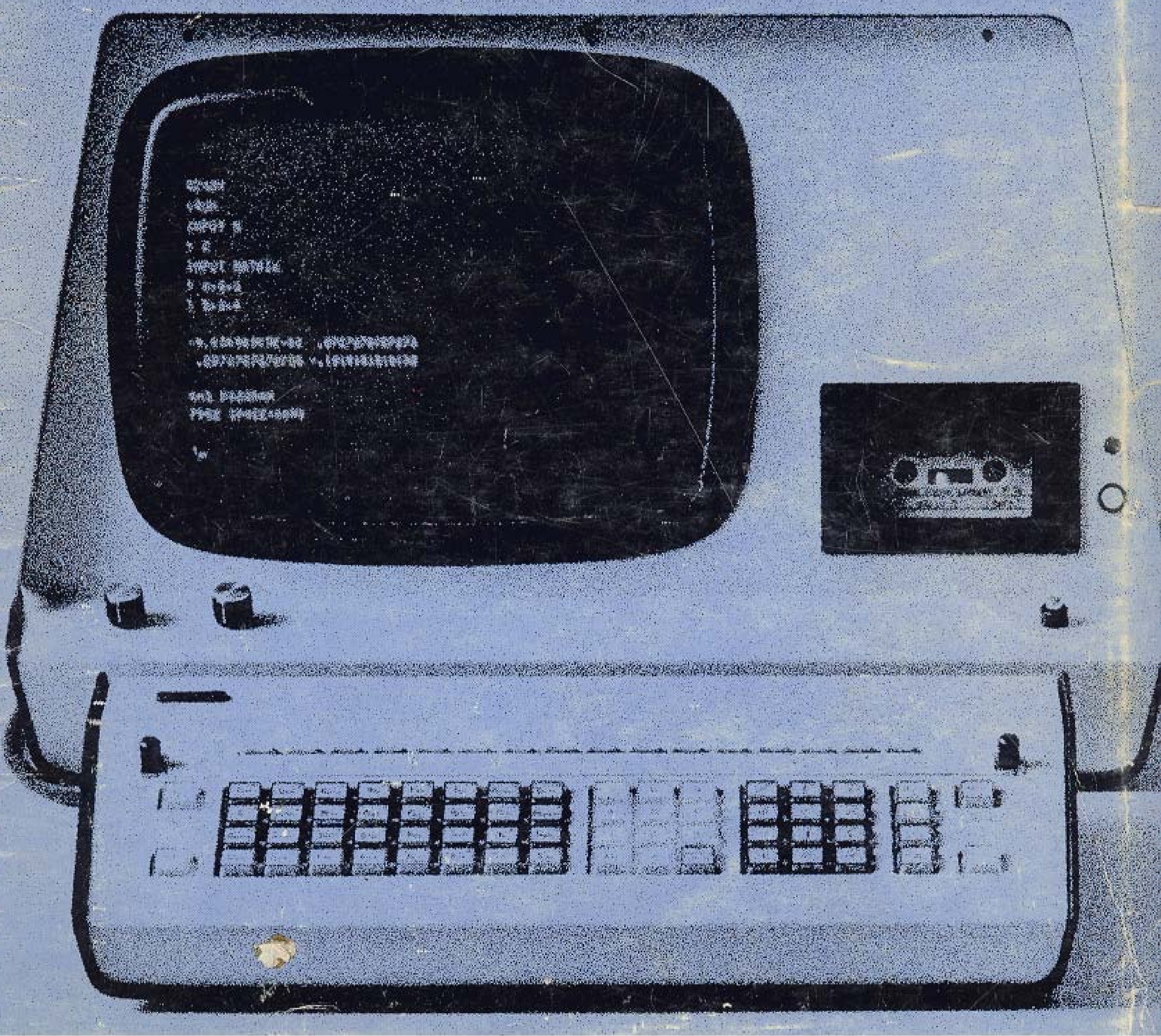
WANG

2200 A/B REFERENCE MANUAL

19

SYSTEM 2200



ALPHABETICAL INDEX

2200 A/B Reference Manual

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

HOW TO USE THIS MANUAL

This manual has been written for the sole purpose of providing quick answers to questions concerning the operation of the System 2200A/B. It is designed for users who are already quite familiar with the System 2200 and its BASIC language instruction set.

The manual is divided into fifteen sections covering all the operational features of the System 2200A/B. The BASIC 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 2200A/B and its BASIC language for the first time, we strongly recommend you first read the Self-Teaching Programming Manual which discusses in detail the operational and programming features of the System 2200A/B.

Once you have completed the Self-Teaching Programming Manual, then and only then should you refer to this manual as a reference guide to individual questions concerning the operation of the System 2200A/B.

INTRODUCTION

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

The manual is divided into fifteen sections, separated by tabs, for ease of section location. The title page for each section has a listing of the contents of the section. Also, a complete Table of Contents is located in the front of the book.

Section I Introduces you to the Model 2216 CRT Executive Display; the System 2200 Central Processing Unit (CPU); the two keyboards, Model 2215 BASIC Keyword Keyboard and Model 2222 Alpha-Numeric Typewriter Keyboard; along with the Model 2219 I/O Extended Chassis. Unpacking, installation and turn-on procedures also are illustrated.

Section II The basic structure and components of the system are covered in this section, such as: 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, Arithmetic Symbols, Numeric Constants, Math Functions, Common Variables, Random Numbers, User Functions and Rational Functions.

Section IV Alphanumeric capabilities are covered in this section, such as: Alpha Strings, Variables, Literal Strings, Alpha Functions, Hexadecimal Literal Strings, Length 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 needed to effectively utilize the system are covered here, arranged in alphabetical order.

Section VIII This section describes the various statements which are used to perform bit and byte, and data conversion operations. All statements are arranged in alphabetical order.

Section IX This section describes the use of tape cassettes, along with file operation techniques.

Section X Describes the various statements needed to effectively operate the plotters in the System 2200A/B.

Section XI Describes the general operating instructions, commands, and statements needed to operate the Mark Sense Card Reader.

Section XII This section describes the Paper Tape Reader, used only with the System 2200B.

Section XIII A description of the various commands and statements used to operate a Teletype interfaced to a System 2200.

Section XIV 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.

Section XV This last section, titled Appendices is divided into four subsections: Appendix A, Specifications; Appendix B, Peripherals; Appendix C, ASCII Codes and what the various codes generate; and Appendix D, a list of the various error messages by title and code.

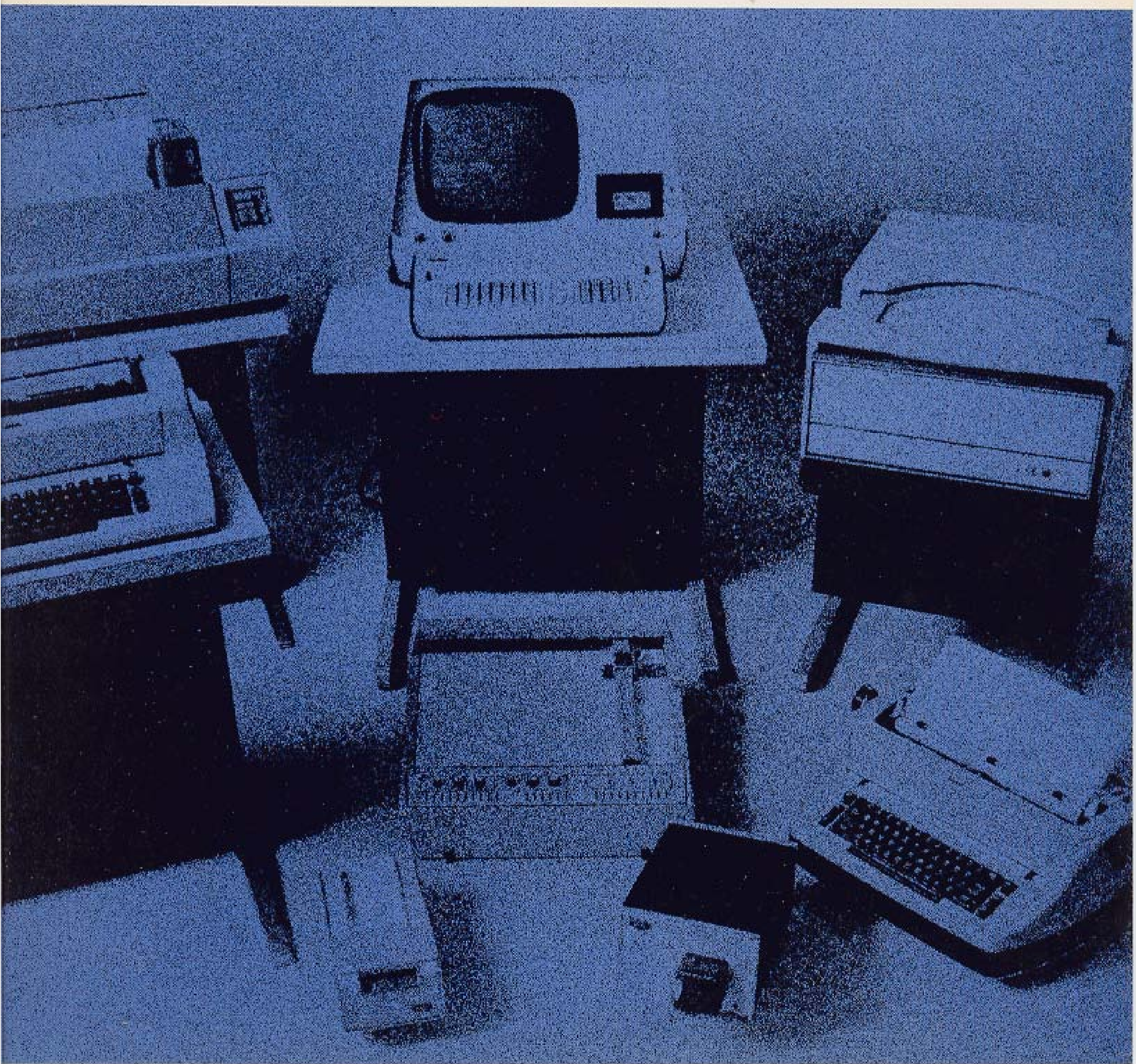
TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

SECTION I
GENERAL SYSTEM INTRODUCTION



Section I

General System Introduction

UNPACKING AND INSPECTION

Carefully unpack your equipment and inspect all units for shipping damage. If damage is noticed, do not proceed. Notify the shipping agency. Check equipment received against the purchase order. Decals specifying model numbers can be found on all Wang equipment, usually on the back of each unit.

After unpacking and verifying the status of your equipment, the following procedures are used to install and turn on your 2200 System.

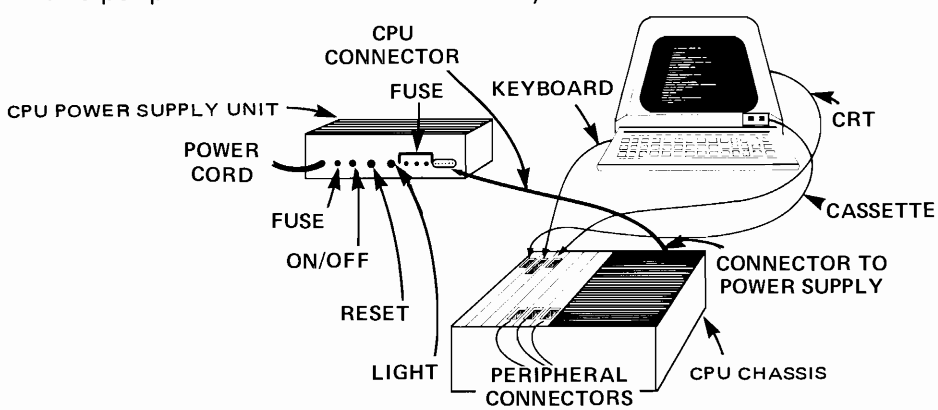
The basic component of the 2200 is the Central Processing Unit (CPU). All other additional pieces of equipment are considered peripherals and are attached to the CPU. The CPU is divided into two parts. The main CPU chassis houses the processor, memory and peripheral connectors. A smaller power supply unit contains the power supply and also 'Power On' and RESET buttons.

INSTALLATION

To install your 2200 System, use the following procedure:

1. Plug all peripherals into CPU chassis. Each peripheral connector on the CPU is labeled for the appropriate device. After each cord is plugged in, make sure the lock clips are snapped in.
2. Plug any peripheral power cords into wall outlets.
3. Plug the main power cord of the CPU chassis into Power Supply Unit, plug the Power Supply Unit into a wall outlet.

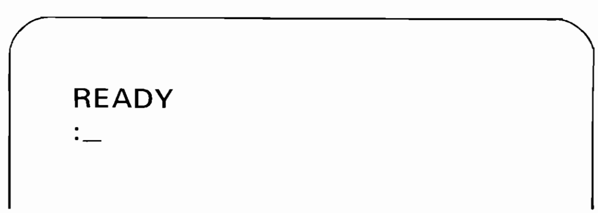
A maximum of 6 peripherals can be attached directly to the standard CPU in this manner.



TURN-ON PROCEDURE

Use the following procedure to turn ON your 2200 System:

1. Turn power switches ON on all peripherals (including CRT).
2. Move the main power switch on Power Supply Unit to the ON position (light on Power Supply Unit illuminates). This process Master Initializes the system.
3. The CRT display appears as illustrated below.

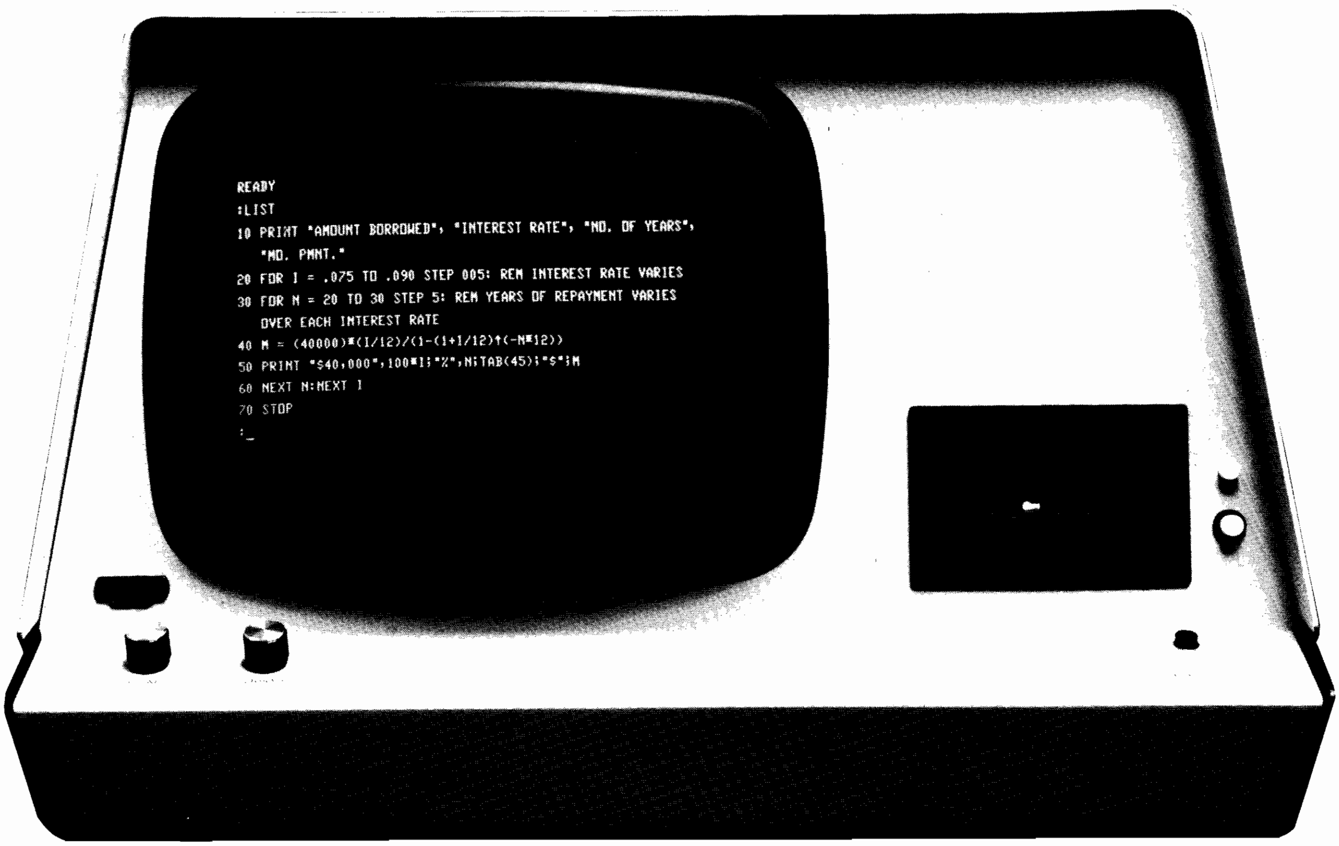


Your 2200 system is now ready to use.

If a system failure should occur, try to restore operation by touching the RESET button on the keyboard or Power Supply Unit. If normal operation is not restored, master initialize the system by turning the power OFF, then ON (power ON/OFF switch on Power Supply Unit). If the system is still non-functional, repeat the installation procedure before calling your Wang Service Representative.

2216 CRT DISPLAY

The CRT display is designed to enable the user to easily write, review, modify, and correct programs. The CRT is composed of an 8 X 10.5 inch screen, and two controls used to set the brightness and contrast of the output as it appears on the screen. The screen itself has a maximum of 16 lines, each 64 characters in length. The CRT display functions similar to a teletype type printer except that 16 lines can be displayed at a time. Lines are displayed sequentially on the screen, each terminated by a carriage return and line feed character. If more than sixteen lines are given at any one time, each new line is added to the bottom of the CRT, moving the previously entered lines up; the line at the top of the CRT display is replaced by the line directly beneath it.



The following CRT commands are issued by outputting the specified code by a PRINT HEX (code); statement.

HEX CODE	COMMAND
01	cursor home
03	clear screen & cursor home
07	bell (CRT option)
08	cursor left (←)
09	cursor right (→)
0A	cursor down (↓)
0C	cursor up (↑)

For example, PRINT HEX(03); clears the CRT screen.

Cleaning the CRT Screen

The CRT screen should be cleaned periodically with a mild soap and water using a soft cloth. Do not use an alcohol pad which might cause damage to the black surface surrounding the screen.

> **WARNING**
> Do not attempt to remove the cover for *any reason* due to the danger of high voltage. Call a Wang Service Representative if any maintenance is required.

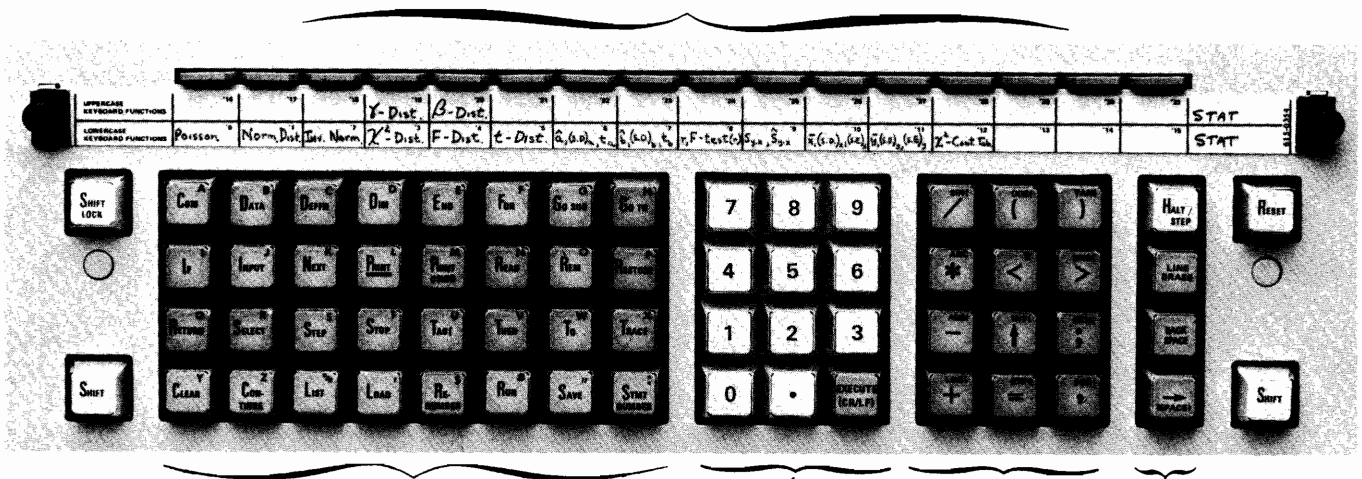
2215 BASIC KEYWORD KEYBOARD

The 2215 keyboard permits most BASIC language words to be entered by single keystrokes. For example, pressing the [L / PRINT] key causes the entire word "PRINT" to be entered.

SHIFT Uppercase characters can be entered into the system by touching one of the two SHIFT keys and then touching the key containing the desired symbol or function. When a SHIFT key is depressed, a SHIFT light goes on until another key is touched, then it goes off. The SHIFT LOCK key (upper left corner) causes the SHIFT to remain on while any number of upper case keys are entered; the SHIFT can be subsequently turned off by touching either SHIFT key. Alternatively the SHIFT key can be held down as on a typewriter, if several uppercase characters are to be entered.

The keyboard is divided into 5 zones.

ZONE 5
SIXTEEN USER DEFINED SPECIAL FUNCTION KEYS



ZONE 1
BASIC LANGUAGE KEYBOARD KEYS AND ALPHA AND SPECIAL CHARACTERS

ZONE 2
NUMERIC ENTRY KEYS

ZONE 3
ARITHMETIC OPERATORS MATH FUNCTIONS, PUNCTUATION SYMBOLS

ZONE 4
EDIT AND ERROR CORRECTION KEYS

ZONE 1 The first zone contains the alphabetic and special characters, most BASIC language words, and the statement number generator key.

STMT NO	. . . automatically sets the statement number of the next line about to be entered, equal to the highest line number of the user program in the system +10.

ZONE 2 The second zone consists of the numeric entry keys and the EXECUTE-CR/LF key.

EXECUTE (CR/LF)	. . . causes the line just keyed in to be entered and processed by the system.

ZONE 3 Zone three contains the arithmetic operators, mathematical functions and punctuation keys.

ZONE 4 Zone four consists of the following special keys, used for entry and system control:

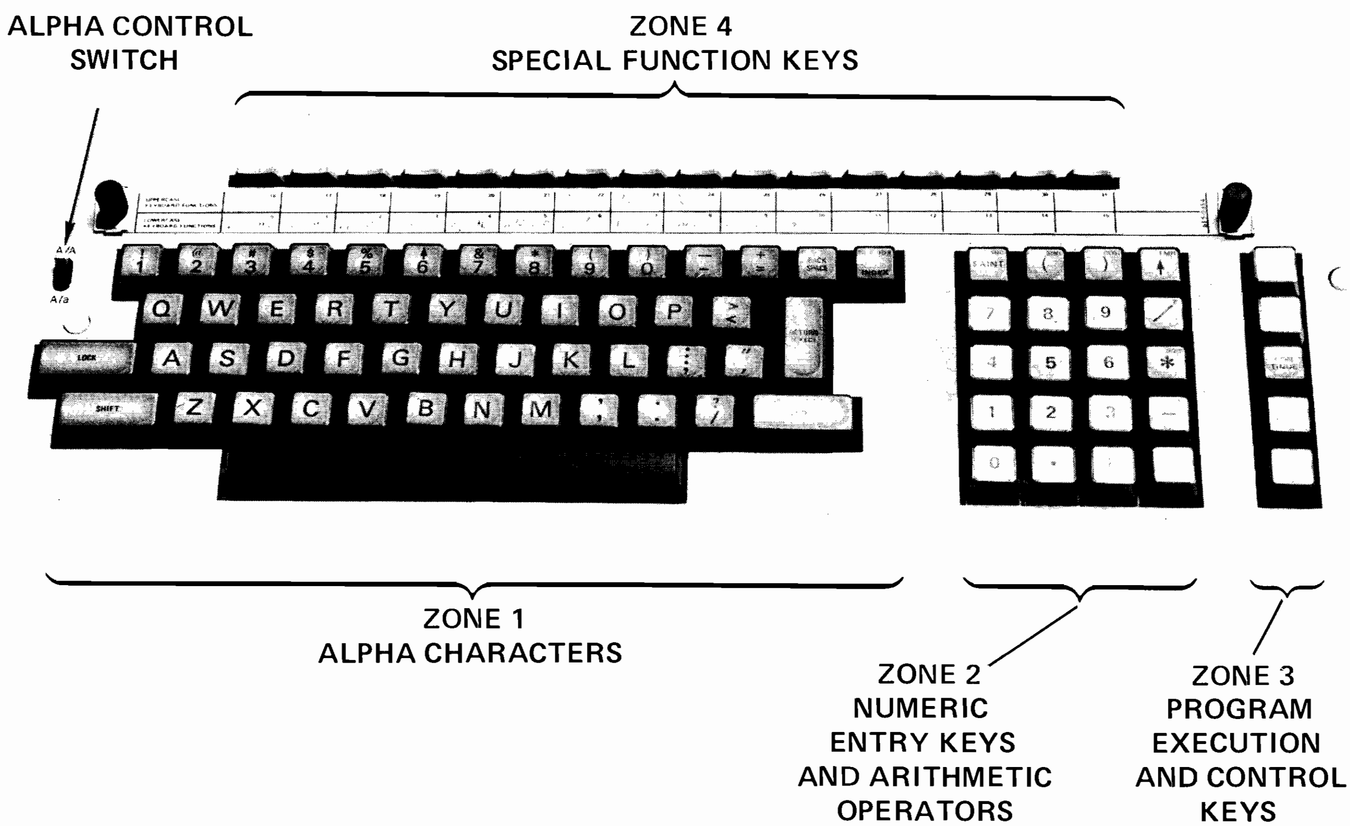
RESET	. . . immediately stops program listing or execution, clears the CRT screen, and returns control to the user; leaving program text and variables intact.
HALT/ STEP	. . . causes program to halt or execute one line at a time each time the key is touched.
LINE ERASE	. . . deletes the line currently being entered.
← (BACK)	. . . backspace – deletes the result of the last keystroke entered.
→ (SPACE)	. . . enters a space character.

ZONE 5 Zone five consists of 16 user defined special function keys for access of up to 32 subroutines or text entry operations.

MODEL 2222 ALPHA-NUMERIC TYPEWRITER KEYBOARD

The 2222 keyboard is designed for users who are already familiar with a standard selectric typewriter, or for those users whose applications require large amounts of alpha input.

The 2222 keyboard is divided into four major zones which are, in some respects, similar to the zones of the 2215; however, the differences lie in the way BASIC words are generated. With the 2222 most BASIC language words must be keyed in one character at a time (similar to a typewriter). This is compared to the keyword section of the 2215 where one keystroke can generate an entire word. Either way, however, takes up the same amount of space in memory.



ZONE 1 Zone 1 of the 2222 keyboard is very similar to a regular selectric typewriter keyboard, which includes all alpha characters, both upper and lowercase, numbers 0-9, and all of the typical special characters.

ALPHA CONTROL SWITCH

An integral part of Zone 1 is the addition of an Alpha Control Switch. The reason for this switch is to more easily write programs in BASIC. This switch acts somewhat similar to a shift key, however, the switch only conditions alpha characters to always be upper case and in no way interferes with the other keys on the keyboards.

DOWN POSITION

In the down position the keyboard acts as a standard typewriter keyboard.

UP POSITION

In the up position the keyboard conditions the system to generate all uppercase alpha characters regardless of the position of the shift key. This is just for the 26 alpha keys and in no way does this condition change the input capabilities of the other keys on the keyboard. For uppercase keys other than alpha characters, the shift key must be used. This would be the normial position setting when entering BASIC programs, since BASIC statement words and variables require uppercase alphabetic characters.

RETURN (EXEC.) . . . causes the line just keyed in to be entered and processed by the system.

BACK SPACE . . . deletes the result of the last keystroke entered.

ZONE 2 Zone 2 contains all the numeric entry keys and arithmetic operators, along with a number of math functions. Immediate mode calculations can be generated using the PRINT key followed by a legal calculating expression. This set of 20 keys is generally considered a "scratch pad" calculator for immediate mode calculations; however, these keys can be used to enter program line numbers, numbers and functions.

ZONE 3 Zone 3 consists of the following special keys used for entry and system control.

RESET . . . immediately stops program listing or execution, clears the CRT screen, and returns control to the user; leaving program text and variables intact.

HALT/ STEP . . . causes program to halt or execute one line at a time each time the key is touched.

LINE ERASE . . . deletes the line currently being entered.

CON- TINUE . . . continues program execution after a "STOP" verb has been executed, or the "HALT/STEP" key has been touched.

RUN . . . initiates execution of the user's program.

> **NOTE:**
>
> *CONTINUE and RUN must be followed by RETURN(EXEC).*

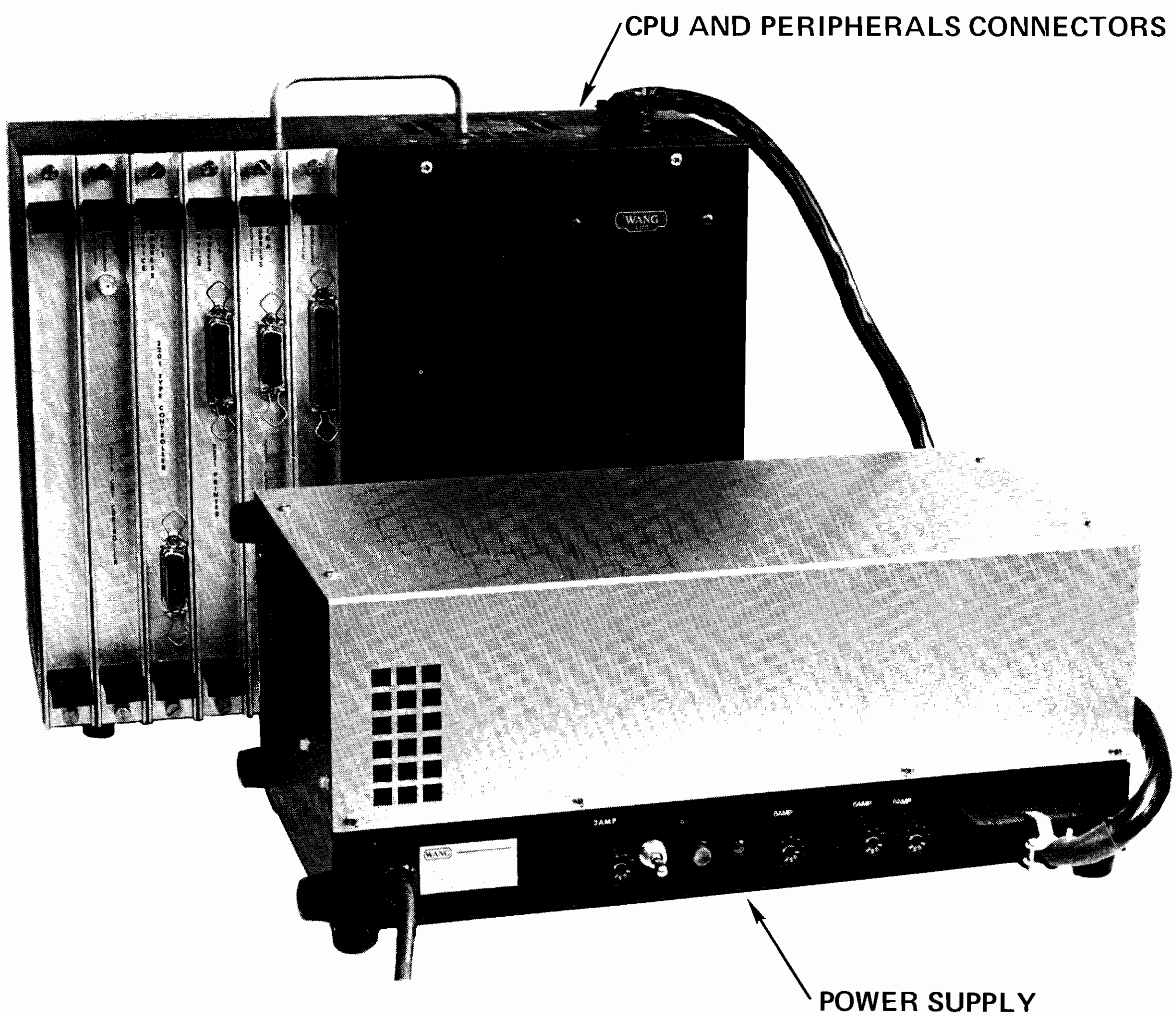
ZONE 4 Zone 4 consists of 16 user defined special function keys for access of up to 32 subroutines or text entry operations.

2200 CENTRAL PROCESSING UNIT (CPU)

The standard 2200-1 Central Processing Unit (CPU) has a user memory (RAM) of 4096 (4K) bytes (8-bit words). This can be increased in increments of 4K up to a maximum of 32K, self-contained in the 2200 chassis.

An outstanding feature of the 2200 system is that the BASIC language compiler is hardwired in a separate section of the calculator, allowing nearly* the entire memory to be accessed by the user.

The CPU contains slots for up to 6 I/O peripheral devices. If more than six peripherals are required, a 2219 I/O Extension Chassis can be used which provides an additional 5 I/O peripheral connector slots.



*Approximately 700 bytes are used for "housekeeping" purposes.

2219 I/O EXTENDED CHASSIS

The peripheral capacity of the System 2200 can be extended to meet the needs of almost any user. The basic system is composed of a CRT, Tape Cassette Drive and Keyboard, leaving three peripheral connectors for other devices (see Figure 1).

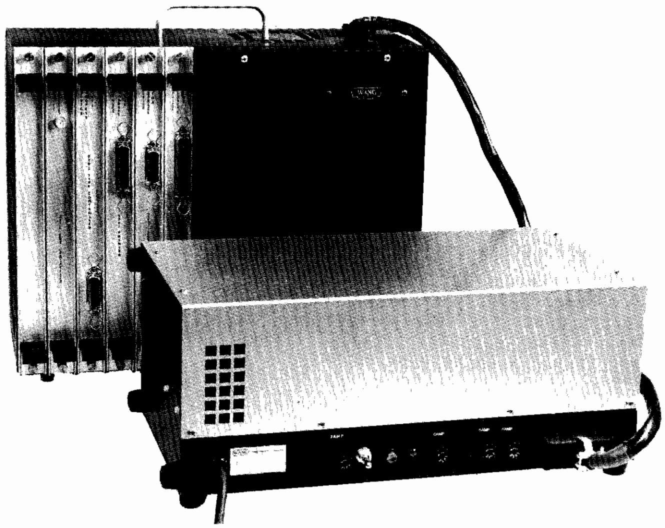


FIG. 1

There are, however, requirements for more than six peripheral devices; when this occurs, the Model 2219 I/O Extended Chassis is used. This adds to the system an additional 5 peripheral connectors by providing a larger CPU chassis. See the illustration below on the installation setup.

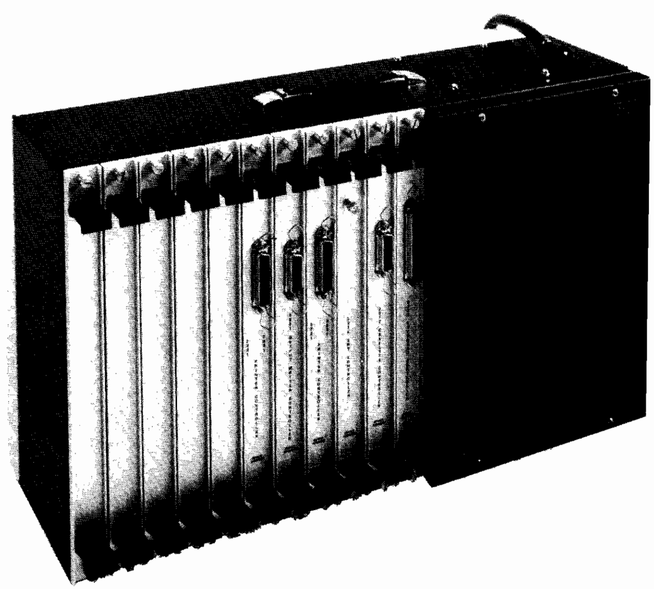
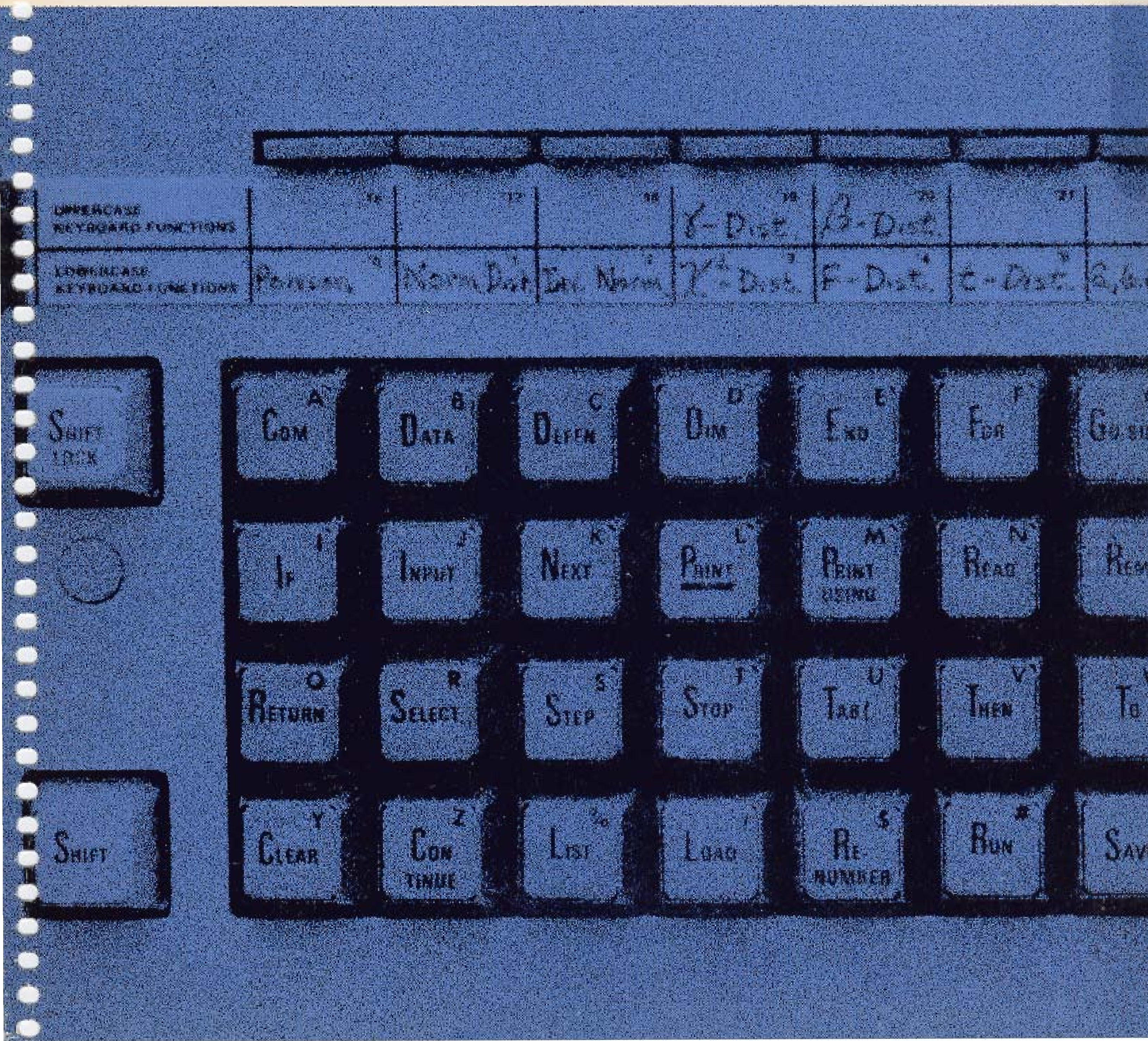


FIG. 2

In Fig. 2 the user has the capability of installing 11 peripheral devices. Any system that has more than 6 peripherals must utilize a 2219 I/O Extended Chassis.

SECTION II
BASIC LANGUAGE STRUCTURE



Section II

BASIC Language Structure

Section II BASIC Language Structure

INTRODUCTION

A BASIC program must have a certain structure - simple though it is. The rules are few and easy to follow. Certain components should be used in the structure of a program. These components include allowable characters, kinds of symbols, and various functions that can be used in BASIC.

LINE NUMBER

Every program line must begin with a line number. It may be 1 to 4 digits in length. Line numbers identify the lines and specify the order in which the program lines are to be executed. These lines do not have to be entered in sequential order; the BASIC system automatically arranges and processes the lines in order according to the line number. Line numbers should be assigned with a suitable increment between consecutive lines for the insertion of additional lines. Line numbers can be entered by pressing the STATEMENT NUMBER key (2215 keyboard only) which automatically generates a new line number, or by manually keying in the digits in the line number. Line numbers must not be preceded by spaces.

BASIC WORDS

BASIC words (i.e., PRINT, NEXT, SAVE, TO) can either be entered as single keystroke entries by pressing the appropriate key or by typing in each character in the word. In either case only 1 byte of memory is required to store the word.

BASIC STATEMENT LINES

Each statement lines is comprised of a line number and at least one statement. A series of statements, separated by colons, may be entered on the same line - with one line number.

Example:

40 X – 2 :Y – 3 :PRINT X, Y

There are two types of statements:

1. An executable statement specifies the action to be performed.

Example:

Q = 8*Y

2. A nonexecutable statement provides information

Example:

DATA 2, –7, 5

One statement line cannot exceed 192 keystrokes.

SPACING

Spaces are customarily used between characters in a program line for readability; the system ignores them. For example, 10 READ A, B, C, D is easier for the programmer to read than 10READA,B,C,D; both, however, are equally clear to the BASIC system. The condensed format conserves user text area space.

COLON

The colon (:) is displayed by the system to indicate that the programmer may proceed to enter program lines. This symbol is also useful for identifying lines in the program listing - those preceded by a colon were entered by the user; all others were system output.

IMMEDIATE MODE

The Wang 2200 BASIC system provides for two modes of operation, PROGRAM and IMMEDIATE.

The IMMEDIATE mode allows the 2200 to be used as a powerful one-line calculator. The BASIC statements are entered with no preceding line numbers. The absence of a line number causes the system to check the line for grammatical correctness and, if no errors exist, to immediately execute the statements in the line. The line is not saved and requires only temporary storage space.

Multi-Statement Immediate Mode Lines

When using more than one BASIC statement on a line, a colon (:) must be placed between each statement. The ability to place several statements on a single line makes the immediate mode a very powerful calculating tool.

Example:

Key IN **FOR I=1 TO 10: PRINT I, LOG(I) :NEXT I CR/LF-EXECUTE**

Ten values of I and LOG(I) would be printed immediately.

PROGRAM MODE

The PROGRAM mode requires each line to be preceded by a line number of from 1 to 4 digits. The presence of the line number causes the system to check the line for grammatical correctness, store the line and await further instructions from the user. In this way, an entire program can be entered line by line, checked for syntax errors, and then saved, listed, or executed by the user.

CR/LF-EXECUTE KEY

RETURN EXEC	OR	EXECUTE CR/LF
2222		2215

Purpose

The CR/LF-EXECUTE key is used in both the immediate mode and the program mode. It must terminate every line of input to the system. When entered, it causes the following:

1. IMMEDIATE MODE - If the statement line does not have a line number in front of it, the line is checked for BASIC grammatical correctness and, if found to be correct, the line is immediately executed.
2. PROGRAM MODE - If the statement line has a line number in front of it, the line is checked for BASIC grammatical correctness and entered into the 2200 memory.
3. COMMANDS - The command is checked for BASIC grammatical correctness and executed.

> **NOTE:**
>
> *If a syntax error is found in either mode the appropriate error code is displayed along with an up arrow symbol pointing out the error. The system then returns control to the user by displaying a colon on the CRT display.*

ILLEGAL IMMEDIATE MODE STATEMENTS

DATA	INPUT	RETURN
DEFFN	KEYIN	STOP
GOSUB	PRINTUSING	% (IMAGE statement)
IF	READ	
	RESTORE	
	IF-END THEN	
	ON	

DEBUGGING AND EDITING FEATURES

Debugging a program on any system can often be a difficult and time-consuming job. The special edit and debug features of the Wang 2200 combined with the sixteen line visual CRT display help make this task much easier.

Character Erasing

Single keystroke entries in the current text line can be removed by touching the backspace key while in lowercase [BACK SPACE] (2222) or [← (BACK)] (2215)

Example:

:120 X=SQR (2+COS(17

Key [BACK SPACE] 4 Times

:120 X=SQR(2

correct remainder of line

:120 X = SQR(2 – COS(17))

Removing the Current Line

The line currently being entered can be removed from the screen by touching the [LINE ERASE] key.

Example:

:300 PRINT "RESULT": A(4 –

Key [LINE ERASE]

: _

Deleting a Line

A previously entered text line can be deleted by keying the line number of that line and the CR/LF-EXECUTE key.

Example:

```
        READY
        :LIST
        10A = 14
        20   PRINT A
        .
        .
        .
Key     20   CR/LF-EXECUTE
Key     LIST CR/LF-EXECUTE
        :LIST
        10A = 14
        :_
```

Replacing a Line

An existing line can be replaced by entering the same line number followed by the new line and CR/LF-EXECUTE.

Renumbering a Program

A program can be renumbered by using the RENUMBER command, so that spaces can be made between closely numbered lines in order to insert additional lines of text.

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

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 printed result 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.

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 TRACE OFF statement is executed. 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.

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
                  :_
```

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 TRACE OFF statement is executed. When in the TRACE mode, printouts will be produced when:

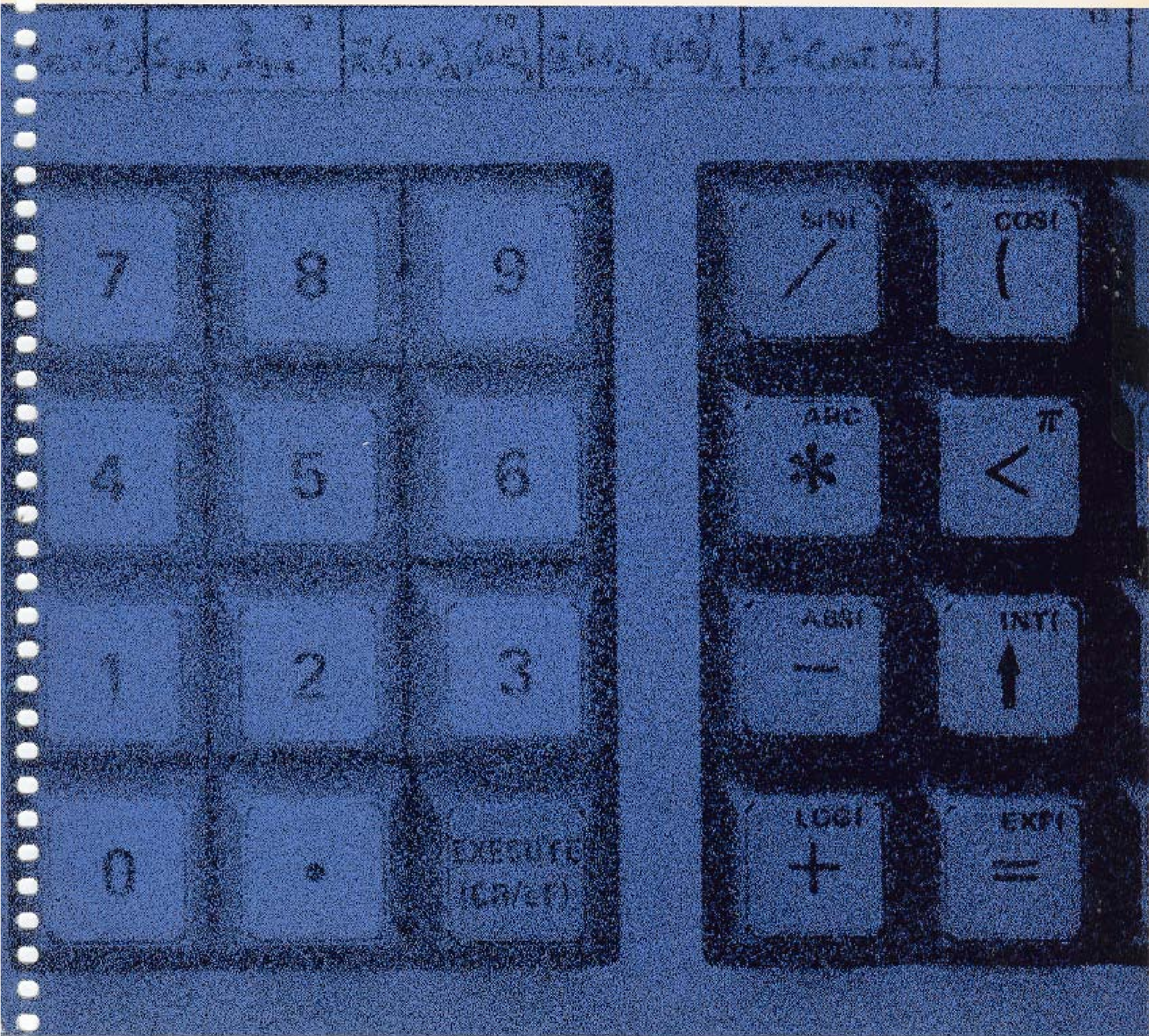
Pause

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. A pause is generated 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 (i.e., 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
:_
```

SECTION III
NUMERIC EXPRESSIONS



Section III

Numeric Expressions

Section III Numeric Expressions

EXPRESSIONS

An expression may be a variable, a function or a constant or any valid combination of variables, functions, and constants connected by arithmetic symbols. An expression may be preceded by plus or minus and may be contained within parentheses. The following examples illustrate BASIC expressions:

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

These are all expressions

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.

There are two types of numeric variables: scalar and array. A scalar numeric variable is designated by a letter or a letter followed by a digit: there are 286 legal scalar variable names.

Example:

A,A4

Array variables are used to define the elements of an array. These variables are used when a single subscript or a double subscript might ordinarily be used.

$(a_1, a_2, a_3, \ldots)$ or by b_{ij}

A numeric array variable consists of a letter or a letter followed by a digit which is the array name, followed by subscripts in parentheses:

A(3), C3(5), B(2,3), D(N, M–2), E1(5), F3(N,M)

For all array variables, the DIM statement is used with the array name and the numeric value subscripts to provide space and specify the dimensions of a complete array of one or two dimensions. The DIM statement must precede the first reference to the variables.

Example:

```
READY
:20  DIM Q(25)      defines the 1-dimensional array Q with 25 elements
:30  READ N
:40  FOR I = 1 to N
:50  READ Q(I)
:55  PRINT Q(I)
: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.

The following rules apply to the use and assignment of array variables:

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 cannot exceed 255.

An array variable and a scalar variable may have the same name; they are independent, unrelated variables. Single subscripted and double subscripted arrays may 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 onto the next program. Common variables are cleared from memory when a CLEAR or CLEAR V command is executed.

Section III Numeric Expressions

The following arithmetic symbols are used in BASIC to write a formula. Operations are executed in sequence from the highest level to the lowest level: (1) operations within parentheses, (2) raising a number to a power, (3) multiplication and division, and (4) addition and subtraction.

SYMBOL	SAMPLE FORMULA	EXPLANATION
↑	A↑B	Raise A to the power of B.
*	A*B	Multiply B by A.
/	A/B	Divide A by B.
+	A+B	Add B to A
-	A-B	Subtract B from A

RELATIONAL SYMBOLS

Relational symbols are used with the IF verb when values are to be compared before processing. For example: 20 IF G < 10 THEN 63 means that if G is less than 10, processing continues at program line 63.

The following relational symbols may be used with 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

USER FUNCTIONS

A user function is a mathematical function of a single variable, which is used several times within a program. Such a function is defined by a DEFFN statement. The format of the function is a letter or a digit, a scalar variable in parentheses, an equals sign, and an expression. (i.e., Y(X) = 2 * X ↑ 2 + 3 * X – 7). A function could be used in a program as follows: The function is defined: 30 DEFFN E (Z1) = EXP (-Z1↑3+5). If the following statement is entered, 40 Q = A/B + FNE(10), the value of 10 is assigned to Z1; the result, EXP (–10↑3+5) will be used in place of the referenced FNE(10) in program line 40.

Section III Numeric Expressions

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 number format error. The following are examples of numeric constants in BASIC:

4, −10, 1432443, −.7865, 24.4563

If the exponential notation, E, is used, the value of the constant is equal to the number to the left of the E multiplied by 10 to the power of the number to the right of the E. For example, 4.5E7 indicates that 4.5 to be multiplied by 10^7.

The magnitude of a numeric constant can be anywhere between 10^{-100} and 10^{+100}.

Invalid Use of Scientific Notation

8.7E5.8	Not valid because of the illegal decimal form of the exponent.
−103.2E99	Not valid because in reduced form, it is equivalent to −1.032E101, an exponent greater than E100.
.87E−99	Not valid because it is equivalent to 8.7E−100, which is less than E−100.

Section III Numeric Expressions

MATHEMATICAL FUNCTIONS

Keyboard Function	Meaning	Example
*SIN(expression)	Find the sine of the expression	SIN(π/3) = .8660254037841
*COS(expression)	Find the cosine of the expression	COS(.693↑2) = .8868799122686
*TAN(expression)	Find the tangent of the expression	TAN(10) = .6483608274585
*ARC SIN(expression)	Find the arcsine of the expression	ARC SIN (.003) = 3.00000450E-03
*ARC COS(expression)	Find the arccosine of the expression	ARC COS (.587) = .943448079441
**ARC TAN(expression)	Find the arctangent of the expression	ARC TAN (3.2) = 1.267911458422
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)	Take the greatest 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)= 8.023552392402
EXP(expression)	Find the value of e raised to the value of the expression	EXP(.33*(5-6))= .7189237334321
SQR(expression)	Find the square root of the expression	SQR(18+6)=SQR(24)= 4.8989794856

*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	–selects degrees for all following calculations.
SELECT	R	CR/LF–EXECUTE	–selects radians for all following calculations.
SELECT	G	CR/LF–EXECUTE	–selects grads for all following 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:

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

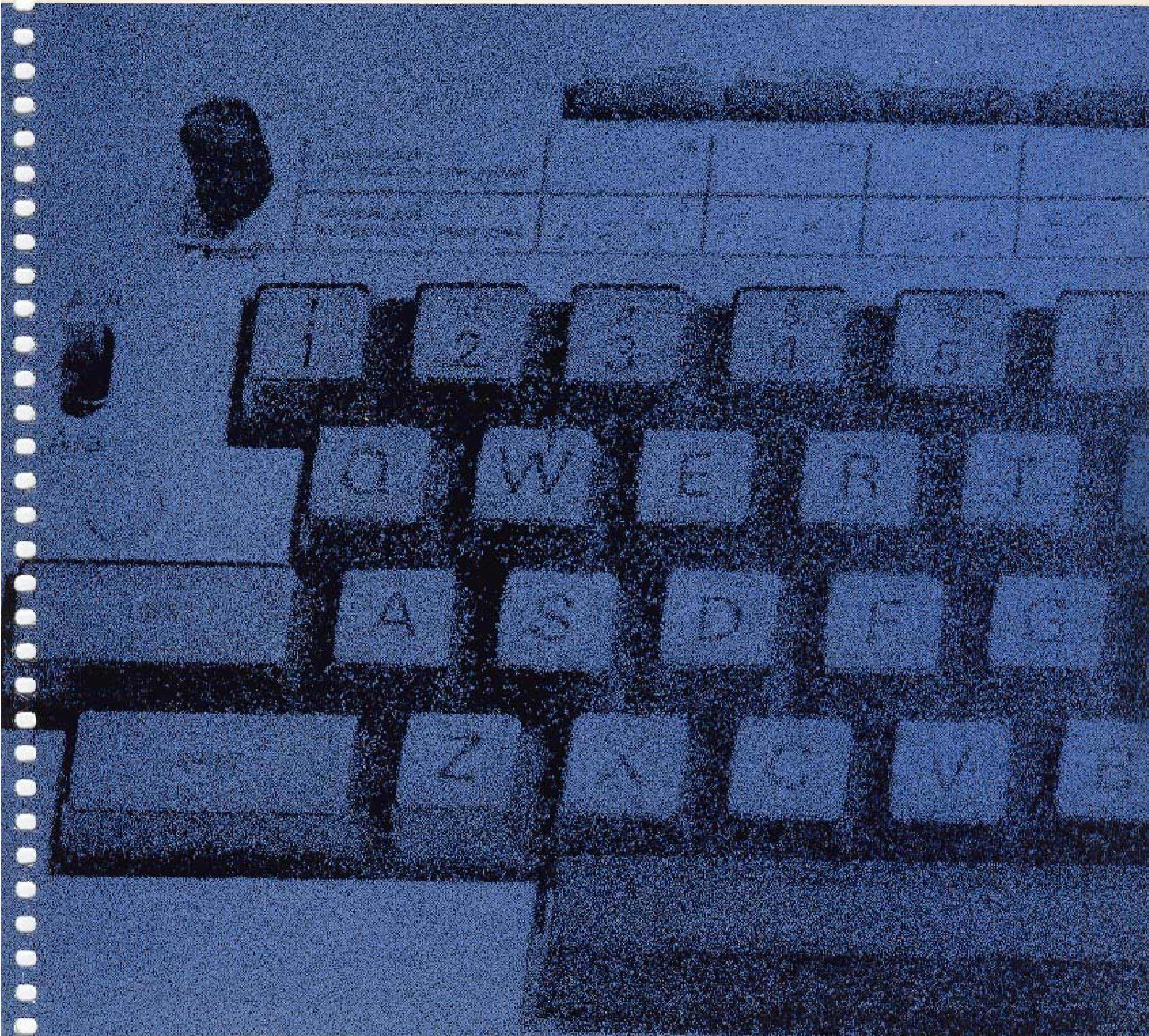
Whenever the system is master initialized (Power On), the random number generator is initialized; the next time RND is used, the first random number in the list will be produced.

ADDITIONAL 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	Binary value of a string character.
LEN	Length of a string.

They are described in detail in Section VIII.

SECTION IV
ALPHANUMERICS



Section IV

Alphanumerics

Section IV Alphanumerics

ALPHANUMERIC STRING VARIABLES

The Wang 2200 provides for an additional form of variable, the alphanumeric string variable. It is distinguished from numeric variables 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'] $ (i.e., A$, B$, C1$)

One-dimensional alphanumeric string array variable
'letter' ['digit'] $ (d_1) (i.e., A$ (3), B$ (N))

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

where d_1 and d_2 are expressions whose values are $\geqslant$ 1 and less than 256.

Each string variable or string array element is initially assigned a value of 1 blank character. Thereafter, it can take the value and length of any alphanumeric character string up to its maximum length. The maximum length of a string variable is assumed to be 16 characters; however, the user may change the maximum length (up to 64) by using a DIM or COM statement. If a string variable receives a string value of less than its maximum length, it reflects that shorter length in all subsequent operations until it receives another value. The end of the alphanumeric value is assumed to be the last nonblank character (except when the value is all blanks, in which case the value is assumed to be one blank).

Example:

```
READY
:10  A$ = "ABC     "
:20  PRINT A$
:_
```

Execution of these statements would print "ABC" with no trailing spaces.

Hence, trailing blanks are not considered part of alphanumeric values.

ALPHANUMERIC LITERAL STRINGS

An alphanumeric literal string is a character string enclosed in double quotation marks. It is used in conjunction with string variables to provide a string value within a BASIC statement.

Example:

```
READY
:10  LET A$="ABCD"
:20  IF B$<"#XYZ" THEN 100
:30  PRINT "NAME=" ;A$
:_
```

When inputting data, the literal string need not be enclosed in quotes. In this case, commas and carriage returns act as string terminators and leading spaces are ignored; hence if commas or leading spaces are to be included in the literal string, the string must be enclosed in quotes.

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

Example:

```
LET A$="ABCDEFGHIJKLMNOPQRST"
```

In this statement A$ only receives the first 18 characters of the literal string (i.e., ABCDEFGHIJKL MNOPQR) if the maximum length of A$ is 18, otherwise it is set to 16 (see DIM; page 79).

EXAMPLES OF STATEMENTS USING STRING VARIABLES

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

LET	LET A$=B$(2) A$="ABCD"
IF	IF A$=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$
PRINT	PRINT A$,B$, "ABCD"
PRINTUSING	PRINTUSING 50,A$,B$, "LAST"
DATASAVE	DATASAVE A$, "GROUP1"
DATA	DATA "ABCD", "EFGH"

NOTE:

When comparing string variables with string literals or other string variables (i.e., IF A$ < "ABCD"), trailing spaces are ignored and only the values of the strings are compared.

STR (STRING) FUNCTION

Wang 2200 BASIC provides a function which permits the user to extract, examine, compare or replace a specified portion of an alphanumeric string. The STR function operates on alphanumeric string variables, and can be used in any BASIC statement where alphanumeric variables are permissible. It has the following format; items enclosed in brackets are optional.

STR (string variable, X_1 [,X_2])

where X_1 = Starting character in sub-string (an expression).

X_2 = Number of consecutive characters desired (an expression; the specification of X_2 is optional).

Example:

STR(A$,3,4)

This statement means take the 3rd, 4th, 5th, and 6th characters of A$.

STR(A$,3)

This statement means, starting with the 3rd character, take the remainder of the string A$.

In BASIC statements, STR functions can be used wherever string variables are used. They may be used on either side of an equal sign or relation. The following examples illustrate use of the string function:

Assuming B$="ABCDEFGH"

10 A$=STR(B$,2,4)	---A$ is set to "BCDE".
20 STR(A$,4) = B$	---Characters 4 through 11 of A$ are set to "ABCDEFGH".
30 STR(A$,3,3)=STR(B$,5,3)	---The 3rd, 4th and 5th characters of A$ are set to "EFG".
40 IF STR(B$,3,2)="AB"THEN 100	---Characters "CD" of B$ are compared to the literal string "AB".
50 READ STR(A$,9,9)	---Characters 9 through 17 of A$ receive the next data value read.

LEN (LENGTH) FUNCTION

Wang 2200 BASIC provides a function, LEN(, which permits the user to determine the number of characters in a given string variable. The LEN function can be used whenever a math function is permitted. The format of the length function is:

LEN(string variable)

Example:

A$ = "ABCD"
LEN(A$) has a value of 4

NOTE:
Trailing blanks are not considered to be part of the value of a string variable.

Examples:

```
100 X = LEN(A$) + 2
110 IF LEN A$(3) > 8 THEN 150
```

HEX (HEXADECIMAL) FUNCTION

The HEX function is a form of literal string that enables a user to use any 8-bit codes in a BASIC program; it may be used wherever literal strings enclosed in double quotes may be used. The HEX function has the following format; items in brackets are optional.

HEX (hexdigit hexdigit [{hexdigit hexdigit } . . .])

where hexdigit = a digit 0 - 9 or a letter A - F.

Example:

Executing the following statement clears the CRT display.

:PRINT HEX (03)

Executing the following statement sets the string variable, A$, equal to the 3 characters: 81_{16}, 82_{16}, and 34_{16}.

A$ = HEX (818234).

Any character can be represented by two hexadecimal digits. A complete list of HEX codes pertaining to the CRT is given in Appendix D.

LOWERCASE LITERALS

A special form of literal string is available for specifying lowercase characters; the literal string is enclosed in single quotes. For example, the following statement

:PRINT "J"; 'OHN'; " D"; 'OE'

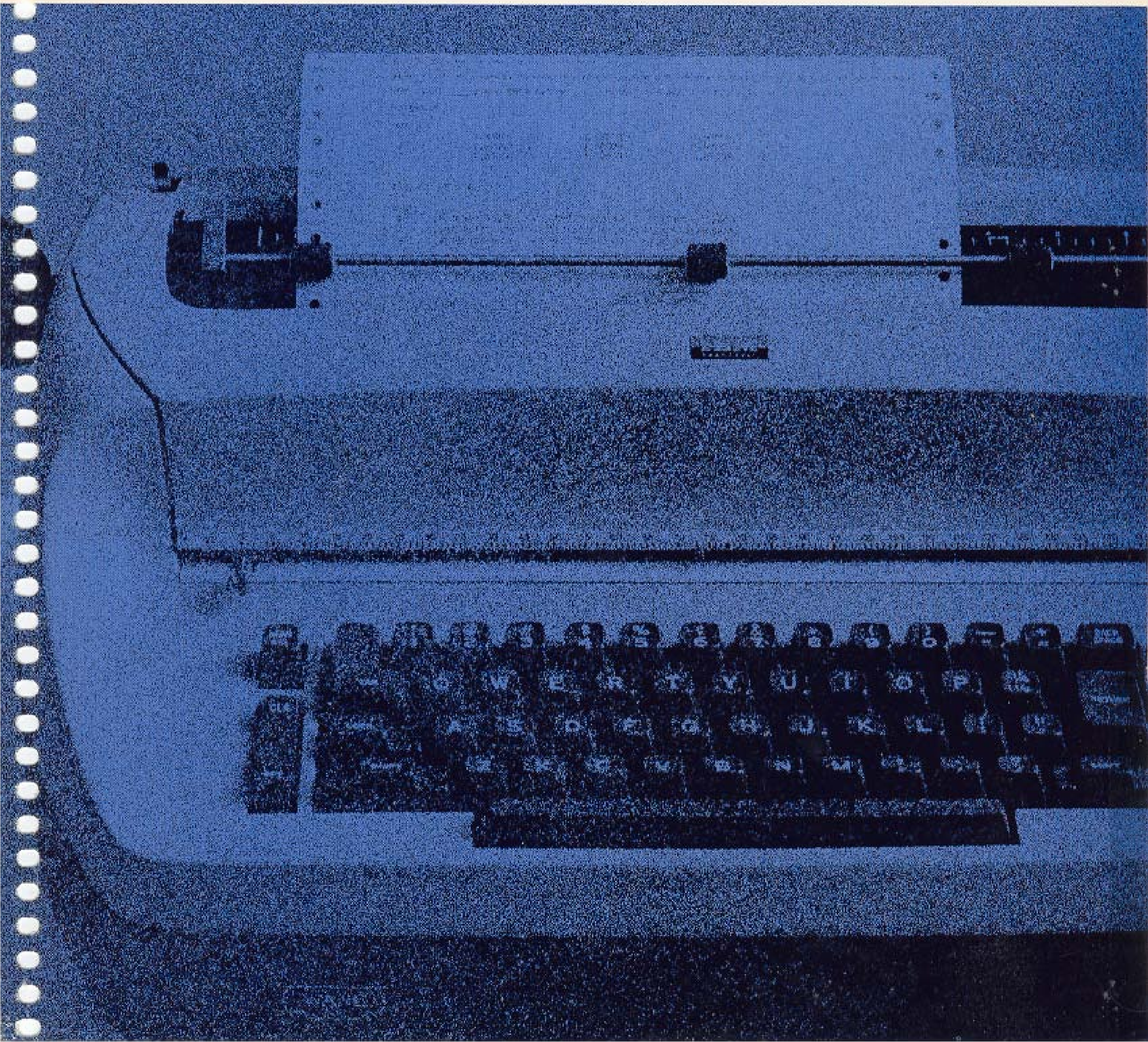
outputs 'John Doe' on peripheral devices capable of printing lowercase letters.

Section IV Alphanumerics

The following characters are valid for use in lowercase literals.

Letters:	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Digits:	0123456789
Special Characters:	blank ! " # $ % & ( ) * + , –
	. / : ; < = > ?

SECTION V
I/O DEVICE SELECTION



Section V

I/O Device Selection

INTRODUCTION

SYSTEM 2200A/B DEVICE SELECTION

Each peripheral I/O device associated with the System 2200 is assigned a unique device address. All device addresses are composed of three-digit hexadecimal numbers. The first hex digit identifies the device type. It is used by the system when controlling the I/O operation. The last two hex digits represent the actual device address, which is used to electronically select the device for operation.

The device type digit is used by the System 2200 to identify what type device is being selected for an I/O operation. The various peripheral devices on the System 2200 often require different control procedures to perform an input/output operation. For example, a type digit of 1 signifies cassettes, a type digit 3 indicates disk. The last two digits correspond to the actual device address which is preset in each device controller card in the System 2200 CPU. For example, if a System 2200 has three cassette drives, three unique addresses are available for cassettes.

When a System 2200 BASIC command or statement which performs an input/output operation is executed, the appropriate device can be selected in one of three ways.

1. DEFAULT (Primary Console Device) - If no device address is specified or selected, the System 2200 automatically provides the device address which is most commonly used for that particular operation.
2. SELECT - The System 2200 SELECT statement can be executed. It assigns device addresses for specified I/O operations.
3. SPECIFICATIONS - The device address can be supplied with the BASIC I/O statement or command, either absolutely or indirectly.

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 Table).

length = An integer < 256 specifying the desired carriage width.

'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 complete list of the System 2200 I/O devices and addresses is shown in the table below.

DEVICE ADDRESSES FOR SYSTEM 2200 PERIPHERALS
(For further detail, see the individual peripheral manuals.)

I/O DEVICE CATEGORIES	DEVICE ADDRESS (S)*	
KEYBOARDS** (2215, 2222)	001, 002, 003, 004	
CRT UNITS** (2216)	005, 006, 007, 008	
CASSETTE DRIVES (2217, 2218)	10A, 10B, 10C, 10D, 10E, 10F	
HIGH-SPEED PRINTERS (2221, 2231)	215, 216	
OUTPUT WRITERS (2201)	211, 212	
THERMAL PRINTER (2241)	211, 212	
PLOTTERS (2202, 2212, 2232)	413, 414	
DISK DRIVES (2230-1, -2, -3) (2240-1, -2)	310, 320, 330	
CARD READERS (2214)	517	
HIGH-SPEED PAPER TAPE READER (2203)	618	
TELETYPES (2207)	019, 01A, 01B	INPUT
	01D, 01E, 01F	OUTPUT
TELETYPE TAPE UNITS	41D, 41E, 41F	PUNCH/ READER
TELECOMMUNICATIONS (2227)	219, 21A, 21B	INPUT
	21D, 21E, 21F	OUTPUT

* In some cases, more than one device address is listed for each device category. Unless otherwise noted, each peripheral device is assigned a unique address; device addresses are assigned sequentially. Therefore is a System 2200 has only one device of a particular category, such as a cassette, it is set up with the first device address listed (10A in the case of the cassette). If it has two cassettes, they are set up with device addresses 10A and 10B. Each device address is printed on the interface card which controls that device.

** All peripherals in this category are assigned to lowest device address shown. They may, however, be assigned unique addresses by customer request.

DEFAULT DEVICE ADDRESS SELECTION

Each System 2200 has five I/O devices designated as the Primary Console Devices for the system. The device addresses of these peripherals are built into the system such that whenever Master Initialization occurs (i.e., power is turned off and then on again), the system automatically is set to those device addresses for I/O operations. The Primary Console Devices normally are:

1. Primary Console INPUT Device:	Keyboard (address 001) (Model 2215 or 2222)
2. Primary Console OUTPUT Device:	CRT (address 005) (Model 2216)
3. Primary Console TAPE Device:	The Primary Cassette (address 10A) (Model 2217)
4. Primary Console DISK Device:	The Primary Disk (address 310)
5. Primary Console PLOTTER Device:	The Primary Plotter (address 413) (Model 2202, 2212, 2232)

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. Device selection is described in the remainder of this section.

When Master Initialization occurs, the Primary Console Device addresses are assigned to all input and output operations. That is, all commands, statements, and other information keyed into the System 2200 are done from the Primary Console Input Device, while all system output is sent to the Primary Console Output Device. All BASIC statements involving cassette operations automatically access the Primary Console Tape Device unless the statements contain either of the two optional parameters, #n, or /xxx, which supply the device address.

Similarly, disk operations reference the Primary Disk Device and PLOT statements reference the primary plotter.

The console device addresses for input/output operations can be changed from the Primary Console Device addresses by using SELECT statements containing the parameters CI (console input), CO (console output), TAPE (console tape cassette drive), DISK (console disk drive) and PLOT (console plotter). Before these parameters can be used, however, the device addresses of the new console devices must be known (see Device Address Table).

To change the console device from the Primary Output Device address (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 carriage width is not specified for console output, PRINT or LIST, the last carriage widths selected for these operations are used. Master Initialization sets these carriage widths to 64 characters.*

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.

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"
```

The INPUT and PRINT parameters

The INPUT and PRINT parameters are used to select device addresses for the INPUT, KEYIN, PRINT, PRINTUSING, and HEXPRINT statements executed in a user's program. 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 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 carriage width 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 212(70)
```

This statement specifies that a line printer (device address = 212) 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 seconds. For example, the following statements generated the indicated pauses:

```
100 SELECT P1       pause  =  1/6 seconds
SELECT P6           pause  =  1 second
SELECT P (or PO)    pause  =  null (i.e., no pause)
```

Again, a pause remains in effect until Master Initialization occurs or until a different pause is selected. Selecting P or PO removes the current pause.

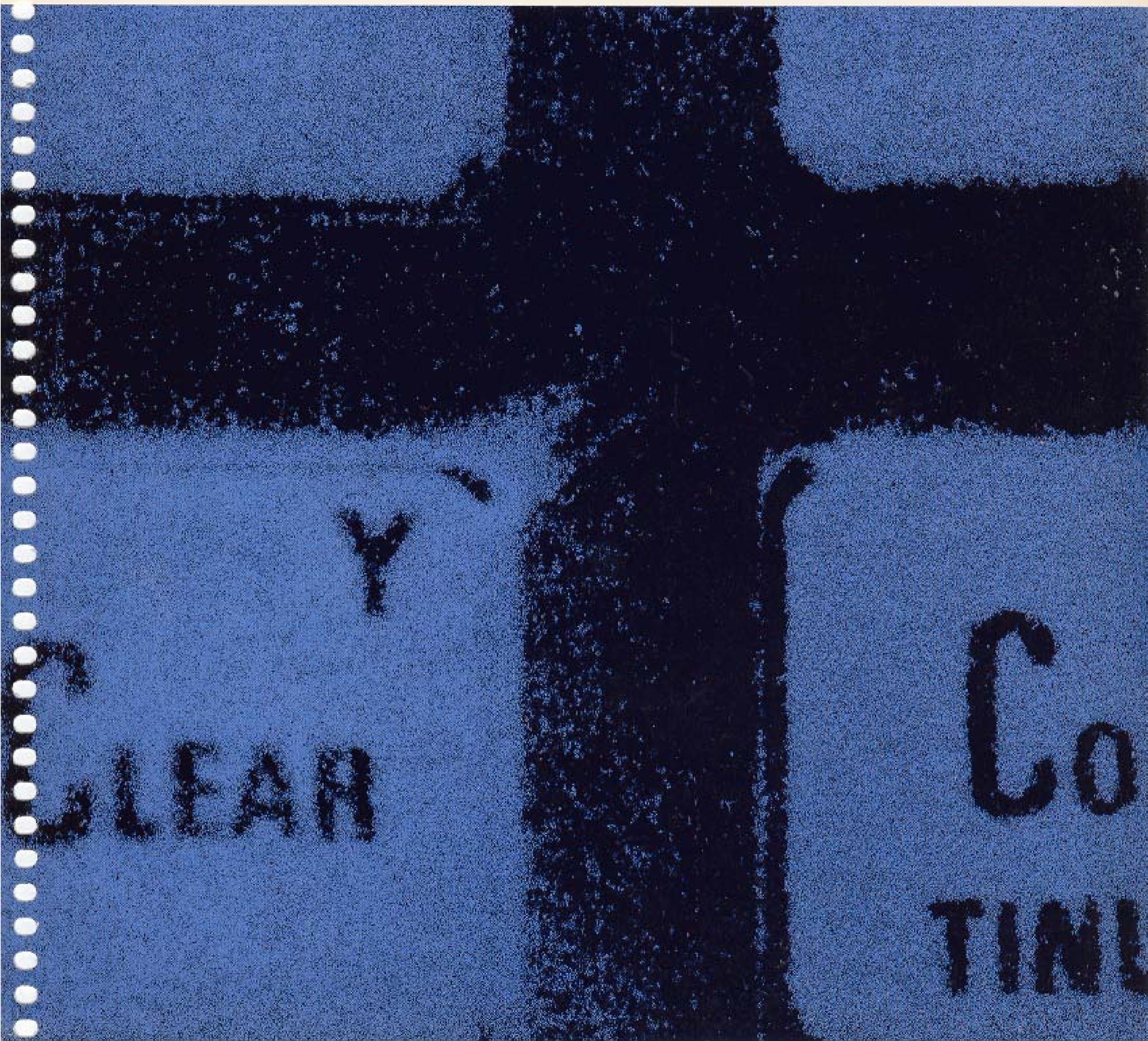
Specifying DEGREES, RADIANS, or GRADS:

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.

SECTION VI
NON-PROGRAMMABLE COMMANDS



Section VI

Non-Programmable Commands

Section VI Non-Programmable Commands

INTRODUCTION

A BASIC command provides the user with a means for communicating 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 in 2200 memory; the SAVE command instructs the system that all program text is to be recorded on a cassette tape, or some other device.

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.

BASIC SYNTAX SPECIFICATION RULES

The following editorial rules are used in this manual to define and illustrate the components of BASIC program statements and system commands.

1. Uppercase letters (A through Z), digits (0 through 9) and special characters (*, /, +, etc.) must always be used for program entry exactly as they are shown in the general form.
2. Information in lowercase letters is to be supplied by the user; for example, in the statement GOSUB 'line number', the line number must be entered by the user.
3. Square brackets, [ ], indicate that the enclosed information is optional. For example,

   RESTORE [expression]

   means that the RESTORE statement verb can be optionally followed by an expression:

   RESTORE
   or RESTORE 2*X

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

   $$\text{COM}\left\{\begin{array}{l}\text{scalar variable}\\ \\ \text{array variable}\end{array}\right\}\text{---}$$

   means that the COM statement elements can be:

   a scalar variable (i.e., C2)
   OR
   an array variable (i.e., D(4,8))
5. Ellipsis, . . . , indicate that the preceding item may occur once or many times in succession. For example,

   INPUT variable, variable, . . .
6. Except within double quotation marks, BASIC syntax ignores blanks.
7. When one or more items appear in sequence, these items or their replacements must appear in the specified order.

Section VI Non-Programmable Commands

GENERAL FORM OF TERMS

The list below defines the language syntax elements used in the command and statement syntax specifications.

alpha array designator ::	= letter[digit] $( )
alpha array variable ::	= letter [digit] $(expression [, expression])
alpha scalar variable ::	= letter [digit] $
alpha variable ::	$= \begin{Bmatrix} \text{alpha array variable} \\ \text{alpha scalar variable} \\ \text{STR function} \end{Bmatrix}$
array designator ::	$= \begin{Bmatrix} \text{alpha array designator} \\ \text{numeric array designator} \end{Bmatrix}$
builtin ::	= one of the following function names: SIN, COS, TAN, ARCSIN, ARCCOS, ARCTAN, ATN, EXP, LOG, SQR, ABS, INT, SGN, RND, LEN.
character string ::	= any string of letters, digits, or symbols not including carriage return, backspace, etc.
device address ::	= hexdigit hexdigit hexdigit
digit ::	= 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9
exponent	$= \text{E}[\{\pm\}]\ \text{digit}\ [\text{digit}]$
expression ::	$= [\{\pm\}]\ \text{term}$
fraction ::	= . integer
function ::	$= \begin{Bmatrix} \text{FN} \begin{Bmatrix} \text{letter} \\ \text{digit} \end{Bmatrix} \\ \text{builtin} \end{Bmatrix} (\text{ expression })$
hexdigit ::	$= \begin{Bmatrix} \text{digit} \\ \text{A, B, C, D, E, or F} \end{Bmatrix}$
integer ::	= digit [digit . . .]
letter ::	= A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,orZ
line number ::	= digit [digit] [digit] [digit]
literal string ::	$= \begin{Bmatrix} \text{"character string not including quotes"} \\ \text{'character string not including single quotes'} \\ \text{HEX (} \{ \text{ hexdigit hexdigit } \} \cdots \text{)} \end{Bmatrix}$
number ::	$= \begin{Bmatrix} \text{integer} \\ \text{fraction} \\ \text{integer fraction} \end{Bmatrix} [\text{ exponent}]$

numeric array designator :: = letter [digit] ( )

numeric array variable :: = letter [digit] (expression [, expression])

numeric scalar variable :: = letter [digit]

numeric variable :: = $\left\{ \begin{array}{l} \text{numeric scalar variable} \\ \text{numeric array variable} \end{array} \right\}$

STR function :: = STR(alpha variable , expression [, expression])

term :: = $\left\{ \begin{array}{l} \text{(expression)} \\ \text{number} \\ \text{function} \\ \text{numeric variable} \end{array} \right\} \left\{ \begin{array}{c} \pm \\ * \\ / \\ \uparrow \end{array} \right\} \text{term}$

variable :: = $\left\{ \begin{array}{l} \text{numeric variable} \\ \text{alpha variable} \end{array} \right\}$

CLEAR

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

Purpose

The CLEAR command reinitializes the user program text and variable areas. CLEAR with no parameters removes all program text and variables from the system. The current console devices are selected for all I/O operations (see SELECT). Also, pause and trace are turned off.

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

General Form:	CONTINUE

Purpose

This command continues program execution whenever the program has been stopped either by a STOP verb or the touching of the HALT/STEP key. The program continues with the program statement immediately following the last executed program statement.

> NOTE:
>
> *An error message is displayed and execution does not continue if the user enters a CONTINUE command after:*
> 1. *A text or table overflow error has occured.*
> 2. *A variable has been entered that has not previously been defined.*
> 3. *A CLEAR V or CLEAR N command has been executed.*
> 4. *Program text has been modified by a CLEAR, CLEAR P, or RENUMBER command having been executed, or a new program line having been entered.*
> 5. *The RESET key has been pressed.*

Example:

CONTINUE

HALT/STEP

General Form:	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 verb 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:*
>
> 1. *A text or table overflow error has occured.*
> 2. *A variable has been entered that has not previously been defined.*
> 3. *A CLEAR V or CLEAR N command has been executed.*
> 4. *Program text has been modified by a CLEAR, CLEAR P, or RENUMBER command having been executed, or a new program line having been entered.*
> 5. *The RESET key has been pressed.*

Suppose the following program is in memory:

Example:

```
.
.
.
:90   GOSUB 200
:100 PRINT "CALCULATE X, Y"
:110 X=1.2: Y=5*Z+X: GOTO 30
.
.
.
```

and we wish to step through the program from line 100 on. TRACE is turned on so that variables receiving new values are displayed.

HALT/STEP

Example:

Turn TRACE mode on	:TRACE
Start stepping at line 100	:GOTO 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 30 X=1.2
Touch HALT/STEP key	: 110: Y=5*Z+X: GOTO 30 Y=21.6
HALT/STEP key	: 110: : GOTO 30 TRANSFER TO 30
	:_

LIST

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.

RENUMBER

General Form:	RENUMBER [line number] [,line number] [,integer]
	where $0 < \text{integer} < 100$

Purpose

The RENUMBER command renumbers the user program. 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

General Form:	RESET

Purpose

The RESET button 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

RUN

General Form:	RUN [line number]

Purpose

The RUN command initiates the execution of the user's program. The system verifies the currently loaded program; variables are scanned and new (not previously entered) common variables and all non-common variables are reset to zero. The pointer to the next data value (to be used in a READ statement) is reset to the first data value in the program. The program statements are then executed in line number sequence.

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. This enables the user to continue a halted program. Program execution must not be started in the middle of a FOR/NEXT loop or a subroutine.

Examples:

```
RUN
RUN 30
```

NOTE:

After a program has been entered or loaded, execution should be initiated by a RUN command to ensure that space is reserved for program variables. Once a program has been RUN, program execution may be restarted by pressing a special function key.

SPECIAL FUNCTION

General Form:	Special Function Key

Purpose

There are 16 special function keys available on the 2215 (or 2222) keyboard. Depressing them in conjunction with the SHIFT key provides up to 32 entry points for the currently loaded BASIC program. 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 special function entry 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

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 execute the following:

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

```
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
:_
```

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
:100 GOSUB' 0 (.092, –.32)
:110 STOP
:RUN 100
X = 9.20000000E–02
Y = –.32
Z = 5.507152

STOP
:_
```

STATEMENT NUMBER

General Form:	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.

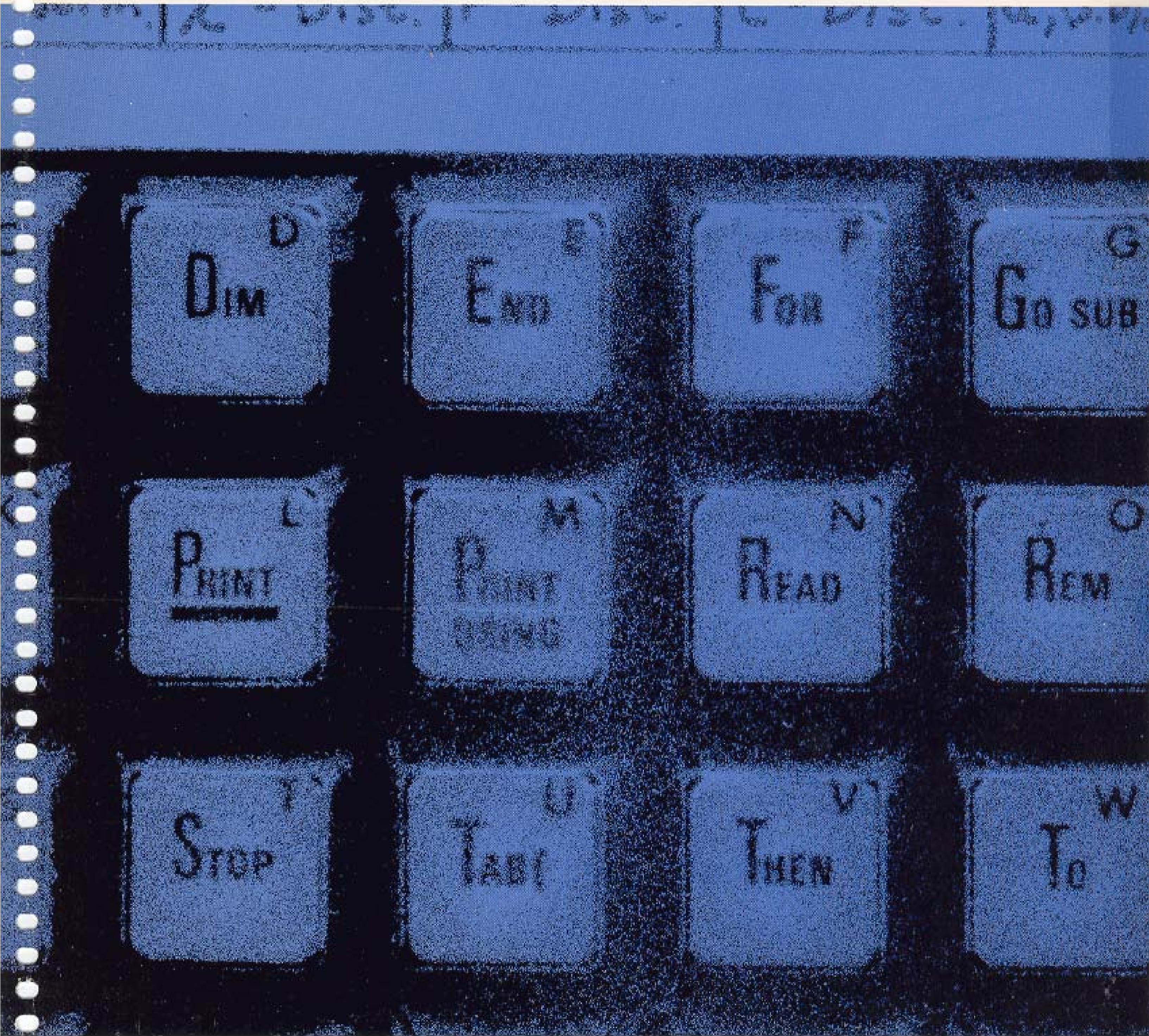
The statement number can also be keyed in manually, using the numeric entry keys. Statement numbers can be any integer from 1 to 4 digits.

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

SECTION VII
GENERAL BASIC COMMANDS



Section VII
General BASIC Statements

BASIC STATEMENTS

A BASIC statement is a special verb or word followed by an expression, variable, or numbers. For example:

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 the formats in which they can be used.

COM

General Form:	COM com element [, com element ...]	
where		numeric scalar variable
	com element =	numeric array variable
		alpha scalar variable [integer]
		alpha array variable [integer]
		$0 < \text{integer} \leqslant 64$

Purpose

The COM statement allows a programmer to store information in memory in an area which can be saved for use in a subsequent program. 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. Common variables must be named with identical attributes in a previous program.
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 integer ($\leqslant 64$) following the alpha scalar (or alpha array) variable specifes 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.

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
```

DATA

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

Purpose

The DATA statement provides the values to be used by the variables in a READ statement. In effect, the READ and DATA statements 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 are 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 numbers; alphanumeric variables must reference literal strings, which must be enclosed in quotation marks.

The RESTORE statement provides a means of reseting the current DATA statement pointer (i.e., reusing 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.

DEFFN

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

Purpose

The DEFFN statement defines a user's unique functions. The DEFFN statement is used to define functions which 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. 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 of Line 30:

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

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; it can refer to other functions. Up to five levels of function nesting are permitted. Two functions cannot refer to each other (an endless loop). A reference cannot be made to a DEFFN statement from an immediate mode statement. The scalar variable used in a DEFFN statement is called a dummy variable. It may have a variable name identical to a real variable used elsewhere in the program or in other DEFFN statements; current values of these 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)
```

DEFFN'

General Form:	DEFFN' integer	["character string" / (variable [,variable . . .])]
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: 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"
```

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 (ERROR 02).

DIM

General Form:	DIM dim element [, dim element . . .]
where	
dim element =	{ numeric array variable / alpha array variable [integer] / alpha scalar variable [integer] }
	$0 < \text{integer} \leqslant 64$

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 indicating the end of a BASIC program. 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
```

FOR

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. 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:

```
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

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
:_
```

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.

Repetitive entries to subroutines without executing a RETURN should not be made. Failure to RETURN causes RETURN information to be accumulated in a table which will eventually cause a table overflow error, (ERROR 02).

Example:

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

The 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
	subroutine argument = { character string in quotes / 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').

Use of GOSUB' is not permitted in immediate execution mode; GOSUB' may not be the last statement in a program.

Repetative entries to subroutines without executing a RETURN should not be made. Failure to return 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
:_
```

GOTO

General Form:	GOTO line number

Purpose

This statement transfers execution to another area of the program. The GOTO statement directs the system to the line number where execution is to continue.

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.

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 GO TO 40
:RUN
136           34

END PROGRAM
FREE SPACE = 3841
```

IF END THEN

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$
:_
```

IF...THEN

General Form:

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

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

Purpose

The IF statement causes the system to skip the normal sequence of program lines and go to the line number following THEN, provided certain conditions are met. This may be described as a conditional GOTO statement, which compares two items.

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 "<" relational operator is interpreted as "earlier in alphabetic order". Actually, the ASCII codes of the characters in the strings (see Appendix D) are compared; 1 is less than A since the ASCII code for 1 is 31 and the ASCII code for A is 41. In any comparison, trailing blanks are ignored, thus, "YES" = "YES   ". An error results if numeric values are compared to alphanumeric 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
```

Image (%)

General Form: % t [{ f t } . . .]

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

f, format specification = [+ / – / $] [#[,] . . .] [.#. . .]] [↑↑↑↑]

Purpose

This statement is used in conjunction with a PRINTUSING statement to provide an image line for formatted output. The Image statement contains text to be printed, along with the format specifications used to format print elements contained in the PRINTUSING statement.

The Image statement may have any printable characters of text inserted before and after print element format specifications. All text characters in the Image statement are printed as long as the final format specification is used. Each format specification in an Image statement is identified by at least one # character. The format specification may begin with the following characters ($, +, –, ., #). Commas (,) may be embedded in the integer portion of the format specification (after the first # character but before the decimal point (.) or up arrow symbols (↑↑↑↑)).

The Image statement must be the only statement on the statement line.

Example:

```
READY
:140% CODE NO. = #### COMPOSITION = ## ###
:670% #### UNITS AT $#,###.## PER UNIT
:800% +#.##↑↑↑↑
```

INPUT

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 types the optional input required message, VALUE OF A, B, and a question mark (?) and waits for the user to supply the two numbers. Program execution then continues. The input request message is 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. If more than one value is entered on a line, they may be separated by commas or entered on separate lines. Several lines may be used to enter the required INPUT data.

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 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.

When inputting 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 quotes.

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$="NO" THEN 50
:40 INPUT "ADDRESS",B$
: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 the system is awaiting input, pressing the special function key will cause the character string associated with that key to be entered.

For example:

```
:10 DEFFN' 01 "COLOR T.V."
:20 INPUT A$
:RUN
?
```

Now, pressing special function key '01 will cause "COLOR T.V." to be entered.

```
? 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. Repetitive subroutine entries via special function keys should not be made unless the subroutine RETURN is always executed. Failure to return from these entries will cause return information to accumulate in a table and eventually cause a table overflow error (ERROR 02).

For example

The program illustrated below 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 = I
:120 RETURN

:RUN
AMOUNT? 7
AMOUNT? 5
AMOUNT? 11
AMOUNT?          (Depress special function key 2)
TOTAL = 23
AMOUNT?
```

KEYIN

SYSTEM 2200B ONLY

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 1st line number.
3. READY WITH SPECIAL FUNCTION KEY - the code representing the special function key (hex 00 - 1F) is stored as the 1st character of the specified alphanumeric variable and execution continues at the second 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 execution mode.

Example:

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

General Form:	[LET] variable [, variable ...] = expression

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:

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

NEXT

General Form:	NEXT numeric scalar variable

Purpose

The NEXT statement signals the end of a loop begun by a FOR statement. The variable in the FOR statement and in its related NEXT statement must be the same.

During execution NEXT causes the index variable to be incremented. If the limit is not exceeded, transfer is made to the statement following the referenced FOR statement. If the limit is exceeded, the statement following the NEXT statement is executed.

In immediate execution mode, the NEXT statement and its corresponding FOR statement must both be in the same statement line.

Example:

```
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
```

ON

SYSTEM 2200 B ONLY

General Form:	ON expression {GOSUB / GOTO} 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
```

PRINT

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

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.

ZONE-FORM:	PRINT print element [, print element . . .] [,]

The output line is divided into as many zones of 16 characters as possible; the four CRT terminal 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
```

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

A semicolon signals that the next print element is to be printed immediately following the last print element, unless the last print element is an expression, in which case a space is inserted between the value of the expression and the next print element. 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
```

A PRINT statement can contain both comma and semicolon element separators. Each separator explicitly determines the amount of space between elements.

A semicolon causes 1 or no spaces to be skipped depending upon whether the previous element was an expression or text string. For example:

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

results in the following printout:

```
X =  2     Y =  3     Z = -4.2
```

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, skipping 1 space if the last print element was an expression. For example, the statements

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

causes the following printout:

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

A PRINT statement with no PRINT element advances the paper or the CRT cursor one line, or it causes the completion of a partially filled line.

Values of expressions are printed in one of two formats depending upon the value:

FORMAT 1:	SM.MMMMMMMME±XX	$10^{-1} > \text{VALUE} \geqslant 10^{+13}$
FORMAT 2:	SZZZZZZ.FFFFFFF	$10^{-1} \leqslant \text{VALUE} < 10^{+13}$

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

In format 2, the decimal point is inserted at the proper position or omitted if the value is an integer. Leading integer digit zeros and trailing fractional digit zeros are omitted.

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

```
FORMAT 1:  2.34762145E-09
           -1.64721000E+22
FORMAT 2:  23.47954890123
           -.6374
           0
           -421
```

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

Positions are numbered 0 to 64 on the CRT, and 0 to 155 (Selectric). The value of the expression in the TAB function is computed, and the integer part is taken. The typewriter is then moved to this position. If it has already passed this position, the TAB is ignored. If the value of the expression is greater than maximum values, the output device 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 2200 system, a built-in carriage width of 64 characters is initially available. If more than 64 characters are printed without a carriage return, an automatic carriage return is generated. This carriage width can be changed to any value ($0 \leqslant$ value < 256) by a SELECT statement, in conjunction with selecting the device address for PRINT.

PRINTUSING

General Form: PRINTUSING line number [, print element t . . .] [;]

where line number = Line number of the corresponding IMAGE statement.

print element = expression
alphanumeric variable
literal string in double quotes

t = comma or semicolon.

Purpose

The PRINTUSING statement permits numeric and alphanumeric values to be printed in a formatted fashion on the output device currently selected for PRINT (see SELECT).

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
```

(PRINTOUT)

```
ANGLE =   2.30 LENGTH = +27.1
```

Example 2:

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

(PRINTOUT)

```
1.0
2.0
3.0
```

Example 3:

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

(PRINTOUT)

```
1.0 2.0 3.0
```

PRINTUSING

Each IMAGE statement format specification has the following general format:

$$\begin{bmatrix} + \\ - \\ \$ \end{bmatrix} \quad [\#[,] \ldots] \; [.[\#\ldots]] \; [\uparrow\uparrow\uparrow\uparrow]$$

The IMAGE statement variable formats can be classified into three general formats:

FORMAT 1 –	Integer	e.g., ###
FORMAT 2 –	Fixed Point Number	e.g., ##.##
FORMAT 3 –	Exponential	e.g., #.##↑↑↑↑

Print elements are formatted according to the following rules:

1. Numeric expression print elements:

   a) If the format specification is not started with a plus (+), minus (–), or dollar sign ($) (i.e., the first format character is a number sign (#) or decimal point (.)) and the expression is negative, a minus (–) sign is edited into the print line and the length of the format increased by one.

   b) If the format specification is started with a plus (+) sign, the sign of the expression (+ or –) is edited into the print line immediately preceding the first significant digit.

   c) If the format specification is started with a minus (–) sign, a blank for positive expressions and a minus (–) sign for negative expressions is edited into the print line immediately preceding the first significant digit.

   d) If the format specification is started with a dollar ($) sign, a dollar ($) sign is edited into the print line immediately preceding the first significant digit.

   e) Commas (,) in the integer portion of the format are edited into the print line as they occur, if a significant digit has been edited prior to their occurrence; otherwise a blank is inserted.

   f) If the length of the value to be printed is less than the length of the format specification (overformatted) the value is right adjusted. If the length of the value to be printed is greater than the length of the format specification (underformatted) the format specification is edited into the print line (i.e., #'s are printed instead of a number).

   g) The expression value is edited according to the format specified in the image statement.

      FORMAT 1 – The integer part of the value is printed truncating any fractions. Leading blanks are inserted.

      FORMAT 2 – The value is printed as a fixed point number, truncating or extending any fraction with zeros and inserting leading blanks according to the format specification.

      FORMAT 3 – The value of the expression is printed as a floating point number. The value is scaled as specified by the format and printed as in formats 1 or 2. (There are, however, no leading blanks.) The exponent is always printed in the 4 character form: E±XX.

2. Alphanumeric string variables or literal string print elements:
   The value of a string variable or a literal string in quotation marks is edited into the print line by replacing each character in the format specification with characters in the text string. The text string is left-justified. If the text string is shorter than the format specifications, blanks are inserted on the right. The text string is truncated on the right if it is longer than the format specifications.

Example 1:

```
            :100 PRINTUSING 200, 1242.3, 73694.23
            :200 %TOTAL SALES = #### VALUE $###,###.##
            :RUN
(PRINTOUT)  TOTAL SALES = 1242 VALUE  $73,694.23
```

Example 2:

```
            :100 PRINTUSING 200, 2.13E-5, 2.3E-9
            :200 % COEFF = +.###↑↑↑↑ ERROR = -##↑↑↑↑
            :RUN
(PRINTOUT)  COEFF = +.213E-04 ERROR = 23E-10
```

Example 3:

```
            :100 PRINTUSING 200, 317.23
            :200 % +#.##
            :RUN
(PRINTOUT)  +#.## (Value too large for format)
```

Example 4:

```
            :100 PRINTUSING 200
            :200 $ PROFIT AND LOSS STATEMENT
            :RUN
(PRINTOUT)  PROFIT AND LOSS STATEMENT
```

Example 5:

```
            :100 PRINTUSING 200, A$, T
            :200 % SALESMAN ######## TOTAL SALES $##,###.##
            :RUN
(PRINTOUT)  SALESMAN J. SMITH           TOTAL SALES $9,237.51
```

READ

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
```

REM

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
```

RESTORE

General Form:	RESTORE [expression]
	where $1 \leq$ 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 could eventually cause the table overflow error (ERROR 02).

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
```

STOP

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 system types STOP followed by the specified character string if one is entered.

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, printouts are produced when:

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

   Printout format: variable = recieved value

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

   Printout format: TRANSFER TO 'line number'

Example 1:

```
          30 LET X, Y, Z(5)=A+SIN(B)/C
produces  the TRACE printout:
          X =
          Y =
          Z ( ) = 29.631
```

Example 2:

```
          :40 READ A, B, C(22), D
produces  A = 9.4
          B = 64.27
          C ( ) =1.37492100E+11
          D = 99.4
```

Example 3:

```
          :100 GOTO 200
produces  TRANSFER TO 200
```

Example 4:

```
          30 GOSUB 10
produces  TRANSFER TO 10
```

Example 5:

```
          :10 FOR I=1 TO 3
          :15 PRINT X(I);
          :20 NEXT I
produces  I=1
          I=2
          TRANSFER TO 15
          I=3
          TRANSFER TO 15
          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
```

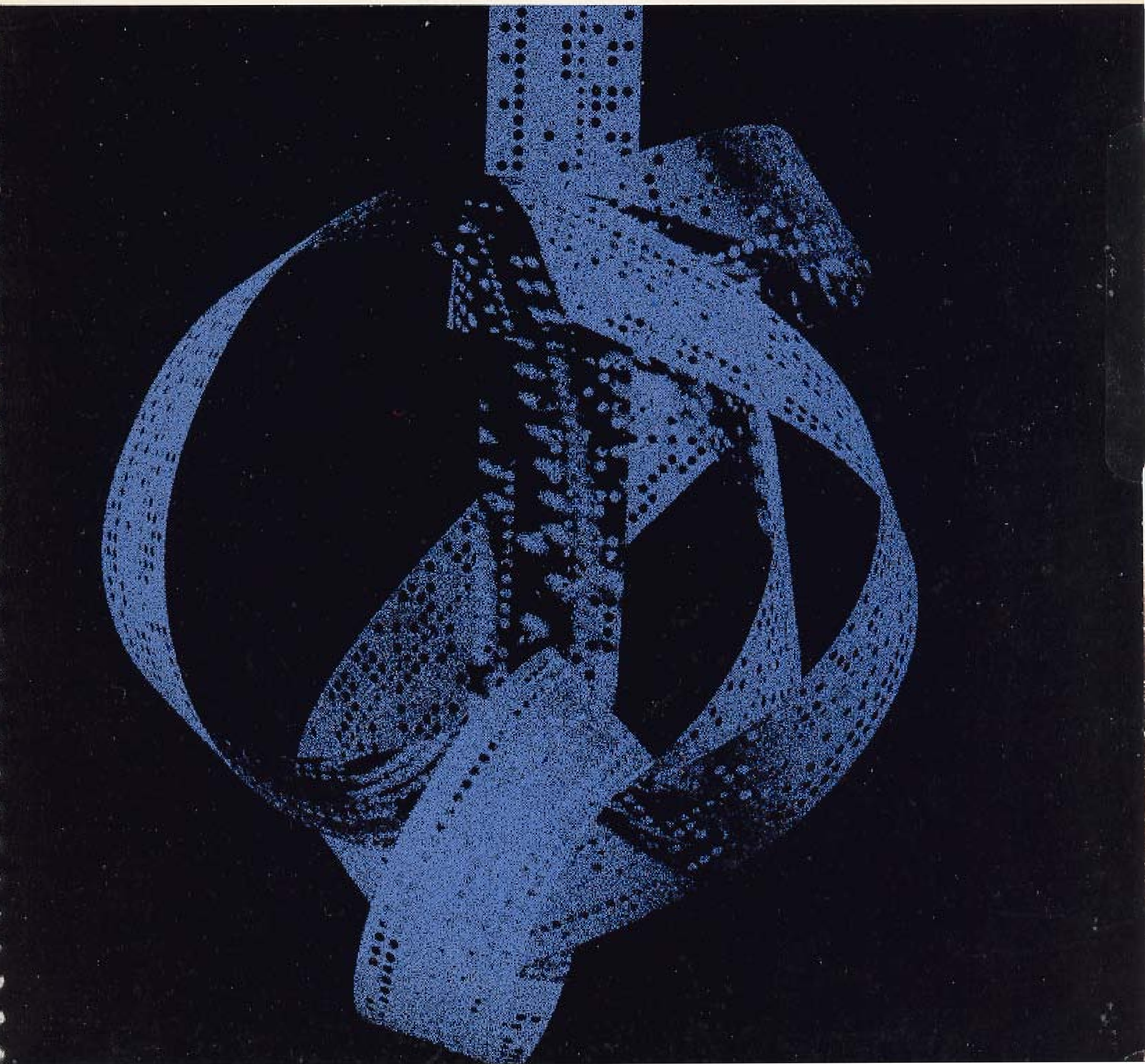
Example 9:

```
: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
```

SECTION VIII
DATA MANIPULATION



Section VIII

Data Manipulation

INTRODUCTION

DATA MANIPULATION

The System 2200B utilizes a number of statements which perform bit and byte manipulation and data conversion. In most cases, the operations are performed on the data values contained in alphanumeric string variables and arrays. When used in this manner, the characters or bytes contained in alphanumeric variables are used in a fashion similar to registers in a computer. With this capability, the System 2200B provides a powerful system for the conversion, editing, and efficient use of data.

Specifically, the statements provide the following capabilities:

1. For data processing, the ability to receive and validate keyed-in numeric data under program control.
2. The ability to receive, convert and process data received in any format from peripheral devices, especially useful when processing data generated by other computer systems or special instrumentation.
3. The ability to store numeric data in an efficient packed format which saves storage space in memory or on cassettes, disks, etc., and reduces the time required to retrieve it.
4. The ability to scan, edit and convert actual System 2200 BASIC programs which are processed as data. Utilities are available to perform a number of program editing and compression functions. For example, blanks and REM statements can be removed from a program to reduce memory storage requirements.
5. The general ability to perform binary, logical and arithmetic operations and built-in conversion operations can save memory and storage, and increase operating speeds.

ADD

SYSTEM 2200B ONLY

General Form:	ADD [C] (alpha variable, {xx / alpha variable})
where:	x = hexadecimal digit (i.e., 0-9 or A-F)
	C = add with carry

Purpose

The ADD statement is used to add (in binary) the value specified by the second argument (an alphanumeric variable or two hex digits) to the 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), ADD (A$, 02)
sets A$ = HEX (0325)

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

If A$ = HEX (02FFFE), 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.

ADD

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

SYSTEM 2200B ONLY

General Form:

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

where: x = hexadecimal digit (i.e., 0 - 9 or A - F)

Purpose

These statements perform the specified logical function (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)

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))
```

BIN

SYSTEM 2200B ONLY

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

Purpose

This statement converts the integer value of the expression to a character (i.e., to a 1 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 a decimal value of 64)
20 BIN(STR(A$, I, 1)) = X*T/2	

General Form:

BOOL x (alpha variable, {xx / alpha variable})

where: x = hexadecimal digit (i.e., 0 - 9, or A - F)

Purpose

The statement BOOL is a generalized logical function that operates on the characters of the entire value of the first alphanumeric variable. All characters in the value 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, 2) A7)

The hex digit following 'BOOL' defines which of the 16 possible logical functions is to be performed (see chart below). The hex digit represents the desired logical result of the following bit combinations:

value #1: 1 1 0 0
value #2: 1 0 1 0

For example, the hex digit 'E' (1110) defines the OR function since (1100) OR'ed with (1010) is (1110). Note, BOOL 6 is equivalent to XOR; BOOL 8 is equivalent to AND; and BOOL E is equivalent to OR. The 16 possible logical functions are listed below.

HEX DIGIT	BIT REPRESENTATION	LOGICAL FUNCTION
0	0000	null
1	0001	not OR
2	0010	
3	0011	complement of value #1
4	0100	
5	0101	complement of value #2
6	0110	exclusive OR
7	0111	not AND
8	1000	AND
9	1001	equivalence
A	1010	value #2
B	1011	value #1 implies value #2
C	1100	value #1
D	1101	value #2 implies value#1
E	1110	OR
F	1111	indentity

BOOL

Two types of logical functions 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 Functions

The logical function specified by the hex digit after 'BOOL' is performed using the character specified by the two hex digits and each character in the entire value of the alphanumeric variable (or portion of alphanumeric variable if the STR function is used). The result becomes the new value of the alphanumeric variable.

Example:

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

String-to-String Logical Functions

The logical function 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)
```

CONVERT

SYSTEM 2200B ONLY

General Form:	
	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 format 1 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 format 2 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 vlaue 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 1 - Fixed Point e.g., ##.##
Format 2 - Exponential e.g., #.##↑↑↑↑

Numeric values are formatted according to the following rules:

1. If the image starts with a plus (+) sign, the sign of the value (+ or –) is edited into the character string.
2. If the image starts with a minus (–) sign, a blank for positive values and a minus (–) for negative values is edited into the character string.
3. If no sign is specified in the image, no sign is included in the character string.
4. If the image has format 1, the value is edited into the character string as a fixed point number, truncating or extending with zeroes any fraction, and inserting leading zeroes 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 image has format 2, the value is edited into the character string as a floating point number. The value is scaled as specified by the image (there are no leading zeroes). The exponent is always edited in the form: E ± XX.

CONVERT

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$, (####.#####)
    (result: A$ = "0012.195000")
```

HEXPRINT

SYSTEM 2200B ONLY

General Form:

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

where:

alpha array designator = alpha array name ( ) e.g., A$( )

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$;
:120    HEXPRINT X$( )
```

INIT

SYSTEM 2200B ONLY

General Form:

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

where: x = hexadecimal digit (i.e., 0 - 9 or A - F)
alpha array designator = alpha array name ( ) e.g., A$( )

Purpose

The INIT statement initializes the specified alphanumeric variable(s) and/or array(s). Each character in the variable or array is set equal to the character specified inside the parentheses. The character may be represented by two hex digits, 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$) B$( )
```

NUM
FUNCTION

SYSTEM 2200B ONLY

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 (alpha-numeric ASCII characters). If the value represents a legal BASIC number, NUM(A$) equals 16, the number of characters in the string variable A$.

PACK

SYSTEM 2200B ONLY

General Form:

PACK (image) {alpha variable / alpha array designator} FROM {numeric array designator / expression}, . . .

where: image = [±] [# . . .] [.] [# . . .] [↑ ↑ ↑ ↑]

0 < number of #'s < 14

array designator = alpha array name ( ) e.g., A$( ), N( )

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 1 – Fixed Point e.g., ##.##
Format 2 – Exponential e.g., #.##↑↑↑↑

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 the image has format 1, the value is edited as a fixed point number, truncating or extending with zeroes any fraction and inserting leading zeroes for nonsignificant integer digits according to the image specification.
7. If the image has format 2, the value is edited as a floating point number. The value is scaled as specified by the image (there are no leading zeroes). The exponent occupies one byte.

Examples of storage requirements:

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

Examples:

```
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( )
```

POS
FUNCTION

SYSTEM 2200B ONLY

General Form:

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

where: x = hexadecimal digit (0 - 9 or A - F)

Purpose

The POS function finds the position of the first character in the specified alphanumeric value that is $<$, $\leqslant$, $=$, $\geqslant$, $>$, or $<>$ the character specified following the relation operation. 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 functions normally are used.

Examples:

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

ROTATE

SYSTEM 2200B ONLY

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)
```

UNPACK

SYSTEM 2200B ONLY

General Form: UNPACK (image) {alpha array designator / alpha variable} TO {numeric array designator / numeric variable} , . . .

where: image = [±] [# . . .] [.] [# . . .] [↑↑↑↑]

0 < number of #'s < 14

array designator = alpha array name ( ) e.g., A$( ), N( )

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:

```
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( )
```

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.

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:

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

SECTION IX
TAPE
CASSETTES

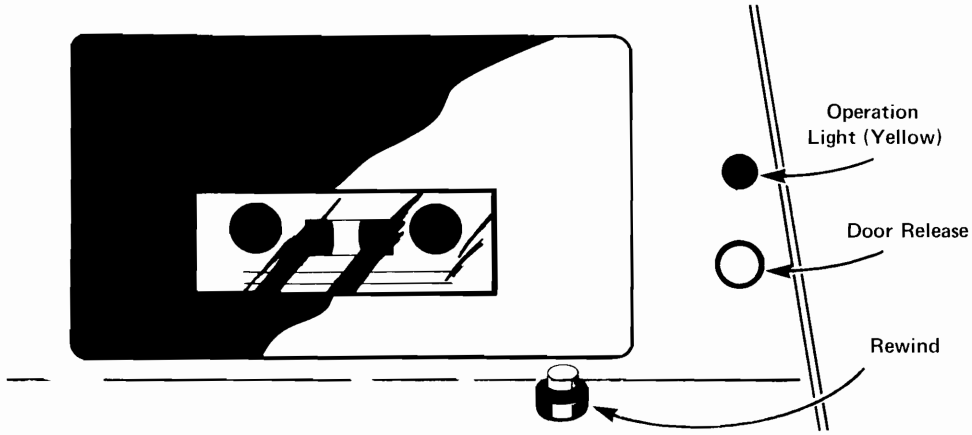
Section IX

Tape Cassettes

Section IX   Tape Cassettes

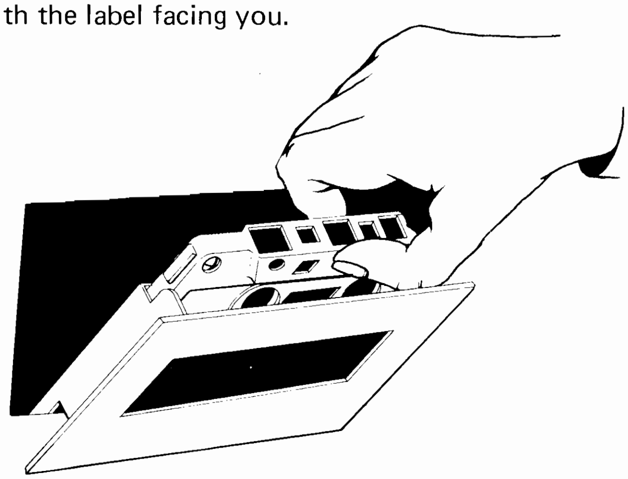
THE 2217 SINGLE TAPE CASSETTE

The 2217 Single Magnetic Tape Cassette Recorder is contained within the housing of the CRT. It is located in the right-hand corner of this housing. The 2217 is a peripheral and therefore is connected to the CPU with a connector cord (at back of the CRT housing). A separate cord is provided with the 2217 which goes to any wall outlet.



MOUNTING AND REMOVING A TAPE CASSETTE

The tape drive is opened by pressing the white push button to the right of the tape. A cassette is loaded into the tape drive with the label facing you.



Once the cassette is in place, the door should be closed.

Before using a tape, it should be rewound. This can be done in two ways: 1) touching the REWIND button on the CRT housing, or 2) keying REWIND CR/LF EXECUTE from the 2215 (or 2222) keyboard.

For example, key [SHIFT LOCK] R E W I N D [SHIFT] [CR/LF EXECUTE]

The second method enables you to rewind a tape under program control.

A tape is removed from the tape drive by opening the tape drive door. Should this door not open, it is due to a double lock activated to prevent a tape from being removed which is not completely rewound.

Whenever the tape drive is in motion the yellow operating light next to the drive is on. Do not try to remove a tape when this light is on.

MAGNETIC TAPE HEAD CLEANING

The magnetic tape cassette requires much the same care as required for cassettes used with home cassette recorders. The cassettes should be kept as free as possible from dust and dirt, and the magnetic heads should be periodically cleaned. The cleaning process is as follows:

The tape reading head is located in the top center of the magnetic tape unit (Figure 1). The head can be lowered to the cleaning position as follows: select the tape unit by keying LOAD, CR/LF. The head will be lowered into the position as shown in Figure 2 (disregard the error).

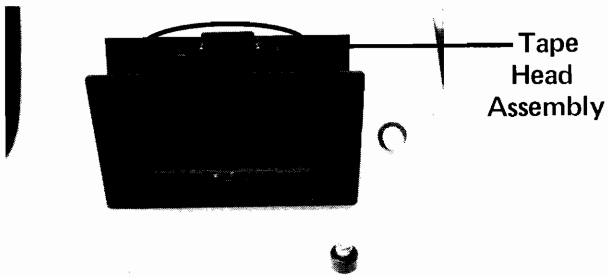


Figure 1

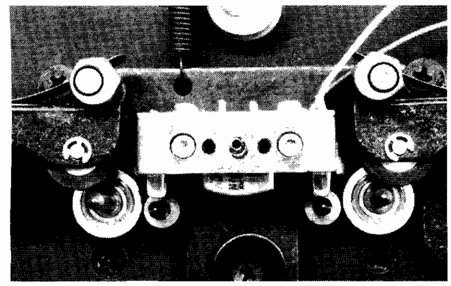


Figure 2

Tear open the foil packet containing the cleaning pad and rub the magnetic tape head gently for a few moments (Figure 3). After cleaning, dispose of the pad in the foil packet, exercising care that it does not touch any painted, shellacked, or plastic surface.

The 2200 can be restored to service by depressing the rewind button. The rewind process restores each head to its normal position (Figure 4).

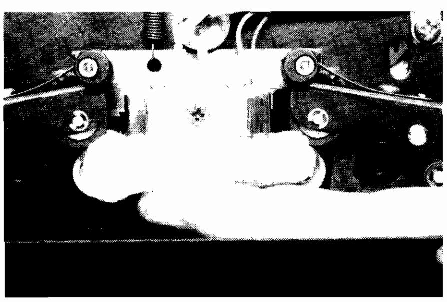


Figure 3

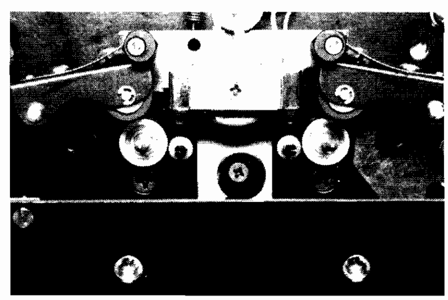


Figure 4

The cleaning operation should be performed every three weeks under normal conditions. In the event that your tapes have become heavily contaminated with dust or dirt, or if the 2200 is operating with the room humidity below 20%, then more frequent cleaning is required because of possible electrostatic attraction of dust and dirt to the tape mechanism.

Cleaning pads can be obtained from your Wang Serviceman.

PROTECTING A PROGRAM ON TAPE

With the System 2200 a new program simply writes over an old program; there is no need to erase the tape. To insure that a good program stored on tape is not written over or lost accidently, the tape can be protected.

To protect a program on tape, flip the orange plastic tab on the bottom right of the tape cassette 180°. When the tab is flipped over, an opening in the tape cassette indicates that the tape is protected.

If you need to write over the data (unprotect the tape) at a later date, flip the orange tab back 180° to cover the opening in the tape cassette.

Section IX Tape Cassettes

TAPE FORMAT

The 2200 provides the capability to record both programs and data onto cassette tape. Both programs and data are recorded on tape in 256 byte physical records. A 2200 user, however, need not worry about formatting a tape since the 2200 does this automatically. For example, if you wish to save a program currently in memory into cassette tape, key:

SAVE CR/LF-EXECUTE

The program is automatically recorded onto cassette tape; as many 256 byte physical records as are necessary are written.

To read back the program, rewind the tape and key:

CLEAR	CR/LF-EXECUTE	(Clears 2200 memory.)
LOAD	CR/LF-EXECUTE	(Loads the program from cassette.)

To insure data exactness, each physical record is recorded twice on tape. Dual recording and read-back is done automatically by the system, and requires no special user considerations.

PROGRAM FILES

When programs are recorded on cassette tape, it is not sufficient to merely record the program lines. It is important for the 2200 system to tell where the beginning and ending records of a program are. Therefore, every time a program is recorded, the 2200 system automatically records a header record before the program, and a trailer record after the program. Each recorded program thus becomes a program file. The figure below illustrates a program file.

HEADER RECORD	1st PROGRAM RECORD	2nd PROGRAM RECORD	------	Nth PROGRAM RECORD	TRAILER RECORD

Header Record

This is a physical record (256 bytes) which contains a control byte identifying it as a header (or beginning record) of a program. It also contains 8 bytes which can be used to store the name of the program, if the program is named when saved. *The remainder of the record is blank.*

Program Record

Each program record is a 256 byte physical record containing a portion of the saved program. It also contains a control byte identifying it as a physical record which contains part of a program (i.e., a program record).

Trailer Record

The trailer record is similar to a program record except that the trailer record has a control byte identifying it as the final physical record of the current program file (i.e., the trailer record).

There are a number of advantages associated with having program files. A program name can be stored in the header record. Thus, on a tape containing a number of programs, a particular program can be searched for by name. For example, a program is saved and named as follows,

SAVE "EVAL1"

it can be automatically searched and loaded by reference to the name of the program:

LOAD "EVAL1"

Program files can also be skipped and backspaced over by simple commands:

```
SKIP 2F         (skip forward over 2 files)
BACKSPACE 3F    (backspace over 3 files)
```

For example, if a user wants to add a 4th program to a cassette tape that already has three, he follows this sequence:

1. Mount the tape in the drive.
2. Depress the manual rewind button, or enter "REWIND".
3. Key SKIP 3F (skip the 3 current program files).
4. Key SAVE "PROG4" (save the program in memory on tape and name it "PROG4").
5. Rewind and remove the tape.

RECORDING DATA ON TAPE

Data is recorded onto a cassette tape by means of a DATASAVE statement. For example, the following statement in a program would record the values of the variables A, B, C$ and the 3rd element of 1-dimensional array D:

100 DATASAVE A, B, C$, D(3)

In addition, the 2200 offers the ability to record and read entire arrays by simply listing the array name followed by a left and right parenthesis, ( ). For example, values of all elements of the arrays A, B, and C$ can be written by:

```
10   DIM A(40), B(10,10), C$(10)
- - -
- - -
100  DATASAVE A( ), B( ), C$( )
```

READING DATA FROM TAPE

Data is read back from tape using a DATALOAD statement. For example:

```
100 DATALOAD A, B, C$, D(3)
200 DATALOAD A( ), B( ), C$( )
```

With the DATALOAD statement, the tape is read and the read values are sequentially assigned to the scalar and array variables listed in the program.

LOGICAL DATA RECORDS

Since all programs and data are recorded on cassette in 256 byte physical records, it is possible for the values of the variable list of a DATASAVE statement to exceed 256 bytes. In this case, two or more physical records are written. The one or more physical records written by the execution of one DATASAVE statement is called a LOGICAL RECORD. When data is read back by a DATALOAD statement, the entire logical record is read, reading physical records sequentially one at a time. If there are more values on a logical record than are called for in a variable list of a DATALOAD statement, the unused values are bypassed, and the tape is positioned at the beginning of the next logical record. For example, 50 logical records consisting of the current values of the arrays A and B could be written with the following program sequence:

```
READY
:90  FOR I = 1TO 50
:100   DATASAVE A( ), B( )
:
:200   NEXT I
:_
```

The logical records can be read back after rewinding the tape, with only the array A specified. In the following example,

```
READY
:400   REWIND
:410   FOR I = 1 TO 50
:420   DATALOAD A( )
:
:500   NEXT I
:_
```

the values of array B on each logical record are bypassed when read.

If more data is required in a variable list of a DATALOAD statement than is found in a logical record, another logical record is read to complete the list. For example, the arrays A and B can be written on separate logical records:

```
100 DATASAVE A( )
110 DATASAVE B( )
```

and both logical records can be read back in one DATALOAD statement:

```
200 REWIND
210 DATALOAD A( ), B( )
```

It is generally better, however, to read back data with a variable list identical in format to the DATASAVE statement which wrote that data.

Logical data records can be skipped and backspaced over. For example,

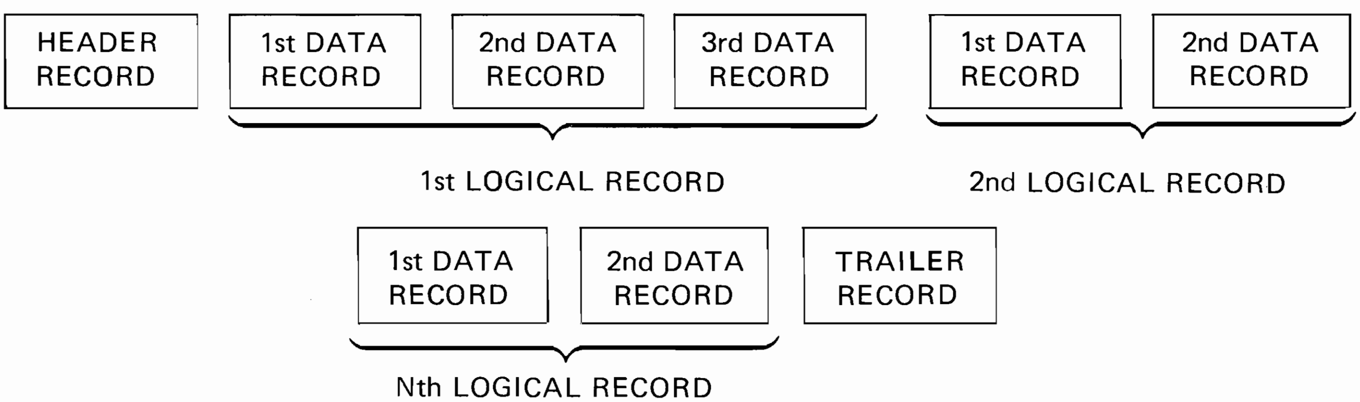
100 SKIP 3	Skip forward over 3 logical records
110 BACKSPACE 2*N	Backspace over 2*N logical records

DATA FILES

A series of logical data records on cassette can be made into a data file, similar to a program file, by preceding the records with a header record and following the records with a trailer record. Unlike program files however, the header and trailer record *are not* automatically generated by the 2200 system. They must be generated by the user's program using special forms of the DATASAVE statement.

DATASAVE OPEN "FILE1"	(Write a data file header record on tape and name the file "FILE1"; data files must be named.)
DATASAVE END	(Write a data file trailer record on tape.)

Therefore, a data file constructed by a series of DATASAVE statements would be as follows:



The header, data records, and trailer record are similar to those in a program file except that the control information in the records identifies them as data file records.

Therefore, a typical sequence for creating a data file could be:

```
:100   DATASAVE OPEN "STATFILE"     (Write header record.)
  .
  .
:150   FOR I = 1 TO N
:160   DATASAVE A, B, C$, D( )      (Write data records.)
  .
  .
:220   NEXT I
  .
:300   DATASAVE END                 (Write a trailer record)
```

Formatting a series of logical records into data files offers the same flexibility as program files. Data files can be searched on a tape by name using a special form of the DATALOAD statement. For example:

```
:100    DATALOAD "SAM"
```

This statement causes the system to search forward on the cassette tape until a data header record with the name "SAM" is found, and leaves the tape positioned to read the first logical record. If the data file to be searched could be either prior to or after the current tape position, a high speed rewind statement can be executed prior to the search:

```
:100    REWIND
:110    DATALOAD "FILE5"
```

Data files and program files can be recorded together on the same tape. The file SKIP and BACKSPACE statements apply to either kind of file. For example:

```
:100    SKIP 3F          (SKIP over the next 3 data or program files.)
:200    BACKSPACE 2F (BACKSPACE over the last two program or
                           data files.)
```

When logical data records are organized as files, record skipping and backspacing have additional features. For example:

```
:300    SKIP END          (SKIP to end of file.)
:400    BACKSPACE BEG  (BACKSPACE to beginning of file.)
```

In addition, because header and trailer records are present, the system prevents skipping over the beginning or end of file when skipping or backspacing logical records. (If more records are specified to be skipped or backspaced than exist in the remainder of the file, the tape stops at the trailer or header record.)

A final, and very important feature of data files is the ability to test for the end of file. In many cases when a data file is read, it is not always known how many records a file contains. When the trailer record is encountered while reading data records, an end of file condition is set and it can be tested by an IF END statement.

```
:200   DATALOAD A, B, C(10,2), D( )
:210   IF END THEN 300
```

In the above example, a transfer is made to statement 300 when a trailer record is read. The tape is repositioned back to the beginning of the trailer record. The end of file condition remains set until a subsequent DATALOAD statement is executed.

REWRITING DATA RECORDS

The 2200 provides a special capability to rewrite individual logical data records within a file. The 2200 system records timing bits in front of all records to insure proper alignment of a record before it is written. A special statement, DATARESAVE, is used to rewrite records. For example, a typical program sequence for rewriting a record might be:

```
:100   DATALOAD "COSTFILE"            (Search to beginning of file.)
  :
:150   DATALOAD A, B, C( ), D$( )     (Read next record.)
:160   IF A = X THEN 200              (Test if record to be rewritten.)
  :
:200   B = C:C(1) = D                 (Modify record.)
:210   BACKSPACE 1                    (Reposition before record.)
:220   DATARESAVE A, B, C( ), D$( )   (Rewrite record.)
```

NOTE:

The tape must be positioned directly in front of the old record to be rewritten. It is also important, when a record is rewritten, that the argument list be identical in format to that of the old record (i.e., the same number and type of variables, in the same order). Although the main requirement is that the rewritten logical record produces the same number of physical records as the old one did, miscalculations and tape formatting errors can be avoided if the argument lists are identical in format. Under no circumstances should records be rewritten using just the DATASAVE statement. Tape errors will result.

SPACE REQUIREMENTS ON CASSETTE

Numeric and alphanumeric data are stored on a cassette in the following format. Each numeric value occupies 9 bytes in the record. Literal string values occupy the length of the string plus 1 byte. Each alphanumeric variable value occupies either the default length (16 bytes) plus 1 additional byte, or the dimensioned length of the variable plus 1 byte. A total of 253 bytes is available for storing data in each physical record. Partial values are not written in a physical block; if a value of a scalar variable or array element to be recorded does not fit into the current physical block, the value is recorded in the next physical block.

DEVICE ADDRESS SPECIFICATIONS

Up to this point, examples have been presented for recording and reading of cassette tapes without a specification of a device address. Since 2200 systems can be purchased with a number of cassette drives, the user may specify what drive he wishes. The following rules apply to device address selection.

1. If no address is specified with Input/Output statements (i.e., LOAD, SAVE, DATALOAD, DATASAVE, SKIP, etc.), the system assumes a cassette tape is implied, and uses the default tape address. Therefore, a System 2200 with just one cassette does not require a cassette device address to be specified.
2. The tape default address is set to 10A when the system is master initialized (power is turned ON). It may, however, be changed by the SELECT statement. For example:

   :SELECT TAPE 10B

   would change the default tape address to 10B. It then remains set to 10B until the system is master initialized (power turned OFF, then ON), or when the address is changed by another SELECT statement.
3. There are two ways of specifying an I/O device address within an I/O statement: (These apply to other devices as well as cassettes.)

   a. Absolute Device Specification

   A three character device address, preceded by a slash (/) character, can be entered in the statement after the statement verb and is followed by a comma(,).

   Example:

   :LOAD/10B, "LINPROG"
   :100 DATASAVE/10C, A( ), B( )
   :110 SKIP/10D, 2F

   b. Indirect Device Address Specification (File Numbers)

   Six storage locations are available in the 2200 system for the assignment of device addresses. They are called file numbers and are referenced as follows: #1, #2, #3, #4, #5, #6.

   File numbers are assigned addresses in a SELECT statement. For example, the following statement

   :100 SELECT #1, 10B, #2 10C

   assigns the device address 10B to #1 and 10C to #2. Thereafter the file number can be used in the I/O statements:

   :LOAD #1
   :DATASAVE #2, A, B, C$
   :BACKSPACE #2, 1

   The device address assigned to the specified file number is used in the I/O statements. File numbers for cassette operations allow the user to reassign cassette drives for all the I/O operations in a program by changing just the SELECT statement.
4. The legal cassette addresses are 10A, 10B, 10D, 10E and 10F. The cassette drive addresses are marked next to the 2217 cassette drive controller plugs on the CPU chassis.

BACKSPACE

CASSETTE STATEMENT

General Form:

```
              ┌ #n,  ┐ ┌ BEG ┐
    BACKSPACE │      │ │ n   │
              └ /xxx,┘ └ nF  ┘
```

Where #n	=	File number to which the device address has been assigned. (#n = #1, #2, #3, #4, #5, or #6)
xxx	=	Device address of cassette
		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 indicated cassette tape 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:

DATALOAD [#n, | /xxx ,] { "name" | argument list }

#n = File number to which device is currently assigned (n is an integer from 1-6)

xxx = 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 list = { alphanumeric variable | numeric variable | alpha or numeric array designator } , . . .

array designator = array name ( ) e.g., A( ), B( ), C2( ), A$( )

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 SYSTEM 2200B ONLY

General Form:

DATALOAD BT [(N=expression)] [#n | /xxx,] alpha array designator

Where:

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

#n = File number to which device is currently assigned. (n is an integer from 1-6)

xxx = 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.

alpha array designator = array name( ) e.g., A$( ), B1$( )

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$( )
DATALOAD BT (N=100) #5, Q$( )
```

DATARESAVE

CASSETTE STATEMENT

General Form:

DATA RESAVE [#n, | /xxx ,] {OPEN "name" | argument list}

where #n = File number to which the device is currently assigned. (n is an integer from 1 to 6)

xxx = 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 list = {literal string | alphanumeric variable | expression | array designator} , . . .

array designator = array name( ) e.g., A$( ), B( ), C2( ), D$( )

Purpose

The DATA RESAVE statement allows the user to rewrite (i.e. update) any complete logical record including the header record, of an existing data file. Rewriting the header record permits the user to rename a file.

REWRITING A DATA RECORD

Rewriting (updating) a logical data record within a file generally involves 3 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.

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)
```

DATASAVE

CASSETTE STATEMENT

General Form:

```
DATASAVE  [ #n,   ]  { OPEN "name"   }
          [ /xxx, ]  { END           }
                     { argument list }
```

where #n = File number to which the device is currently assigned. (n is an integer from 1 to 6)

xxx = Device Address of cassette 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 list = { literal string | alphanumeric variable | expression | array designator } , . . .

array designator = array name( ) e.g., A( ), B( ), C2( )

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)
```

DATASAVE BT

CASSETTE STATEMENT SYSTEM 2200B ONLY

General Form:

DATASAVE BT [R] [([N=expression],* [H])] [#n, | /xxx,] alpha array designator

Where:

N	=	100 or 256 (size of block to record)
H	=	record header block (0's timing mark)
#n	=	File number to which the device is currently assigned. (n is an integer from 1 to 6)
xxx	=	Device Address of cassette 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.
alpha array designator	=	array name ( ) e.g., A$( )
R	=	resave

*A comma must separate the N and H parameters if both are specifed.

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$( )
DATASAVE BT (H) #6, Q$( )
```

LOAD

CASSETTE COMMAND

General Form:

```
LOAD  [ #n,   ]   ["name"]
      [ /xxx, ]
```

Where #n = File number to which a device address is currently assigned. (n = an integer from 1 to 6)

xxx = 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" = 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; this is described on the next page.

Example:

```
LOAD
LOAD "LINREGR"
LOAD#1, "PROGRAM1"
LOAD/10B
LOAD#4
```

LOAD

CASSETTE STATEMENT

General Form:

```
              ┌#n,  ┐   ┌      ┐
        LOAD  │     │   │"name"│   [ line number 1 ] [, line number 2 ]
              └/xxx,┘   └      ┘
```

where #n = file number to which the device is currently assigned. (n is an integer from 1 to 6)

xxx = device address of cassette.

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. (If a program is stored prior to current tape position, the user should give a REWIND command first.)

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 will result 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
```

REWIND

CASSETTE STATEMENT

General Form:

REWIND [#n | /xxx]

where #n = logical file number to which a device address has been assigned (n is integer from 1 to 6).

xxx = device address of cassette

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 indicated cassette to be rewound.

Example:

```
REWIND
100 SELECT #2 10B
110 REWIND #2
30 REWIND
40 REWIND /10C
```

SAVE

CASSETTE COMMAND

General Form:

SAVE [#n, | /xxx,] [P] ["name"] [line number [, line number]]

where #n = File number to which device address is assigned (#1 → #6).
xxx = Device address of desired output tape.

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

SKIP

CASSETTE STATEMENT

General Form:

```
SKIP [#n,  ]  {END}
     [/xxx,]  {n  }
              {nF }
```

where #n = File number to which a cassette device address has been assigned; n is an integer from 1 to 6.

xxx = Device address of cassette

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 $\geqslant 1$)

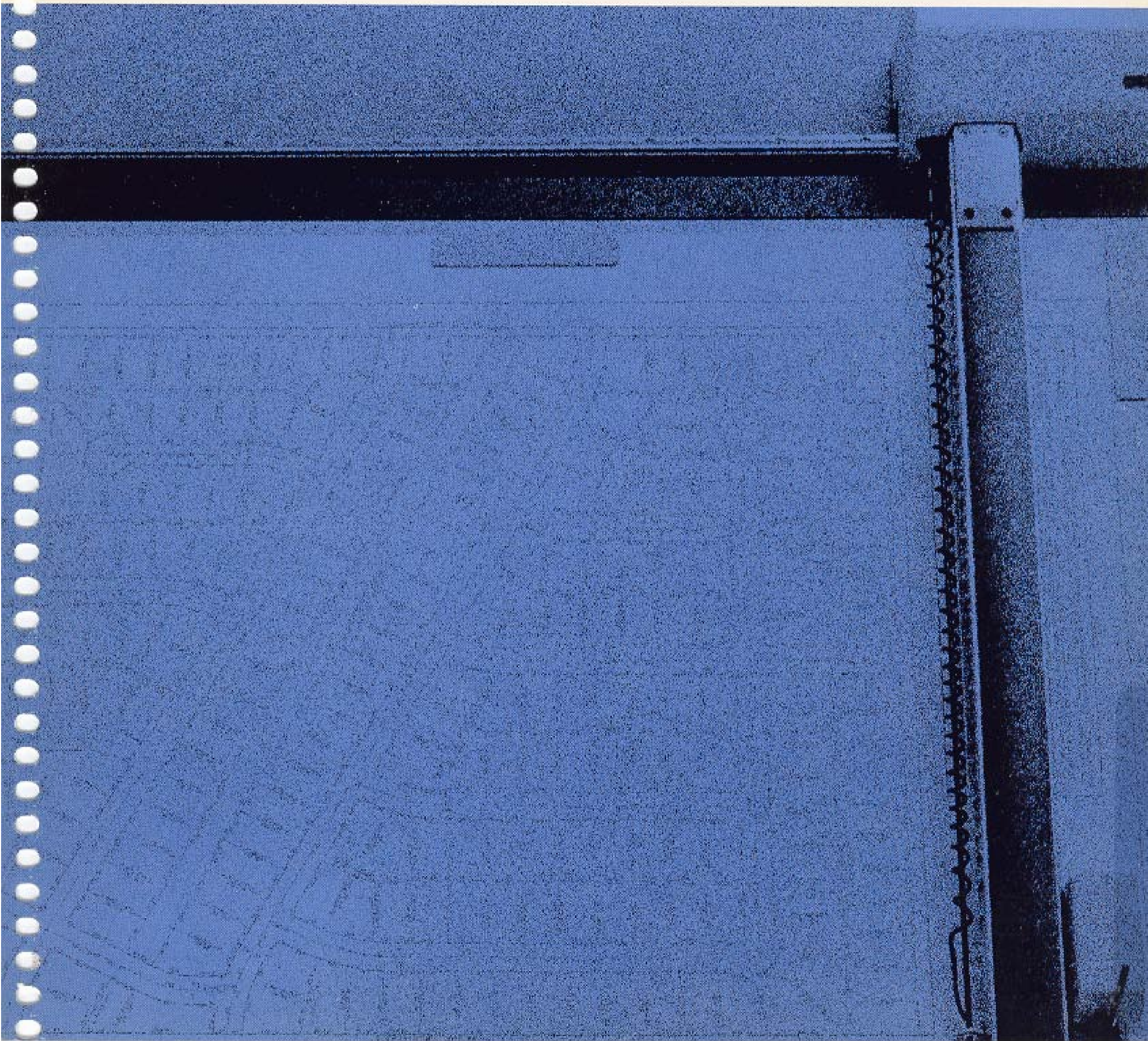
Purpose

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
```

SECTION X
PLOTTERS



Section X

Plotters

2202 PLOTTING OUTPUT WRITER SYSTEM 2200B ONLY

General Form:

```
                                             ('null'         )
PLOT [expression 0] < [expression 1],[expression 2], {literal string } > [,<....]
                                             (alpha variable )
```

where: expression 0 — represents the replication factor, or the number of times the values enclosed in <> are plotted
$(1 \leqslant \text{expression } 0 < 1000)$
If omitted, expression 0 is assumed to be 1.

expression 1 — represents Δx increments of .01″
$(-1000 < \text{expression } 1 < 1000)$
If omitted, expression 1 is assumed to be 0.

expression 2 — represents Δy increments of .01″
$(-1000 < \text{expression } 2 < 1000)$
If omitted, expression 2 is assumed to be 0.

All 3 expressions are truncated to integer values.

'null' implies move the Δx and Δy distance, specified in expression 1 and expression 2, without plotting.

Literal string, alpha variable represent character or characters to be plotted or printed.

Purpose

This statement positions the typing element 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 location. At the new position a literal string or alpha variable value is plotted or printed if one is present in the PLOT statement. If there is no argument following expression 2 ('null'), no character is plotted.

A one character literal string or alpha variable will be plotted while more than one character will cause a print. Plotting implies no space before the character while printing moves the typing element over 1 space before each character. Therefore, if more than one character is supplied to plot, the plot element will move an additional n times .1″ in the +x direction, where n is the number of characters.

The normal plotting character is the centered dot which is specified by its hexadecimal notation, HEX (FB). For example,

```
PLOT < 10, 20, HEX(FB) >
```

Examples:

10 PLOT < 10, 20, "*" >	Advance Δx = 10 increments (of .01″) and Δy = 20 increments (of .01″) and plot a *.
10 C = 40: D = 50 20 PLOT < C -10, D +20, >	Advance Δx = 30 increments and Δy = 70 increments without plotting.
10 PLOT <-50, 100,"ANGLE">	Advance Δx = –50 increments and Δy = 100 increments and print ANGLE.
10 A$ = "*" 20 PLOT < 10, 20, A$ >	Advance Δx = 10 increments and Δy = 20 increments and plot a *.
PLOT 25 <, 20,HEX(FB) >	Advance Δy = 20 increments and plot a centered dot. Do this 25 times.

```
10  N = 10
20  PLOT N < 10, , "ABC" >
```

Advance Δx = 10 increments and print ABC. Do this 10 times.

```
10  PLOT < X, Y, "VALUE" > , < 40, 60, "-" >, < A + 10, B, C$ >
```

The above is an example of multiple PLOT arguments in the same statement. They are processed sequentially from left to right.

2212 ANALOG FLATBED PLOTTER (10" x 15") SYSTEM 2200B ONLY

General Form:

$$\text{PLOT [expression 0]} < \text{[expression 1],[expression 2],} \left\{\begin{array}{l} \text{'null'} \\ \text{literal string} \\ \text{alpha variable} \\ \text{U} \\ \text{D} \\ \text{C} \\ \text{S} \\ \text{R} \end{array}\right\} > [,<\ldots.]$$

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 is assumed to be 1.

expression 1 — normally represents Δx increments of .015"*
(−1000 < expression 1 < 1000)
If omitted, expression 1 is assumed to be 0.

expression 2 — normally represents Δy increments of .01"*
(−1000 < expression 2 < 1000)
If omitted, expression 2 is assumed to be 0.

All 3 expressions are truncated to integer values.

For Plotting

'null' (i.e., no argument) and U imply move the Δx and Δy distance, specified in expression 1 and expression 2, with the pen up.

D implies draw a line while moving the Δx and Δy distance specified in expression 1 and expression 2.

R (RESET) moves the pen to the zero position as manually set on the plotter.

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.

*These values assume full-scale plotting.

Purpose

When used with plot arguments this statement moves the plot pen 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, 'null') or down (D). When a literal string or alpha variable is supplied as the argument, the movement is made with the pen up to the new position and then the characters are plotted.

Several special plotter control arguments are available. R resets the plotter to the 0,0 position. U and D indicate movement with the pen up or down. C and S are used to set the plotter character size and spacing as follows:

C sets the character size which is specified in expression 1. This must be an integer value from 1 to 15, where 1 is the smallest (.10" X .13")* and 15 is the largest size (1.5" X 1.95")*.

S sets the horizontal and vertical spacing between characters. Expression 1 specifies the horizontal spacing (which should be 10 times the character size to prevent overlap) and expression 2 specifies the vertical spacing (which should be 13 times the character size).

*These values assume full-scale plotting.

Examples:

Moving the plot pen

PLOT < 10, 20, D >	Plot (pen down) moving Δx = 10 (times .015") and Δy = 20 (times .01").
PLOT < A, B, U >	Advance (pen up) Δx = integer value of A (times .015") and Δy = integer value of B (times .01").
PLOT < , , R >	Reset to 0,0 position.

Setting plot conditions

PLOT < 9, , C >	Set character size to 9.
PLOT < 10, –20, S >	Set character spacing to 10 in the horizontal direction and –20 vertically.

Plotting characters

10 PLOT < , , "DEGREES" >	Plot the characters DEGREES.
10 A$="DEGREES" 20 PLOT < X, Y, A$ >	Advance (pen up) Δx = x and Δy = y and plot the characters DEGREES.

Replication and multiple arguments on one line

10 PLOT 10 < X, Y, D >	Plot x and y 10 times.
10 N = 30 20 PLOT N <–10, 20, U >	Advance Δx = –10 and Δy = 20 thirty times.

10 PLOT < X, Y, U >, < 10, 20 D >, < A + 10, –B, U >, < , , R >

The above is an example of multiple arguments in one PLOT statement. They are processed sequentially from left to right.

> NOTE:
>
> *Alphanumeric characters should not be printed when the plot pen is at the zero reference point. The pen position is considered the center of a character, so at the zero reference point the left half and bottom half of a character will not print correctly. Therefore, the plot pen should be diagonally above and to the right of the zero reference point before a character is printed.*

General Form:

$$\text{PLOT [expression 0]} < \text{[expression 1],[expression 2],} \begin{Bmatrix}\text{'null'}\\ \text{U}\\ \text{D}\\ \text{R}\end{Bmatrix} > [,<\ldots.]$$

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 is assumed to be 1.

expression 1 — represents Δx increments of .0025"
(–1000 < expression 1 < 1000)
If omitted, expression 1 is assumed to be 0.

expression 2 — represents Δy increments of .0025"
(–1000 < expression 2 < 1000)
If omitted, expression 2 is assumed to be 0.

All 3 expressions are truncated to integer values.

For Plotting

'null' (i.e., no argument) and U imply move the Δx and Δy distance, specified in expression 1 and expression 2, with the pen up.

D implies draw a line while moving the Δx and Δy distance specified in expression 1 and expression 2.

R (RESET) moves the pen to the zero position on the plotter.

Purpose

When used with plot arguments this statement moves the plot pen 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, 'null') or down (D).

One additional plotter control argument is available. R resets the plotter to the 0,0 position with the pen up.

Examples:

Moving the plot pen

PLOT < 10, 20, D > — Plot (pen down) moving Δx = 10 (times .0025") and Δy = 20 (times .0025").

PLOT < A, B, U > — Advance (pen up) Δx = integer value of A (times .0025") and Δy = integer value of B (times .0025")

PLOT <, , R > — Reset to 0,0 position

Replication and multiple arguments on one line

```
10  PLOT 10< X, Y, D >
```

Plot x and y 10 times

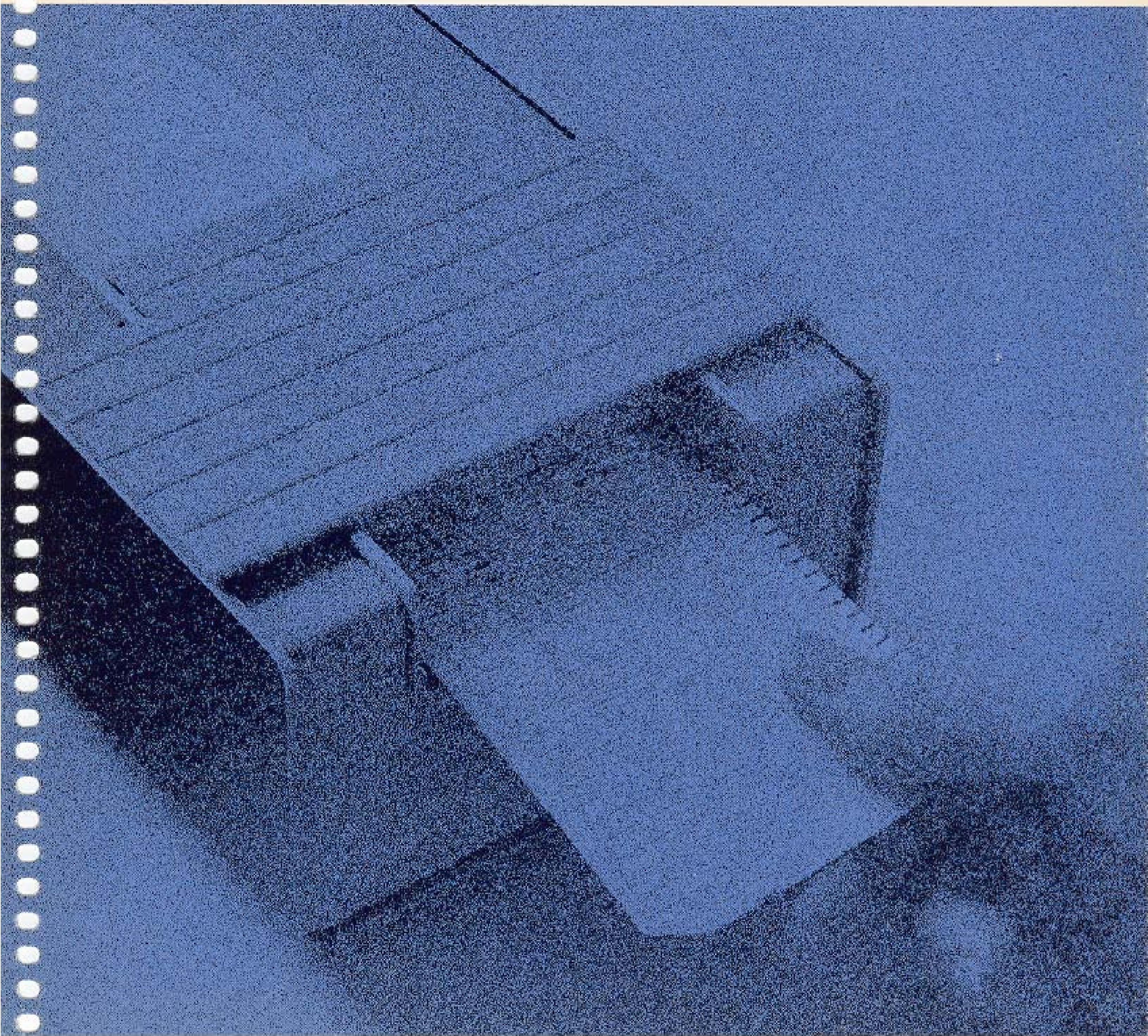
```
10  N = 30
20  PLOT N <-10, 20, U >
```

Advance Δx = – 10 and Δy = 20 thirty times

```
10  PLOT < X, Y, U >,< 10, 20, D>,< A + 10, -B, U >,< , , R >
```

The above is an example of multiple arguments in one PLOT statement. They are processed sequentially from left to right.

SECTION XI
MARK SENSE CARD READER



Section XI

Mark Sense Card Reader

INTRODUCTION

MODEL 2214 MARK SENSE CARD READER

The Model 2214 Mark Sense Card Reader is an economical System 2200 option which enables marked cards containing BASIC programs or data to be read into the System 2200. With this peripheral, the cards are manually fed into the reader and are automatically stacked in a hopper after reading. The card format allows 40 columns to be marked on each card, with eight data or character bits per column.

Two general modes of operation are used on the System 2200 for reading marked sense cards with the Model 2214:

1. Console Input Operation
   In the console input mode, the card reader device address is selected for console input operation (in lieu of the keyboard). This permits the card reader to function similar to the System 2200 keyboard. Thus programs can be entered from cards, similar to the way they are keyed in, and data can be entered, similar to the way it is keyed in, in response to an input statement. An advantage of the console input is read data is automatically displayed on the CRT screen, and error detection and recovery is simplified.
2. Peripheral Device Operation
   In the peripheral device mode, data or programs are read from cards in the same manner as other peripheral devices using, LOAD, DATALOAD and DATALOAD BT BASIC commands and statements. The required data format on the card and operating procedures are more flexible in this mode.

Procedure:

(1) Select the Mark Sense Card Reader for Console Input by entering:

SELECT CI 517 (execute)

(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:

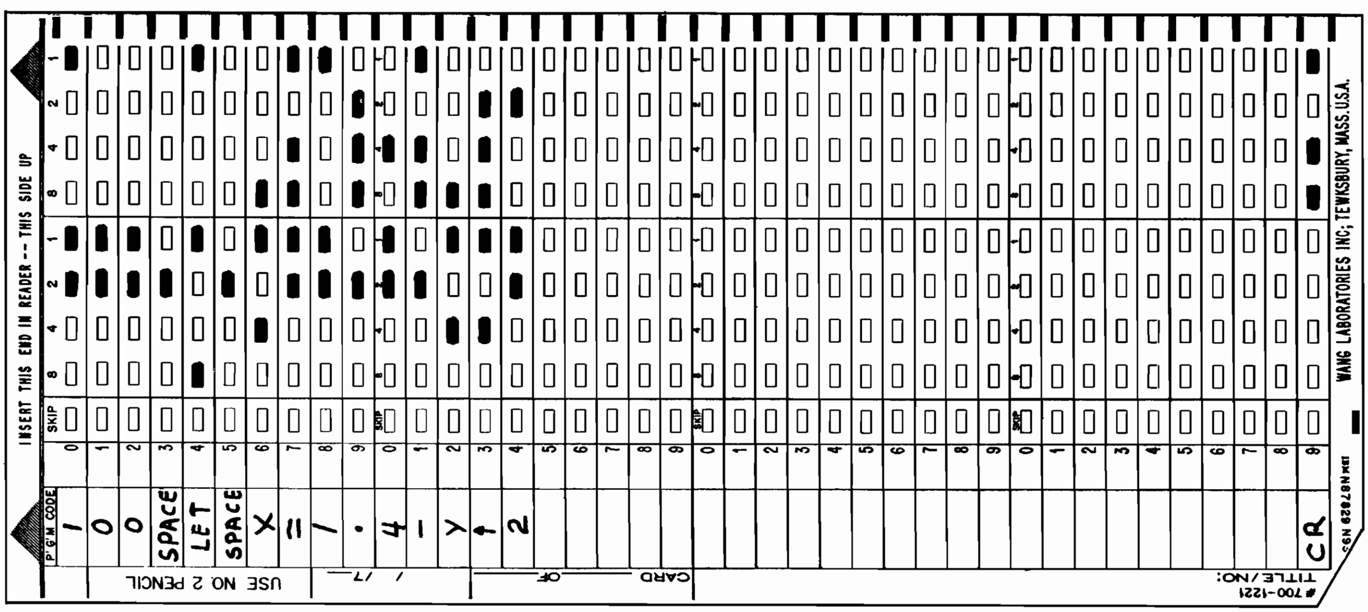
SELECT CI 001

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:

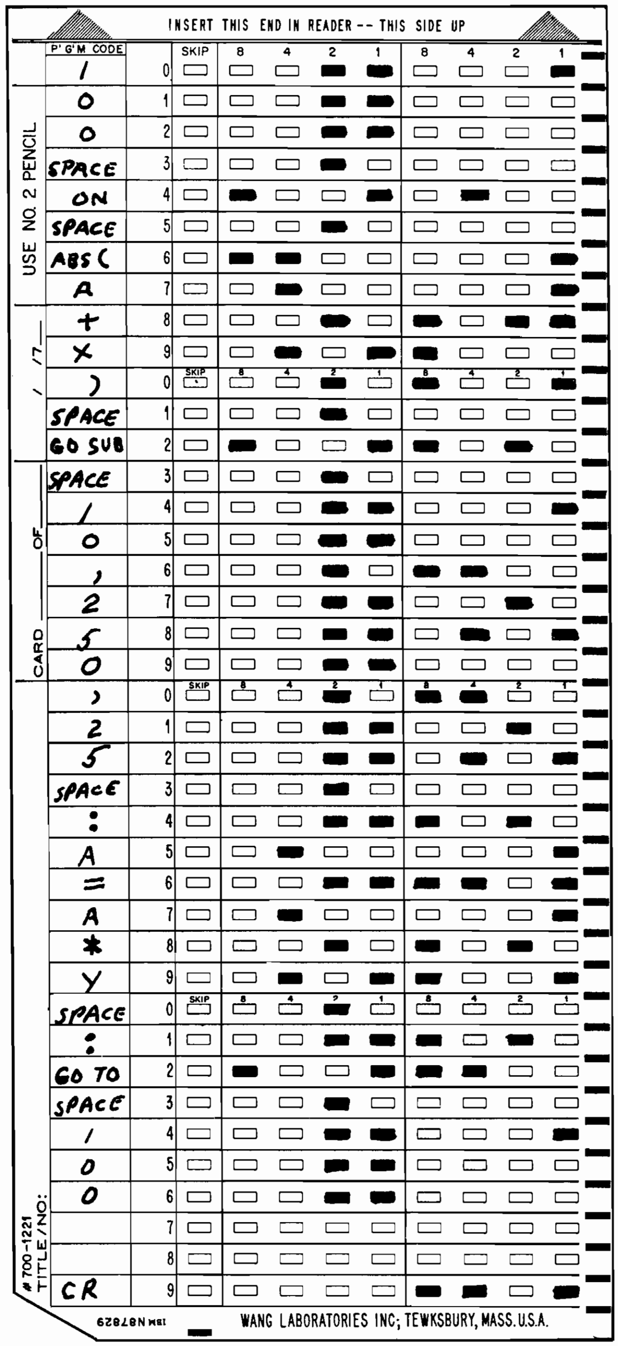
Program and command lines are marked on each card in ASCII character code format. Special single column codes (TEXT ATOMS, see Table at the end of this section) can be marked for BASIC statement verbs, functions, etc. Unmarked columns are ignored. A carriage return character should be marked at the end 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:



Example:

100 ON ABS(A+X) GOSUB 10, 250, 25: A=A*Y :GOTO 100



INPUT

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

Purpose

This statement allows the user to supply data via the 2214 Mark Sense Card Reader during the running of a program already stored in memory. The INPUT statement used with the Mark Sense Card Reader operates in a similar fashion as with the keyboard. In a program, INPUT is first selected by specifying the Mark Sense Card Reader Device Address and cards containing the data are read instead of keying in the data. The program then normally reselects INPUT back to the keyboard address. 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 A, B
        or
40  INPUT "ENTER VALUES OF A, B", A, B
50  SELECT INPUT 001
```

Statement 30 selects the card reader 517, for INPUT commands. When the INPUT statement is executed, this device is selected. The system then displays either a question mark or the optional input request message 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, the systems then proceeds the statement 50 where INPUT operations are selected back to the keyboard, address 001.

Use of the INPUT statement with the Mark Sense Card Reader is similar to its use with 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. (It would therefore be necessary to either place only one value on each card or to program restart procedures to allow an entire card with several values to be reread. The restart procedure would be to depress the RESET button and run the program starting at the INPUT statement line).

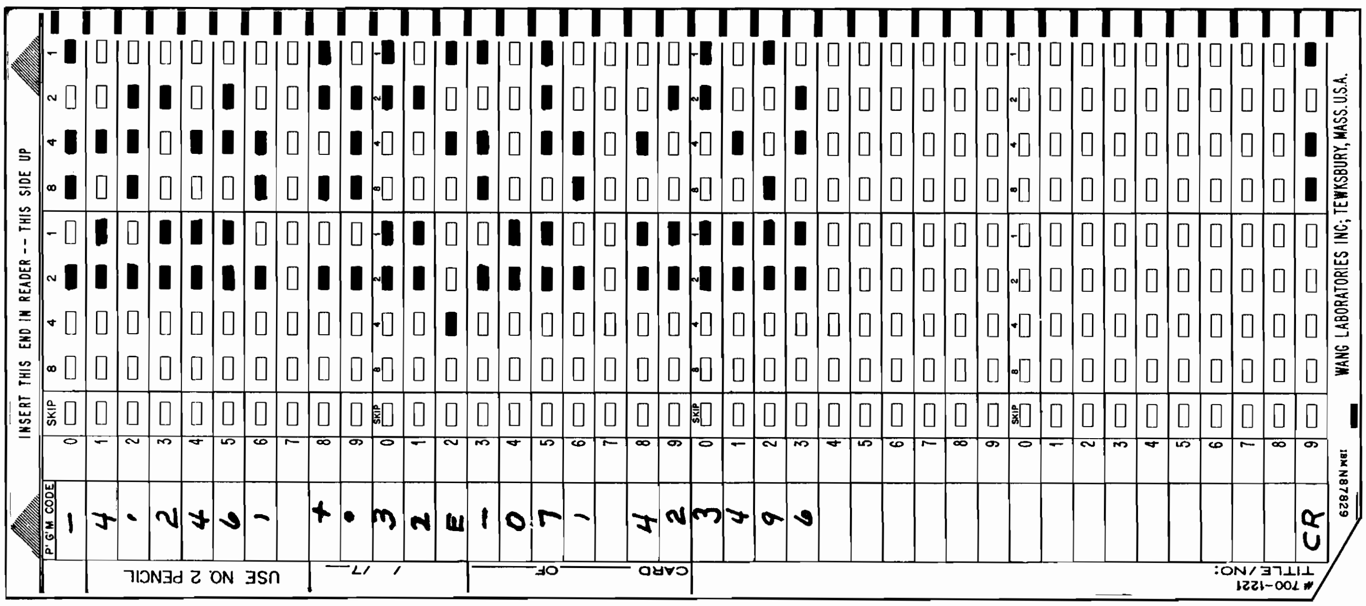
A user may terminate any INPUT statement sequence without supplying all required input values by simply entering a card with a carriage return character marked on it with no marked information preceding it. This would cause the system to proceed to the next program statement. The input list variables which have not received data will remain unchanged.

Card Format:

Data values are marked on the card in ASCII character code format (see Table at the end of this section). A carriage return character, HEX (0D), should be marked on the last column of the card (column 40). If more than one value is entered on a card, they should be separated by commas. Numeric data is marked in free-format (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 will be ignored and commas will act as string terminators. If leading blanks or commas are to be included, enclose the string in quotes. Space character 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.

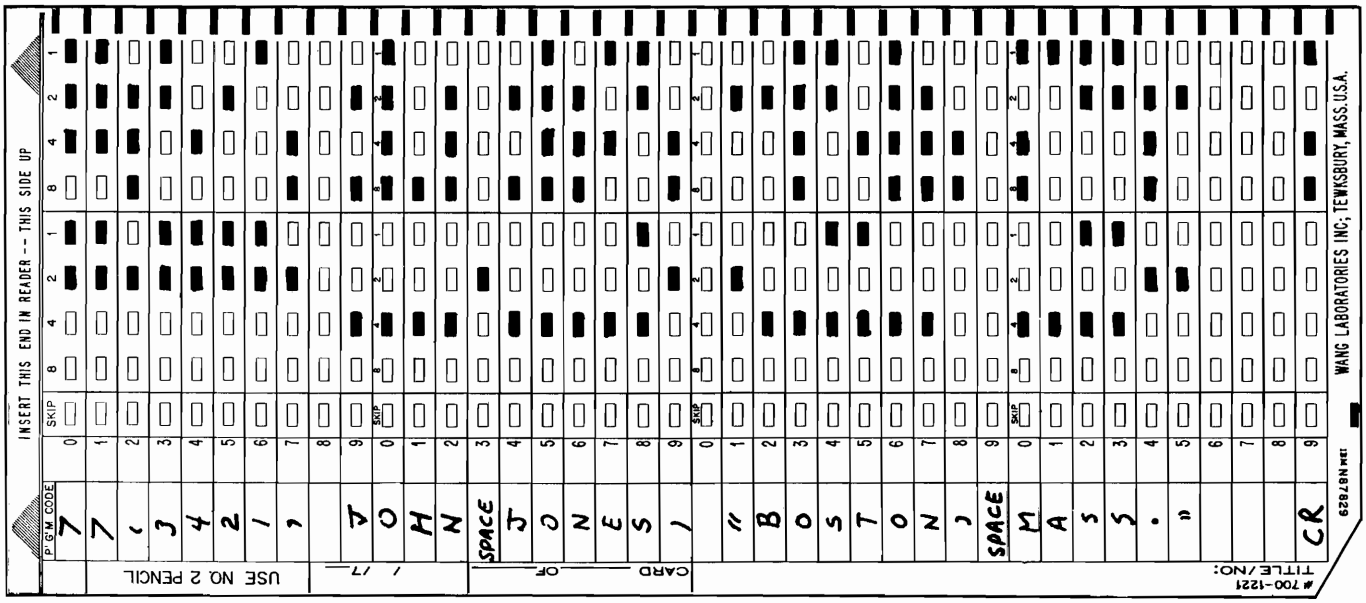
Example:

INPUT A, B2 C(3)



Example:

INPUT A, B$, C$(1,4)



NOTE:

In the above examples, the data also could be marked on three separate cards (with one value on each card). The commas separating data would not be required. Each card could, however, require a carriage return character in the last column.

DATALOAD

2214 MARK SENSE CARD READER, SYSTEM 2200B ONLY

General Form:

DATALOAD $\begin{bmatrix} \#n, \\ /xxx, \end{bmatrix}$ argument list

where:	#n	= logical file number to which a device address has been assigned (n is integer from 1 to 6).
	xxx	= device address of card reader (517).
	argument list	= $\left\{\begin{matrix}\text{variable} \\ \text{array designator}\end{matrix}\right\}$, . . .
	array designator	= array name( ) (e.g., N( ), A$( ))

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 values are marked by ASCII in any legal BASIC free-form format (i.e., 4.2, –732.71, 21.2+E07). Space characters and unmarked columns are ignored. Alphanumeric values are also marked in ASCII code. Quotes are not required. 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 will be successively read from one or more cards until all variables in the list are satisified 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 will then cause a valid transfer.

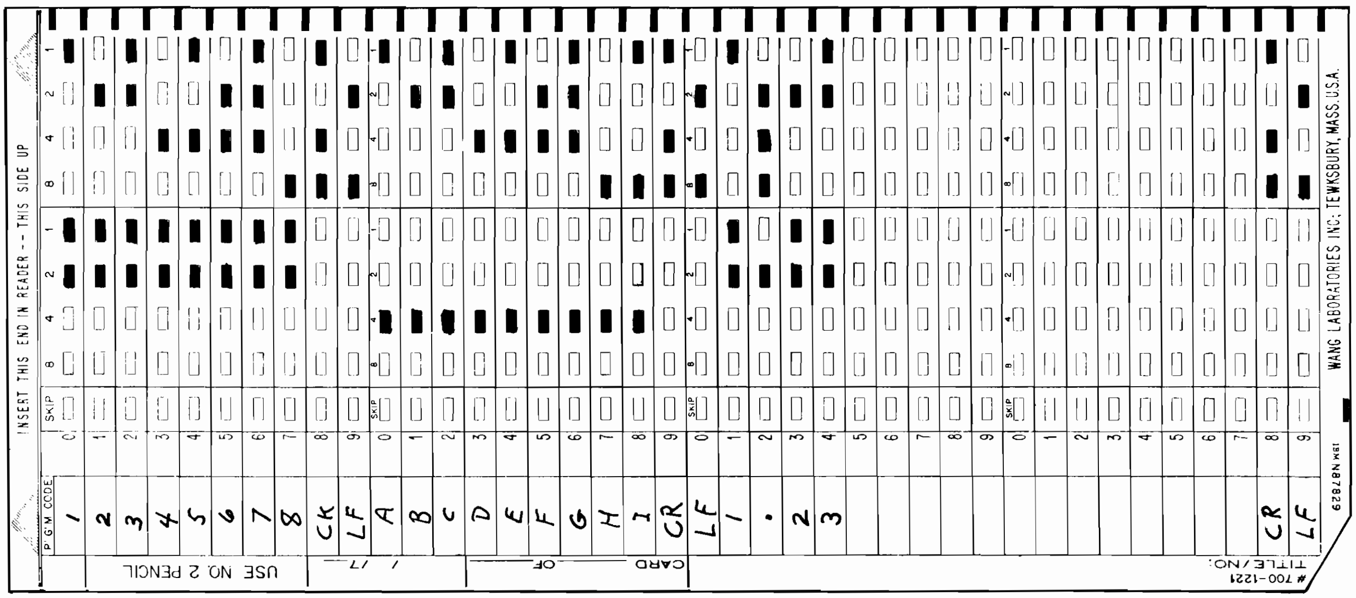
If the SKIP position is marked in any card column, that column will be ignored.

If a read error occurs which produces an illegal number format, an error message will be displayed, and the program will be terminated, the program can be restarted at the DATALOAD statement and all cards reread.

If no device address is specified, the address currently selected for TAPE will be used. This should be selected to 517.

Examples:

```
SELECT TAPE 517
DATALOAD X, Y, A$, B$
DATALOAD #3, N( ), A$, B$
DATALOAD /517, A1$( ), X, A$
```



DATALOAD BT

2214 MARK SENSE CARD READER, SYSTEM 2200B ONLY

General Form:

DATALOAD BT [([N=expression] ,[L= {xx / alpha-variable}] ,[S= {xx / alpha-variable}])] [#n, / /xxx,] {alpha-variable / alpha array designator} *

where:

N = number of characters to read (this is 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 character is used.
x = hexadecimal digit.
#n = file number to which a device address has been assigned (#1 – #6).
xxx = device address of card reader.

*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.

Examples:

```
DATALOAD BT (N = 40) /517, A$
SELECT TAPE 517
DATALOAD BT (N = 40) A$( )
SELECT #1 517
DATALOAD BT (N = 40, L = FF, S = 99) #1, B$
```

INFORMATION FOR THE MODEL 2214 MARK SENSE CARD READER

8-BIT HEX CODES (TEXT ATOMS) FOR THE 2200 BASIC LANGUAGE WORDS

BASIC WORD	HEXADECIMAL	BINARY	BASIC WORD	HEXADECIMAL	BINARY
#PI	CC	11001100	GOTO	9C	10011100
ABS(	C1	11000001	HEX(	D2	11010010
AND	8A	10001010	IF	9F	10011111
ARC	CB	11001011	INPUT	99	10011001
ATN(	D4	11010100	INT(	C5	11000101
BA	BE	10111110	KEYIN	88	10001000
BACKSPACE	AB	10101011	LEN(	D5	11010101
BEG	B3	10110011	LET	91	10010001
BIN(	DE	11011110	LIMITS	86	10000110
BT	DA	11011010	LIST	80	10000000
CI	B5	10110101	LOAD	A1	10100001
CLEAR	81	10000001	LOG(	C6	11000110
CO	B8	10111000	MAT	A8	10101000
COM	A6	10100110	MOVE	AD	10101101
CONTINUE	84	10000100	NEXT	9D	10011101
CONVERT	AE	10101110	NUM(	DD	11011101
COS(	C3	11000011	OFF	BA	10111010
DA	BD	10111101	ON	94	10010100
DATA	97	10010111	OPEN	B4	10110100
DBACKSPACE	BB	10111011	OR	8B	10001011
DC	BF	10111111	PLOT	A4	10100100
DEFFN	CE	11001110	PLOT(SEL)	AF	10101111
DIM	93	10010011	POS(	DF	11011111
DISK	8E	10001110	PRINT	A0	10100000
DSKIP	89	10001001	PRINTUSING	A7	10100111
END	96	10010110	RE	D6	11010110
EXP(	C4	11000100	READ	98	10011000
FN	C0	11000000	REM	A2	10100010
FOR	9E	10011110	RENUMBER	83	10000011
GOSUB	9A	10011010	RESTORE	A3	10100011

8-BIT HEX CODES (TEXT ATOMS) FOR THE SYSTEM 2200 BASIC LANGUAGE WORDS

BASIC WORD	HEXADECIMAL	BINARY	BASIC WORD	HEXADECIMAL	BINARY
RETURN	9B	10011011	STOP	95	10010101
REWIND	A9	10101001	STR(	D3	11010011
RND(	C9	11001001	TAB(	CD	11001101
RUN	82	10000010	TAN(	CA	11001010
SAVE	85	10000101	TAPE	8F	10001111
SCRATCH	AC	10101100	TEMP	8D	10001101
SELECT	A5	10100101	THEN	B1	10110001
SGN(	C8	11001000	TO	B2	10110010
SIN(	C7	11000111	TRACE	90	10010000
SKIP	AA	10101010	VAL(	DC	11011100
SQR(	C2	11000010	VERIFY	BC	10111100
STEP	B0	10110000	XOR	8C	10001100

INFORMATION FOR THE MODEL 2214 MARK SENSE CARD READER

ASCII CODES FOR THE 2200 BASIC LANGUAGE CHARACTERS AND SYMBOLS

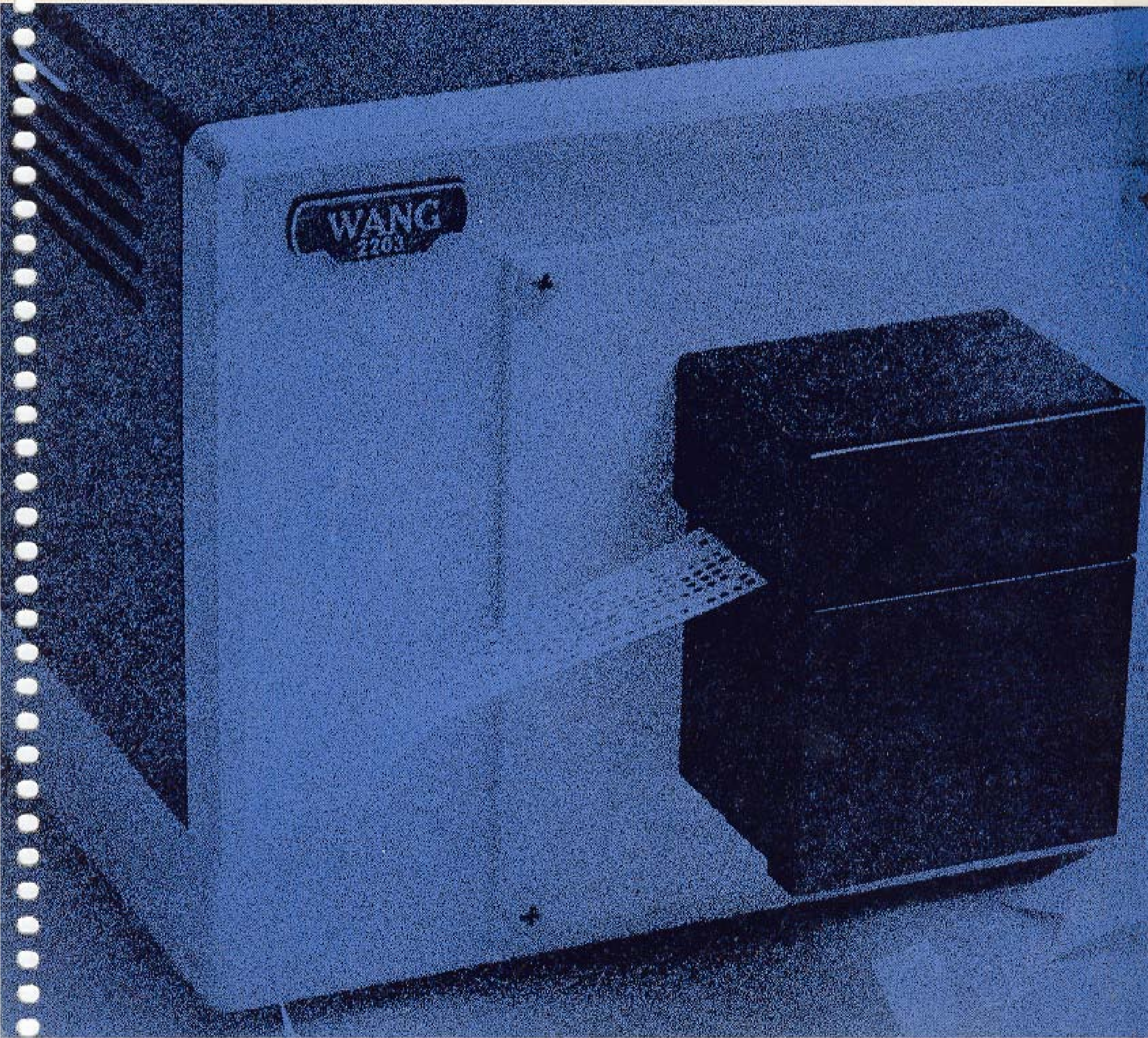
CHARACTER CODE

BASIC SYMBOL	HEXADECIMAL	BINARY	BASIC SYMBOL	HEXADECIMAL	BINARY
!	21	00100001	@	40	01000000
#	23	00100011	A	41	01000001
$	24	00100100	a	61	01100001
%	25	00100101	B	42	01000010
&	26	00100110	b	62	01100010
(	28	00101000	[	5B	01011011
)	29	00101001	]	5D	01011101
*	2A	00101010	C	43	01000011
+	2B	00101011	c	63	01100011
, (comma)	2C	00101100	CARRIAGE RETURN	0D	00001101
–	2D	00101101	D	44	01000100
.	2E	00101110	d	64	01100100
/ (slash)	2F	00101111	"	22	00100010
0	30	00110000	E	45	01000101
1	31	00110001	e	65	01100101
2	32	00110010	F	46	01000110
3	33	00110011	f	66	01100110
4	34	00110100	G	47	01000111
5	35	00110101	g	67	01100111
6	36	00110110	H	48	01001000
7	37	00110111	h	68	01101000
8	38	00111000	I	49	01001001
9	39	00111001	i	69	01101001
:	3A	00111010	J	4A	01001010
;	3B	00111011	j	6A	01101010
<	3C	00111100	K	4B	01001011
=	3D	00111101	k	6B	01101011
>	3E	00111110	L	4C	01001100
?	3F	00111111	l	6C	01101100

ASCII CODES FOR THE SYSTEM 2200 BASIC LANGUAGE CHARACTERS AND SYMBOLS

BASIC SYMBOL	HEXADECIMAL	BINARY	BASIC SYMBOL	HEXADECIMAL	BINARY
LINE FEED	0A	00001010	T	54	01010100
M	4D	01001101	t	74	01110100
m	6D	01101101	U	55	01010101
N	4E	01001110	u	75	01110101
n	6E	01101110	V	56	01010110
O	4F	01001111	v	76	01110110
o	6F	01101111	W	57	01010111
P	50	01010000	w	77	01110111
p	70	01110000	X	58	01011000
Q	51	01010001	x	78	01111000
q	71	01110001	X-OFF	13	00010011
R	52	01010010	X-ON	11	00010001
r	72	01110010	Y	59	01011001
S	53	01010011	y	79	01111001
s	73	01110011	Z	5A	01011010
' (single quote)	27	00100111	z	7A	01111010
SPACE	20	00100000	↑	5E	01011110

SECTION XII
PAPER TAPE READER



Section XII

Paper Tape Reader

SYSTEM 2200B ONLY

DATALOAD

2203 PUNCHED TAPE READER, SYSTEM 2200B ONLY

General Form:

DATALOAD $\begin{bmatrix} \#n, \\ /xxx, \end{bmatrix}$ argument list

where #n = Logical file number to which a device address has been assigned (n is integer from 1 to 6).

xxx = Device address of paper 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 list = $\begin{Bmatrix} \text{variable} \\ \text{array designator} \end{Bmatrix}$, . . .

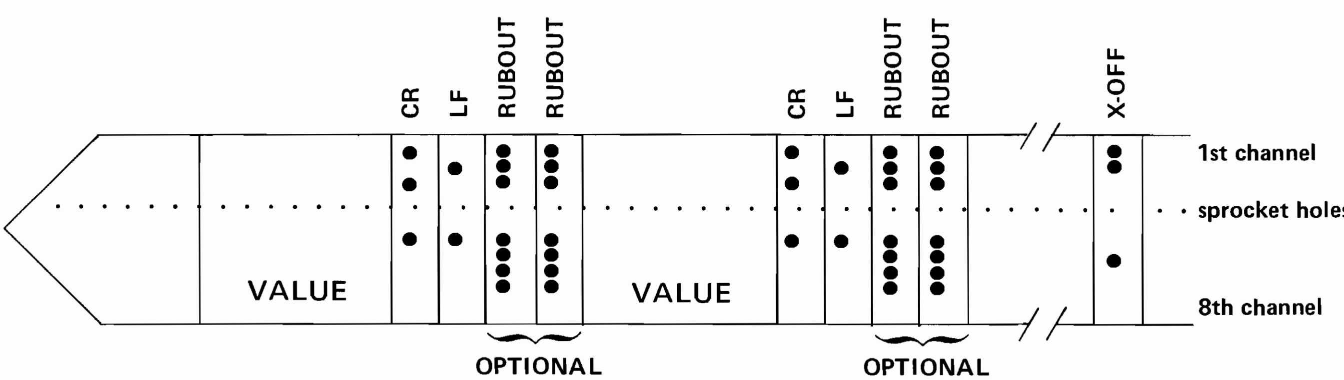
array designator = array name ( ) e.g., A( ), A$( ).

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

2203 PUNCHED TAPE READER, SYSTEM 2200B ONLY

General Form:

DATALOAD BT [R] [([N=expression], [L={xx / alpha-variable}], [S={xx / alpha-variable}])] [#n, / /xxx,] {alpha variable / alpha array designator} *

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.
x	=	Hexadecimal digit (i.e., 0-9 or A-F).
#n	=	Logical file number to which a device address has been assigned (n is integer from 1 to 6).
xxx	=	Device address of paper tape reader. If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.

*Commas must separate N, L, S arguments if more than one is present.

Purpose

This statement reads a paper 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 paper tapes in any format to be read by the System 2200. The data read then can be converted into a form usable by the System 2200 using the System 2200 data manipulation statements.

Examples:

```
DATALOAD BT /618, A$
DATALOAD BTR (L=FF, S=OD) #1, A$(  )
DATALOAD BT (N=100) A$( )
DATALOAD BT (N=200, L=00, S=A$) /618, B$(  )
```

LOAD

2203 PUNCHED TAPE READER SYSTEM 2200B ONLY

General Form:

LOAD [#n | /xxx]

where #n = File number to which a device address is currently assigned (n is an integer from 1 to 6).

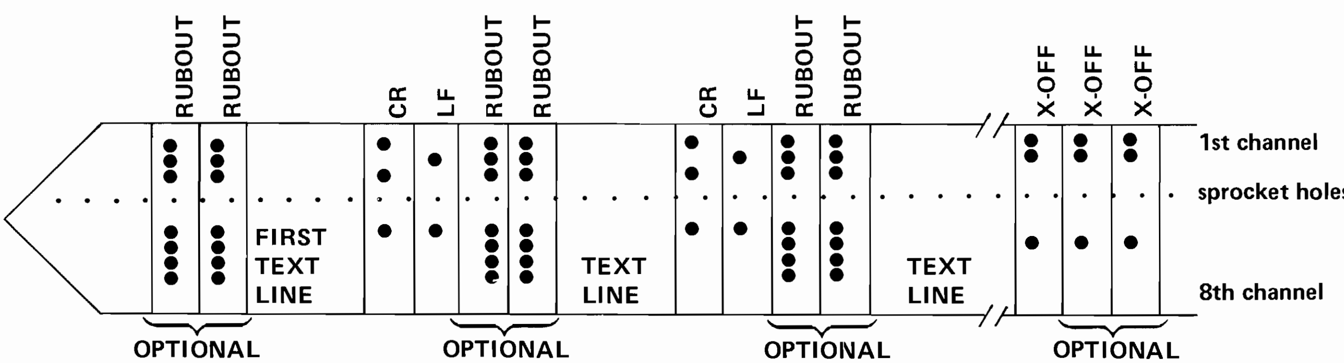
xxx = Device address of device to load from.

If neither of the above is specified, the default device address (the device address currently assigned to TAPE (see SELECT)) is used.

Purpose

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 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
```

LOAD

2203 PUNCHED TAPE READER SYSTEM 2200B ONLY

General Form:

LOAD [#n, | /xxx,] [line number 1 [,line number 2]]

where #n = File number to which the device is currently assigned (n is an integer from 1 to 6).

xxx = 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.

line number 1 = The line number of the first line to be deleted from the program currently in memory, before loading the new program. After loading, execution continues at the line number 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

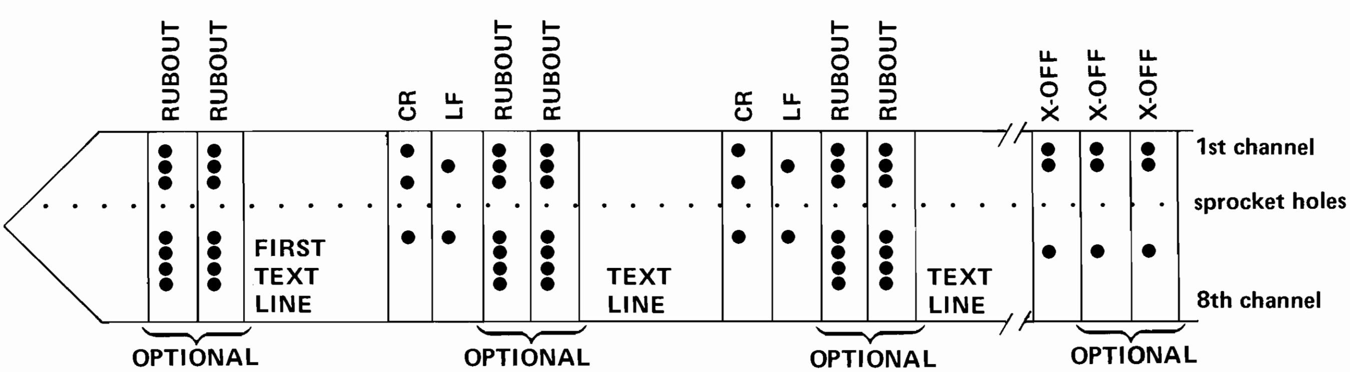
This is a BASIC program statement which, in effect, produces an automatic combination of the following:

STOP	(stop current program execution)
CLEAR P	[line number 1 [line number 2]] (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 normal user intervention. Common variables are passed between program segments. LOAD must be the last statement on a statement line.

To be read, the paper 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.

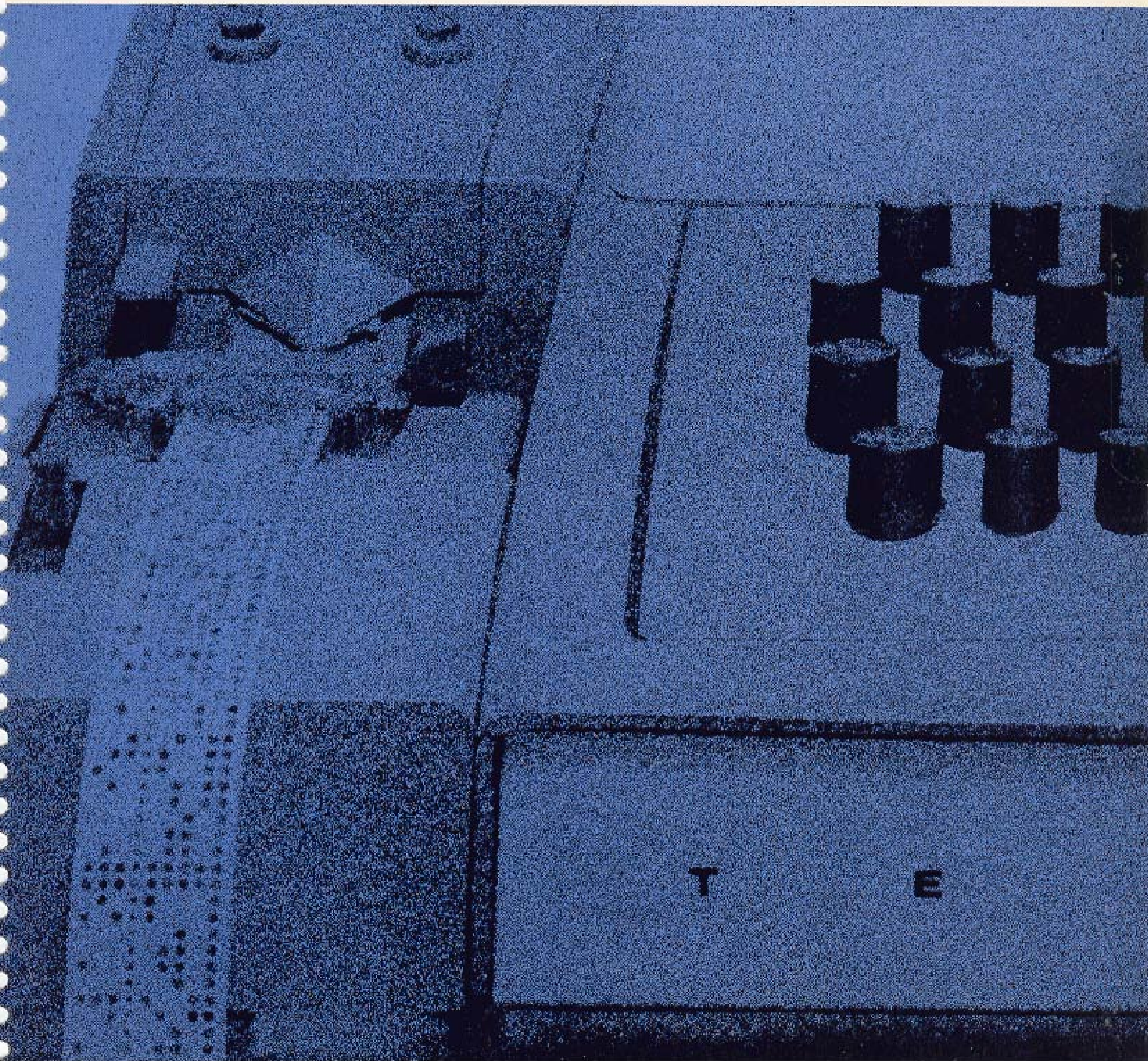
Text lines are punched in ASCII character code and are separated by CR LF RUBOUT RUBOUT; the rubouts are optional but are punched when a program is saved on Teletype. The program is terminated by an X-OFF character. LOAD reads only the first seven channels of the paper tape; the 8th bit is always read as 0. Nonpunched frames and RUBOUTS are ignored when reading the tape.

In immediate execution 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
```

SECTION XIII
TELETYPE



Section XIII

Teletype

INTRODUCTION

MODEL 2207 TELETYPE INTERFACE CONTROLLER

The Model 2207 Teletype Interface is used to interface a Teletype to the System 2200. In this configuration, the Teletype keyboard and printer are used in a similar manner to the System 2200 keyboard or CRT, providing the appropriate device address is selected for console input, console output, and print operations. Therefore, most console, input, and print operations which apply to a System 2200 keyboard and CRT also apply to the Teletype keyboard and printer. The CRT cursor and screen control operations, however, are not valid on a Teletype; HALT/STEP and RESET are produced by the Teletype BREAK and ESC keys, respectively. For editing, the backarrow (←) key acts as a backspace key to delete the last character entered and the backslash (\) key acts as a line erase key.

The 2207 Teletype Interface has two device addresses associated with it: one for input and one for output. For paper tape read or punch operations, the output address with device type 4 is always used. The following device addresses are used to select the Teletype.

019 (or 01A, 01B)	– Teletype keyboard input
01D (or 01E, 01F)	– Teletype printer output
41D (or 41E, 41F)	– Teletype paper tape read or punch operations

In addition, a number of System 2200B BASIC statements and commands are provided to utilize the paper tape reader and punch. They are presented in this section.

DATALOAD

TELETYPE STATEMENT SYSTEM 2200B ONLY

General Form:

DATALOAD [#n, /xxx,] argument list

where #n = Logical file number to which a device address has been assigned (n is integer from 1 to 6).

4xx = 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 list = {variable / array designator} , . . .

array designator = array name ( ) e.g., A$( ), B( )

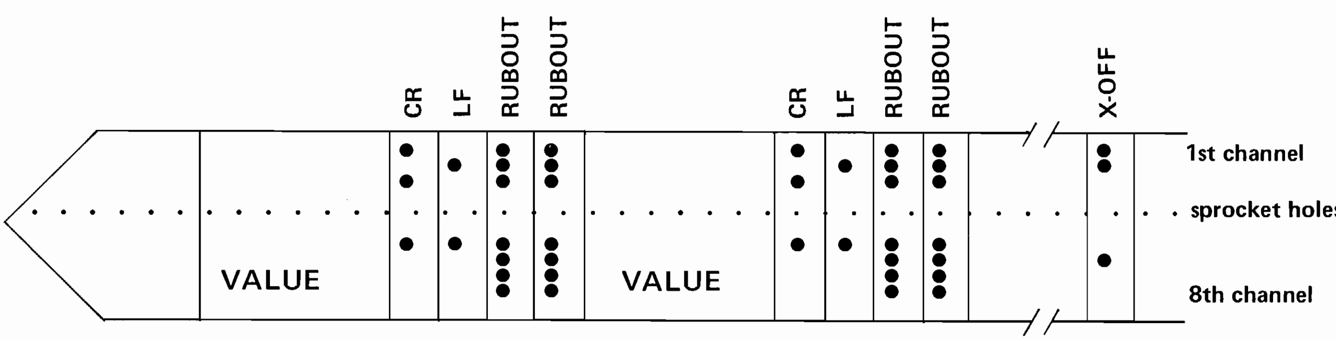
Purpose

This statement reads values from the Teletype paper 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 paper 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

TELETYPE STATEMENT SYSTEM 2200B ONLY

General Form:

DATALOAD BT $\left[\left(\left[N\text{=expression}\right], \left[L=\begin{Bmatrix}xx\\ \text{alpha-variable}\end{Bmatrix}\right], \left[S=\begin{Bmatrix}xx\\ \text{alpha-variable}\end{Bmatrix}\right]\right)\right] \left[\begin{matrix}\#n,\\ /4xx,\end{matrix}\right] \begin{Bmatrix}\text{alpha variable}\\ \text{alpha array designator}\end{Bmatrix}$ *

where 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 code.

x = Hexadecimal digit (i.e., 0-9 or A-F).

#n = Logical file number to which a device address has been assigned (n is integer from 1 to 6).

4xx = 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.

*Commas must separate N, L, S arguments if more than one is present.

Purpose

This statement reads a paper tape 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 paper tape are read.

The System 2200 automatically sends out an X-ON character to start the Teletype 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 by $\geqslant 3$. 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 $\geqslant 3$. 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 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 not equal to the leader code is recognized).

DATALOAD BT permits paper tapes in any format to be read by the System 2200. The data read then can be converted into a form usable by the System 2200 using System 2200 data manipulation statements.

Examples:

```
DATALOAD BT /41D, A$
DATALOAD BT (L=FF, S=OD) #1, A$( )
DATALOAD BT (N = 100) A$( )
DATALOAD BT (N=20, L=00, S=A$) A1$( )
```

DATASAVE

TELETYPE STATEMENT SYSTEM 2200B ONLY

General Form:

$$\text{DATASAVE} \begin{bmatrix} \#n, \\ /4xx, \end{bmatrix} \begin{Bmatrix} \text{OPEN "name"} \\ \text{END} \\ \text{argument list} \end{Bmatrix}$$

where #n = Logical file number to which a device address has been assigned (n is integer from 1 to 6).

4xx = 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.

$$\text{argument list} = \begin{Bmatrix} \text{literal string} \\ \text{alpha variable} \\ \text{expression} \\ \text{array designator} \end{Bmatrix} , \ldots$$

array designator = variable name ( ) (e.g., A$( ), B( )

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

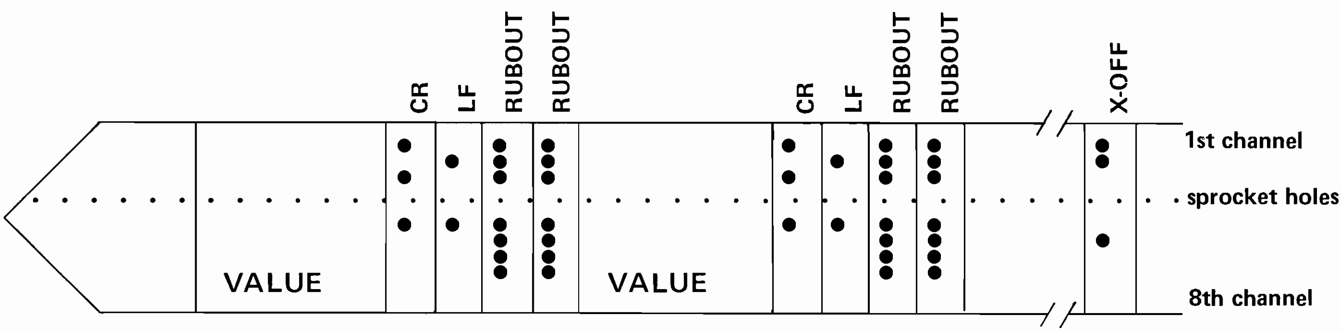
This statement causes the values specified in the argument list to be punched on paper tape. Numeric values are written in a form identical to that resulting from a PRINT statement:

Format 1: SM.MMMMMMMME+XX $10^{-1} > \text{value} \geqslant 10^{+13}$
Format 2: SZZZZZZ.FFFFFFF $10^{-1} \leqslant \text{value} < 10^{+13}$

where:
M = mantissa digits
X = exponent digits
F = fraction digits
Z = integer digits
S = minus sign if value $<$ 0, or blank if value $\geqslant$ 0

Alphanumeric values are written identically to the character string data they contain; trailing spaces in values of alphanumeric variables are not written. Alphanumeric values must not contain any of the following characters; CR, RUBOUT, X-OFF, null. The OPEN parameter writes leader code (50 null characters). The END parameter terminates the data file by punching an X-OFF character and trailer code (50 null characters).

The paper tape is punched in the following format:



Values are punched in ASCII character code and are separated by CR LF RUBOUT RUBOUT.

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

TELETYPE STATEMENT SYSTEM 2200B ONLY

General Form:

DATASAVE BT [#n, | /4xx,] {alpha variable | alpha array designator}

where #n = Logical file number to which a device address has been assigned (n is integer from 1 to 6).

4xx = 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.

alpha array designator = alpha array name ( ) (e.g., A$( ))

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.

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$( )
```

LOAD

TELETYPE COMMAND SYSTEM 2200B ONLY

General Form:

LOAD [#n, | /4xx,]

where #n = File number to which a device address is currently assigned (n - an integer from 1 to 6).

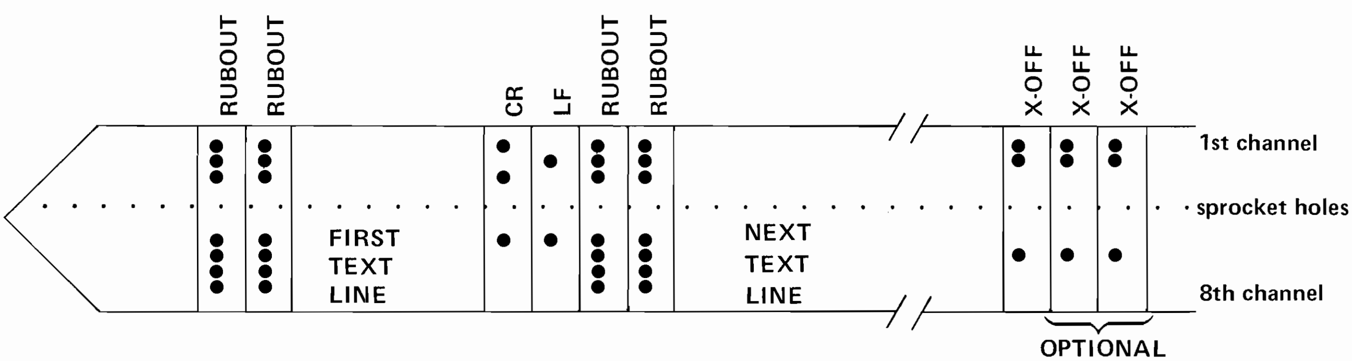
4xx = Device address of device to load from. (41D, 41E, or 41F)

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
```

LOAD

TELETYPE STATEMENT SYSTEM 2200B ONLY

General Form:

LOAD [#n, or /4xx,] [line number 1 [,line number 2]]

where #n = File number to which the device is currently assigned (n is an integer form 1 to 6).

4xx = 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 program currently in memory, before loading the new program. After loading, execution continues at the line whose number is equal to 'line number 1'. An error will result 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

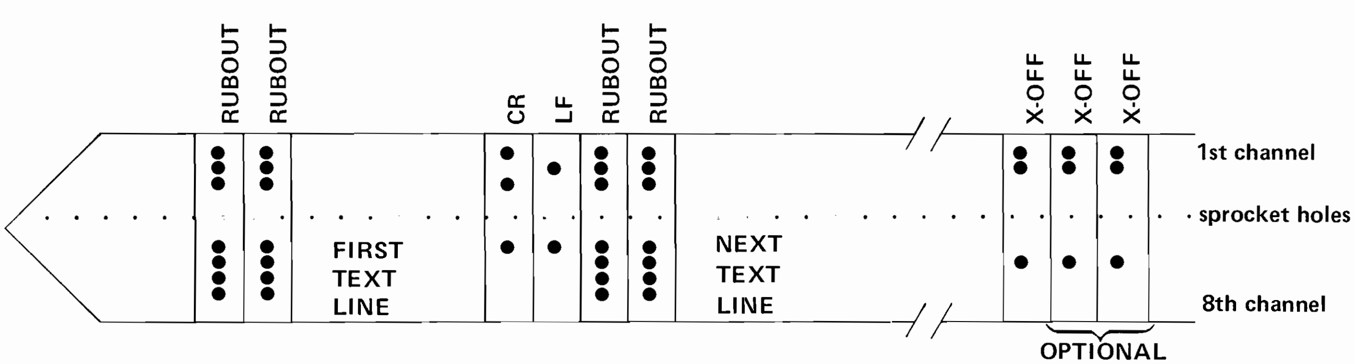
This is a BASIC program statement which, in effect, produces an automatic combination of the following:

STOP	(stop current program execution)
CLEAR P	[line number 1 [,line number 2]] (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 normal user intervention. Common variables are passed between program segments. LOAD must be the last statement on a statement line.

To be read, the paper tape must conform to the following format:



The LOAD statement must not be within a FOR/NEXT loop or subroutine; an error results when the NEXT or RETURN statement is encountered.

LOAD

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 execution 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
```

SAVE

TELETYPE COMMAND SYSTEM 2200B ONLY

General Form:

SAVE [#n, | /4xx,] [line number [,line number]]

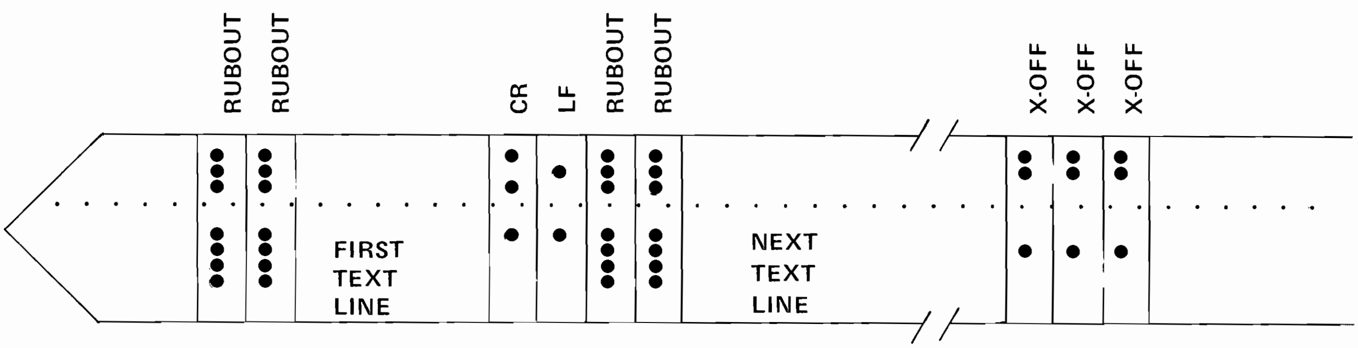
where	#n	= File number to which device address is assigned (#1 to #6).
	4xx	= Device address of Teletype. (41D, 41E, or 41F)
		If neither of the above is specified, the default device address (the device address currrently assigned to TAPE, (see SELECT)) is used.
	1st line number	= Starting line number 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:

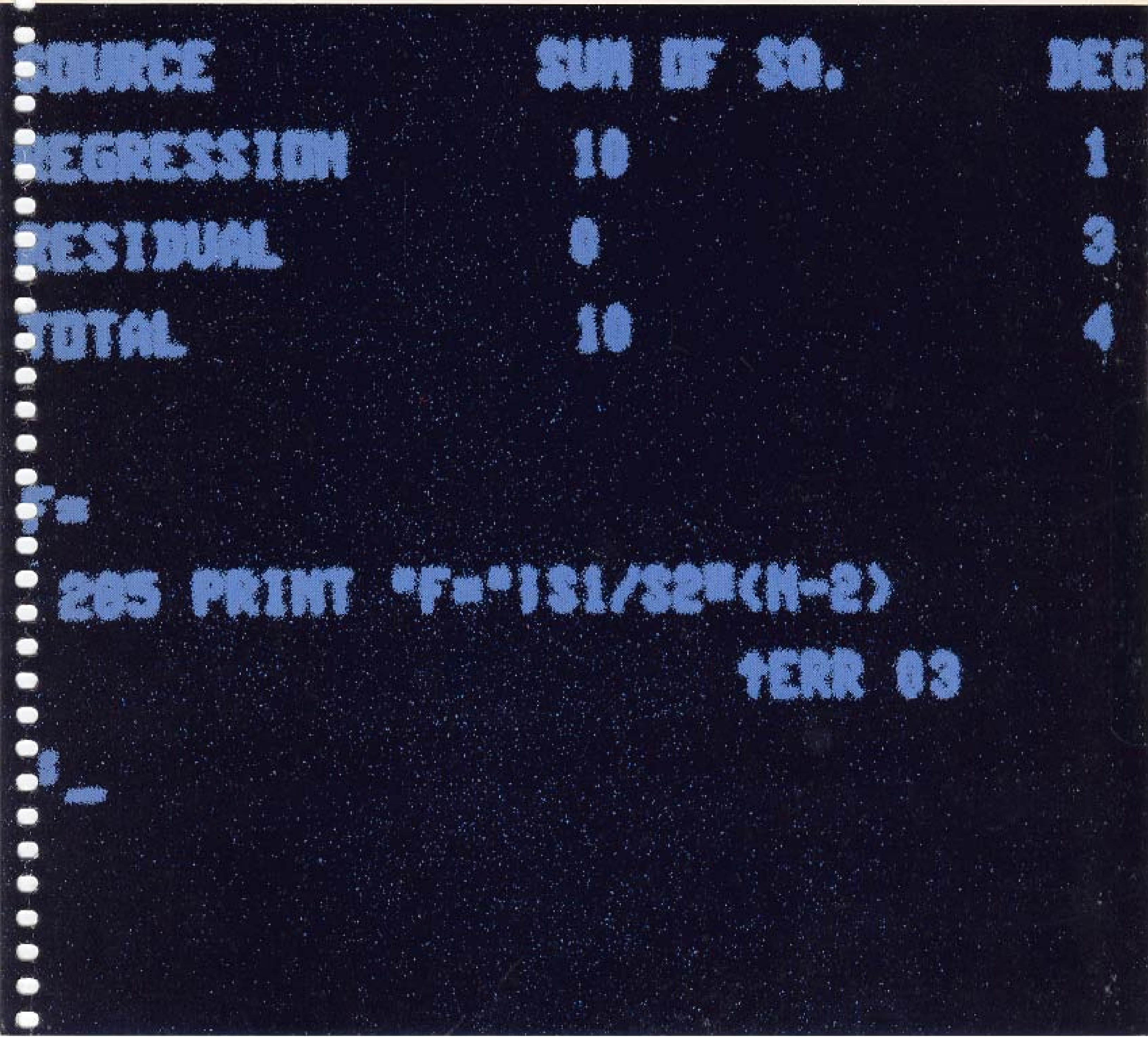


Text lines are punched in ASCII character code and are separated by CR LF RUBOUT RUBOUT. The program is terminated by 3 X-OFF's.

Examples:

```
SAVE
SAVE #3
SAVE /41D
SAVE /41D, 100, 200
SAVE #5, 400
```

SECTION XIV
ERROR CODES



Section XIV

Error Codes

Section XIV Error Codes

The Wang System 2200 BASIC checks for and displays syntax errors as each line is entered. The user may then correct the error before proceeding with his program. When any error is detected, the line being scanned by the system is displayed and on the next line, an "↑" symbol is placed at the point of the error followed by the error message number.

The following example shows the format of the System 2200 error pointer:

```
:10  DIM A(P)
            ↑ ERR 13
```

The user may then refer to the listing of error messages to identify the error 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.*

THREE TYPES OF ERRORS CAN OCCUR

A Syntax Error

Results when the required format of a System 2200 BASIC statement is violated. Pressing a sequence of keys not recognized as an accepted combination results in this type of error. Syntax errors in a statement are recognized and noted, as soon as the execute key is touched to enter a statement. Examples of this type of error include mispelling verbs, illegal formats for numbers, operators, parentheses, and the improper use of punctuation.

Example:

```
:10  DEFFN . (X) = 3*X↑2 - 2*X↑3
             ↑ ERR 21
```

Section XIV   Error Codes

An Error of Execution

Results when an illegal arithmetic operation is performed, or the execution of an illegal statement or programming procedure is attempted when a program is executed. This type of error differs from a Syntax Error. The statement itself uses the proper syntax. However, the execution of the statement is impossible to perform and leads to an error condition. Typical errors of this type include illegal branches, arithmetic overflow or underflow, illegal "FOR" loops, etc.

Example:

(Branch to non-existant statement number)

```
:100 GOTO 110
:105 PRINT "VALUES = " ;A, B, C
:120 END
:RUN
 100 GOTO 110
          ↑ERR 11
```

A Programming Error

The 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 internal operating system tables and variables has been used up (storage of variables, values, etc.) or a repetative program loop which illegally allows system tables to fill up was encountered. An example of the latter would be jumping out of FOR loops or subroutines without completing them.

Action: Shorten or correct and/or chain the program by using COM statements and continue.

Example:

```
:10  DIM A(19), B(10, 10), C(10, 10)
:RUN
↑ERR 02
```

(the table space required for variables exceeded the table limit for variable storage as line 10 was processed)

User must compress program and variable storage requirements.

CODE 03

Error: **Math Error**

Cause:

1. EXPONENT OVERFLOW. The resulting magnitude of the number calculated was greater than or equal to 10^{100}. (+ , – , * , / , ↑ , TAN , EXP).
2. DIVISION BY ZERO.
3. NEGATIVE OR ZERO LOG FUNCTION ARGUMENT.
4. NEGATIVE SQR FUNCTION ARGUMENT.
5. INVALID EXPONENTIATION. An exponentiation, (X↑Y) was attempted where X was negative and Y was not an integral, producing an imaginary result, or X and Y were both zero.
6. ILLEGAL SIN, COS, OR TAN ARGUMENT. The function argument exceeds $2\pi \times 10^{11}$ 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:

```
:10 DEF FNA V) = SIN(3*V-1)
              ↑ERR 04
:10 DEF FNA(V) + SIN(3*V-1)          (Possible Correction)
```

CODE 05

Error: **Missing Right Parenthesis**

Cause: A right ( )) parenthesis was expected.

Action: Correct statement text.

Example:

```
:10 Y = INT(1.2↑5
                  ↑ERR 05
:10 Y = INT(1.2↑5)                    (Possible Correction)
```

CODE 06

Error: **Missing Equals Sign**

Cause: An equals sign (=) was expected.

Action: Correct statement text.

Example:

```
:10  DEFFNC(V) – V + 2
                 ↑ERR 06
:10  DEFFNC(V) = V+2                  (Possible Correction)
```

CODE 07

Error: **Missing Quotation Marks**

Cause: Quotation marks were expected.

Action: Reenter the DATASAVE OPEN statement correctly.

Example:

```
:DATASAVE OPEN TTTT"
                    ↑ERR 07

:DATASAVE OPEN "TTTT"                (Possible Correction)
```

CODE 08

Error: **Undefined FN Function**

Cause: An undefined FN function was referenced.

Action: Correct program to define or reference the function correctly.

Example:

```
:10 X=FNC(2)
:20 PRINT "X";X
:30 END
:RUN
10 X=FNC(2)
       ↑ERR 08
:05 DEFFNC(V)=COS(2*V)               (Possible Correction)
```

CODE 09

Error: **Illegal FN Usage**

Cause: More than five levels of nesting were encountered when evaluating an FN function.

Action: Reduce the number of nested functions.

Example:

```
:10 DEF FN1(X)=1+X       :DEF FN2(X)=1+FN1(X)
:20 DEF FN3(X)=1+FN2(X)     :DEF FN4(X)=1+FN3(X)
:30 DEF FN5(X)=1+FN4(X)     :DEF FN6(X)=1+FN5(X)
:40 PRINT FN6(2)
:RUN
10 DEF FN1(X)=1+X        :DEF FN2(X)=1+FN1(X)
               ↑ERR 09
:40 PRINT 1+FN5(2)                    (Possible Correction)
```

CODE 10

Error: **Incomplete Statement**

Cause: The end of the statement was expected.

Action: Complete the statement text.

Example:

```
:10 PRINT X"
              ↑ERR 10
:10 PRINT "X"
      OR
:10 PRINT X                           (Possible Correction)
```

CODE 11

Error: **Missing Line Number or Continue Illegal**

Cause: The line number is missing or a referenced line number is undefined; or the user is attempting to continue program execution after one of the following conditions: A text or table overflow error, a new variable has been entered, a CLEAR command has been entered, the user program text has been modified, or the RESET key has been pressed.

Action: Correct statement text.

Example:

```
:10 GOSUB 200
             ↑ERR 11
:10 GOSUB 100                         (Possible Correction)
```

CODE 12

Error: **Missing Statement Text**

Cause: The required statement text is missing (THEN, STEP, etc.).

Action: Correct statement text.

Example:

```
:10 IF I+12*X,45
               ↑ERR 12
:10 IF I=12*X THEN 45                 (Possible Correction)
```

CODE 13

Error: **Missing or Illegal Integer**

Cause: A positive integer was expected or an integer was found which exceeded the allowed limit.

Action: Correct statement text.

Example:

```
:10 COM D(P)
            ↑ERR 13
:10 COM D(8)                       (Possible Correction)
```

CODE 14

Error: **Missing Relation Operator**

Cause: A relational operator (<, =, >, <=, >=, <>) was expected.

Action: Correct statement text.

Example:

```
:10 IF A-B THEN 100
           ↑ERR 14
:10 IF A=B THEN 100                (Possible Correction)
```

CODE 15

Error: **Missing Expression**

Cause: A variable, or number, or a function was expected.

Action: Correct statement text.

Example:

```
:10 FOR I=, TO 2
           ↑ERR 15
:10 FOR I=1 TO 2                   (Possible Correction)
```

CODE 16

Error: **Missing Scalar**

Cause: A scalar variable was expected.

Action: Correct statement text.

Example:

```
:10 FOR A(3)=1 TO 2
            ↑ERR 16
:10 FOR B=1 TO 2                   (Possible Correction)
```

CODE 17

Error: **Missing Array**

Cause: An array variable was expected.

Action: Correct statement text.

Example:

```
:10 DIM A2
          ↑ERR 17
:10 DIM A(2)                       (Possible Correction)
```

CODE 18

Error:	**Illegal Value for Array Dimension**
Cause:	The value exceeds the allowable limit. For example, a dimension is greater than 255 or an array variable subscript exceeds the defined dimension.
Action:	Correct the program.

Example:

```
:10 DIM A(2,3)
:20 A(1,4) = 1
:RUN
 20 A(1,4) = 1
        ↑ERR 18
:10 DIM A(2,4)                    (Possible Correction)
```

CODE 19

Error:	**Missing Number**
Cause:	A number was expected.
Action:	Correct statement text.

Example:

```
:10 DATA L
          ↑ERR 19
:10 DATA 1                        (Possible Correction)
```

CODE 20

Error:	**Illegal Number Format**
Cause:	A number format is illegal.
Action:	Correct statement text.

Example:

```
:10 A=12345678.234567                        (More than 13 digits of mantissa)
                     ↑ERR 20
:10 A=12345678.23456              (Possible Correction)
```

CODE 21

Error:	**Missing Letter or Digit**
Cause:	A letter or digit was expected.
Action:	Correct statement text.

Example:

```
:10 DEF FN.(X)=X↑5-1
           ↑ERR 21
:10 DEF FN1(X)=X↑5-1              (Possible Correction)
```

CODE 22

Error: **Undefined Array Variable**

Cause: An array variable is referenced in the program which was not defined properly in a DIM or COM statement (i.e., an array variable was not defined in a DIM or COM statement or has been referenced both as a 1-dimensional and as a 2-dimensional array).

Action: Correct statement text.

Example:

```
:10 A(2,2) = 123
:RUN
 10 A(2,2) = 123
      ↑ERR 22
:1 DIM A(4,4)                    (Possible Correction)
```

CODE 23

Error: **No Program Statements**

Cause: A RUN command was entered but there are no program statements.

Action: Enter program statements.

Example:

```
:RUN
 ↑ERR 23
```

CODE 24

Error: **Illegal Immediate Mode Statement**

Cause: An illegal verb or transfer in an immediate execution statement was encountered.

Action: Re-enter a corrected immediate execution statement.

Example:

```
 IF A = 1 THEN 100
 ↑ERR 24
```

CODE 25

Error: **Illegal GOSUB/RETURN Usage**

Cause: There is no companion GOSUB statement for a RETURN statement, 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 loop.

Action: Correct the program.

Example:

```
:10 PRINT "I=";I
:20 NEXT I
:30 END
:RUN
I = 0
 20 NEXT I
           ↑ERR 26
:5 FOR I=1 TO 10                 (Possible Correction)
```

CODE 27

Error: **Insufficient Data**

Cause: There is insufficient data for READ statement requirements.

Action: Correct program to supply additional data.

Example:

```
:10 DATA 2
:20 READ X,Y
:30 END
:RUN

 20 READ X,Y
              ↑ERR 27
:11 DATA 3                       (Possible Correction)
```

CODE 28

Error: Data Reference Beyond Limits

Cause: The data reference in a RESTORE statement is beyond the existing data limits.

Action: Correct the RESTORE statement.

Example:

```
:10 DATA 1,2,3
:20 READ X,Y,Z
:30 RESTORE 5
     ....
     ....
     ....
:90 END
:RUN
 30 RESTORE 5
               ↑ERR 28
:30 RESTORE 2                    (Possible Correction)
```

CODE 29

Error: Illegal Data Format

Cause: The data format for an INPUT statement is illegal (format error).

Action: Reenter data in the correct format starting with erroneous number or terminate run with the RESET key and run again.

Example:

```
:10 INPUT X,Y
     ....
     ....
     ....
:90 END
:RUN
:INPUT
?1A,2E-30
  ↑ERR 29
?12,2E-30                        (Possible Correction)
```

CODE 30

Error: Illegal Common Assignment

Cause: A COM statement variable definition was preceded by a non-common variable definition.

Action: Correct program, making all COM statements the first numbered lines.

Example:

```
:10 A=1 :B=2
:20 COM A,B
:99 END
:RUN

 20 COM A,B
          ↑ERR 30
:10[CR/LF–EXECUTE]               (Possible Correction)
:30 A=1 :B=2
```

CODE 31

Error: **Illegal Line Number**

Cause: The 'statement number' key was pressed producing a line number greater than 9999; or in renumbering a program with the RENUMBER command a line number was generated which was greater than 9999.

Action: Correct the program.

Example:
```
:9995 PRINT X,Y
:[line number key]
 ↑ERR 31
```

CODE 33

Error: **Missing HEX Digit**

Cause: A digit or a letter from A - F was expected.

Action: Correct the program text.

Example:
```
:10 SELECT PRINT 00P
                  ↑ERR 33
:10 SELECT PRINT 005          (Possible Correction)
```

CODE 34

Error: **Tape Read Error**

Cause: The system was unable to read the next record on the tape; the tape is positioned after the bad record.

CODE 35

Error: **Missing Comma or Semicolon**

Cause: A comma or semicolon was expected.

Action: Correct statement text.

Example:
```
:10 DATASAVE #2   X,Y,Z
                 ↑ ERR 35
:10 DATASAVE #2,X,Y,Z         (Possible Correction)
```

CODE 36

Error: **Illegal Image Statement**

Cause: No format (e.g. #.##) in image statement.

Action: Correct the statement text.

Example:
```
:10 PRINTUSING 20, 1.23
:20% AMOUNT =
:RUN
:10 PRINTUSING 20,1.23
                 ↑ERR 36
:20% AMOUNT = #####           (Possible Correction)
```

CODE 37

Error: **Statement Not Image Statement**

Cause: The statement referenced by the PRINTUSING statement is not an image statement.

Action: Correct the statement text.

Example:
```
:10 PRINTUSING 20,X
:20 PRINT X
:RUN
:10 PRINTUSING 20,X
                  ↑ERR37
:20% AMOUNT = $#,###.##        (Possible Correction)
```

CODE 38

Error: **Illegal Floating Point Format**

Cause: Fewer than 4 up arrows were specified in the floating point format in an image statement.

Action: Correct the statement text.

Example:
```
:10 % ##.##↑↑↑
              ↑ ERR 38
:10 % ##.##↑↑↑↑
```

CODE 39

Error: **Missing Literal String**

Cause: A literal string was expected.

Action: Correct the text.

Example:
```
:10 READ A$
:20 DATA 123
:RUN
20 DATA 123
        ↑ERR 39
20 DATA "123"                   (Possible Correction)
```

CODE 40

Error: **Missing Alphanumeric Variable**

Cause: An alphanumeric variable was expected.

Action: Correct the statement text.

Example:
```
:10 A$, X = "JOHN"
        ↑ERR 40
:10 A$, X$ = "JOHN"
```

CODE 41

Error: **Illegal STR(Arguments**

Cause: The STR(function arguments exceed the maximum length of the string variable.

Example:
```
:10 B$ = STR(A$, 10, 8)
                    ↑ERR 41
:10 B$ = STR(A$, 10, 6)        (Possible Correction)
```

CODE 42

Error:	**File Name Too Long**
Cause:	The program name specified is too long (a maximum of 8 characters is allowed).
Action:	Correct the program text.
Example:	:SAVE "PROGRAM#1"
	↑ERR 42
	:SAVE "PROGRAM1" (Possible Correction)

CODE 43

Error:	**Wrong Variable Type**
Cause:	During a DATALOAD operation a numeric (or alphanumeric) value was expected but an alphanumeric (or numeric) value was read.
Action:	Correct the program or make sure proper tape is mounted.
Example:	:DATALOAD X, Y
	↑ERR 43
	:DATALOAD X$, Y$ (Possible Correction)

CODE 44

Error:	**Program Protected**
Cause:	A program loaded was protected and, hence, cannot be SAVED or LISTED.
Action:	Execute a CLEAR command to remove protect mode, (but, program will be scratched).

CODE 45

Error:	**Statement Line Too Long**
Cause:	A statement line may not exceed 192 keystrokes.
Action:	Shorten the statement line being entered.

CODE 46

Error:	**New Starting Statement Number Too Low**
Cause:	The new starting statement number in a RENUMBER command is not greater than the next lowest statement number.
Action:	Reenter the RENUMBER command correctly.
Example:	50 REM – PROGRAM 1
	62 PRINT X, Y
	73 GOSUB 500
	:
	:RENUMBER 62, 20, 5
	↑ERR 46
	:RENUMBER 62, 60, 5 (Possible Correction)

CODE 47

Error:	**Illegal Or Undefined Device Specification**
Cause:	The #n device specifications in a program statement is undefined.
Action:	Define the specified device numbers.
Example:	:SAVE #2
	↑ERR 47
	:SELECT #2 10A
	:SAVE #2 (Possible Correction)

CODE 48

Error: **Undefined Keyboard Function**

Cause: There is no mark (DEFFN') in a user's program corresponding to the keyboard function key depressed.

Action: Correct the program.

Example:
```
:[keyboard function key #2]
↑ERR 48
```

CODE 49

Error: **End of Tape**

Cause: The end of tape was encountered during a tape operation.

Action: Correct the program or make sure the tape is correctly positioned.

Example:
```
100 DATALOAD X, Y, Z
                   ↑ERR 49
```

CODE 50

Error: **Protected Tape**

Cause: A tape operation is attempting to write on a tape cassette that has been protected (by tab on bottom of cassette tape).

Action: Mount another cassette or "unprotect" the tape cassette by covering the punched hole on the bottom of the cassette with the tab.

Example:
```
SAVE /103
        ↑ERR 50
```

CODE 51

Error: **Illegal Statement**

Cause: The System 2200 does not have the capability to process this BASIC statement.

Action: Do not use this statement.

CODE 52

Error: **Expected Data (Nonheader) Record**

Cause: A DATALOAD operation was attempted but the device was not positioned at a data record.

Action: Make sure the correct device is positioned correctly.

CODE 53

Error: **Illegal Use of HEX Function**

Cause: The HEX(function is being used in an illegal situation. The HEX function may not be used in a PRINTUSING statement.

Action: Do not use HEX function in this situation.

Example:
```
:10 PRINTUSING 200, HEX(F4F5)
                          ↑ ERR 53
:10 A$ = HEX(F4F5)
:20 PRINTUSING 200,A$            (Possible Correction)
```

CODE 54

Error: **Illegal Plot Argument**

Cause: An argument in the PLOT statement is illegal.

Action: Correct the PLOT statement.

Example:

```
100  PLOT < 5, , H >
                   ↑ ERR 54
100  PLOT < 5, , C >               (Possible Correction)
```

CODE 55

Error: **Illegal BT Argument**

Cause: An argument in a DATALOAD BT or DATASAVE BT statement is illegal.

Action: Correct the statement in error.

Example:

```
100  DATALOAD BT (M=50) A$
                  ↑ERR 55
100  DATALOAD BT (N=50) A$     (Possible Correction)
```

CODE 56

Error: **Number Exceeds Image Format**

Cause: The value of the number being packed or converted is greater than the number integer digits provided for in the pack or convert image.

Action: Change the image specification.

Example:

```
100  PACK (##) A$ FROM 1234
                         ↑ ERR 56
100  PACK (####) A$ FROM 1234   (Possible Correction)
```

CODE 57

Error: **Illegal Disk Sector Address**

Cause: Illegal disk sector address specified; value is negative or greater than 32767. (The System 2200 cannot store a sector address greater than 32767.)

Action: Correct the program statement in error.

Example:

```
100  DATASAVE DAF (42000 ,X) A,B,C.
                         ↑ ERR 57
100  DATASAVE DAF (4200 ,X) A,B,C    (Possible Correction)
```

CODE 58

Error: **Expected Data Record**

Cause: A program record or header record was read when a data record was expected.

Action: Correct the program.

Example:
```
100 DATALOAD DAF(0,X) A,B,C
                       ↑ERR 58
```

CODE 59

Error: **Illegal Alpha Variable For Sector Address**

Cause: Alphanumeric receiver for the next available address in the disk DA instruction is not at least 2 bytes long.

Action: Dimension the alpha variable to be at least two characters long.

Example:
```
10   DIM A$1
100  DATASAVE DAR( ) ,A$ ) X,Y,Z
                           ↑ERR 59
10   DIM A$2                         (Possible Correction)
```

CODE 60

Error: **Array Too Small**

Cause: The alphanumeric array does not contain enough space to store the block of information being read from disk or tape or being packed into it. For cassette tape and disk records, the array must contain at least 256 bytes (100 bytes for 100 byte cassette blocks).

Action: Increase the size of the array.

Example:
```
10   DIM A$(15)
20   DATALOAD BT A$( )
                    ↑ERR 60
10   DIM A$(16)               (Possible Correction)
```

CODE 61

Error: **Disk Hardware Error**

Cause: The disk did not recognize or properly respond back to the System 2200 during read or write operation in the proper amount of time.

Action: Run program again. If error persists, re-initialize the disk; contact Wang service 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$( )** **↑ERR 62**

CODE 63

Error:	**Missing Alpha Array Designator**
Cause:	An alpha array designator (e.g., A$( )) was expected. (Block operations for cassette and disk require an alpha array argument.)
Action:	Correct the statement in error.
Example:	**100 DATALOAD BT A$** **↑ERR 63** **100 DATALOAD BT A$( )** (Possible Correction)

CODE 64

Error:	**Sector Not On Disk**
Cause:	The disk sector being addressed is not on the disk. (Maximum legal sector address depends upon the model of disk used.)
Action:	Correct the program statement in error.
Example:	**100 MOVEEND F = 10000** **↑ERR 64** **100 MOVEEND F = 9791** (Possible Correction)

CODE 65

Error:	**Disk Hardware Malfunction**
Cause:	A disk hardware error occurred (i.e., the disk is not in file ready position. This could occur, for example, if the disk is in LOAD mode or power is not turned on).
Action:	Insure disk is turned on and properly setup for operation. Set the disk into LOAD mode and then back into RUN mode, with the RUN/LOAD selection switch. The check light should then go out. If error persists call your Wang Service personnel. (Note, the disk should never be left in LOAD mode when running.)
Example:	**100 DATALOAD DCF A$,B$** **↑ERR 65**

CODE 66

Error: **Format Key Engaged**

Cause: The disk format key is engaged. (The key is normally engaged only when formatting a disk pack.)

Action: Turn off the format key.

Example:
```
100 DATASAVE DCF X,Y,Z
                  ↑ERR 66
```

CODE 67

Error: **Disk Format Error**

Cause: A disk format error was detected on disk read or write. The disk is not properly formatted such that sector addresses can be read.

Action: Format the disk again.

Example:
```
100 DATALOAD DCF X,Y,Z
                  ↑ERR 67
```

CODE 68

Error: **LRC Error**

Cause: A disk longitudinal redundancy check error occurred when reading a sector. The data may have been written incorrectly, or the System 2200/Disk Controller could be malfunctioning.

Action: Run program again. If error persists, re-write the bad sector. If error still persists, call Wang Service 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 pack. 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 subsequently the sector was incorrectly written on disk (i.e., the disk pack was never initially formatted).

Action: Format the disk if it was not done. If the disk was formatted, re-write the bad sector, or reformat the disk. If error persists call Wang Service personnel.

Example:

```
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"
           ↑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
                        ↑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
     ↑ERR 75
```

CODE 76

Error: **Missing < or > (Plot Enclosures)**

Cause: The required PLOT enclosures are not in the PLOT statement.

Action: Correct the statement in error.

Example:

```
100  PLOT A, B, "*"
              ↑ERR 76
100  PLOT <A, B, "*">          (Possible Correction)
```

CODE 77

Error: **Starting Sector Greater Than Ending Sector**

Cause: The starting sector address specified is greater than the ending sector address specified.

Action: Correct the statement in error.

Example:

```
10   COPY FR(1000, 100)
                   ↑ERR 77
10   COPY FR(100, 1000)          (Possible Correction)
```

CODE 78

Error: **File Not Scratched**

Cause: A file is being renamed that has not been scratched.

Action: Scratch the file before renaming it.

Example:

```
SAVE DCF (LINREG") "LINREG2"
                ↑ERR 78
SCRATCH F "LINREG"                (Possible Correction)
SAVE DCF ("LINREG") "LINREG2"
```

CODE 79

Error: **File Already Catalogued**

Cause: An attempt was made to catalogue a file with a name that already exists in the catalogue index.

Action: Use a different name.

Example:

```
SAVE DCF "MATLIB"
            ↑ERR 79
SAVE DCF "MATLIB1"          (Possible Correction)
```

CODE 80

Error: **File Not In Catalog**

Cause: The error may occur if one attempts to address a non-existing file name or to load a data file as a program or open a program file as a data file.

Action: Make sure you're using the correct file name; make sure the proper disk pack is mounted.

Example:
```
LOAD DCR "PRES"
          ↑ERR 80
LOAD DCF "PRES"           (Possible Correction)
```

CODE 81

Error: **/XXX Device Specification Illegal**

Cause: The /XXX device specification may not be used in this statement.

Action: Correct the statement in error.

Example:
```
100 DATASAVE DC /310, X
                 ↑ERR 81
100 DATASAVE DC #1, X           (Possible Correction)
```

CODE 82

Error: **No End Of File**

Cause: No end of file record was recorded on file and therefore could not be found in a SKIP END operation.

Action: Correct the file.

Example:
```
100D SKIP END
         ↑ERR 82
```

CODE 83

Error: **Disk Hardware Failure**

Cause: A disk address cannot be properly transferred from the System 2200 to the disk when processing MOVE or COPY.

Action: Run program again. If error persists, call Wang 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)
              ↑ERR 84
```

CODE 85

Error: **Read After Write Error**

Cause: The comparison of read after write to a disk sector failed. The information was not written properly.

Action: Write the information again. If error persists, call Wang Field Service personnel.

Example:
```
100  DATASAVE DCF$ X, Y, Z
                  ↑ERR 85
```

CODE 86

Error: **File Not Open**

Cause: The file was not opened.

Action: Open the file before reading from it.

Example:
```
100  DATALOAD DC A$
                 ↑ERR 86
10   DATALOAD DC OPEN F "DATFIL"          (Possible Correction)
```

CODE 87

Error: **Common Variable Required**

Cause: The variable in the LOAD DA statement, used to receive the sector address of the next available sector after the load, is not a common variable.

Action: Define the variable to be common.

Example:
```
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"
             ↑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 equal sign in a MAT multiplication or transposition statement.

Action: Correct the statement.

Example:

```
:10  MAT A=A*B
               ↑ERR 91
:10  MAT C=A*B                         (Possible Correction)
```

CODE 92

Error: **Illegal Redimensioning Of Array**

Cause: The space required to redimension the array is greater than the space initially reserved for the array.

Action: Reserve more space for array in DIM or 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**

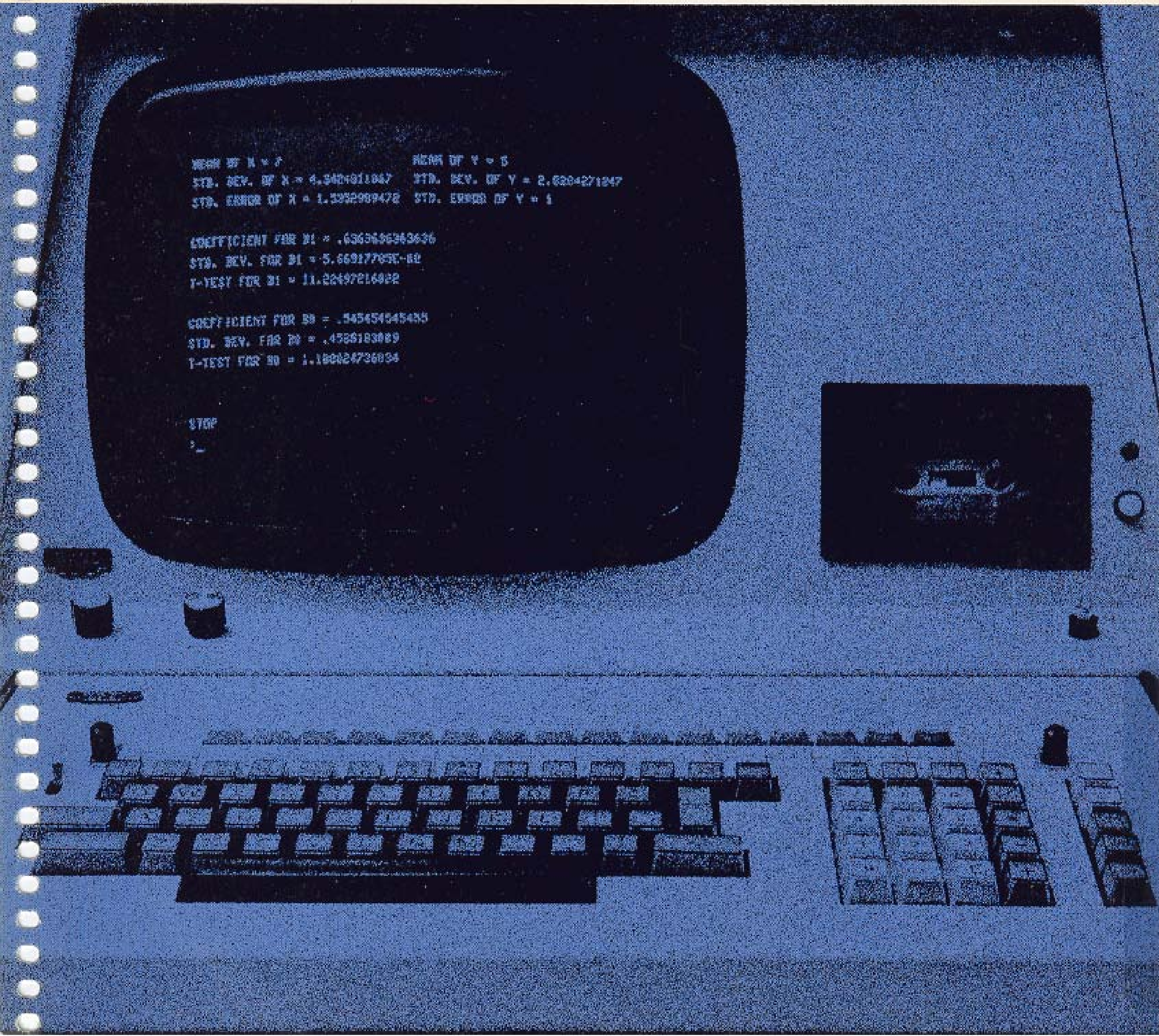
Cause: An asterisk (*) was expected.

Action: Correct statement text.

Example:

```
:10  MAT C=(3)B
              ↑ERR 94
:10  MAT C=(3)*B                         (Possible Correction)
```

SECTION XV
APPENDICES



Section XV

Appendices

APPENDIX A

SPECIFICATIONS

CRT (Cathode Ray Tube) – Model 2216

Unit Size
Height . . . . 14 in. (35.6 cm)
Depth . . . . 16 in. (40.6 cm)
Width . . . . 21½ in. (54.6 cm)

Display Size
Height . . . . 8 in. (20.3 cm)
Width . . . . 10½ in. (26.7 cm)

Capacity
16 lines, 64 characters/line

Character Size
Height . . . . 0.20 in. (0.51 cm)
Width . . . . 0.12 in. (0.30 cm)

Weight
36 lbs (16.3 kg)

System 2200 Power Requirements
115 VAC or 230 VAC ± 10%
50 or 60 Hz ± ½ cycle

System 2200 Operating Environment
50°F to 90°F (10°C to 32°C)
40% to 60% relative humidity

TAPE DRIVE –**Model 2217**

Stop/Start Time
0.09/0.05 sec

Capacity
522 bytes/ft (1712 bytes/m)

Recording Speed
7.5 IPS (19.05 cm/sec)

Search Speed
7.5 IPS (19.05 cm/sec)

Transfer Rate
326 characters/sec (approx.)

Inter-record Gap
0.6 in. (1.52 cm)

(Capacity and transfer rate include gaps and redundant recording.)

CPU (Central Processing Unit) – System 2200, Model A or B.

Built-in Functions

Mathematical & Trigonometric Functions*

EXP	e to the power of x
LOG	Natural Log
SQR	Square Root
π	Pi
SIN	Sine
COS	Cosine
TAN	Tangent
ARCSIN	Inverse Sine
ARCCOS	Inverse Cosine
ARCTAN	Inverse Tangent
RND	Random Number Generator

Logical & Data Manipulation Functions

ABS	Absolute Value of a Number
INT	Integer Value of a Number
	1, 0, or +1 if a number is negative, 0, or positive.
STR	Selection of one or more characters in an alphanumeric string.
HEX	Hexadecimal Values
LEN	Length of Alphanumeric Variable

CPU (Central Processing Unit) – System 2200, Model A **or** B **(Continued)**

Variable Formats

Scalar Numeric Variable.
Numeric 1- and 2-dimension Array Variables.
Alphanumeric String Variable.
Alphanumeric 1- and 2-dimensional String Arrays.

Average Execution Times (Milliseconds)

Add/Subtract	0.8
Multiply/Divide	3.87/7.4
Square Root/e^x	46.4/25.3
$\text{Log}_e x/X^y$	23.2/45.4
Integer/Absolute Value	0.24/0.02
Sign/Sine	0.25/38.3
Cosine/Tangent	38.9/78.5
Arctangent	72.5
Read/Write Cycle	1.6μ sec

(Average execution times were determined using random number arguments with 13 digits of precision. Average execution times will be faster in most calculations with arguments having fewer significant digits.)

APPENDIX A

SPECIFICATIONS (Cont.)

Capacity

Memory Size
4,096 bytes (expandable to 32K)

Peripheral Capacity
6 (expandable to 11 max)

Dynamic Range
10^{-99} to 10^{+99}

Subroutine Stacking
No Limit

***CPU Size**

Height	9¾ in. (24.8 cm)
Depth	16 in. (40.6 cm)
Width	17 in. (43.2 cm)

Weight
24 lbs (10.9 kg)

Power Supply Size

Height	7¾ in. (19.7 cm)
Depth	8¾ in. (22.2 cm)
Width	19 in. (48.3 cm)

Weight
34 lbs (15.4 kg)

KEYBOARD

Model 2215

Height	3 in. (7.62 cm)
Depth	10 in. (25.4 cm)
Width	17½ in. (44.5 cm)

Weight
7 lbs (3.2 kg)

Model 2222

Height	3 in. (7.62 cm)
Depth	10 in. (25.4 cm)
Width	19½ in. (49.5 cm)

Weight
7½ lbs (3.4 kg)

**Trigonometric arguments in radians, degrees or gradians.*

Wang Laboratories reserves the right to change specifications without prior notice.

APPENDIX B

AVAILABLE PERIPHERALS

2201	Output Writer
2202*	Plotting Output Writer
2203*	Punched Paper Tape Reader
2212*	Analog Flatbed Plotter (10" x 15")
2214	Mark Sense Card Reader
2215	BASIC Keyword Keyboard
2216	CRT Executive Display
2217	Single Tape Cassette Drive
2216/2217	Combined CRT Executive Display/Single Tape Cassette Drive
2218	Dual Tape Cassette Drive
2219	I/O Extended Chassis
2221	Line Printer (132 Column)
2222	Alpha-Numeric Typewriter Keyboard
2230-1*	Fixed/Removable Disk Drive (1,228,800 bytes)
2230-2*	Fixed/Removable Disk Drive (2,457,600 bytes)
2230-3*	Fixed/Removable Disk Drive (5,013,504 bytes)
2231	Line Printer (80 Column)
2234*	Hopper-Feed Punched Card Reader
2232*	Digital Flatbed Plotter (31" x 42")
2240-1*	Dual Removable Flexible Disk Drive (262,144 bytes)
2240-2*	Dual Removable Flexible Disk Drive (524,288 bytes)
2241	Thermal Printer (80 Column)
2207*	I/O Interface Controller (RS-232-C)
2227	Standard Telecommunications Controller
2290	CPU/Peripheral Stand
2250	I/O Interface Controller (8 Bit Parallel)
2252*	Input Interface Controller (BCD 10-Digit-Parallel)
OPTION 1	Matrix ROM
OPTION 2	General I/O ROM
OPTION 3	Character Edit ROM

*Peripheral used with the System 2200B only. A System 2200A can be upgraded to a System 2200B upon request at a nominal charge.

APPENDIX C

WANG SYSTEM 2200 ASCII CHARACTER CODE 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		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	

APPENDIX D

LISTING OF ERROR MESSAGES

CODE 01	TEXT OVERFLOW
CODE 02	TABLE OVERFLOW
CODE 03	MATH ERROR
CODE 04	MISSING LEFT PARENTHESIS
CODE 05	MISSING RIGHT PARENTHESIS
CODE 06	MISSING EQUALS SIGN
CODE 07	MISSING QUOTATION MARKS
CODE 08	UNDEFINED FN FUNCTION
CODE 09	ILLEGAL FN USAGE
CODE 10	INCOMPLETE STATEMENT
CODE 11	MISSING LINE NUMBER OR CONTINUE ILLEGAL
CODE 12	MISSING STATEMENT TEXT
CODE 13	MISSING OR ILLEGAL INTEGER
CODE 14	MISSING RELATION OPERATOR
CODE 15	MISSING EXPRESSION
CODE 16	MISSING SCALAR
CODE 17	MISSING ARRAY
CODE 18	ILLEGAL VALUE
CODE 19	MISSING NUMBER
CODE 20	ILLEGAL NUMBER FORMAT
CODE 21	MISSING LETTER OR DIGIT
CODE 22	UNDEFINED ARRAY VARIABLE
CODE 23	NO PROGRAM STATEMENTS
CODE 24	ILLEGAL IMMEDIATE MODE STATEMENT
CODE 25	ILLEGAL GOSUB/RETURN USAGE
CODE 26	ILLEGAL FOR/NEXT USAGE
CODE 27	INSUFFICIENT DATA
CODE 28	DATA REFERENCE BEYOND LIMITS
CODE 29	ILLEGAL DATA FORMAT
CODE 30	ILLEGAL COMMON ASSIGNMENT
CODE 31	ILLEGAL LINE NUMBER
CODE 33	MISSING HEX DIGIT
CODE 34	TAPE READ ERROR
CODE 35	MISSING COMMA OR SEMICOLON
CODE 36	ILLEGAL IMAGE STATEMENT
CODE 37	STATEMENT NOT IMAGE STATEMENT
CODE 38	ILLEGAL FLOATING POINT FORMAT
CODE 39	MISSING LITERAL STRING
CODE 40	MISSING ALPHANUMERIC VARIABLE
CODE 41	ILLEGAL STR(ARGUMENTS
CODE 42	FILE NAME TOO LONG
CODE 43	WRONG VARIABLE TYPE
CODE 44	PROGRAM PROTECTED
CODE 45	STATEMENT LINE TOO LONG
CODE 46	NEW STARTING STATEMENT NUMBER TOO LOW
CODE 47	ILLEGAL OR UNDEFINED DEVICE SPECIFICATION
CODE 48	UNDEFINED KEYBOARD FUNCTION
CODE 49	END OF TAPE
CODE 50	PROTECTED TAPE
CODE 51	ILLEGAL STATEMENT
CODE 52	EXPECTED DATA (NONHEADER) RECORD
CODE 53	ILLEGAL USE OF HEX FUNCTION
CODE 54	ILLEGAL PLOT ARGUMENT
CODE 55	ILLEGAL BT ARGUMENT
CODE 56	NUMBER EXCEEDS IMAGE FORMAT
CODE 57	ILLEGAL SECTOR ADDRESS
CODE 58	EXPECTED DATA RECORD
CODE 59	ILLEGAL ALPHA VARIABLE FOR SECTOR ADDRESS
CODE 60	ARRAY TOO SMALL
CODE 61	DISK HARDWARE ERROR
CODE 62	FILE FULL
CODE 63	MISSING ALPHA ARRAY DESIGNATOR
CODE 64	SECTOR NOT ON DISK
CODE 65	DISK HARDWARE MALFUNCTION
CODE 66	FORMAT KEY ENGAGED
CODE 67	DISK FORMAT ERROR
CODE 68	LRC ERROR
CODE 71	CANNOT FIND SECTOR
CODE 72	CYCLIC READ ERROR
CODE 73	ILLEGAL ALTERING OF A FILE
CODE 74	CATALOG END ERROR
CODE 75	COMMAND ONLY (NOT PROGRAMMABLE)
CODE 76	MISSING < OR > (PLOT ENCLOSURES)
CODE 77	STARTING SECTOR > ENDING SECTOR
CODE 78	FILE NOT SCRATCHED
CODE 79	FILE ALREADY CATALOGED
CODE 80	FILE NOT IN CATALOG
CODE 81	/XXX DEVICE SPECIFICATION ILLEGAL
CODE 82	NO END OF FILE
CODE 83	DISK HARDWARE FAILURE
CODE 84	NOT ENOUGH MEMORY FOR MOVE OR COPY
CODE 85	READ AFTER WRITE ERROR
CODE 86	FILE NOT OPEN
CODE 87	COMMON VARIABLE REQUIRED
CODE 88	LIBRARY INDEX FULL
CODE 89	MATRIX NOT SQUARE
CODE 90	MATRIX OPERANDS NOT COMPATIBLE
CODE 91	ILLEGAL MATRIX OPERAND
CODE 92	ILLEGAL REDIMENSIONING OF ARRAY
CODE 93	SINGULAR MATRIX
CODE 94	MISSING ASTERISK

To help us to provide you with the best manuals possible, please make your comments and suggestions concerning this publication on the form below. Then detach, fold, tape closed and mail to us. All comments and suggestions become the property of Wang Laboratories, Inc. For a reply, be sure to include your name and address. Your cooperation is appreciated.

TITLE OF MANUAL:

COMMENTS:

Fold

Fold

(Please tape. Postal regulations prohibit the use of staples.)

WANG

Fold

FIRST CLASS
PERMIT NO. 16
Tewksbury, Mass.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

— POSTAGE WILL BE PAID BY —

WANG LABORATORIES, INC.
836 NORTH STREET
TEWKSBURY, MASSACHUSETTS 01876

Attention: Marketing Department

Fold

Cut along dotted line.

Printed in U.S.A.

ALPHABETICAL INDEX

WANG LABORATORIES (CANADA) LTD.
180 Duncan Mill Road
Don Mills, Ontario M3B 1Z6
TELEPHONE (416) 449-7890
TELEX 06-966546

WANG EUROPE, S.A.
Buurtweg 13
9412 Ottergem
Belgium
TELEPHONE: 053/74514
TELEX: 26077

WANG ELECTRONICS LTD.
1, Olympic Way, 4th Floor
Wembley Park,
Middlesex, England
TELEPHONE: 01/903/6755
TELEX: 923498

WANG FRANCE SARL
47, Rue de la Chapelle
Paris 18, France
TELEPHONE 203.27.94 or 203.25.94

WANG LABORATORIES GMBH
Moselstrasse No. 4
6000 Frankfurt am Main
West Germany
TELEPHONE (0611) 252061-64

WANG SKANDINAVISKA AB
Fredsgaten 17
S-172-23
Sundbyberg 1, Sweden
TELEPHONE 08-98-12-45

WANG NEDERLAND B.V.
Damstraat 2
Utrecht, Netherlands
TELEPHONE 030-930947

WANG PACIFIC LTD.
61, King Yip Street, 1st Floor
Kwun Tong Kowloon, Hong Kong
TELEPHONE 3-434231/2

WANG INDUSTRIAL CO., LTD.
110-118 Kuang-Fu N. Rd.
Taipei, Taiwan
Republic of China
TELEPHONE 784181-3

WANG GESELLSCHAFT MBH
Grinzinger Allee 16
1190 Vienna 19
Austria
TELEPHONE (0222) 32.42.43

WANG COMPUTER PTY. LTD.
25 Bridge Street
Pymble, NSW 2073
Australia
TELEPHONE 449-6388

WANG INTERNATIONAL TRADE, INC.
836 North Street
Tewksbury, Massachusetts 01876
TELEPHONE (617) 851-4111
TWX 710-343-6769
TELEX 94-7421

WANG COMPUTER SERVICES
836 North Street
Tewksbury, Massachusetts 01876
TELEPHONE (617) 851-4111
TWX 710-343-6769
TELEX 94-7421

24 Mill Street
Arlington, Massachusetts 02174
TELEPHONE (617) 648-8550

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

Printed in U.S.A.
700-3038C
1-74-3M
Price $10.00