDUSTRIAL CO., LTD.
110-118 Kuang-Fu N. Road
Taipei, China
TELEPHONE 784181-3
Telex: 21713

WANG GESELLSCHAFT M.B.H.
Formanekgasse 12-14
A-1190 Vienna, Austria
TELEPHONE 36.32.80
Telex: 74640

WANG S.A./A.G.
Markusstrasse 20
CH-8042 Zurich 6
Switzerland
TELEPHONE 41.1.26.6866
Telex: 59151

WANG COMPUTER PTY. LTD.
55 Herbert Street
St. Leonards, 2065
Australia
TELEPHONE 439-3511
Telex: 25469

WANG ELECTRONICS LTD.
1 Olympic Way, 4th Floor
Wembley Park,
Middlesex, England
TELEPHONE 01/903/6755
Telex: 923498

WANG FRANCE S.A.R.L.
Tour Gallieni, 1
78/80 Ave. Gallieni
93170 Bagnolet, France
TELEPHONE 33.1.3602211
Telex: 68958F

WANG LABORATORIES GMBH
Moselstrasse 4
6000 Frankfurt AM Main
West Germany
TELEPHONE (0611) 252061
Telex: 04-16246

WANG DE PANAMA (CPEC) S.A.
Apartado 6425
Calle 45E, No. 9N. Bella Vista
Panama 5, Panama

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.
700-3038G
2-76-5M
Price $15.00