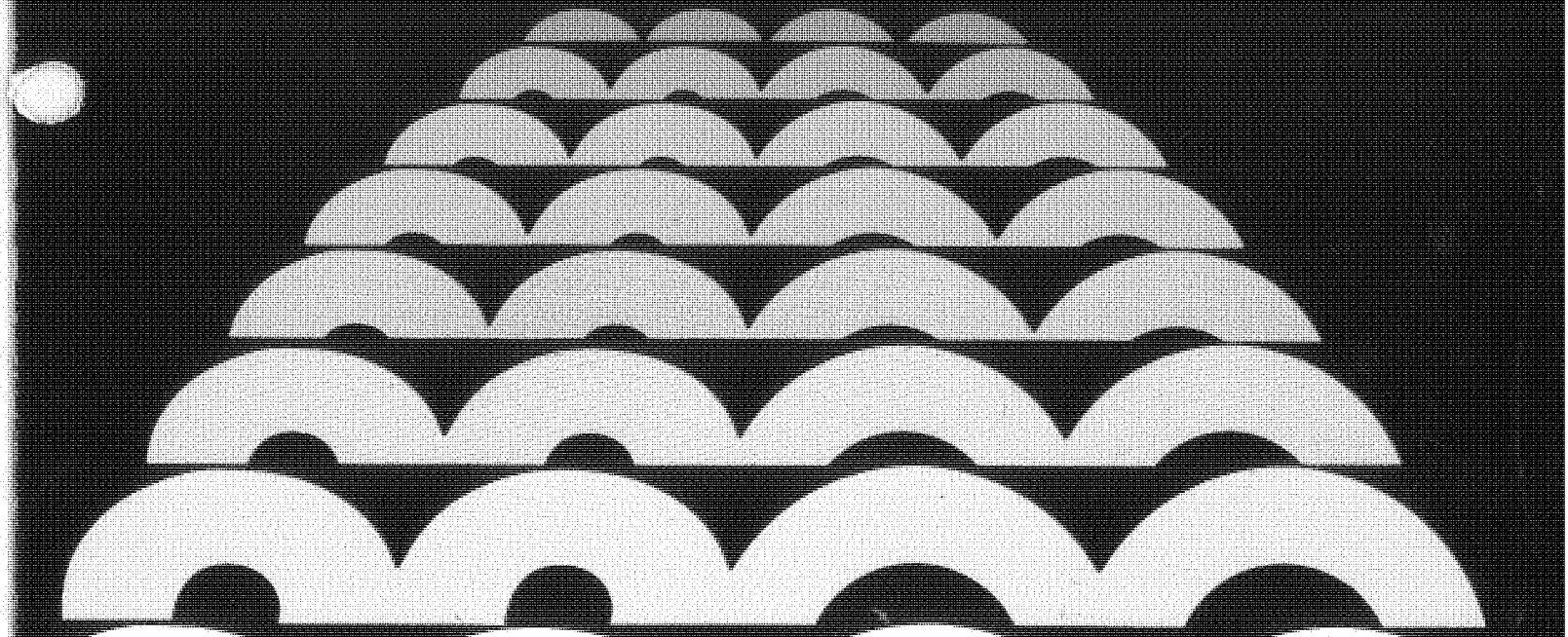
WANG

BASIC-2 Language Reference Manual



2200

BASIC-2
Language Reference Manual

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

Disclaimer of Warranties and Limitation of Liabilities

The staff of Wang Laboratories, Inc., has taken due care in preparing this manual; however, nothing contained herein modifies or alters in any way the standard terms and conditions of the Wang purchase agreement, lease agreement, or rental agreement by which this equipment was acquired, nor increases in any way Wang's liability to the customer. In no event shall Wang Laboratories, Inc., or its subsidiaries be liable for incidental or consequential damages in connection with or arising from the use of this manual or any programs contained herein.

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

HOW TO USE THIS MANUAL

This manual is intended for programmers using Wang Systems which support the BASIC-2 Language (2200VP, 2200MVP). It contains information on internal machine organization, all statements (except disk statements), and error messages. It should be used in conjunction with the introductory manual for your system, as well as the *Basic-2 Disk Reference manual.* this book is not intended as a tutorial; that purpose is served by the manual *Programming in BASIC.*

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION TO WANG BASIC-2

CHAPTER 2 MEMORY ORGANIZATION

CHAPTER 3 EDITING AND DEBUGGING FEATURES

CHAPTER 4 NUMERIC OPERATIONS

CHAPTER 5 ALPHANUMERIC STRING MANIPULATION INSTRUCTIONS

CHAPTER 6 BINARY AND PACKED DECIMAL ARITHMETIC OPERATORS

CHAPTER 7 THE SELECT STATEMENT

CHAPTER 7 THE SELECT STATEMENT (Continued)

CHAPTER 8 PROGRAMMABLE INTERRUPT FEATURE

CHAPTER 9 ERROR CONTROL FEATURES

CHAPTER 10 SYSTEM COMMANDS

CHAPTER 11 GENERAL-PURPOSE BASIC-2 STATEMENTS

CHAPTER 11 GENERAL-PURPOSE BASIC-2 STATEMENTS (Continued)

CHAPTER 12 DATA CONVERSION STATEMENTS

CHAPTER 13 MATH MATRIX STATEMENTS

CHAPTER 14 SORT STATEMENTS

CHAPTER 14 SORT STATEMENTS (Continued)

CHAPTER 15 GENERAL I/O STATEMENTS

CHAPTER 16 2200MVP OPERATING SYSTEM AND SPECIAL LANGUAGE FEATURES

CHAPTER 16 2200MVP OPERATING SYSTEM AND SPECIAL LANGUAGE FEATURES (Continued)

APPENDICIES

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION TO WANG BASIC-2

1.1 THE BASIC-2 LANGUAGE

Wang BASIC-2 is a general-purpose high-level programming language developed by Wang Laboratories for use on the 2200VP and 2200MVP series of central processing units. BASIC-2 is a modified version of the original Dartmouth BASIC language which offers all of the important features of the original BASIC as well as numerous new features and enhancements implemented by Wang Laboratories. The result is a programming language which is powerful, extremely versatile, and well suited for both technical and commercial applications. BASIC-2, like the original BASIC, also is designed to be easily learned by beginning programmers.

1.2 COMPATIBILITY WITH EARLIER VERSIONS OF WANG BASIC

BASIC-2 was developed with the aid of experience gained in the implementation of earlier versions of BASIC on 2200 series central processors. Particular attention was paid to improving the clarity of the syntax in many instructions and to implementing new features which help to better support the tasks of writing, documenting, and debugging programs in BASIC-2.

Although the vast majority of instructions carried over from earlier versions of Wang BASIC have undergone some syntax revisions in BASIC-2, BASIC-2 preserves almost complete downward compatibility with earlier versions of the language. This compatibility is achieved by supporting both the BASIC-2 syntax and the previous BASIC syntax for each instruction. In general, therefore, programs written in earlier versions of Wang BASIC can be loaded and run without modification in a system equipped with a BASIC-2 interpreter. There are very few exceptions to this compatibility. The exceptions, as well as the earlier versions of 2200 BASIC syntax now supported in BASIC-2, are detailed in Appendix C.

1.3 THE SYSTEM 2200 CPU's

The BASIC-2 language is supported on two models of the Wang 2200 series central processing units (CPU's) — the 2200VP and the 2200MVP. The 2200VP is a single-user system which offers high performance and a wide range of available peripherals and options. Its general characteristics are described in the *2200VP Introductory Manual*. The 2200MVP is a multiprogramming system which supports multiple users, each of whom is allocated one or more sections of memory, called "partitions," in which to store programs and data. Its general characteristics are described in the *2200MVP Introductory Manual* and in Chapter 16 of this manual.

In general, the versions of the BASIC-2 language supported on the 2200VP and 2200MVP are identical. However, several special language features have been added to BASIC-2 on the MVP to better support the multiprogramming environment. These include statements which enable individual users to gain exclusive control over commonly used I/O devices (such as printers) for a period of time, to process jobs in both "foreground" and "background" modes, and to define and access "global" partitions. The additional MVP instructions are described in Chapter 16, section 16.10. Both the 2200VP and the 2200MVP support earlier versions of Wang BASIC, subject to the incompatibilities documented in Appendix C.

Each user on the 2200MVP has one or more sections of memory (partitions) allocated for his or her exclusive use. In addition, certain partitions may be designated as "global," meaning that the program text and variables stored in them can be accessed by several users. Each partition functions in effect as an independent, single-user system. For each partition, therefore, the description of memory organization in Chapter 2 holds true. Chapter 16, section 16.3 provides a more general description of how memory is partitioned on the MVP and how individual partitions can communicate with one another.

The 2200MVP system can be loaded with the 2200VP Operating System (OS) and BASIC-2 interpreter, in which case, the MVP essentially becomes a 2200VP single-user system. This feature permits 2200VP peripherals not supported by the MVP OS to be used on MVP systems.

1.4 TYPES OF BASIC-2 INSTRUCTIONS

The BASIC-2 language consists of a large group of instructions of various types, including statements, commands, operators, and system-defined functions. Of these, the two most important classes of instructions are statements and commands. Statements are programmable instructions used to write programs in BASIC-2. Commands are used by the operator to control system operations directly from the keyboard and generally are not programmable. System-defined functions and operators are used to construct numeric or alphanumeric expressions within BASIC statements.

System Commands

System commands are instructions which provide the operator with control of major system functions directly from the keyboard. Commands enable the operator to perform functions such as initiating program execution, clearing system memory, listing the program in memory, and renumbering the program in memory, among others. Commands are entered and executed immediately by the operator; they are not stored in memory as part of a program. (A very few commands are programmable, but the majority of commands are illegal in Program Mode.) The system commands are listed below:

CLEAR
CONTINUE
HALT/STEP Key
LIST
LOAD
RESET Key
RENUMBER
RUN
SAVE
STMT NUMBER Key*
SPECIAL FUNCTION Keys

Statements

A statement is a programmable instruction which serves as the fundamental building block of programs written in BASIC-2. Every line in a BASIC-2 program consists of one or more statements, each of which directs the system to perform a specific operation or sequence of operations. In most cases, a statement includes one or more expressions which provide the information to be operated on by the statement. An expression may consist of numeric or alphanumeric data, variables containing such data, or a combination of functions or operators and data. For example, the statement

```
10 PRINT A+B-5
```

instructs the system to evaluate the expression "A+B-5" and then output the result to a CRT display or printer.

Executable and Nonexecutable Statements

While most statements direct the system to perform certain tasks when they are executed (and hence are referred to as "executable statements"), there is a second class of statements whose sole

* Not available on all terminals.

purpose is to provide the system with needed information or to aid in program documentation. These statements, called "nonexecutable statements", initiate no system action when encountered during program execution.

The following are nonexecutable statements:

```
COM
DATA
DEFFN
DIM
IMAGE (%)
REM
```

1.5 BASIC-2 PROGRAMS

A BASIC-2 program consists of one or more numbered program lines, each of which consists of one or more statements. Each program line must begin with a line-number. Numbered lines are not executed immediately upon entry, but are stored in memory for execution at a later time. A line which does not begin with a line-number, however, is executed immediately upon entry and is not stored in memory.

The legal range of program line-numbers is from 0 to 9999. Line-numbers with fewer than four digits do not need to be padded with leading zeros, although this is not illegal. For example, the following lines are equivalent:

```
0005 PRINT A+B-5
     or
5 PRINT A+B-5
```

Line-numbers may not be preceded by any character other than a space. Spaces which precede a line-number are deleted by the system when the program line is stored in memory.

Execution of a BASIC-2 program always proceeds in line-number sequence from the lowest numbered line through the highest numbered line unless the normal sequence of execution is altered by a program branch instruction. Although program lines are executed in line-number sequence, however, they do not need to be entered in line-number sequence. As each line is entered, the system automatically inserts it in proper line-number sequence. Lines may therefore be entered in any convenient order.

Multiple-Statement Lines

BASIC-2 permits the specification of more than one statement on a program line. Because each statement is a separate and independent instruction, individual statements on the same line must be separated by colons. For example, the three statements

```
30 A=A-1
40 PRINT A
50 GOTO 100
```

could as well be written in a single line:

```
30 A=A-1: PRINT A: GOTO 100
```

In the second case, line 30 contains three separate statements which cause the system to: (1) decrement the value of A by 1, (2) print the value of A, and (3) branch to line 100. Multiple-statement lines can contribute to more efficient use of memory and somewhat faster program execution. The use of multiple-statement lines also allows program statements to be logically grouped for more readable programs.

Spaces

Spaces can be included within a program line to enhance its readability, but they are ignored by the system when the program is executed. For example, the line

```
10FORI=1TO10STEP2
```

is equivalent, from the system's point of view, to the line

```
10 FOR I = 1 TO 10 STEP 2
```

although the latter is obviously much easier to read. Spaces within a program line are, in general, stored along with the line in memory, each space occupying one byte of memory. However, spaces preceding or embedded within line-numbers or BASIC words are deleted by the system. In addition, a space following a line-number, some BASIC words, or a statement separator does not occupy any memory.

Maximum Program Line Length

A single program line may occupy several lines in the CRT display. The maximum length of a program line which can be entered into memory is determined by the amount of memory available for buffering the line.

When the operator signals that a line which has been keyed in is to be permanently stored in memory (by touching the RETURN Key), the system determines whether there is enough free space to accommodate the line in memory. If not, an error message (ERR A01) is returned, and the program line is not saved in main memory. The first 256 bytes of the line are, however, stored temporarily in a special buffer, enabling the operator to recall the line and, if possible, edit it to fit in the available space. For a more detailed discussion of how programs are stored in memory, refer to Chapter 2, section 2.5. For an explanation of how to edit a program line which does not fit in memory, see Chapter 3, section 3.3.

> **NOTE:**
> Although program lines entered in memory are subject only to the restrictions of available memory, a further restriction is imposed for programs which are to be stored on disk. The maximum length of a program line which can be stored on disk is 253 bytes. If a program line longer than 253 bytes is entered, the system will display an error message (Error A05). This error message is intended solely to warn the operator that the program in its current form cannot be saved on disk. The error does not prevent the program from executing. See Chapter 2, section 2.5 for a more detailed discussion of this problem.

1.6 THE RETURN(EXEC) KEY

Every system keyboard has one or more keys labelled RETURN or EXEC. The RETURN Key has a special significance to the system: it signals that operator entry is complete, and the system may proceed to take appropriate action. When keyed following the entry of a program line, RETURN causes the system to store the line in memory. When keyed following entry of a system command or a statement which is not preceded by a line-number, RETURN causes the system to immediately execute the command or statement. Whenever operator entry is requested by an INPUT or LINPUT statement, RETURN is used to signal that the required information has been entered and may be processed.

1.7 IMMEDIATE MODE OPERATIONS

Instructions which are executed immediately upon entry rather than stored in memory as part of a program for subsequent execution are said to be executed in "Immediate Mode." System commands are almost universally executed in Immediate Mode (most commands cannot legally be included in a program). Although statements are, by definition, programmable instructions, most statements also can be executed in Immediate Mode simply by entering them without a preceding line-number. The statement most commonly used in Immediate Mode is the PRINT statement, which is utilized to display the results of evaluating an expression. For example, to evaluate the expression $(AxB)^2$, it is not sufficient merely to enter

```
(A*B)↑2
```

Instead, the expression must be included in a PRINT statement which will output the result to the CRT display:

```
PRINT (A*B)↑2
```

It is even possible to execute multistatement lines in Immediate Mode. For example, the statement

```
FOR I=1 TO 10: PRINT LOG(I): NEXT I
```

displays the natural logarithms of integers 1-10. This capability makes the system a powerful one-line calculator when used in Immediate Mode.

> **NOTE:**
>
> Execution of an Immediate Mode PRINT statement does not affect the contents of variables in memory, even if those variables are referenced in the PRINT statement. Other Immediate Mode statements (in particular, assignment statements) can, however, alter the contents of variables in memory. Care should be taken in the selection of variables referenced in such statements if the alteration of one or more variables may affect the operation of the program currently in memory.

Certain statements cannot be executed in Immediate Mode. They are:

DATA	LINPUT
DEFFN(')	ON GOSUB(')/GOTO
ERROR	READ
GOSUB(')	RETURN
INPUT	STOP

In addition, statements that reference program line-numbers are not allowed in Immediate Mode, except as noted.

Although a line executed in Immediate Mode is not permanently stored in memory, it is temporarily held in a special buffer area following execution and may be recalled for editing immediately after it is executed. There are, however, some system commands — in particular, LOAD and CLEAR — which cannot always be recalled for editing. In addition, there are a number of conditions which cause the Immediate Mode line to be completely or partially destroyed before it can be recalled. These conditions are described in Chapter 2, section 2.5.

The special buffer area is 256 bytes in size. In the unlikely event that an Immediate Mode line exceeds 256 bytes in length, only the first 256 bytes are saved; the remaining bytes are lost when the line is executed. See Chapter 3, section 3.3 for a detailed discussion of editing Immediate Mode lines.

1.8 BASIC-2 KEYWORDS

Instructions may be entered into the system from most keyboards in two ways. Any instruction may be typed in letter by letter. The PRINT statement, for example, can be entered by typing the letters P-R-I-N-T. Alternatively, if your system is equipped with keyword keyboards, most of the commonly used instructions can be entered with a single keystroke. Most alphabetic, numeric, and special-character keys on the system keyboard also are labelled with a BASIC instruction. When the keyboard is placed in Keyword/A Mode, an instruction can be entered by keying SHIFT and touching the appropriate key. Most system commands occupy their own keys on the right-hand side of the keyboard and do not require the use of SHIFT.

Instructions which can be entered with a single keystroke are referred to as BASIC-2 "keywords." It is immaterial to the system whether an instruction is entered letter by letter or as one keyword.

1.9 THE ASSIGNMENT STATEMENT

One of the most common and fundamental operations performed in a program is the assignment of a value to a variable in memory. In BASIC-2, there are a number of statements which assign values to variables (INPUT, LINPUT, GOSUB', KEYIN, and READ statements are a few examples); the most explicit means of assigning a value to a variable, however, is the "assignment statement."

The assignment statement is identified by the assignment operator "=". For example, the statement

```
100 A = 50
```

assigns the numeric constant "50" to the numeric-variable "A". This statement may be read "replace the current value of A with 50" or simply "assign the value 50 to A."

It is perhaps an unfortunate feature of the BASIC language that the equality sign, "=", is used *both* as the assignment operator *and* as a relational operator denoting equality. These two meanings of the equality sign must be carefully distinguished. In particular, the assignment operator may be used in an assignment statement such as

```
200 A = A + 1
```

which is read "add 1 to the current value of A, and assign the sum to A." Thus, if A=5 prior to execution of statement 200, then A=6 following statement execution. Clearly, such a statement would be impossible if "=" were interpreted as the equality operator.

Many forms of BASIC require the BASIC keyword LET in an assignment statement. In BASIC-2, the word LET is optional and generally is omitted. For example, the following statements are both equally valid:

```
100 A = 5
    or
100 LET A = 5
```

When an assignment statement is processed, the system first evaluates the expression on the right-hand side of the assignment operator and assigns its value to the variable on the left-hand side of the operator. Any valid expression may be specified on the right-hand side of an assignment operator. The expression may be simple, consisting only of a constant or variable, or it may be quite complex. For example:

```
100 N = SQR(B1) + A*C/D
```

In this case, the expression "SQR(B1)+A*C/D" is evaluated, and the resulting value is assigned to N.

Because BASIC-2 recognizes two distinct data types, numeric data and alphanumeric data, there are actually two different types of assignment statements. A numeric assignment statement assigns the value of a numeric expression to a numeric-variable (for example, N=50), while an alphanumeric assignment statement assigns the value of an alphanumeric expression to an alphanumeric-variable (for example, A$="ABC"). Alphanumeric data is assigned on a character-by-character basis.

The same value can be assigned to several variables in an assignment statement; this feature is useful for initializing variables at the start of a program. For example, the statement

```
100 A,N,X = 1
```

assigns the value 1 to the variables A, N, and X.

CHAPTER 2 MEMORY ORGANIZATION

2.1 INTRODUCTION

This chapter describes the logical structure of internal memory in a 2200VP single-user CPU and in each partition of a 2200MVP multiuser CPU. Its intent is to clarify certain important concepts concerning the storage of program text and variable data in user memory and to explain in general terms the techniques employed by the system to locate critical information in memory. The following topics are discussed:

- The three phases of Program Mode operation (Entry Phase, Resolution Phase, and Execution Phase) and the concept of "text atomization."
- Data storage in memory (constant data, numeric- and alphanumeric-variables, scalar- and array-variables, one- and two-dimensional arrays, and common and noncommon data).
- The structure of the "internal stacks" used for the storage of system control information.
- The meaning of "free space" in memory and its importance for the programmer. In particular, the END statement, the SPACE function, and Errors A01 and A02 are treated.

The manner in which the 2200MVP memory is divided into individual partitions and the techniques available for interpartition communication are described in Chapter 16, sections 16.2 and 16.3.

Although a great deal of the information contained in this chapter is "background" material not directly required to write programs in BASIC-2, it will enable the programmer to develop a better understanding of the system's logical structure. Such a general understanding can, in turn, help the programmer to produce more efficient code by designing his programs to use the system's architecture most efficiently. Many of the topics discussed in Chapter 2 can also contribute to a better understanding of specific system features, including certain types of error conditions and certain BASIC statements (such as COM CLEAR and RETURN CLEAR) whose functions are meaningful only in the context of the 2200VP's (or MVP's) particular architecture.

2.2 THE PHASES OF PROGRAM MODE OPERATION

The process of entering and executing a BASIC-2 program or Immediate Mode line is composed, from the system's point of view, of three distinct phases: Entry Phase, Resolution Phase, and Execution Phase. During each phase, a specific set of actions (which includes checking for certain types of errors) is performed by the system.

Entry Phase

As its name implies, Entry Phase encompasses the entry of BASIC-2 program text into memory, usually from the keyboard or a disk. In most cases, a program is initially entered from the keyboard. When the operator has entered a line and keyed RETURN, the line is atomized and scanned for syntax errors. An appropriate error message is displayed if an error is found. If the line is error free and is not preceded by a line-number, it is immediately resolved and executed. If the line is preceded by a line-

number, it is stored in memory. A numbered program line is stored in memory whether or not it contains syntax errors.

BASIC-2 programs in memory are syntactically analyzed and executed by the BASIC-2 "interpreter." The interpreter scans each line of program text, "interprets" each BASIC-2 statement in the line, and performs the appropriate action or sequence of actions defined for that statement. When the BASIC-2 statement PRINT is encountered in a program line, for example, the interpreter initiates the sequence of actions defined for the PRINT statement.

Many systems which use an interpreter to execute programs in a high-level language simply store the program text in memory exactly as it is entered by the programmer (assuming it is syntactically correct). Each line is then scanned for character strings corresponding to legal words in the language. The Wang BASIC-2 interpreter, however, adds a further refinement to this process. Legal BASIC-2 keywords are not stored in memory exactly as they are keyed in from the keyboard; instead, each keyword is converted into a one-byte "text atom" by the interpreter before it is stored in memory.

The "text atom" for each BASIC-2 keyword is a unique 8-bit code which the interpreter regards as equivalent to the keyword itself. For example, the text atom for the keyword PRINT is hex A0. Whenever the interpreter encounters a hexadecimal A0 code in scanning a text line, it "interprets" this code as equivalent to the keyword PRINT and takes appropriate action. Of course, the interpreter is smart enough to recognize the difference between the BASIC-2 keyword PRINT and the character string "PRINT". The BASIC-2 word is atomized; the character string is not (it is stored as the five-letter string P-R-I-N-T). Similarly, when the interpreter is scanning program lines during execution, it recognizes the difference between the text atom A0 and the character A0 specified in a literal (e.g., HEX(A0)).

The value of text atomization is twofold. First, it permits more efficient storage of program text since most BASIC-2 keywords in a program require only a single byte of storage. PRINT, for example, requires only one byte when atomized; it would require five bytes if stored character by character. Secondly, atomization permits faster program execution since program lines are more compact and can be more rapidly scanned, with keywords more readily identified and "interpreted."

Program lines may be entered in any order; when a new line having a unique line-number is entered, the operating system automatically ensures that it is inserted in proper line-number sequence in the resident program. When a new line is entered with a line-number identical to that of an existing line in memory, the original line is removed and the new line replaces it in memory. This is the process of "line replacement." When a line-number is entered and followed only by a carriage return (i.e., the operator has entered the line-number and keyed RETURN), the corresponding line is removed from memory, but not replaced by a new line. This is the process of "line deletion."

Resolution Phase

Resolution Phase is entered just prior to program execution and is initiated by the execution of a RUN or LOAD RUN command or a LOAD statement. Its main functions are to reserve space in memory for all referenced variables and to ensure that all referenced line-numbers (e.g., in GOTO, GOSUB, etc. statements) exist. Resolution Phase consists of a complete sequential scan of the program. If the entire pass is error free, program execution immediately begins. If an error is detected, the system outputs an error message, aborts the resolution procedure, inhibits program execution, and returns to Entry Phase. The program is considered to be "unresolved."

In general, every BASIC-2 program must be resolved initially. Once resolved, a program in memory need not be resolved again unless modifications are made to existing program text or new program text is added. Note that while there are a number of techniques for initiating program execution, only the RUN and LOAD RUN commands and the LOAD statement (program overlaying) initiate program resolution prior to execution. Thus, one of these instructions must be used for the first execution of a program; subsequently, a technique which does not cause resolution (such as depressing a Special Function Key) can be used to reexecute the program.

Execution Phase

In general, Execution Phase is entered only after an error-free resolution has been completed (although it can be entered directly via a Special Function Key). During Execution Phase, program lines

are executed in line-number sequence except when program execution is transferred to a specified line by a branch statement (GOTO, GOSUB, FOR...NEXT, IF...THEN, etc.). Multiple statements on a line are executed sequentially from left to right. Program execution continues until one of the following conditions occurs:

- A STOP or END statement is executed.
- The last statement in the program is executed.
- An error occurs (unless the error termination is suppressed with a SELECT ERROR or ERROR statement; see Chapter 9, section 9.3).
- The HALT/STEP Key or RESET button is pressed by the operator.

2.3 DATA STORAGE IN MEMORY

Two types of data can be stored in memory and operated on in BASIC-2: numeric data and alphanumeric data. Numeric values can participate in arithmetic operations (addition, subtraction, etc., and math functions); each numeric value is stored in a system-defined format called Wang internal numeric format and always occupies exactly eight bytes. Alphanumeric values cannot participate in standard arithmetic operations; they are stored on a character-by-character basis in memory, and the amount of memory they require is specified by the programmer. BASIC-2 provides a variety of instructions for logically testing, manipulating, and converting alphanumeric values by changing the values of specific bits within each character.

Numeric and alphanumeric values may be specified in a program as constants or stored as variable data. Numeric constants can be expressed in fixed-point or exponential format (the rules and restrictions for each format are explained in Chapter 4, section 4.2). Alphanumeric constants can be specified as character strings in quotes, called alpha literal strings (e.g., "ABC"), or as HEX literal strings (e.g., HEX(0A0D)). Alpha and HEX literals are covered in Chapter 5, sections 5.4 and 5.5, respectively.

Numeric-and Alphanumeric-Variables

In addition to numeric and alphanumeric constants, data can be stored in memory as the values of variables. Variables are defined sections of data memory which can be assigned values by the programmer and whose values can be changed and examined under program control. In BASIC-2, two classes of variables can be defined, corresponding to the two types of data: numeric-variables and alphanumeric-variables. Numeric-variables can contain only numeric values, and alphanumeric-variables (also sometimes called "string variables") can contain only alphanumeric values. On the 2200MVP, a special category of alpha- and numeric-variables additionally can be defined, called "global variables." Global variables are discussed in Chapter 16, section 16.9. Each variable must be identified with a unique variable-name. In BASIC-2, variable-names consist of a single uppercase letter (A-Z) or a single letter followed by a single digit (0-9). Alphanumeric-variable-names are distinguished from numeric-variable-names by the presence of a dollar sign ("$") immediately following the variable-name. (For example, N is a numeric-variable-name, while N$ is an alpha-variable-name; similarly, A3 is a numeric-variable-name, while A3$ is an alpha-variable-name.) Examples of valid variable-names are:

Numeric	Alphanumeric
A	A$
A1	F0$
B0	P9$
N8	S1$

A numeric-variable-name and an alpha-variable-name always identify different variables, even if the names (i.e., the letters or the letters and digits) are identical. For example, N1 and N1$ identify two different variables, one numeric and one alphanumeric, and both may be used without ambiguity in the same program.

Scalar-Variables and Array-Variables

Within each of the two classes of variables (alpha and numeric), two different types of variables can be defined: scalar-variables and array-variables. A scalar-variable is one which can contain only one value. A numeric-scalar-variable can contain a single numeric value (the range of numeric values which can be stored in a variable is defined in Chapter 4, section 4.2); a numeric-scalar-variable always is exactly eight bytes in length. An alpha-scalar-variable can contain a single, contiguous character string and can range from one byte to 124 bytes in length. If the length of an alpha-scalar-variable is not defined by the programmer in a DIM or COM statement, the system assigns a default length of 16 bytes to the variable.

An array-variable is really a collection of scalar-variables identified by a common name. Each scalar-variable contained in the array is referred to as an "element" of the array and can be identified by specifying the array-name, followed by a subscript (or a pair of subscripts) which locates the element within the array. For example, the fifth element in array N( ) could be specified as N(5). Note that the subscript is enclosed in parentheses immediately following the array-name. The names of array-variables are formed in exactly the same way as the names of scalar-variables (that is, a letter or a letter and a digit). Since scalar-variables are different from array-variables, the same name (i.e., the same letter or the same letter and digit) may be used both as a scalar-variable-name and as an array-variable-name. Thus, N( ) names an array-variable, while N names a scalar-variable.

In general, any reference to an array-variable must consist of the array-name immediately followed by parentheses. If the parentheses enclose an expression or a pair of expressions, the expressions are interpreted as the subscripts of a particular element in the array. In the example cited above, N(5) identified the fifth element of array N( ). If the entire array rather than a particular element of the array is to be referenced, the array-name must be followed by empty parentheses, e.g., N( ) or A$( ). This special technique for referencing an entire array is called the "array-designator." An array can have the same name as a scalar-variable, but the array must (with few exceptions) always be referenced with an array-designator to indicate that an array rather than a scalar-variable is meant. For example:

N6	identifies a numeric-scalar-variable.
N6$	identifies an alpha-scalar-variable.
N6( )	identifies a numeric-array.
N6$( )	identifies an alpha-array.

One-Dimensional Arrays and Two-Dimensional Arrays

Array-variables are of two types: one-dimensional and two-dimensional. A one-dimensional array is a "list" of variables, all identified by the same name. A two-dimensional array is a "table" of variables, all identified by the same name.

One-dimensional arrays are also called "lists," "vectors," "column vectors," and, since each element is identified by a single subscript, "singly subscripted arrays." In general, the term preferred by a programmer is the one which makes the most intuitive sense for his application: programmers involved in data processing tend to prefer "list," while those programming mathematical applications tend to favor "vector." To avoid prejudicing the discussion in favor of either group, the more neutral term "one-dimensional array" will be used consistently throughout this manual except in Chapter 13, which is explicitly devoted to mathematical applications.

A one-dimensional array can be conceived as a list or column of variables ("elements"), each occupying its own slot or "row" in the column. Consider, for example, the representation of array N( ) in Figure 2-1.

	N( )
Row 1	N(1)
Row 2	N(2)
Row 3	N(3)
Row 4	N(4)
Row 5	N(5)

Figure 2-1.
The One-Dimensional Array N( )

Note that N( ) contains a total of five elements, and that each element is identified by specifying its row in the column (e.g., element N(3) is located in row 3).

It is not difficult to generalize the scheme in Figure 2-1 to contain two or more columns. When this is done, the result is a two-dimensional array. Two-dimensional arrays are also called "tables," "matrices," and, because each element is identified by a pair of subscripts, "doubly subscripted arrays." In this manual, the term "two-dimensional arrays" will (with the exception of Chapter 13) be used consistently.

A two-dimensional array can be conceived as a table consisting of two or more columns of elements. Consider, for example, the representation of the two-dimensional array M( ) in Figure 2-2.

	M( ) Column 1	Column 2
Row 1	M(1,1)	M(1,2)
Row 2	M(2,1)	M(2,2)
Row 3	M(3,1)	M(3,2)
Row 4	M(4,1)	M(4,2)
Row 5	M(5,1)	M(5,2)

Figure 2-2.
The Two-Dimensional Array M( )

Note that M( ) consists of two columns of elements, with five rows in each column, for a total of 10 elements. In this case, it is not sufficient to identify each element by its row since the row may be in column 1 or column 2. A second subscript is required to identify the column. The convention followed when referencing a particular element in a two-dimensional array is always to specify the row first and then the column. Thus, M(3,2) identifies the element in the third row of column 2.

Maximum Array Size

There are, of course, certain restrictions placed upon the size of an array. The primary restriction is imposed by the available memory size: an array may not contain more bytes than the available memory allows. For one-dimensional arrays, the available memory size is the only restriction; since the maximum possible memory size for a program is less than 64K or 65,535 bytes, the maximum possible number of elements in a one- dimensional array approaches 65,535. Note that this number is a theoretical maximum which could never be reached in practice. For two-dimensional arrays, a second restriction is imposed in addition to memory size. The system uses a single byte to represent each subscript of a two-dimensional array internally. Since the maximum number which can be represented in one byte (eight bits) is 255, each subscript of a two-dimensional array is restricted to a maximum value of 255. This restriction can be expressed in a different way by stating that each row in a two-dimensional array is limited to a maximum of 255 elements and each column is limited to a maximum of 255 rows. Thus, the maximum total number of elements in a two-dimensional array is 255 x 255 or 65,025 elements.

These restrictions on array size can be summarized formally in the following way:

One-Dimensional Arrays e.g., N(d)	**Two-Dimensional Arrays** e.g., M(d1,d2)
1<= d < 65536	1< = d1,d2 < 256

The arrays N( ) and M( ) used in Figures 2-1 and 2-2 are numeric-arrays in which each element has a fixed length of eight bytes. Thus N( ), which consists of five elements, has a total of 5x8 = 40 bytes. Similarly, M( ), which has 10 elements, occupies 10x8 = 80 bytes. Clearly, the maximum limits described above cannot be approached with a numeric-array since, for example, a one-dimensional numeric-array with 65,535 elements would occupy a total of 65,535 x 8 = 524,280 bytes, a number which greatly exceeds the largest available memory size. (Indeed, each variable also requires a few bytes of control information in memory, inflating this total somewhat further.) The critical consideration when defining arrays must always be the amount of memory space available since that represents the only practical restriction on array size in most cases.

For alphanumeric-arrays, the length of each element is not fixed by the system. Instead, it can be set by the programmer to any length between one byte and 124 bytes, inclusive. An alpha-array consisting of five elements each one byte in length would occupy five bytes, while an array of five elements each 124 bytes in length would occupy 5 x 124 = 620 bytes (not counting control bytes; see below). In this case, as with numeric-arrays, the determining factor in restricting array size is the available memory space.

Control Bytes Used for Scalar- and Array-Variables

The preceding discussion of array memory requirements failed to take into account the system "overhead" associated with each scalar- and array-variable. In order to identify each variable in memory, the system automatically inserts several bytes of control information at the beginning of the variable area. These additional bytes are completely "transparent" to the programmer (that is, they are used exclusively by the operating system and are never seen by the BASIC-2 program), and they represent a fixed overhead for each variable defined in a program. The number of control bytes required differs according to whether the variable is alpha or numeric, scalar or array. There are four possible conditions:

Variable Type	Number of Control Bytes
numeric-scalar	4
alpha-scalar	5
numeric-array	6
alpha-array	7

Figure 2-3.
System Overhead for Different Variable Types

Note that the control bytes are *not* replicated for each element of an array; they are specified only once, prior to the first element of the array in memory. Thus, a numeric-array with five elements has exactly the same system overhead (6 bytes) as a numeric-array with 500 elements.

To compute the total number of bytes, T, actually required for a variable in memory, the following formula can be used:

$$T = N * L + O$$

where:

N = the total number of elements in the array (N=1 for scalar-variables).

L = the length of the scalar-variable or each array- element (for numeric-variables, L=8; for alpha-variables, $1 <= L <= 125$).

O = system overhead (see Figure 2-3).

For example, suppose the following variables are defined:

```
10 DIM A$(2,2)26, N(6), F$2
```

For the alpha-array A$( ), the total storage requirement is:

T = 4x26+7 = 111 bytes

For the numeric-array N( ):

T = 6x8+6 = 54 bytes

For the alpha-variable F$:

T = 1x2+5 = 7 bytes

> **NOTE:**
>
> To make its housekeeping easier, the system always reserves an even number of bytes for a variable. If the total computed storage requirement is an odd number of bytes, therefore, the system adds 1 to the total. For example, the total storage requirement for A$( ) above is actually 111 + 1 or 112 bytes. Similarly, F$ occupies 8 bytes rather than 7.

Variable Definition: The DIM and COM Statements

Throughout the preceding discussion, no mention was made of the techniques used to define variables in a program. In general, there are three ways to define a variable:

- Implicit definition by reference in a BASIC-2 statement (scalar-variables only).
- Explicit definition in a DIM or COM statement (scalar- and array-variables).
- Implicit definition in a Math Matrix statement (array-variables only).

Scalar-variables do not need to be explicitly defined. A scalar-variable used in a program is automatically or "implicitly" defined when it is first referenced in the program. Both numeric- and alphanumeric-scalars can be implicitly defined in this way (the system assigns the default length of 16 bytes to an implicitly defined alpha-scalar).

Array-variables must, in general, be defined explicitly. The only exception to this general rule occurs with the Math Matrix statements, which provide a technique for implicit definition of arrays (see Chapter 13, section 13.2). Any reference to an undefined array in a program produces an error. Explicit variable definition can be carried out with a DIM or COM statement. For example, the statement

```
20 DIM N(5), M(5,2), A$(10,10)100
```

defines three arrays. N( ) is a one-dimensional numeric-array of five elements; M( ) is a two-dimensional numeric-array of five rows and two columns (10 elements); A$( ) is a two-dimensional alpha-array of 10 rows and 10 columns (100 elements), with each element 100 bytes in length.

If an alpha-scalar is to be assigned a length other than the default length of 16 bytes, the length must be defined explicitly. For example, the statement

```
30 DIM F$124, A4$3
```

defines the alpha-scalars F$ with a length of 124 bytes and A4$ with a length of three bytes. Numeric-scalars can, for purposes of documentation, be specified in a DIM statement, but their length is always fixed at eight bytes and cannot be changed by the programmer.

The COM statement can be used in a manner analogous to DIM for explicitly defining variables. For example, the statement

```
20 COM N(5), M(5,2), A$(10,10)100
```

defines the three arrays N( ), M( ), and A$( ) exactly as the DIM statement on line 20 on the previous page. There is, however, an important difference between DIM and COM: the variables defined by DIM are located in the noncommon area of memory and are called "noncommon variables," while the variables defined by COM are located in the common area of memory and are called "common variables." The distinction between common and noncommon variables has great significance, particularly in program overlaying applications: noncommon variables are automatically cleared from memory when a program overlay is loaded in, while common variables are not affected by an overlay operation.

The Variable Table

The area of memory in which variables are stored is called the Variable Table. The Variable Table begins at the "bottom" of memory and expands upward as variables are defined. Space is allocated in the Variable Table for all defined variables during program resolution. In this process, the Variable Table is constructed by allocating space for variables in the sequence in which they are defined in the program. As each new variable is defined, it is automatically allocated space at the top of the Variable Table. Thus, the first variable defined occupies the first or bottommost position in the table, the next variable defined occupies the space immediately above it, etc. Consider, for example, the following pair of statements:

```
10 COM N,M(2,5),L$
20 DIM A$(10,10)100,R(3),B$25
```

Figure 2-4 below shows the Variable Table resulting from statements 10 and 20:

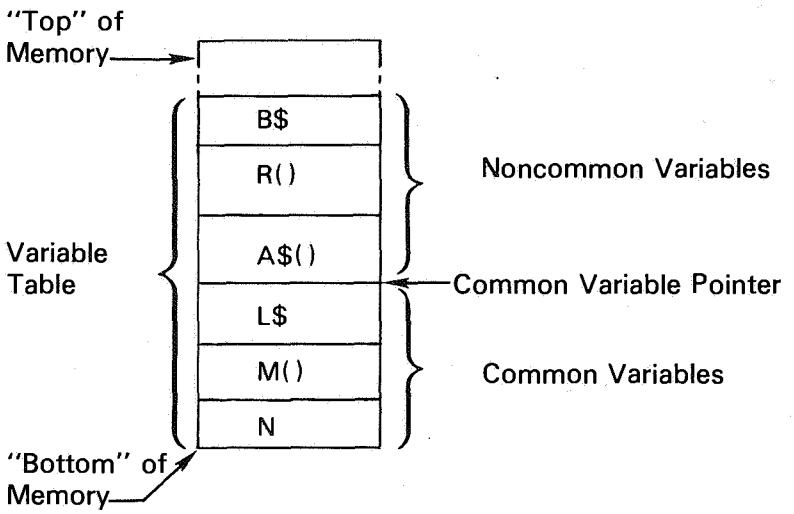


Figure 2-4.
The Variable Table in Memory

Note that the first variable defined, N, is assigned the first position in the table, the second variable defined, M( ), is assigned the next position, etc.

Common and Noncommon Variables

The Variable Table contains two types of variables: common variables and noncommon variables. All common variables are placed in the table first, followed by the noncommon variables. Common variables are distinguished from noncommon by a special pointer, called the Common Variable Pointer (CVP), which always points immediately above the last common variable in the table. All variables below the Common Variable Pointer are common variables; all those above it are noncommon variables. In Figure 2-4, for example, N, M( ), and L$ are common, while A$( ), R( ), and B$ are noncommon.

It must be seen that this technique for distinguishing common from noncommon imposes a severe restriction on the order in which these variables are defined. In particular, *all* common variables *must* be defined in the program before *any* noncommon variables are defined. When the first noncommon variable is defined in a program, the system sets the Common Variable Pointer in the Variable Table and regards all subsequent variables as noncommon. If the program attempts to define common variables subsequently in the program, an error is signalled and program resolution is terminated.

When a program overlay is loaded into memory with a LOAD statement, all noncommon variables are cleared from memory, while the common variables are not affected. Common variables are therefore extremely useful for storing common data which is to be used by several program modules.

The functions of the several forms of the CLEAR command also can be understood more clearly with reference to Figure 2-4. When a CLEAR command with no parameters is executed, all of memory, including program text and the entire Variable Table, is cleared. A CLEAR P command clears only program text, without affecting the Variable Table. CLEAR V, conversely, clears the entire Variable Table, but leaves the program intact. CLEAR N, finally, clears all noncommon variables (all those above the Common Variable Pointer) but does not affect the common variables below that point.

The COM CLEAR Statement

It can be seen that to move a variable from the common to the noncommon area, or vice versa, it is necessary only to move the Common Variable Pointer up or down in the Variable Table. For example, suppose the CVP in Figure 2-4 is moved "down" one variable. Then Figure 2-5 would result:

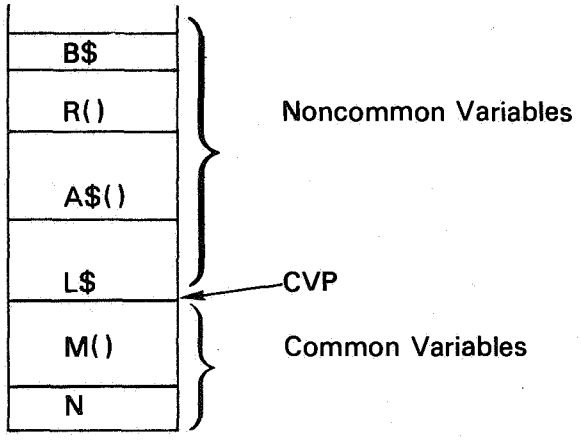


Figure 2-5.
Variable Table in Figure 2-4 Following Downward Movement of CVP

Note that the last common variable in Figure 2-4, L$, has now become noncommon in Figure 2-5 as a result of the movement of the CVP. It sometimes occurs in highly modular systems that variables required for common data in earlier modules are no longer needed in later modules. In such cases, it would be convenient to have a means of redefining the unwanted common variables as noncommon so that they can be cleared when the subsequent module is loaded in. Less frequently, it may be desirable to redefine one or more noncommon variables as common. In either case, the procedure involves simply moving the CVP up or down within the Variable Table. A special BASIC-2 statement is available to shift the position of the CVP: COM CLEAR. COM CLEAR has three different forms: COM CLEAR with a specified common variable, COM CLEAR with a specified noncommon variable, and COM CLEAR with no parameters.

A COM CLEAR statement with a specified common variable moves the CVP "down" the Variable Table to the point immediately below the specified variable. The specified variable and all common variables above it (i.e., all variables defined after it in the program) are thereby shifted into the noncommon area and redesignated as noncommon variables. For example, if the Variable Table in Figure 2-4 is assumed in memory, Figure 2-6 results from executing statement 100 below:

```
100 COM CLEAR M()
```

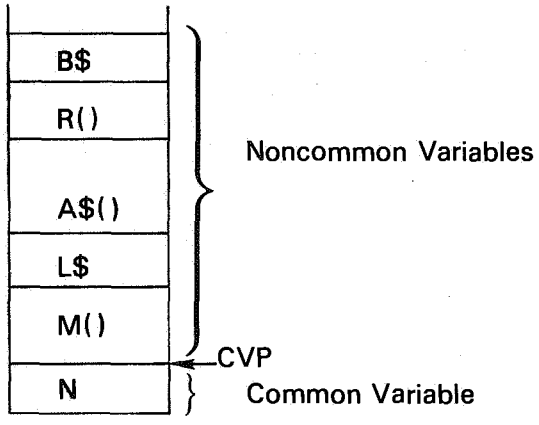


Figure 2-6.
Result of Executing a COM CLEAR M() Statement

Notice that the CVP has moved to a point below the specified common variable, M(), in effect redesignating M() and any succeeding common variable as noncommon. A subsequent program overlay will clear those variables from memory.

A COM CLEAR statement with a specified noncommon variable moves the CVP "up" the Variable Table to the point immediately below the specified variable. All noncommon variables below the specified variable (i.e., all variables defined *before* it in the program) are thereby shifted into the common area and redesignated as common variables. The specified variable itself and all variables defined *after* it in the program remain noncommon. For example, if the Variable Table in Figure 2-6 is assumed, Figure 2-7 results when statement 200 below is executed:

```
200 COM CLEAR B$
```

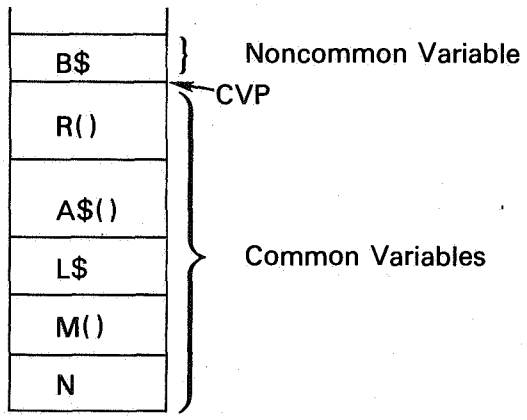


Figure 2-7.
Result of Executing a COM CLEAR B$ Statement

Note that the CVP has been moved to a point immediately below the specified noncommon variable, B$, in effect redesignating all noncommon variables defined prior to B$ as common. These variables will not be altered by a subsequent program overlay.

A COM CLEAR statement with no parameters simply shifts the CVP to the bottom of memory, in effect redesignating *all* currently defined common variables as noncommon. A subsequent program overlay will clear all variables from memory.

2.4 THE INTERNAL "STACKS"

The system maintains a pair of internal "stacks" in memory for the temporary storage of certain control information: the Operator Stack and the Value Stack. On the 2200VP, there is one Value Stack and one Operator Stack. On the 2200MVP, each partition has its own Value Stack and Operator Stack. The Operator Stack is stored in a reserved section of memory; it grows or shrinks in size as information is added to or deleted from it, but it has a fixed maximum limit beyond which it cannot expand. The Value Stack is located in the middle area of user memory above the Variable Table and below the BASIC-2 program text (see Figure 2-8). The Value Stack, too, increases or decreases in size as information is added or deleted; its maximum size is determined by the amount of memory occupied by variables and program text.

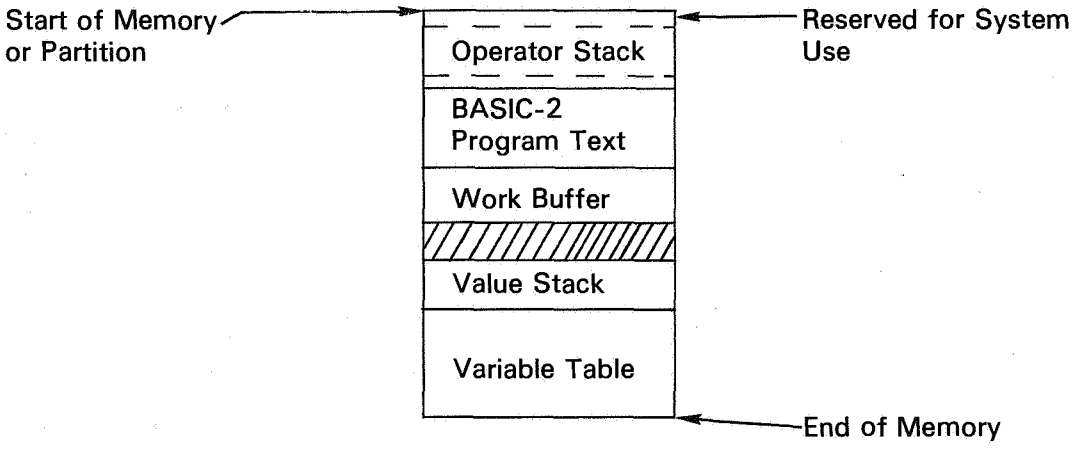


Figure 2-8.
Schematic Representation of Memory, Showing Relative Locations
of Operator Stack, Program Text, Work Buffer, Value Stack,
and Defined Variables

The Operator Stack and the Value Stack are used in conjunction by the system to maintain certain critical information, including return addresses for subroutines and FOR/NEXT loops. When a subroutine call is issued (with GOSUB, GOSUB', ON GOSUB, etc.), the system must have a means of "remembering" the location of the statement following the GOSUB so that a return can be made to that statement when subroutine execution is finished. Before branching to the subroutine, therefore, the system saves the address of the next sequential statement in the Value Stack and simultaneously inserts an item in the Operator Stack identifying this address as a subroutine return address. Similarly, when a FOR...TO statement is executed, the address of the statement following FOR...TO and other required information is saved in the Value Stack, along with a control byte or "atom" in the Operator Stack identifying the information in the Value Stack as loop parameters. When the corresponding NEXT statement is executed, the system branches back to the appropriate loop address to begin the next iteration of the loop. The two stacks also have a third use in expression evaluation. In the process of evaluating an expression, the system stores the expression operators in the Operator Stack and places operand values and intermediate results in the Value Stack. It is always the case, therefore, that the two stacks are used in conjunction; whenever an item of information is placed in one stack, a corresponding item is placed in the other.

Information is stored sequentially in the stacks, starting at the bottom. As each statement which utilizes the stacks is executed, the associated data for that statement is placed in the stacks. For example, consider the program outline in Figure 2-9:

```
Subroutine                100 GOSUB 500
  Call                      .
                            .
                            .
                          500 FOR I=1 TO 10
                          510 FOR J=1 TO 20
Subroutine                  .
                            .
                            .
                          640 NEXT J
                          650 NEXT I
                            .
                            .
                            .
                          700 RETURN
```

Figure 2-9.
Program Schema Illustrating Statements Which Require the Use of the Internal Stacks

This program contains three statements requiring the use of the internal stacks: the GOSUB statement at line 100 and the two FOR...TO statements at lines 500 and 510. Return information is placed into the stacks in the order in which the corresponding statements are executed; Figure 2-10 illustrates the contents of the stacks following the execution of line 510 in the example program.

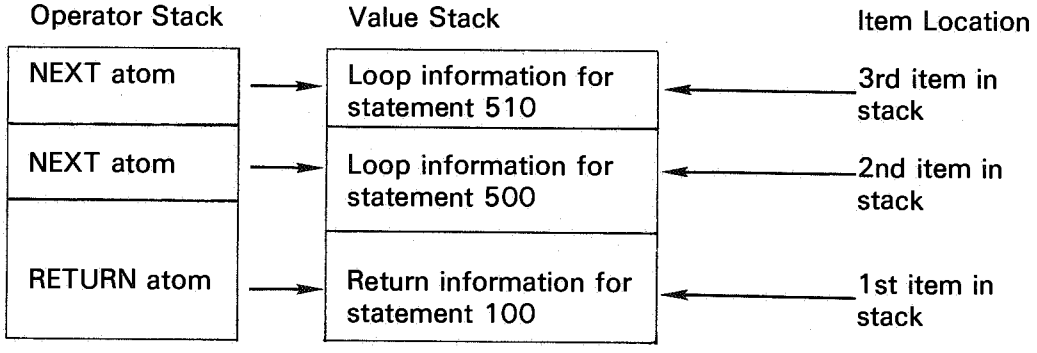


Figure 2-10.
Contents of Internal Stacks for the Program in Figure 2-9

Information normally is deleted from the stacks in the following ways:

- Execution of a subroutine RETURN statement.
- Normal termination of a FOR/NEXT loop.
- Execution of a RETURN CLEAR statement.

Expression analysis information is deleted from the stacks as soon as evaluation of the expression is completed.

A RETURN statement causes a branch to the statement following the last-executed GOSUB (or GOSUB') and automatically clears return information for that subroutine from the stacks. Normal termination of a loop causes execution to continue at the statement following NEXT and clears loop control information for that loop from the stacks.

It is a characteristic of stacks that information is always added or deleted at the top of the stack. It is not possible, therefore, to delete an item from the middle of the stack, leaving a "hole" in the stack. Whenever an item is deleted, therefore, any items *above* it in the stack are also deleted.

In general, this restriction on adding to and deleting from the stack causes no programming difficulties since the last item added typically is the first to be deleted. In Figure 2-9, for example, the innermost loop (starting at line 510 and ending at line 640) typically would be the first to terminate. Its control information would then be deleted from the top of the stacks. Next, the outermost loop (lines 500-650) would terminate, clearing its data from the top of the stacks. Finally, the RETURN statement would execute, clearing the subroutine return information from the stacks.

Notice, however, that complications can arise when the normal order of execution is not followed in the program. Suppose, for example, the RETURN statement at line 700 in Figure 2-9 is moved up to line 630. The RETURN would now be executed before normal termination of either loop. In this case, the stack items for the subroutine *and* for both loops are deleted from the stacks since the subroutine item is the bottommost item in the stack. Any attempt to reexecute the NEXT statement for either loop would now produce an error.

Stack Overflow (ERR A04)

If additions are continually made to the stacks without corresponding deletions, the stacks eventually will become filled, and a Stack or Memory Overflow error will result. In normal program execution, this condition arises if the programmer attempts to nest more levels of subroutines or FOR/NEXT loops than the system can handle. Although the number of subroutines or loops which can be nested in a program is dependent upon a variety of considerations (including the amount of expression analysis being done in the program), the maximum ranges up to 64 in the best cases. In practice, however, no program would purposefully employ so many levels of nesting. A far more common cause of stack overflow is failure to follow normal programming procedure when using loops or subroutines. Repeatedly exiting from subroutines without executing RETURN statements, for example, or branching out of loops before normal loop termination occurs causes information to accumulate in the stacks with no corresponding deletions. If either of these operations is repeated continually, a stack overflow eventually will result.

Because the Operator Stack has a fixed maximum size (in contrast to the Value Stack, whose size is determined by the available memory space), it generally overflows before the Value Stack, triggering a Stack Overflow error, ERR A04. In the relatively rare situations where memory is filled nearly to capacity, the Value Stack may have less space available to it than does the Operator Stack and will overflow first. In this case, a Memory Overflow error (ERR A02) is signalled. Either error condition causes both stacks to be automatically cleared of their entire contents.

Preventing Stack Overflow with RETURN CLEAR

A common programming technique involves the use of keyboard Special Function Keys to initiate execution of subroutines in memory. If these subroutines conclude with ordinary RETURN statements, the RETURN halts program execution and returns control to the keyboard — a situation which may not be desirable. If RETURN statements are not used to terminate the subroutines, however, a stack overflow eventually will result. This problem can be avoided by using the RETURN CLEAR statement.

RETURN CLEAR is effectively a "dummy" RETURN statement which deletes subroutine information from the stacks *without* returning control to the statement following the last GOSUB/GOSUB' (or, in the case of subroutines called via Special Function Keys, to the keyboard). When a subroutine is ended with RETURN CLEAR, the control information for that subroutine is deleted from the stacks, and program execution continues at the next sequential statement following RETURN CLEAR. If the "ALL" parameter is used (i.e., RETURN CLEAR ALL), the stacks are cleared of their entire contents.

Automatic Clearing of the Stacks

The stacks are automatically cleared or "flushed" of their entire contents by the following conditions:

- Master Initialization.
- CLEAR.
- RESET.
- RUN.
- Program overlay (LOAD statement).
- Addition of a new variable to the Variable Table.
- Execution of an END statement in Program Mode.
- Execution of the last program statement.
- Occurrence of any Execution Phase error condition which causes an error message to be displayed. (Errors which are intercepted by SELECT ERROR or ERROR do not flush the stacks.)

2.5 MEMORY ORGANIZATION AND THE CONCEPT OF "FREE SPACE"

One of the most important concepts regarding internal memory is that of "free space." The amount of free space available determines whether the programmer can add new program lines or variables and whether he can successfully run his program once they are entered. To more easily understand the concept of free space, it will be helpful to have a graphic representation of internal memory organization. Figure 2-11 provides such a representation.

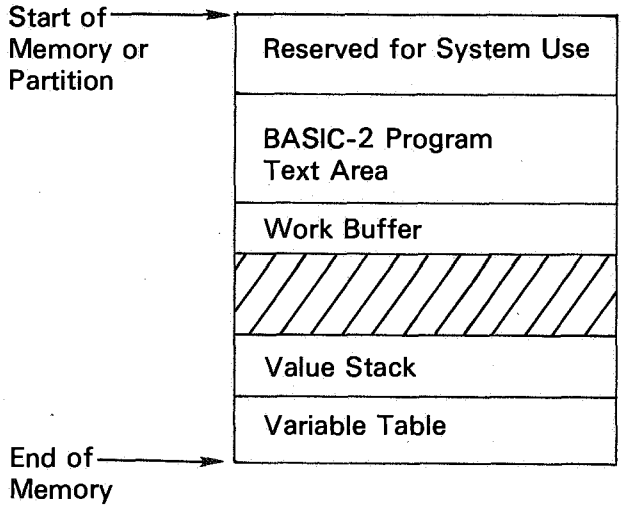


Figure 2-11.
Internal Memory Organization

The Variable Table starts at the "bottom" of memory and grows upward; whenever a new variable is defined, it is added at the top of the Variable Table. The Value Stack "floats" above the Variable Table, being pushed progressively upward as the Variable Table grows. Conversely, the BASIC-2 program begins at the "top" of memory and grows downward as new program lines are added. The Work Buffer "floats" below the program, moving progressively downward as the text area expands.

From this description, it can be seen that the amount of free space is equal to the area between the top of the Value Stack and the bottom of the Work Buffer. The Value Stack, however, is not of fixed size; it expands and contracts in size during the course of program execution, and its size is zero prior to program execution.

To provide the programmer with a completely accurate picture of how much free space is available in memory at any time before or during program execution, the system supports separate instructions which calculate free space values excluding or including the space occupied by the stack. These instructions are the END statement and the SPACE function.

Free Space Values Computed by END and SPACE

The difference between the free space values returned by END and SPACE can be summarized as follows:

- The END statement automatically terminates program execution and returns the amount of memory not currently occupied by program text or data. Space occupied by the Value Stack is *not* figured in this value. (Both the Value Stack and the Operator Stack are automatically flushed by an END statement executed in Program Mode; however, they are *not* flushed by END in Immediate Mode.)

- The SPACE function returns the amount of memory not currently occupied by program text or data *minus* the amount occupied by the Value Stack. This value represents the actual amount of free space in memory at any point during program execution.

Notice that if the Value Stack is empty (as it would be, for example, before program execution begins or after it has been "flushed" by RESET or some other technique), the free space values returned by END and SPACE will be equal. As the Value Stack increases in size during program execution (as loops and subroutines are executed, expressions evaluated, etc.), the free space value returned by SPACE will decrease, while that returned by END remains constant. The relationship between the END and SPACE values is graphically illustrated in Figure 2-12.

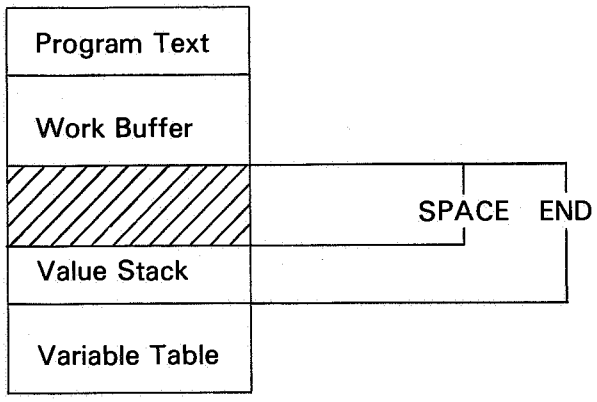


Figure 2-12.
Free Space Values Returned by END and SPACE

It is important for the programmer to recognize that a situation may arise in which there is enough free space to enter a program, but not execute it since the Value Stack initially occupies zero bytes, yet may expand during program execution. To determine how much free space actually is available, free space must be checked by SPACE during program execution when the Value Stack attains its maximum size. Typically, the Value Stack reaches maximum size when the program executes the innermost loop in a series of nested loops. SPACE can be executed in the innermost loop to determine how much free space is available at that point.

The Work Buffer Area in Memory

The Work Buffer "floats" at the end of the program text area in memory. It is used to temporarily store information transferred into memory from the keyboard as well as for temporary storage of data for certain system functions such as LIST DC, MOVE, and COPY. Immediate Mode lines and system commands transferred to the Work Buffer are immediately executed and then flushed, while numbered program lines are threaded into the program text and saved.

The Work Buffer can become as large as necessary (subject to available space) to contain an entered line. In every case, however, the system reserves a fixed minimum of 192 bytes for the Work Buffer. No program text (numbered statement lines) or variables may be stored in this minimum buffer, although it may be used by Immediate Mode statements and system commands; thus, it always represents the minimum distance between the bottom of the program text area and the top of the Variable Table. When the addition of a new program line or variable threatens to overlap into the minimum buffer area, a Memory Overflow error is signalled, and the program line or variable is not stored.

Before computing free space (with END or SPACE), the system automatically subtracts 192 bytes from the available space for the minimum Work Buffer. Thus, even when END and SPACE return free space values of zero, there remains a minimum of 192 bytes available for the Work Buffer (see Figure 2-13).

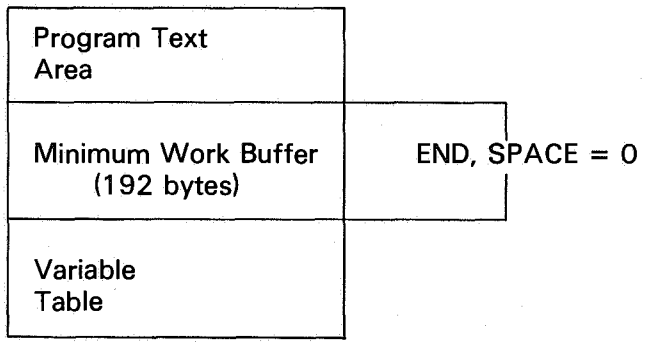


Figure 2-13.
"Memory Full" Condition (END, SPACE=0), Showing Reserved Minimum 192 Bytes for Work Buffer

The reason for always reserving a minimum space for the Work Buffer may be obvious. If the system did not protect such an area, it would be possible for the programmer to fill memory so completely without getting an error that it would be impossible to run the program (since there would not even be enough space to enter a RUN command). Because the program could not be saved on disk (no room to enter a SAVE command), the programmer's only alternative would be to clear memory by Master Initializing the system and start all over. By ensuring that there is always space available to enter a command or Immediate Mode statement, even when memory is legally "full," the system protects the programmer against such a calamity.

The Meaning of "Negative" Free Space (SPACE Function)

Although the system ensures that a minimum of 192 bytes always remain unoccupied by program text or variables in memory, it does permit the Value Stack to utilize a portion of this minimum buffer area. Up to 128 bytes of the 192-byte minimum Work Buffer can be used by the Value Stack. A program can, therefore, be run even when memory is legally "full" since additions to the Value Stack during execution can overlap into the reserved buffer area. Note that in this case the SPACE function would return a negative free space value. To understand why this is so, consider Figure 2-14:

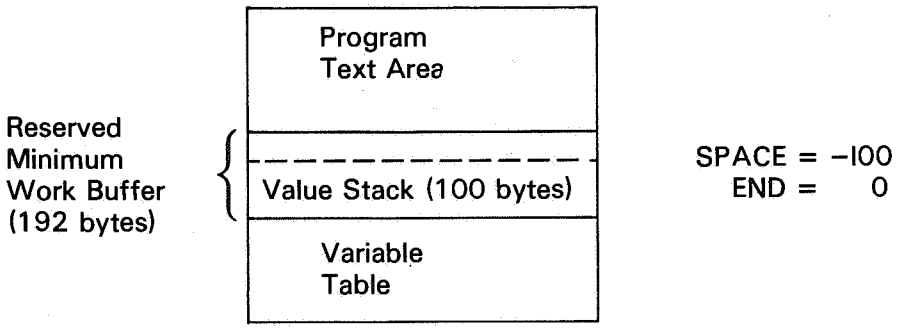


Figure 2-14.
"Negative" Free Space

Before free space is computed by SPACE or END, 192 bytes are subtracted from the available space for the Work Buffer. Thus, when SPACE and END return free space values of zero, there are still 192 bytes available, part of which can be used by the Value Stack. When memory is so fully packed that the Value Stack must occupy part of the minimum buffer area, its size is subtracted from zero by SPACE, yielding a negative free space figure. Thus, a free space value of −100 returned by SPACE indicates that memory is legally "full," but 100 bytes of the reserved minimum buffer have been used by the stack. Since a maximum of 128 bytes of the minimum buffer area can be used by the stack, SPACE cannot return a value less than −128. When the Value Stack requires more than 128 bytes of the buffer, a Memory Overflow error is signalled.

Memory Overflow Errors (ERR A01, A02)

Memory overflow occurs when all available free space in memory is exhausted. This situation can arise either when (a) there are fewer than 192 bytes between the bottom of the program text area and the top of the Variable Table or (b) there are fewer than 64 bytes between the bottom of the text area and the top of the Value Stack. Situation (a) is signalled by ERR A01, and situation (b) by ERR A02.

ERR A01

ERR A01 is signalled when the Variable Table approaches within 192 bytes of the text area. The system reserves a minimum Work Buffer between the Variable Table and the text area of 192 bytes; when the addition of a new program line or variable threatens to encroach upon this area, ERR A01 is signalled, and the new variable is not entered or the new line is not threaded into the text.

In general, ERR A01 does *not* cause any resident program text or variables to be destroyed. An exception to this general rule is the situation in which an existing line is replaced by a new, longer line. In this case, ERR A01 can occur after the original line has been deleted, but before the new line is inserted in the text area (assuming the new line caused the overflow), with the result that the original line is lost.

ERR A01 generally calls for serious action on the part of the programmer. Existing program text or variables must be cleared to make room for the new entry by executing a CLEAR P, CLEAR N, or CLEAR V command, or the program must be saved on disk and cleared from memory. It is possible to recall the newly entered program line which caused ERR A01 and either shorten and reenter it or assign it the line-number of an existing (unwanted) line in memory and enter it in place of the unwanted line. Note, however, that a newly entered program line is *not* threaded into the program text

following ERR A01 and is, therefore, regarded as an Immediate Mode line rather than a program line for purposes of RECALL. Entering the line-number and depressing EDIT, RECALL will not recall the new line; it must be recalled as an Immediate Mode line (simply depress EDIT and then RECALL).

ERR A02

ERR A02 is signalled when the Value Stack threatens to occupy more than 128 of the 192 bytes reserved for the Work Buffer (i.e., when there are fewer than 64 bytes remaining between the top of the Value Stack and the bottom of the text area). This error typically occurs during program execution when memory is nearly or completely filled with program text and variables. In such a situation, the Value Stack may expand into the reserved minimum buffer area; if it threatens to occupy more than 128 bytes of the reserved minimum area, ERR A02 is signalled, and the Value Stack and Operator Stack are flushed. Recovery from ERR A02 requires careful analysis of the program. One possible course of action is to delete some program text or variables in order to provide more space for expansion of the stack. Often, however, the problem is one of program design. The program may either attempt to nest too many levels of subroutines or loops or may repeatedly branch out of subroutines or loops without executing appropriate RETURN or NEXT statements. In these cases, adding more space for the Value Stack will simply allow the Operator Stack to overflow first, producing an ERR A04. The program logic must be modified, perhaps by using RETURN CLEAR statements at critical points to clear information from the stacks.

Note that, with the exception of line replacement, memory overflow does not cause the destruction of any information resident in memory. In addition, the system ensures that there is always enough space in memory for the programmer to enter an Immediate Mode statement or system command (such as CLEAR or SAVE) which will permit him to correct the overflow condition by modifying or saving the resident program without having to Master Initialize the system.

2.6 THE SPACEK FUNCTION

The SPACEK function returns the total user memory size, divided by 1,024. On the 2200VP this is the total system memory size, including the system overhead. (Thus, for a 32K 2200VP, SPACEK=32.)

Before a 2200MVP is partitioned (see Chapter 16, section 16.2), SPACEK returns the total amount of memory available for partitioning. On a 32K 2200MVP, for example, SPACEK returns a value of 29 since 3K is devoted to system overhead. After partitioning, SPACEK is equal to the partition size (a 16K partition would return SPACEK=16).

CHAPTER 3 EDITING AND DEBUGGING FEATURES

3.1 INTRODUCTION

Both the BASIC-2 language and the 2200VP and 2200MVP CPU's offer a variety of useful editing and debugging features. The powerful editing capabilities of the system enable the operator to edit program lines, Immediate Mode lines, and input data values both during and after entry. In addition, the extensive debugging features of the BASIC-2 language facilitate the tasks of identifying and isolating bugs in a BASIC-2 program. This chapter describes the complete range of editing and debugging features available on the 2200VP and 2200MVP.

In order to enter and/or edit a line of text, the operator must be able to key in information from the keyboard. There are two basic modes of system operation in which control is passed to the keyboard: Text Entry Mode and Edit Mode. In the third mode of operation, Program Mode, the system is under program control, and the keyboard (except for the HALT/STEP and RESET Keys) is locked out.

3.2 EDIT FEATURES — TEXT ENTRY MODE

The system is in Text Entry Mode during all Console Input operations, which include the entry of program lines, Immediate Mode lines, or system commands; the system is awaiting Console Input whenever the colon (:) is displayed at the beginning of a line. The system also enters Text Entry Mode when data is requested for an INPUT statement (in which case a question mark (?) is displayed indicating a request for operator entry) or when a DEFFN' subroutine is accessed from the keyboard via a Special Function Key. In all of these cases, the operator is free to enter any keyboard characters.

As characters are entered during Text Entry Mode, they are displayed on the CRT and simultaneously placed in a temporary storage area in memory called the "Work Buffer." When the text line has been entered, RETURN must be keyed to terminate the entry operation and pass control to the system, which then processes the line. If the entered line is a numbered program line, it becomes part of the resident program, but is not executed; if it is an Immediate Mode line or system command, it is immediately executed and saved for possible recall. If the text line consists of one or more data values entered in response to an INPUT request or as arguments passed to a DEFFN' subroutine, the individual values are assigned to receiving variables in the INPUT or DEFFN' variable list.

In Text Entry Mode, the system provides some limited capabilities for editing a text line which is currently being entered (i.e., before RETURN has been keyed). Once RETURN has been keyed, the line must be recalled and edited in Edit Mode. Edit Mode is described in the next section. The edit features of Text Entry Mode, covered in this section, include the following:

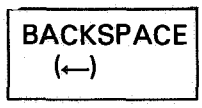
- Character deletion (BACKSPACE).
- Line erasure (LINE ERASE).
- Program line deletion.
- Program line replacement.

Character Deletion: BACKSPACE

A character in the text line currently being entered can be deleted by depressing the BACKSPACE Key. Each depression of the BACKSPACE Key moves the CRT cursor one position to the left, blanks out the character at that position on the CRT, and deletes the character from the Work Buffer. For example, suppose the operator incorrectly types a character while entering the following line:

```
:100 PRINT "ENTER Y OR N]_
```

In this line, a right bracket (]) has been erroneously entered instead of a double quote ("). One depression of the BACKSPACE Key will backspace the cursor one position and delete the "]" character:




```
:100 PRINT "ENTER Y OR N_
```

The operator can now enter the correct character (double quote).

Line Erasure: LINE ERASE

The LINE ERASE Key can be used to erase the entire line or a portion of the line currently being entered. For example, suppose the following text has been erroneously entered:

```
:10 PRINT "ENTER Y OR N_
```

Depressing LINE ERASE (labelled ERASE on some keyboards) clears this line from the CRT and deletes it from the input buffer:


```
:_
```

Deleting a Program Line from Memory

A program line stored in memory can be deleted by keying in the line-number followed immediately by RETURN. For example, the entry

```
:20 RETURN
```

deletes line 20 from memory.

Replacing a Program Line in Memory

A program line in memory can be replaced with a new line by assigning the line-number of the original line to the new line. When the new line is stored, it automatically replaces the original line in memory. For example, suppose the following lines are in memory:

```
10 INPUT A,B
20 PRINT SQR(A↑2 + B↑2)
:_
```

Entering a new line with line-number 20 causes the new line to replace the existing line 20:

```
:20 PRINT LOG(A) — LOG(B) (RETURN)
```

3.3 EDIT FEATURES — EDIT MODE

The system's most powerful editing capabilities are accessible to the operator in Edit Mode. There are two means of placing the system in Edit Mode:

- By depressing the EDIT Key, located at the top of the keyboard to the right of the Special Function Keys.
- By execution of a LINPUT statement.

When the 2200VP enters Edit Mode, it signals this fact by displaying an asterisk (*). The 2200MVP signals Edit Mode by a blinking cursor. In subsequent examples in this manual, Edit Mode is indicated by the asterisk even though this symbol is not displayed on the MVP. The system can be removed from Edit Mode in the following ways:

- By depressing the EDIT Key a second time.
- By keying RETURN.
- By keying LINE ERASE (*unless* Edit Mode was entered via a LINPUT statement).

When the 2200VP leaves Edit Mode, the asterisk is changed to a colon (:) for Console Input operations, to a question mark (?) for INPUT, or to a space for LINPUT. When the 2200MVP leaves Edit Mode, the cursor will no longer blink. A second depression of the EDIT Key simply returns the system to the Text Entry Mode without executing the edited line (it remains in the Work Buffer). Keying RETURN terminates the editing operation and passes control to the system, which then processes the line. Because the EDIT Key can be used to enter and leave Edit Mode without passing control to the system, the operator is free to reenter Text Entry Mode from Edit Mode after a line has been edited and key in additional text prior to keying RETURN. Edit Mode supports the following types of editing operations:

- Editing a numbered program line during entry.
- Recalling a program line from memory and editing it.
- Editing an Immediate Mode line or system command during entry.
- Recalling from memory and editing the last-executed Immediate Mode line or system command.
- Editing a text line consisting of data values entered in response to an INPUT or LINPUT request or keyed in following depression of a Special Function Key to serve as argument(s) for a DEFFN' subroutine.
- Recalling and editing the last-entered data text line.
- Concatenating two or more program lines.

While the system is in Edit Mode, Special Function Keys '04-'15 perform specific editing functions. The remaining Special Function Keys are disabled in Edit Mode.

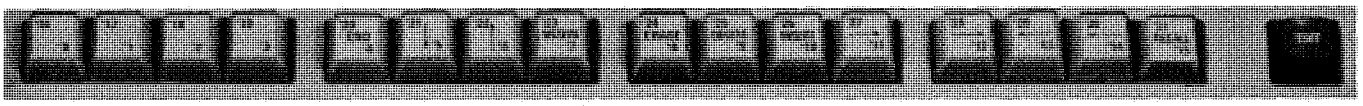


Figure 3-1.
Portion of the Special Function Strip Showing Edit Keys
and Special Function Keys Used in Edit Mode

The special Edit Keys have the following significance in Edit Mode:

Table 3-1.
Edit Mode Keys

RECALL	Recalls the specified program line, the last-executed Immediate Mode line, or the variable value from memory for editing.
←	Moves cursor one position to the left.
→	Moves cursor one position to the right.
< - - - - -	Moves cursor five positions to the left.
- - - - - >	Moves cursor five positions to the right.
-INSERT-	Makes space for insertion of one character in text line.
-DELETE-	Deletes one character from text line.
-ERASE-	Erases the remaining characters in the line, starting at the current cursor position.
-BEGIN-	Moves cursor to beginning of current text line.
-END-	Moves cursor to end of current text line.
↑	Moves cursor up to the previous CRT line. (Only if current text line occupies more than one line on the CRT.)
↓	Moves cursor down to the next CRT line. (Only if current text line occupies more than one line on the CRT.)

Editing Program Lines, Immediate Mode Lines, and Data Values upon Entry

If an error is made in the course of keying in a numbered program line, an Immediate Mode line, or a line of data values, the operator can switch immediately to Edit Mode before executing the line and make the necessary changes. (See the following subsections on cursor positioning, character deletion, and character insertion.) When all the necessary corrections have been made, the operator can return to Text Entry Mode by depressing EDIT again or execute the edited line by keying RETURN.

Recalling a Stored Program Line from Memory with RECALL

Before it can be edited, a text line must be placed in the Work Buffer. If an error is discovered during entry before RETURN has been keyed, the line is present in the Work Buffer and can be edited directly. If an error is discovered after a program line has been stored in memory, however, the line must be recalled to the Work Buffer before it can be edited. This is accomplished by entering the line-number of the erroneous line, depressing EDIT, and then depressing the RECALL Key. RECALL fetches the specified line from memory into the Work Buffer and simultaneously displays it on the CRT screen for editing by the operator.

It is important to note that all changes made by the operator are made to the copy of the program line which is recalled to the Work Buffer. No changes are actually made to the program in memory until the operator keys RETURN. When RETURN is keyed, the edited line replaces the original line in memory.

Recalling an Executed Immediate Mode Line with RECALL

Immediate Mode lines and system commands, unlike program lines, are not permanently stored in memory. The system does, however, temporarily store an Immediate Mode line in a special section of memory following execution. The same area of memory is reused for each successive Immediate Mode line; an Immediate Mode line therefore remains in memory only until a new line is executed.

The last-executed Immediate Mode line can be recalled to the Work Buffer for editing by depressing EDIT and then the RECALL Key with no line-number. RECALL fetches the Immediate Mode line which currently is stored in the temporary storage area in memory, places it in the Work Buffer, and simultaneously displays it on the CRT screen. The line may then be edited and reexecuted.

The temporary storage area in memory used for Immediate Mode lines has a fixed length of 256 bytes. Thus, only the first 256 bytes of an Immediate Mode line can be stored. In the unlikely event an Immediate Mode line exceeds 256 bytes, the additional bytes are lost when the line is executed and cannot be recalled and edited.

Note that the Work Buffer must be clear in order to recall an Immediate Mode line. If one or more characters have been keyed into the buffer after executing an Immediate Mode line, the line cannot be recalled. (If a line-number is in the buffer when RECALL is depressed, the specified line is recalled; if any other characters are in the buffer, nothing is recalled.) Characters inadvertently keyed in after the Immediate Mode line is executed must be cleared from the buffer by keying LINE ERASE prior to depressing RECALL. Once RETURN is keyed, the current contents of the Work Buffer are saved, and the previous Immediate Mode line stored in memory is destroyed. If the displayed line is blank when RETURN is keyed, a line of blanks is saved in memory from the Work Buffer.

It should be noted that the system commands LOAD and CLEAR cannot generally be recalled for editing. In addition, the following conditions can cause an Immediate Mode line to be partially or completely destroyed before it can be recalled:

- RESET.
- Initiation of program execution (RUN, HALT/STEP, or Special Function Key).
- Entry of a program line.
- Execution of a new Immediate Mode line.

Recalling Entered Data with RECALL

Data entered in response to an INPUT request or as arguments for a DEFFN' subroutine accessed from the keyboard can be recalled to the input buffer for editing with RECALL. A line of data is stored in the same temporary storage area used for Immediate Mode lines and can be recalled by depressing RECALL with no line-number (the display must be clear).

Cursor Movement Keys (←,→, <- - - - -, - - - - - >, ↑,↓, END, BEGIN)

The eight cursor movement keys are used to position the CRT cursor to a desired point in the text line being edited. The functions of these keys are described in Table 3-1. It should be emphasized, however, that the cursor movement keys have no effect on data in the Work Buffer; they merely move the cursor around in the Work Buffer, enabling the operator to insert or delete characters at any point.

BACKSPACE (< -) and LINE ERASE Keys in Edit Mode

In Edit Mode, the BACKSPACE (<-) and LINE ERASE Keys function exactly as they do in Text Entry Mode. That is, BACKSPACE moves the cursor one space to the left and deletes the character at that position from the Work Buffer, while LINE ERASE erases the entire contents of the Work Buffer. The functions of these keys should, therefore, be carefully distinguished from those of the apparently similar Edit Keys and ERASE. The BACKSPACE Key moves the cursor one position to the left, but does *not* delete any characters from the Work Buffer. The ERASE Key erases only that portion of the text line from the cursor position to the end of the line.

Deleting Characters from a Text Line: DELETE and ERASE

Characters can be deleted from a text line in Edit Mode with the DELETE and ERASE Keys. Each time the DELETE Key is depressed, it deletes the character at the current cursor position and moves the remainder of the line one position to the left to fill the space vacated by the deleted character. For example, given the erroneous line

```
*20 PRINT A$(2)+B
```

depressing the DELETE Key once produces

```
*20 PRINT A(2)+B
```

The ERASE Key erases all characters from the cursor position to the end of the line. For example, given the line

```
*20 PRINT A$(2)+B
```

depressing the ERASE Key produces

```
*20 PRINT A_
```

Inserting Characters in a Text Line: INSERT

Characters can be inserted into a text line in Edit Mode with the INSERT Key. Each depression of the INSERT Key inserts one space into the text line at the current cursor position. The remainder of the line is shifted one position to the right to accommodate the space. One space must be inserted for each character to be inserted into the line. When enough spaces have been inserted to make room for the new characters, the new characters must then be keyed in, starting at the cursor position.

For example, given the line

```
*20 PRINT A1)+B
```

depressing the INSERT Key once produces

```
*20 PRINT A_1)+B
```

The operator can now correct the line by keying in a left parenthesis [(]:

```
*20 PRINT A(1)+B
```

The corrected line can be saved in memory by keying RETURN. RETURN also returns the system to Text Entry Mode.

Multiple-Character Insertion (VP Only)

If the INSERT Key is not used to insert spaces in a line, characters keyed into a line while the system is in Edit Mode simply replace existing characters in the line. The use of INSERT is somewhat inconvenient for multiple-character insertions, however, since it requires that INSERT be repeatedly depressed to insert spaces before the characters themselves can be entered. An alternative method (on the VP) of inserting characters in a text line involves switching from Edit Mode into Text Entry Mode with the EDIT Key and inserting the new characters in Text Entry Mode. The procedure is as follows:

1. Depress the EDIT Key once, placing the system in Edit Mode.
2. Recall the line (if necessary), and position the cursor to the point at which character insertion is to begin.

3. Depress the EDIT Key a second time, returning the system to the Text Entry Mode. The asterisk (*) is replaced either by a colon (:), a question mark (?), or a blank.

4. Key in the characters to be inserted. As each character is entered, the text line automatically is shifted one space to the right to accommodate the new character. Existing characters in the line are not overwritten.

A character string defined with a Special Function Key also can be inserted in a text line with the technique described above. The cursor must be positioned at the insertion point, and the EDIT Key depressed to return the system to Text Entry Mode. In Text Entry Mode, the Special Function Keys regain their user-defined meanings; thus, depressing a Special Function Key which has been defined for text entry causes its associated character string to be inserted in the text line starting at the current cursor position. The text line is automatically shifted right to accommodate the entire character string, and no characters in the line are overwritten. For example, suppose the following definition is made:

```
:10 DEFFN' 10 "HEXOF("
```

Assume further that the cursor is positioned (in Edit Mode) to the specified position in the program line:

```
*100 PRINT A$)
```

Depressing the Edit Key, followed by Special Function Key '10 produces the following:

```
*100 PRINT HEXOF(A$)
```

Changing a Program Line-Number

The line-number of a program line can be changed in Edit Mode by recalling the line, moving the cursor to the beginning of the line with the BEGIN Key, and then editing the line-number. It must be emphasized, however, that when the edited line is saved in memory, it is saved as a new line (since it has a new line-number). The original line, with the original line-number, is not automatically deleted from memory. For example, if line 20 is recalled, its line-number changed to 30, and then resaved, both lines 20 and 30 (identical except for their different line-numbers) are stored in memory. The original line (20) can be removed only by deleting it (key 20 followed by RETURN) or replacing it with a new line 20.

Concatenating Program Lines

Two or more program lines in memory can be combined (concatenated) into a single program line in Edit Mode. The procedure is as follows:

1. Recall the first program line to be concatenated into the Work Buffer by depressing EDIT, entering the line-number, and depressing RECALL.

2. At the end of the first program line, key in a colon (:), followed by the line-number of the program line which is to be concatenated to the currently displayed line.

3. Depress RECALL. The specified line is recalled from memory and concatenated to the line currently in the Work Buffer. Note that, up to this point, nothing has been changed in memory. This procedure may be repeated to concatenate multiple lines into a single, complex line.

4. Store the new line in memory by keying RETURN. The new line replaces only the first recalled line in memory (since it has the line-number of the first recalled line). The subsequent program lines recalled and concatenated to the original line are not cleared from memory when the new line is saved. Each must be cleared by entering its line-number and keying RETURN.

Assume, for example, that the following two lines are in memory:

```
10 INPUT A,B
20 PRINT A*SQR(B)
```

Line 20 can be concatenated with line 10 by recalling line 10 in Edit Mode and entering ":20" at the end:

```
*10 INPUT A,B :20_
```

Depressing RECALL at this point concatenates line 20 with line 10:

```
RECALL

*10 INPUT A,B :PRINT A*SQR(B)_
```

When RETURN is keyed, the new extended version of line 10 replaces the original line 10 in memory. Line 20, however, is not affected. A listing discloses this fact:

```
:LIST
10 INPUT A,B :PRINT A*SQR(B)
20 PRINT A*SQR(B)
:_
```

Line 20 must be deleted in the normal fashion.

3.4 PROGRAM DEBUGGING FEATURES

The process of locating and identifying errors in a BASIC-2 program often involves careful analysis of the program's performance at critical stages in program execution. To aid the programmer in this task, the system provides a variety of debugging tools. They include:

- The STOP statement and the CONTINUE command.
- The HALT/STEP Key.
- The TRACE statement.
- Special forms of the LIST command.
- The RENUMBER command.

In this chapter, the functions of the above instructions and the HALT/STEP Key are briefly sketched. More detailed discussion of each instruction is provided under the general form of the instruction in a separate chapter.

Stopping and Resuming Program Execution: STOP and CONTINUE

Frequently in debugging a program it is advantageous to stop program execution at a particular point to permit the operator to examine and modify the values of critical variables. One way in which this can be done is with the STOP statement. Execution of a STOP statement causes the system to suspend program execution and output the word STOP, optionally followed by a user-supplied message and/or the line-number of the STOP statement. The operator is then free to examine the current values of variables with an Immediate Mode PRINT statement and modify them to observe the effect on subsequent processing. The number of STOP statements which may be used in a program

is not restricted. Strategic placement of STOP statements will enable the operator to monitor the program's performance at critical points.

Program execution can be resumed following a STOP statement by keying CONTINUE (RETURN). CONTINUE restarts program execution at the statement immediately following STOP, provided modifications have not been made to the program by the operator.

The STOP statement is described in Chapter 11, section 11.2, and the CONTINUE command is covered in Chapter 10, section 10.2.

Halting and Stepping Through a Program: HALT/STEP

The HALT/STEP Key has, as its name implies, a dual function: to temporarily halt program execution and to step through a program, executing one statement at a time. Keying HALT/STEP once causes program execution to halt at the completion of the currently executing statement. The operator is then free to examine the values of critical variables with an Immediate Mode PRINT statement and, if necessary, to modify the values of variables. Normal program execution can then be resumed by keying CONTINUE (RETURN).

Alternatively, the operator can step through program execution by keying HALT/STEP a second time. Each time HALT/STEP is keyed, it causes the next statement to be listed and executed and then halts execution. It is frequently helpful to step through a program in this manner, observing the action taken by the program following execution of each statement. HALT/STEP becomes an even more powerful analytical tool if it is used to step through a program in Trace Mode (see the next paragraph).

The HALT/STEP Key is further discussed in Chapter 10, section 10.2.

Tracing Through Program Execution: TRACE

Certain critical internal program operations, including variable assignments and program branches, can be monitored by the programmer if the program is executed in Trace Mode. In Trace Mode, the system automatically outputs the variable name and value each time a new assignment is made and the line-number to which execution is transferred each time a branch is made. Trace Mode is turned on by executing a TRACE statement and turned off by executing a TRACE OFF statement. (CLEAR and RESET also turn off Trace Mode.)

The programmer can obtain a more comprehensive picture of what is happening in the program by stepping through program execution with HALT/STEP in Trace Mode. A printed copy of the Trace output can be produced by selecting the printer for Console Output operations prior to executing Trace. If Trace output is displayed on the CRT, it can be slowed to a readable rate by invoking a pause with a SELECT P statement. Device selection for Console Output and pause selection are explained in Chapter 7, section 7.3.

The TRACE command is covered in Chapter 10, section 10.2.

Listing and Cross-Referencing a Program: LIST

The LIST command has a variety of forms which provide the programmer with an arsenal of listing and cross-referencing capabilities:

- LIST produces a listing of all or selected portions of program text in memory.
- LIST D produces a "decompressed" program listing, with multiple-statement lines broken up for readability so that each statement is printed on a separate line.
- LIST # produces a cross-reference listing of program line-numbers referenced in the program.
- LIST V produces a cross-reference listing of variables referenced in the program.
- LIST ' produces a cross-reference listing of marked subroutines (DEFFN') defined and referenced in the program.

- LIST T produces a cross-reference listing of all program lines containing a specified text string.
- Two other special forms of LIST are also available: LIST DT lists the contents of the Device Table, and LIST I lists the contents of the Interrupt Table in memory. These commands are discussed in Chapters 7, section 7.9 and 8, section 8.10, respectively.

The LIST command is discussed in Chapter 10, section 10.2.

Renumbering Program Lines: RENUMBER

If it is necessary to make room for new lines to be inserted into a program, the program can be renumbered with the RENUMBER command. RENUMBER permits the programmer to specify the increment between successive line-numbers. In this way, adequate space can be left between successive lines to allow for the insertion of as many new lines as needed. RENUMBER also automatically changes all program references to line-numbers (in GOTO, GOSUB, IF...THEN, etc., statements) in accordance with the new numbering scheme.

The RENUMBER command is covered in Chapter 10, section 10.2.

CHAPTER 4 NUMERIC OPERATIONS

4.1 INTRODUCTION

In BASIC-2, a distinction is made between numeric data and alphanumeric data. The two types of data are stored in different types of variables. Only numeric data can participate in arithmetic operations, although the system does provide some limited binary and packed decimal arithmetic operations which are performed on data in alphanumeric format. Alphanumeric data is discussed in Chapter 5; the binary and packed decimal math features are covered in Chapter 6. Numeric operations are the subject of the present chapter.

4.2 NUMERIC VALUES

Numeric values are stored in a special floating-point format containing 13 digits, a sign, and a signed two-digit integer exponent. (The internal numeric format is described in greater detail in Chapter 2, section 2.3.) A numeric value can be stored in memory in the form of a constant or as the value of a numeric-variable; it may be entered by the operator or produced as the result of evaluating a numeric expression. In any case, the legal range of numeric values which can be handled by the system is shown below:

$$-10^{100} < \text{value} <= -10^{-99}, 0, 10^{-99} <= \text{value} < 10^{100}$$

Numeric values entered from the keyboard may be specified in either fixed-point or exponential format. The rules for entering legal numeric values in each format are as follows:

- Fixed-Point Format — A numeric value entered in fixed-point format may contain a maximum of 13 digits, sign, and decimal point. Embedded spaces in the value are ignored. The sign of the value must precede the digits; unsigned values are treated as positive numbers. If no decimal point is entered, it is assumed to be at the right of the last digit. Examples of legal numeric entries in fixed-point format appear below:

  12.003
  +143000
  -.00725

- Exponential Format — A numeric value entered in exponential format may contain a maximum of 13 digits, sign, decimal point, and a signed one- or two-digit exponent which is denoted by the letter "E". When exponential format is used, the value is equal to the number entered times 10 to the power of the exponent. The number must conform to the rules for fixed-point values (see above). The exponent must be an integer value; it may contain a maximum of two digits and a sign and must be preceded by the letter "E". If a sign is specified, it must precede the exponential digits. An unsigned exponent is

regarded as positive. Examples of legal numeric entries in exponential format appear below:

```
4.56E5
-125021 E+10
+23.005E-07
-1.026 E-85
```

If more than 13 digits are entered for a numeric value, the system signals an error and the number is rejected. Leading zeros before the decimal point are ignored.

4.3 NUMERIC CONSTANTS

A constant is a legal numeric value which appears in a BASIC statement. Constants must conform to the format rules for legal numeric entries detailed in the previous section. They must not be enclosed in quotation marks since values enclosed in quotes are treated as alphanumeric character strings rather than numeric values. Constants are simple expressions and may be used wherever numeric expressions are permitted. The value of a constant is not altered during program execution. Some examples of constants appear below:

```
10 N = 1.256
20 PRINT 165    *N
30 K = 5.7001E-03
```

4.4 NUMERIC-VARIABLES

Numeric-variables are used to store numeric data in memory. Unlike constants, whose values are fixed and cannot be changed during program execution, variables can be assigned new values during program execution by a variety of different statements.

Because numeric values are stored in a special floating-point format, they require a special class of variables different from those used to store alphanumeric data. Numeric- variables are of two types: scalar and array. A numeric-scalar- variable can be used to store a single numeric value; it is designated by a letter (A-Z) or a letter followed by a digit (0-9). There are a total of 286 legal numeric-scalar- variable-names. Examples of legal numeric-scalar-variable-names are:

```
A, N, B1, F9
```

A numeric-array-variable consists of a group of array elements, identified by a single array name. Each array element can be assigned a single numeric value; thus, an array can be used to store and process multiple numeric values. The names used for numeric-array-variables are the same as those used for numeric-scalar-variables, that is, a letter (A-Z) or a letter followed by a digit (0-9). Array-variables may be one- dimensional (single-subscripted) or two-dimensional (double- subscripted); a particular element in an array is identified by specifying its subscript(s) in parentheses following the array name. Examples of legal numeric-array-variable-names are:

```
A(3), N(1,2), F7(6), B1(9,9)
```

A scalar-variable and an array-variable may both have the same name since they are independent variables, but a one-dimensional array-variable and a two-dimensional array- variable may not have the same name in the same program.

It is necessary to reserve space in memory for a numeric-array-variable before any data can be assigned to it. A DIM or COM statement must be used for this purpose. For example, the statement

```
20 DIM N(10), Q1(5,5)
```

reserves space for the one-dimensional array-variable N, containing 10 elements, and for the two-dimensional variable Q1, containing 25 elements. The numbers in parentheses in this case refer not to specific elements, but to the total number of elements in the array. These numbers are called the array

dimensions. In general, the following rules apply when reserving space for array-variables:

1. The numbering of array elements starts at one rather than zero; thus, a subscript of zero, as in

   A(0)

   is not permitted.

2. The single dimension of a one-dimensional array may not exceed 65535, and each dimension of a two-dimensional array may not exceed 255. (Memory size will impose a more severe restriction.)

4.5 NUMERIC EXPRESSIONS

A numeric expression may consist of a single variable or constant, or it may consist of a series of variables and constants separated by arithmetic operators and numeric functions. Numeric expressions can be evaluated in a variety of different BASIC statements. Most commonly, expressions are evaluated and their values assigned to variables in assignment (LET) statements, or they are evaluated and their values printed or displayed in PRINT statements. Some examples of numeric expressions are:

```
10 A =[B]
20 PRINT [4*A+B]
30 N =[N+SQR(A+B↑2)]
40 B1,B2,B3 =[50]
50 ON [3*J-1] GOTO 100, 200, 300
```

4.6 ARITHMETIC OPERATORS

The following arithmetic symbols or "operators" are used to perform mathematical operations in BASIC:

+	Addition ("A+B" means "Add B to A")
−	Subtraction ("A-B" means "Subtract B from A")
*	Multiplication ("A*B" means "Multiply A by B")
/	Division ("A/B" means "Divide A by B")
↑	Exponentiation ("A↑B" means "Raise A to the power of B")

Order of Evaluation

Expressions are evaluated from left to right. For example, the expression A+B−C is evaluated by adding B to A and then subtracting C from the sum. When different types of operators are used in an expression, the following priorities in evaluation are observed:

First	—Exponentiation (↑) is performed from left to right.
Second	—Multiplication and division (*,/) are performed from left to right.
Third	—Addition, subtraction, and negation (+,−) are performed from left to right.

Altering the Normal Order of Evaluation with Parentheses

The normal order in which an expression is evaluated can be altered through the use of parentheses. When parentheses are included in an expression, the portion of the expression enclosed within parentheses is always evaluated first. Note the different results obtained in evaluating the following two expressions, identical except for parentheses:

	Expression	Result
a)	PRINT 5*2↑2	Raise 2 to the 2nd power (=4) and multiply the result by 5. Answer: 20.
b)	PRINT (5*2)↑2	Multiply 5 by 2 (=10) and raise the result to the 2nd power. Answer: 100.

In constructing expressions, parentheses may be nested within parentheses; there is no practical limit to the number of pairs of parentheses used in this manner. The portion of the expression enclosed within the innermost set of parentheses always is evaluated first.

A BASIC expression may not contain two consecutive arithmetic operators. For example, the statement

```
20 PRINT A↑−J
```

violates BASIC syntax and is illegal. Parentheses must be used to separate the operators, thus:

```
20 PRINT A ↑ (−J)
```

A case in which the use of parentheses can be particularly critical is that of raising a negative number to an even power. For example, the statement

```
30 PRINT −3↑2
```

yields the result −9 because the exponentiation is performed prior to the negation. In order to raise -3 to the second power, the statement must be modified by using parentheses:

```
30 PRINT (−3)↑2
```

In this case, the unary negation is performed first and then the exponentiation, yielding the desired result (9).

4.7 ROUND/TRUNCATE OPTION

The results of all arithmetic operations (+, −, *, /, ↑) as well as the square root (SQR) and modulo (MOD) functions normally are rounded to 13 significant digits. Alternatively, results can be truncated to 13 significant digits by executing the statement

```
SELECT NO ROUND
```

Once this statement is executed, all results are truncated to 13 digits. Rounding can be restored subsequently with the statement

```
SELECT ROUND
```

Rounding is selected automatically when the system is Master Initialized.

4.8 SYSTEM-DEFINED NUMERIC FUNCTIONS

The system provides a variety of built-in trigonometric and mathematical functions. They are summarized in Table 4-1 below; those which require some elucidation are discussed further on the following pages.

Note that trigonometric functions normally are evaluated in radians unless the system is explicitly instructed to use degrees or grads. (See Table 4-1 and the discussion of the trig functions in section 4.9 for further details.)

Table 4-1.
System-Defined Numeric Functions

Function	Meaning	Examples
INT(x)	Finds the greatest integer value of the expression.	INT(13.5)=13 INT(−5.2)=−6
FIX(x)	Finds the integer portion of the expression.	FIX(13.5)=13 FIX(−5.2)=−5
ABS(x)	Finds the absolute value of the expression.	ABS(7↑2)=49 ABS(−6.536)=6.536
SGN(x)	Returns the value 1 if the expression is positive, −1 if negative, and 0 if zero.	SGN(8.15)=1 SGN(−.123)=−1 SGN(0)=0
MOD(x,y)	Finds the remainder of x/y.	MOD(8,3)=2 MOD(4.2,4)=.2
ROUND(x,n)	Finds the value of x rounded to the nth decimal place if n>0, rounded to the nearest integer if n=0, rounded −(n)+1 places to the left of the decimal point if n<0.	ROUND(1.234,2)=1.23 ROUND(5.06,1)=5.1 ROUND(5.5,0)=6 ROUND(−3.8,0)=−4 ROUND(−3.2,0)=−3 ROUND(1256,−2)=1300 ROUND(1256,−1)=1260
RND(x)	Produces a random number between 0 and 1.	RND(1)=.8392246561935
SQR(x)	Finds the square root of the expression.	SQR(18+7)=5 SQR(25)=5
MAX(x,y,...,z)	Finds the maximum value among the specified expressions or numeric array(s).	MAX(1,3,2)=3
MIN(x,y,...,z)	Finds the minimum value among the specified expressions or numeric array(s).	MIN(−6,−3,−1)=−6

Table 4-1 (Cont'd)
System-Defined Numeric Functions

Function	Meaning	Examples
LGT(x)	Finds the common logarithm (log base 10) of the expression.	LGT(1000)=3
LOG(x)	Finds the natural logarithm (log base e) of the expression.	LOG(3052)=8.023552392404
EXP(x)	Finds the value of e raised to the power of the expression (i.e., the natural antilog of expression).	EXP(.34*(5-6))= .7117703227626 EXP(1)=2.718281828459
#PI	Assigns the value 3.14159265359 in place of the expression.	4*#PI=12.56637061436
SIN(x)	Finds the sine of the expression.	SIN(#PI /3)= .8660254037847
COS(x)	Finds the cosine of the expression.	COS(.693↑2)= .8868799122688
TAN(x)	Finds the tangent of the expression.	TAN(10)= .6483608274591
ARCSIN(x)	Finds the arcsine of the expression.	ARCSIN(.003)= 3.00000450E-03
ARCCOS(x)	Finds the arccosine of the expression.	ARCCOS(.587)= .9434480794406
ARCTAN(x) or ATN(x)	Finds the arctangent of the expression.	ARCTAN(3.3)= 1.276561761684

4.9 NOTES ON THE NUMERIC FUNCTIONS

Most of the functions described in Table 4-1 are common mathematical or trigonometric functions and require no further explanation. A few which do warrant some additional discussion are covered in the following paragraphs.

INT, FIX Functions

The INT function is the "greatest integer" function, often depicted in ordinary algebra with square brackets (e.g., [5.6], [−3]). For integer values, the INT of a value is identical to the original value (e.g., INT (4)=4, INT(-3)=-3). For noninteger values, INT returns the greatest integer which is less than the value. Some examples of the INT function follow:

```
INT(3.8)      =  3
INT(6.1)      =  6
INT(−2.6)     = − 3
INT(−2.1)     = − 3
INT(30)       =  30
INT(−30)      = −30
```

The FIX function returns the integer portion of a value. For positive values, FIX operates exactly like INT, truncating the decimal portion and returning the integer. For negative numbers, FIX operates on the absolute value of the number, truncating the decimal portion and returning the integer. The true sign of the number is restored following the truncation. Thus, the FIX function can be expressed in terms of the following functions:

```
SGN(x) * INT(ABS(x))
```

Some examples of the FIX function appear below:

```
FIX(45)       = 45
FIX(−3.2)     = − 3
FIX(9.9)      =  9
FIX(−3.8)     = −3
```

MAX, MIN Functions

The MAX and MIN functions can have one or more arguments, each of which is either a numeric expression or a numeric-array- variable. The MAX function returns the value of the largest expression or numeric-array-element in the list of arguments; the MIN function returns the smallest value. For example, assume

```
    A(1)    =  5
    A(2)    =  3
    A(3)    = −10
and X       = −12
```

Then

```
MAX (A())          =   5
MAX (7,2,A())      =   7
MAX (3,A())        =   5
MAX (−2*X, A())    =   24
MIN (A())          =  −10
MIN (A(),−50)      =  −50
MIN (A(),X)        =  −12
```

MOD Function

MOD is a function of two expressions which returns the remainder when the first expression is divided by the second. MOD simulates modulo arithmetic if the second expression has a positive integer value. (Note that if the magnitude of the first expression is significantly greater than that of the second, modulo is essentially meaningless. Also, if the second expression is zero, MOD returns the first expression.) The MOD function is equivalent to the expression

```
x — y * INT(x/y)
```

Some examples of the MOD function appear below:

```
MOD(10,5)   =   0
MOD(15,4)   =   3
MOD(6.2,6)  =   .2
```

RND (Random Number) Function

The RND function produces random numbers with values between 0 and 1. RND may be regarded as a means of extracting a random number between 0 and 1 from a fixed "list" of such numbers. Only two types of arguments are distinguished by RND: zero and nonzero. Whenever the RND function is used with a zero argument, it always extracts the first number from the random number list. If the argument is not zero, RND extracts the next random number from the random number list. If RND is first executed with a nonzero argument, however, it produces a number from a random location in the random number "list." When the RND function is executed for a second time with a nonzero argument, it produces the next number in the list, etc. Each time RND is executed with a nonzero argument, therefore, it produces a new random number. If it is necessary to reuse the same series of random numbers (e.g., when debugging a program), an RND function with a zero argument can be used to "reset" the list to the first random number. (The value of the RND argument has, apart from the fact that it is nonzero, no relation to the random number produced.)

The routine in the following example prints out the first 101 random numbers in the random number list each time the program is run. If line 10 is deleted, the program will produce a different set of random numbers each time it is run:

```
10 PRINT RND(0)
20 FOR N = 1 TO 100
30 PRINT RND(1)
40 NEXT N
```

Following Master Initialization of the system or execution of a CLEAR command, the first use of RND with a nonzero argument produces a number from an arbitrary location in the list. Subsequently, a second use of RND with a nonzero argument produces the next number from the list, etc. The use of RND with a zero argument resets the list pointer to the first number in the list.

On the 2200MVP, each partition has its own list of random numbers. Resetting the list to the beginning by using RND(0) only affects the partition which executed the RND(0).

ROUND Function

The ROUND function can be used to round a value to a specified decimal or integer position. ROUND has two arguments: the first argument is the expression whose value is to be rounded, and the second argument is the "rounding factor":

```
ROUND(5.374, 2)
```

Value to be rounded — Rounding factor

If the "rounding factor" is not an integer value, its fractional portion is automatically truncated. The "rounding factor" has a different significance, depending upon whether its truncated value is greater than, less than, or equal to zero.

1. Rounding factor > 0.

   If the rounding factor is greater than zero, it specifies the decimal place to which the value is to be rounded. A rounding factor of 1 causes the value to be rounded to the nearest tenth; a rounding factor of 2 causes rounding to the nearest hundreth, etc. For example:

```
ROUND(3.6839, 1)  =  3.7
ROUND(3.6839, 2)  =  3.68
ROUND(3.6839, 3)  =  3.684
```

2. Rounding factor = 0.

   A rounding factor of 0 causes the value to be rounded to the nearest integer. For example:

```
ROUND(3.25, 0)    =   3
ROUND(-2.2, 0)    =  -2
ROUND(6.5, 0)     =   7
```

3. Rounding factor < 0.

A negative rounding factor causes the value to be rounded to a specified integer place (i.e., a specified position to the left of the decimal point). In this case, the absolute value of the rounding factor plus one (factor + 1) specifies the integer position for rounding. Thus, a rounding factor of -1 causes the value to be rounded to the nearest ten; a rounding factor of -2 causes rounding to the nearest hundred, etc. For example:

```
ROUND(651, -1)    =   650
ROUND(651, -2)    =   700
ROUND(-1601, -3)  =  -2000
```

The ROUND function is equivalent to the following expression:

```
SGN(X)*INT(ABS(X)*10↑(FIX(N))+.5)/10↑(FIX(N))
```

where "X" is the value to be rounded and "N" is the rounding factor.

SGN (Sign) Function

The SGN function performs a numeric comparison of the argument with zero. SGN returns a result of -1 if the argument is less than zero, 0 if the argument equals zero, or +1 if the argument is greater than zero. For example:

```
SGN(.0001)  =   1
SGN(-9.76)  =  -1
SGN(0)      =   0
```

Trig Functions (SIN, COS, TAN, ARCSIN, ARCCOS, ARCTAN)

The trigonometric functions SIN, COS, and TAN and their inverse functions, ARCSIN, ARCCOS, and ARCTAN, can be calculated in one of three modes: radians, degrees, or grads (360°= 400 grads). Normally, all trig functions are performed in radians. If degrees or grads are required, they must be specified with a SELECT statement prior to performing trig calculations. The following SELECT statements are used:

SELECT D — Use degrees in all subsequent trig calculations.

SELECT G — Use grads in all subsequent trig calculations.

Radian measure is automatically selected whenever the system is Master Initialized or a CLEAR command is issued. Radian measure can also be explicitly selected by executing a SELECT R statement.

4.10 SPECIAL-PURPOSE NUMERIC FUNCTIONS

A second group of numeric functions is available for certain special-purpose operations. These functions are summarized in Table 4-2 below and described in detail in other chapters.

Table 4-2.
Special-Purpose Numeric Functions

Function	Meaning
LEN	Determines length of a character string. For example, X=LEN(A$). (See Chapter 5, section 5.9.)
NUM	Determines whether a character string is a legal representation of a BASIC number. For example, X=NUM(A$). (See Chapter 5, section 5.9.)
POS	Finds the position of the first (or last) character in a character string which meets a specified condition. For example, X=POS(A$="$"). (See Chapter 5, section 5.9.)
VAL	Computes the decimal equivalent of a one- or two- byte binary value. For example, X=VAL(A$). (See Chapter 5, section 5.9.)
VER	Verifies that a character string conforms to a specified format. For example, Y=VER(B$,"####"). (See Chapter 5, section 5.9.)
ERR	Returns the error code of the last error condition. For example, X1=ERR. (See Chapter 9, section 9.3.)
SPACE	Determines the amount of free space available in memory after deducting the current size of the Value Stack. For example, Z= SPACE. (See Chapter 2, section 2.5.)
SPACEK	Returns the total user memory size, divided by 1,024. (Thus, for a 32K 2200VP system, SPACEK= 32.) For example, Z1=SPACEK. (See Chapter 2, section 2.6.)

4.11 COMPUTATIONAL ERRORS

Computational errors may be produced in the course of performing arithmetic operations or evaluating numeric functions. Normally, when a computational error other than underflow occurs, the system displays an error message and terminates program execution. An alternative method of handling computational errors is available, however, which enables a program to respond to errors under program control without terminating program execution. This technique involves the use of the SELECT ERROR statement and the ERR function and is described in detail in Chapter 9, section 9.3.

CHAPTER 5 ALPHANUMERIC STRING MANIPULATION INSTRUCTIONS

5.1 ALPHANUMERIC CHARACTER STRINGS

In addition to its ability to manipulate and operate upon numeric values, the BASIC-2 language also provides an extensive capability for processing information in the form of alphanumeric character strings. A character string is a sequence of characters treated as a unit. A character string may consist of any combination of keyboard characters, including the letters A-Z, the numbers 0-9, and special symbols such as "+", "-", and "$". Characters not found on the keyboard can be represented in the form of hexadecimal codes. Typical examples of character strings are names, addresses, and report headings.

Character strings are represented in a program in two basic forms:

1. As the values of alphanumeric string variables or portions of string variables.

2. As literal strings.

5.2 ALPHANUMERIC STRING VARIABLES

Alphanumeric character strings are stored and processed in a special type of variable called the alphanumeric string variable or simply the alpha-variable. Alpha-variables are distinguished from numeric-variables by the presence of a dollar sign ("$") following the variable name. For example, the variable name "A" represents a numeric-variable, while the variable name "A$" represents an alpha-variable; similarly, "N3" is numeric and "N3$" is alphanumeric. Note that a numeric- variable and an alphanumeric-variable are different variables, even if both have the same name. Data stored in an alpha-variable can be operated on with logical operators and alphanumeric functions. It can also participate in binary or packed decimal arithmetic operations, but cannot take part in standard numeric arithmetic operations. Only numeric data can be used in arithmetic operations. (See Chapter 4 for a discussion of numeric operations.)

Alpha-variables are of two types: alpha-scalar-variables and alpha-array-variables. An alpha-scalar-variable can store a single character string ranging from one to 124 characters in length. An alpha-array-variable consists of one or more array elements, each of which can store a character string ranging from one to 124 characters in length. Array-variables are useful because they enable the programmer to reference a collection of data with a single array name. (Under certain conditions, the separate character strings stored in the elements of an alpha- array can be treated together as a single contiguous character string. The technique involved is explained in section 5.7.)

Alphanumeric-array-variables are further divided into two classes: one-dimensional arrays and two-dimensional arrays. A one-dimensional array (also called a "single-subscripted array") resembles a list because it has a single column of elements. A two-dimensional array (also called a "double-subscripted array") resembles a table because it has both rows and columns of elements. The structure of arrays is discussed in greater detail in Chapter 2, section 2.4.

Scalar-variables and array-variables are regarded by the system as different types of variables, while one-dimensional and two-dimensional array-variables are different but related kinds of arrays. Thus, the same name may be used in a program for both an alpha-scalar-variable and an alpha-array-variable, but the same name cannot be used for a one-dimensional alpha-array and a two-dimensional alpha-array in the same program. For example, the variable names A$ and A$(5) can both be used in the same program, but A$(5) and A$(6,6) cannot.

5.3 ALPHANUMERIC-VARIABLE LENGTH

An alphanumeric-variable identifies a unique location in memory reserved for the storage of alphanumeric data. The system reserves space for each variable during program resolution, a procedure in which the program is scanned for all variable references. The amount of space reserved for each variable can be specified by the programmer in a DIM or COM statement. The maximum length of an alpha-scalar-variable or of an element in an alpha-array is 124 bytes, while the minimum length is one byte in each case. If the programmer does not explicitly dimension an alpha-scalar-variable in a DIM or COM statement, the system automatically reserves 16 bytes for the variable. Similarly, if the programmer does not specify an element length when dimensioning an alphanumeric-array, the system automatically reserves 16 bytes for each element of the array.

The length of an alpha-variable or array element specified in a DIM or COM statement is called its "defined" length. In many cases, however, the character string stored in an alpha-variable will not occupy the entire defined length. The end of the value of an alpha-variable is normally assumed to be the last nonblank character. When the value of the alpha-variable is all blanks, however, its value is assumed to be one blank. Hence, trailing blanks generally are not considered part of the value of an alpha-variable. For example:

```
:10 A$="ABC      "
:20 PRINT A$;"DEF"
:RUN
ABCDEF
```

Note that the trailing blanks of A$ were not printed in the example above.

The character string stored in an alpha-variable is called the "current value" of the alpha-variable, and its length, up to the first trailing blank, is called "the current length" of the variable. The length function, LEN, determines the current length of an alpha-variable. For example:

```
:10 A$="ABCD      "
:20 PRINT LEN(A$)
:RUN
4
```

In the example above, notice that the LEN function does not consider trailing blanks to be part of the value of the alpha-variable.

Most alphanumeric instructions operate on the current length of an alpha-variable. In some cases, however, the entire defined length of the variable may be used. It is important, therefore, to understand the distinction between defined length and current length.

The STR Function

Frequently it is desirable to examine or operate on a specific portion of the value of an alpha-variable. The STR (STRing) function is used for this purpose. STR permits the programmer to define a substring of one or more consecutive characters within an alpha-variable. *The substring defined by a STR function can be used wherever alpha-variables are legal.* For example, assume

```
A$ = "ABCDEFG"
```

then

STR(A$,1,4)	Defines a substring beginning with the first byte of A$, four bytes in length ("ABCD").
STR(A$,5,3)	Defines a substring beginning with the fifth byte of A$, three bytes in length ("EFG").
STR(A$,2,2) = "12"	Defines a substring beginning with the second byte of A$, two bytes in length, and assigns the characters "12" to these byte positions. A$ now contains "A12DEFG".
STR(A$)	Defines a substring beginning with the first byte of A$ and ending with the last byte of A$, including any trailing spaces ("ABCDEFG ").

Example:

```
:10 A$="ABCDEFG"
:20 PRINT STR(A$,1,4)
:30 B$=STR(A$,5,3): PRINT B$
:40 STR(A$,2,2) = "12": PRINT A$
:50 PRINT STR(A$);"X"
:RUN
ABCD
EFG
A12DEFG
A12DEFG       X
```

The STR function is discussed in detail in section 5.9.

5.4 ALPHANUMERIC LITERAL STRINGS

An alphanumeric literal string is a character string enclosed in double quotation marks (" "). Literal strings can be specified as constant data, usually in a PRINT statement, to create headings or titles. For example:

```
:10 PRINT "VALUE OF X="; X
```

In line 10, the character string "VALUE OF X=" is a literal string which is printed exactly as it appears. The character "X" immediately following the semicolon and not enclosed in quotes is a numeric-variable-name.

Literal strings also can be assigned to alphanumeric-variables. For example:

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

A literal string may be up to 255 characters in length. However, when the value of a literal string is stored in an alpha-variable, it is truncated to the defined length of the alpha-variable. For example:

```
:10 DIM A$5
:20 A$="123456789"
:30 PRINT A$
:RUN
12345
```

In the example above, notice that the value is truncated to five characters because the defined length of A$ is five bytes.

The minimum length of a literal string is one; the null string (" ") is not allowed. An alphanumeric literal string may contain any character, including the colon, with the exceptions of double quotation marks ("), a carriage return (RETURN), and the characters represented by codes HEX (FA) — HEX (FF), which are reserved for system use.

An additional form of the literal string, the hexadecimal literal string, also is legal in BASIC-2.

5.5 HEXADECIMAL LITERAL STRINGS

Hexadecimal literal strings are a special form of literal string consisting of one or more hexadecimal codes specified in a HEX function. (See the discussion of the HEX function in section 5.9.) Hexadecimal codes are composed of a pair of hexadecimal digits (0-9 or A-F); these codes are particularly useful for representing special characters not found on the system keyboard and system control codes. For example, hexadecimal codes can be used to control the CRT display. The hex code 03 is a control code which clears the CRT and homes the cursor; thus, the statement

```
:50 PRINT HEX(03);
```

will, upon execution, clear the CRT display. Hex literal strings are legal wherever alphanumeric literal strings are allowed. In particular, they can be assigned to alpha-variables in an assignment statement. For example:

```
:60 A$ = HEX(313233)
:70 PRINT A$
:RUN
123
```

The characters "123" are printed since they are represented by the hex codes 31, 32, and 33.

5.6 CONCATENATION OF STRINGS

The concatenation operator (&) combines two strings; one string is put directly after another, without intervening characters. The two strings combined by the concatenation operator are treated as a single string. For example:

```
:10 A$="WANG"
:20 B$="LABS."
:30 C$=A$ & B$
:40 PRINT C$
:RUN
WANGLABS.
```

Literal strings can be concatenated with values of alpha-variables. For example:

```
:10 A$="WANG"
:20 B$="LABS."
:30 C$=A$ & " " & B$
:40 PRINT C$
:RUN
WANG LABS.
```

Any legal alphanumeric operand, including hex literal strings, can be concatenated with alpha-literals or alpha-variables. For example:

```
:10 A$ = "SMITH"
:20 C$ = A$ & HEX(2C) & "JOHN"
:30 PRINT C$
:RUN
SMITH,JOHN
```

The concatenation operator may be used only on the right-hand side of an alphanumeric assignment (LET) statement; it is not legal in any other statement. Although more than one concatenation operator may be used in the same assignment statement, the concatenation operator may *not* appear in combination with other alphanumeric operators in the same statement.

5.7 USE OF THE ALPHA-ARRAY AS A SCALAR-VARIABLE

Wherever alpha-variables are allowed, an alphanumeric-array-variable can be referenced as though it were a scalar-variable containing a single contiguous character string. An alpha-array, however, can not be referenced as a scalar- variable in certain statements which always treat arrays on an element-by-element basis. When used as a scalar-variable, the array is denoted with an alpha-array designator, which consists of the array name followed by closed parentheses (e.g., A$( ), B4$( ), etc.).

The ability to reference an entire array as a scalar- variable enables the program to operate easily on extremely long character strings since the size of an alpha-array is limited only by the memory size of the machine. A portion of the string can be referenced by using the STR function (e.g., STR (A$( ),4,4)).

The order of the elements in an nxm array is (1,1), (1,2),...,(1,m), (2,1), (2,2),..., (2,m),...,(n,1),(n,2),...,(n,m). For example, assume that the array A$ is dimensioned as a 2x3 array with each element four bytes in length:

```
DIM A$ (2,3)4
```

Assume further that the following assignments are made to the elements of A$( ):

```
A$(1,1) = "ABCD"
A$(1,2) = "EFGH"
A$(1,3) = "IJKL"
A$(2,1) = "MNO"
A$(2,2) = "P"
A$(2,3) = "Q"
```

The array A$( ) has the following form:

	1	2	3
1	ABCD	EFGH	IJKL
2	MNOb	Pbbb	Qbbb

where the character "b" denotes a "space". Trailing spaces in every element but the last are considered part of the value of the array. If the last element of the array is completely blank, however, any trailing spaces in the penultimate element are not considered part of the value of the array. For example:

```
:PRINT A$( )
ABCDEFGHIJKLMNO P  Q
:PRINT LEN(A$( ))
 21
```

Notice that the LEN function counts to the *last nonblank character* in the array. Trailing spaces in A$(2,3) are *not* counted as part of the value of A$( ), although trailing spaces in A$(2,1) and A$(2,2) *are* counted because they precede the last nonblank character ("Q"). In effect, these spaces are regarded as embedded spaces when the array is treated as a single contiguous character string. If "Q" is removed from A$(2,3), the length of A$( ) as computed by LEN drops from 21 to 17 because the trailing spaces in A$(2,2) do not precede any nonblank character and are no longer counted as part of the value of A$( ).

5.8 ALPHANUMERIC EXPRESSIONS

Just as numeric expressions can be used on the right-hand side of numeric assignment statements, alphanumeric expressions can be used on the right-hand side of alpha assignment statements. The alphanumeric assignment statement has the form:

```
alpha-variable = alpha expression
```

The variable on the left-hand side of the equal sign is called the "receiver-variable" because it receives a value. For example, if

```
A$ = B$ AND C$
```

A$ is the receiver. The statement is read as "let A$ = B$, then logically AND C$ with the current value of A$." The general form of an alpha expression is:

```
[alpha-operand] [alpha-operator alpha-operand...]
```

but the null expression is not allowed.

Any number of alpha operators and operands may be combined in a single expression, with the exception of the concatenation operator. The concatenation operator may *not* be combined with other operators in the same expression. An alpha expression is processed from left-to-right, one term at a time. Parentheses can *not* be used to alter the order of processing. Each operator performs the specified binary operation on the defined length of the receiver-variable and the value of the operand following the operator. The receiver is then set equal to the result of the operation. For example:

```
A$ = AND B$ OR ALL (F0)
```

logically ANDs the value of B$ with A$, then logically ORs each byte of A$ with hex F0. The result is stored in A$.

Alphanumeric Expression Operators

ADD	binary addition without carry (see Chapter 6).
ADDC	binary addition with carry (see Chapter 6).
AND	logical AND (see section 5.9).
BOOLh	specified logical Boolean operation (see section 5.9).
DAC	decimal add with carry (see Chapter 6).
DSC	decimal subtract with carry (see Chapter 6).
OR	logical OR (see section 5.9).
SUB	binary subtract without carry (see Chapter 6).
SUBC	binary subtract with carry (see Chapter 6).
XOR	logical exclusive OR (see section 5.9).
&	concatenation (see section 5.6).

Alphanumeric Expression Operands

Types of Operands	Examples
alpha-variable	A$,B1$,C$(2),D$( )
alphanumeric literal string	"JOHN SMITH"
HEX literal string	HEX(0B)
STR function	STR (A$,I,J)
BIN function	BIN(I+1)
ALL function	ALL(FF)

5.9 GENERAL FORMS OF THE ALPHANUMERIC AND SPECIAL-PURPOSE FUNCTIONS AND OPERATORS

General forms of the alpha functions and operators are shown on the following pages, arranged alphabetically for ease of reference. Note that the special alpha operators ADD, SUB, DAC, and DSC, which are used for binary and packed decimal operations, are not included here, but are covered in Chapter 6. Note also that the special-purpose numeric functions LEN, NUM, POS, VAL, and VER *are* covered in this chapter because they operate on alphanumeric arguments and are used primarily in the analysis and manipulation of alphanumeric string data.

ALL Function

General Form:

receiver = [. . .] ALL (hh | alpha-variable | literal-string) [. . .]

Where:

receiver = alpha-variable [,alpha-variable] . . .

h = hexadecimal digit (0-9 or A-F)

Purpose:

The ALL function defines a character string of unlimited length in which every character is equal to the character specified in the function. The character may be specified with a pair of hexdigits or as the first character of a literal string or alpha-variable. The ALL function can be used to initialize alphanumeric-variables and arrays; it is legal only in the alpha-expression portion of an alphanumeric assignment statement. (See the discussion of alpha expressions and the alpha assignment statement in section 5.8.)

For example, the statement

```
A$( ) = ALL(00)
```

sets each character in A$( ) equal to binary 0. ALL is also useful in logical operations for changing certain bits in every character of an alpha-variable or array. For example, the statement

```
A$ = B$ AND ALL(7F)
```

sets A$ = B$ and masks off the high-order bit of each character in A$ by ANDing each character with HEX(7F).

Examples of Valid Syntax:

```
10 A$ = ALL(".")
20 B$ = AND ALL(0F)
30 STR(C$,2,3) = B$ XOR ALL(FF) ADD C$
40 A$ = ALL(B$)
```

AND,OR,XOR Operators

General Form:

receiver = [. . .] { AND / OR / XOR } operand [. . .]

Where:

receiver = alpha-variable [,alpha-variable] . . .

operand = { alpha-variable / literal-string / ALL / BIN }

Purpose:

The logical operators AND, OR, and EXCLUSIVE OR (XOR) perform the specified logical operations on the value of an alpha-variable. These operators may be used only in the alpha-expression portion of an alphanumeric assignment statement. (See the discussion of alpha expressions and the alpha assignment statement in section 5.8.) The value of the operand immediately following the logical operator and the defined length of the receiver-variable are operated upon, and the result is assigned to the receiver-variable. For example, the statement

```
:10 A$ = AND B$
```

logically ANDs the value of B$ with A$; the result is stored in A$.

The logical operations are performed on a character-by-character basis from left to right, starting with the leftmost character in each field. If the defined length of the operand is shorter than that of the receiver, the remaining characters of the receiver are not changed. If the defined length of the operand is longer than that of the receiver, the operation terminates when the last character in the receiver has been operated on. Note that if the operand is equal to or longer than the receiver-variable, the entire contents of the receiver-variable, including trailing spaces, are operated on. If the operand is an alpha-variable, its entire contents, including any trailing spaces, are used. (Trailing spaces normally are not considered to be part of the value of an alphanumeric-variable.) A portion of the alpha-variable can be operated on by using the STR function to define a substring in the variable. For example, the statement

```
:10 STR(A$,3,2) = XOR B$
```

operates only on the third and fourth bytes of A$.

The logical operators AND, OR, and XOR also can be used in the IF...THEN statement to separate multiple conditions (see Chapter 11, section 11.2).

Examples of Valid Syntax:

```
10 A$ = AND HEX(7F)
20 A$ = OR B$
30 STR(A$,1,2) = XOR ALL(FF)
40 C$ = A$ AND B$
```

BIN Function

General Form:

receiver = [. . .] BIN(expression [,2]) [. . .]

Where:

receiver = alpha-variable [, alpha-variable] . . .
0 < = value of expression < 256 if "2" omitted
0 < = value of expression < 65536 if "2" included

Purpose:

The BIN function converts the integer value of the expression to a one-byte binary number if "2" is not specified or to a two-byte binary number if the "2" parameter is included. BIN is the inverse of the VAL function. The BIN function can be used only in the alpha-expression portion of an alphanumeric assignment statement. (See the discussion of alpha expressions and the alpha assignment statement in section 5.8.) BIN is especially useful for code conversion and conversion of numbers from internal decimal format to binary.

Examples of Valid Syntax:

```
10 A$ = BIN(65)
```

(Sets A$="A" since the binary value of decimal 65 is the character code for the letter "A".)

```
20 STR(A$,I,2) = BIN(X,2)
30 C$ = D$ ADD BIN(X*T/2)
```

BOOL Operator

General Form:

receiver = [...] BOOLh operand [...]

Where:

receiver = alpha-variable [,alpha-variable] . . .

h = hexadecimal digit (0-9 or A-F)

operand = { alpha-variable | literal-string | ALL | BIN }

Purpose:

BOOL is a generalized logical operator which performs a specified operation on the value of the receiver-alpha-variable. The operation to be performed is specified by the hexadecimal digit following BOOL (see Table 5-1 which follows). BOOL may be used only in the alpha-expression portion of an alpha assignment statement (see the discussion of alpha expressions and the alpha assignment statement in section 5.8). The value of the operand and the value of the receiver-variable are operated upon, and the result is assigned to the receiver-variable. For example, the statement

```
:10 A$ = BOOL7 B$
```

logically not-ANDs the value of B$ with the value of A$ and assigns the result to A$.

The logical operations are performed on a character-by- character basis from left to right, starting with the leftmost character in each field. If the defined length of the operand is shorter than that of the receiver-variable, the remaining bytes of the receiver-variable are not changed. If the defined length of the operand is equal to that of the receiver-variable, the entire values of both, including any trailing spaces, are operated on. (Note that trailing spaces normally are not considered part of the value of an alpha-variable.) If the operand is longer than the receiver-variable, the operation terminates when the last byte of the receiver-variable has been operated on. A specified portion of an alpha-variable can be operated on if the portion is defined with a STRing function. For example, the statement

```
:10 STR(A$,3,2) = BOOL9 B$
```

operates only on the third and fourth bytes of A$.

In every case, the logical operation to be performed is identified by the hex digit following BOOL. A total of 16 logical operations are available (see Table 5-1 below). The hex digit used to identify each operation is a kind of mnemonic which represents the logical result of performing the operation on the following bit combinations:

receiver-variable:	1100 (hex C)
operand:	1010 (hex A)

For example, the hex digit "E" identifies the OR operation. When 1100 is ORed with 1010, the result is 1110 or hex digit E. Note that several commonly used BOOL operations are available as separate operators: BOOLE is equivalent to OR, BOOL6 TO XOR, and BOOL8 to AND. The 16 possible logical functions are listed in Table 5-1 below.

Table 5-1.
BOOLh Logical Functions

Hex Digit	Binary Representation	Logical Function
0	0000	null
1	0001	not-OR
2	0010	operand does not imply receiver
3	0011	complement of receiver
4	0100	receiver does not imply operand
5	0101	complement of operand
6	0110	exclusive OR
7	0111	not-AND
8	1000	AND
9	1001	equivalence
A	1010	receiver = operand
B	1011	receiver implies operand
C	1100	operand = receiver
D	1101	operand implies receiver
E	1110	OR
F	1111	identity

Examples of Valid Syntax:

```
10 A$=BOOL1 B$
20 B$=C$ BOOL7 D$
30 STR(A$,3,2) = BOOL9 ALL(FF)
40 A$ = BOOLB HEX(F0F0F0)
```

HEX Literal

General Form:

HEX(hh [hh...])

Where:

h = hexadecimal digit (0-9 or A-F)

Purpose:

The HEX literal string permits the use of any eight-bit character code in a BASIC program. It may be used wherever alphanumeric literal strings enclosed in double quotes are allowed. Each character in the literal string is represented by two hexadecimal digits. If the HEX literal contains an odd number of hex digits or any characters other than hex digits or spaces, an error will result.

The HEX literal often is used to define control codes which do not appear on the keyboard for transmission to peripheral devices. For example, the statement

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

clears the CRT screen.

Any character can be represented by a pair of hex digits. A complete chart of hex codes used by the CRT is given in Appendix F. See the appropriate peripheral manual for codes pertaining to other devices.

Examples of Valid Syntax:

```
10 A$=HEX(0C0A0A)
20 IF A$ > HEX(7F) THEN 100
30 B$=A$ & HEX(000000)
40 PRINT HEX(0E);"TITLE"
```

LEN Function

General Form:

LEN (alpha-variable)

Purpose:

The LENgth function, LEN, is a numeric function which determines the number of characters in the value of an alphanumeric-variable. LEN is a special-purpose numeric function which uses an alphanumeric argument, but returns a numeric value as a result. LEN may be used wherever numeric functions are legal. (See the discussion of numeric functions in Chapter 4, section 4.8.)

Trailing spaces are not considered by LEN to be part of the current value of an alpha-variable. LEN scans the variable and returns the number of characters up to the first trailing space. All characters up to the first trailing space, including leading and embedded spaces, are treated as part of the value. In the special case of a variable which contains all blanks, the length is 1. For example:

```
:10 A$ = "ABCD  "
:20 PRINT LEN(A$)
:RUN
 4

:10 A$ = "A BCD     "
:20 PRINT LEN(A$)
:RUN
 5

:10 A$ = "     "
:20 PRINT LEN(A$)
:RUN
 1
```

A LEN function of a STR function returns the length parameter specified in the STR function, irrespective of the contents of the variable. For example:

```
:10 A$ = "AB     "
:20 PRINT LEN(STR(A$,,5))
:RUN
 5
```

Examples of Valid Syntax:

```
10 X = LEN(A$) + 2
20 IF LEN(A$(3)) < 8 THEN 100
30 X = LEN(A$( ))
```

NUM Function

General Form:

NUM (alpha-variable)

Purpose:

NUM is a numeric function which determines the number of sequential ASCII characters in the specified alphanumeric- variable which represent a legal BASIC number. A BASIC number consists of numeric characters in a standard format. A numeric character is defined to be one of the following:

blanks
digits 0-9
decimal point (.)
leading plus and minus signs (+,-)
letter E

Counting of numeric characters begins with the first character of the specified variable or STR function. The count ends when a nonnumeric character occurs, when the sequence of numeric characters fails to conform to standard BASIC number format, or when all characters in the specified variable have been scanned. 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 the numeric portion of an alphanumeric value. (Note that the BASIC representation of a number cannot have more than 13 significant mantissa digits.) NUM is particularly useful in applications where it is desirable to numerically validate data entered under program control.

NUM is a special-purpose numeric function which utilizes an alphanumeric argument, but returns a numeric result. The NUM function may be used wherever numeric functions are legal. (See the discussion of numeric functions in Chapter 4, section 4.8.)

Examples:

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

Comments:

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

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

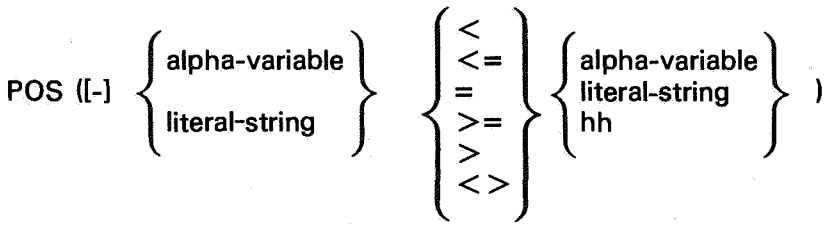
X = 4 since the sequence of numeric characters fails to conform to standard BASIC number format once the "+" character is encountered.

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

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 an entered value (alphanumeric ASCII characters). If the value represents a legal BASIC number, NUM(A$) equals 16, the number of characters in the string variable A$.

POS Function

General Form:



Where:

h = hexadecimal digit (0-9 or A-F)

Purpose:

The POSition function, POS, is a numeric function which returns the position of the first character in an alpha-variable or literal string which satisfies a specified relation to a given value. The value against which comparison is made follows the relational operator and may be specified as the first character of a literal string, the first character of the value of an alpha-variable, or as a pair of hexadecimal digits.

The position of the first character in the alpha-variable which satisfies the given relation is returned as a numeric value. If an alpha-variable is specified, its entire (defined) length, including trailing spaces, is scanned by POS. If no character is found to satisfy the specified relation, POS = 0.

The ability to scan a literal string with the POS function can be useful when a fixed table of characters must be compared to a single character in a variable. In this case, the use of a literal string saves assigning the data to another variable and results in code which is clearer and more self-explanatory. For example:

```
:10 DIM A$1
:20 LINPUT "LOAD,SAVE,VERIFY, OR DISPLAY", A$
:30 ON POS("LSVD" = A$) GOSUB 100,200,300,400:ELSE GOSUB 500
```

If a minus sign (-) immediately precedes the alpha-variable being searched, POS returns the position of the *last* character satisfying the given relation rather than the first. For example:

```
:10 DIM A$64
:20 A$ = "MR. SAM ADAMS, BOSTON, MASS. 01906"
:30 PRINT POS(A$ = ".")
:40 PRINT POS(-A$ = ".")
:RUN
3
28
```

POS is a special-purpose numeric function which uses an alphanumeric argument, but returns a numeric value as a result. The POS function may be used wherever numeric functions are legal. (See the discussion of numeric functions in Chapter 4, section 4.8.)

Examples of Valid Syntax:

```
10 X=POS(A$="$")
20 PRINT POS(STR(A$,I,J)=HEX(0D))
30 IF POS(A$( )="T")=0 THEN 100
40 B$=STR(A$,POS(A$="+"))
50 Y=POS(-B$<C$)
60 X=POS(A$=0A)
70 T=POS("123456789"=T$)
80 J=POS(HEX(0102040810204080)>A$)
90 L=POS(-"0:A a"<=STR(B$,N,1))
```

STR Function

General Form:

STR(alpha-variable [,[s][,n]])

Where:

s = an expression, of which the integer portion specifies the position of the starting character of the substring in the alpha-variable.

n = an expression, of which the integer portion specifies the number of characters in the substring.

s,n >0; s+n-1 <maximum no. of characters in alpha-variable.

Purpose:

The STRing function, STR, defines a substring of an alpha-variable. STR permits a specified portion of the alphanumeric value to be examined, extracted, or changed. For example, the statement

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

sets B$ equal to the 3rd, 4th, 5th, and 6th characters of A$.

If the "s" parameter is omitted, the default value is 1 and the substring starts with the first character in the alpha-variable.

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

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

If both "s" and "n" are omitted, the entire value of the alpha-variable is used, including any trailing spaces.

If the STR function is used on the left-hand side of the equal sign in an assignment statement and the value to be received is shorter than the specified substring, the substring is padded with trailing spaces. For example:

```
:10 A$="123456789"
:20 STR(A$,3,5)="ABC"
:30 PRINT A$
:RUN
12ABC  89
```

NOTE:

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

Examples of Valid Syntax:

```
10 A$=STR(B$,2,4)
20 STR(D1$,I,J)=B$
30 IF STR(A$,3,5)<STR(B$,3,5) THEN 100
40 READ STR(A$( ),9,9)
50 PRINT STR(C$,3)
60 D$=STR(A$( ),S,N)
70 LINPUT STR(A$,,10)
```

VAL Function

General Form:

VAL ({ alpha-variable / literal-string } [,2])

Purpose:

VAL is a numeric function which converts the binary value of the first byte of a specified alphanumeric value to a numeric value. If the "2" parameter is included, VAL converts the binary value of the first two bytes of the specified alphanumeric value to a two-byte numeric value. The VAL function is the inverse of the BIN function.

VAL is a special-purpose numeric function which utilizes an alphanumeric argument, but returns a numeric result. The VAL function can be used wherever numeric functions are legal. (See the discussion of numeric functions in Chapter 4, section 4.8.)

VAL is particularly useful in code conversion and table lookup operations since the converted number can be used either as a subscript to retrieve the corresponding code from an array or with the RESTORE statement to retrieve codes or data from DATA statements. The two-byte conversion performed by VAL also is useful when working with two-byte binary sector addresses in disk operations.

Examples:

```
:PRINT VAL (HEX(20))
 32

:A$=HEX(1234)
:PRINT VAL(A$,2)
 4660
```

Examples of Valid Syntax:

```
10 X = VAL(A$)
20 PRINT VAL("A")
30 Y = VAL(B$,2)
40 RESTORE VAL(STR(A$,I,1))+1
50 B$ = A$(VAL(C$)+1)
60 IF VAL(X$,2)>1024 THEN 100
```

VER Function

General Form:

VER({ alpha-variable, / literal-string, } format-specification)

Where:

format-specification = { alpha-variable / literal-string }

Purpose:

The VERify function, VER, is a numeric function used to verify that the value of an alphanumeric-variable or literal string conforms to a specified format. The first variable or literal string in the function is verified against the format specified by the second variable or literal string (the "format-specification"). The VER function returns the number of successive characters in the value being verified which conform to the format-specification. Each character in the defined length of the alpha-variable or literal string is verified by checking it against the character set associated with the specified format-character in the format-specification. If a character in the value being verified does not appear in the specified format character set, it is regarded as an illegal character and causes a termination of the VER operation.

Format-Character Definitions

A	=	Alphabetic only (A-Z or a-z).
#	=	Numeric only (0-9).
N	=	Alphabetic or numeric (A-Z, a-z, or 0-9).
H	=	Hexadecimal (0-9 or A-F).
P	=	Packed decimal.
+	=	Sign (+, -, or blank).
X	=	Any character.
Other	=	Only the specified character.

The VERify operation terminates when an illegal character is found, when the end of the value (*including* any trailing spaces) is encountered, or when the end of the format- specification is reached.

VER is a special-purpose numeric function which utilizes an alphanumeric argument, but returns a numeric result. The VER function may be used wherever numeric functions are legal. (See the discussion of numeric functions in Chapter 4, section 4.8.)

Examples:

```
Assume A$ = "$012.45AB"
Then:
    VER (A$, "$###.##AA") = 9
    VER (A$, "$###.####") = 7
    VER (A$, "AAAAAAAAA") = 0
    VER (STR(A$,6,4), "NNNN") = 4
```

CHAPTER 6 BINARY AND PACKED DECIMAL ARITHMETIC OPERATORS

6.1 INTRODUCTION

This chapter contains general forms for alpha operators used to perform binary and packed decimal arithmetic on alphanumeric data (literal strings or the contents of alpha-variables). These instructions do not treat alphanumeric data as character strings, but as binary or packed decimal values or a series of values. A list of instructions which perform either addition or subtraction on binary or packed decimal values follows:

ADD[C]	—	Performs binary addition on a pair of binary values with or without carry propagation between bytes.
DAC	—	Performs decimal addition on a pair of packed decimal numbers.
DSC	—	Performs decimal subtraction on a pair of packed decimal numbers.
SUB[C]	—	Performs binary subtraction on a pair of binary values.

The binary operators can be particularly useful when manipulating binary counters or disk addresses. The packed decimal operators may be useful for extended-precision decimal addition and subtraction since operations can be performed on packed decimal numbers of almost unlimited length. Packed decimal numbers can be packed two digits per byte in an alpha-variable, and the only effective limit to the size of an alpha-variable is the amount of available memory.

> NOTE:
>
> The term "alpha-variable" includes *both* scalar- *and* array-variables.

Decimal to Binary Conversion and Two's Complement Notation

In order to perform binary operations on numbers on the 2200VP/MVP, the decimal values must first be converted to binary values. Conversion of a nonnegative integer to binary form can be accomplished in the following manner (assuming X is a numeric value):

Conversion to a One-Byte Binary Value (0<=X<256)

```
10 DIM A$1
.
.
.
100 A$=BIN(X)
```

Conversion to a Two-Byte Binary Value (0<=X<65536)

```
10 DIM A$2
.
.
.
100 A$=BIN(X,2)
```

Conversion to Binary When the Decimal Value Exceeds Two Bytes

```
10 DIM A$7
.
.
.
90 REM % NUMERIC TO BINARY CONVERSION
100 FOR I = LEN(STR(A$)) -1 to 1 STEP -2
110 Q = INT(X/65536)
120 STR(A$,I,2) = BIN(X-65536*Q,2)
130 X=Q
140 NEXT I
150 IF I>1 THEN STR(A$,,1)=BIN(X)
160 RETURN
```

If a negative number is to be either an operand or the result of a binary arithmetic operation, the number must be represented in two's complement form after conversion to binary format. Using the two's complement form ensures the operator that the obtained result will be signed correctly. The following two examples demonstrate how to obtain a two's complement representation of a binary number.

Example 1:

```
10 DIM A$7,B$7
.
.
.
200 B$= ALL(00) SUBC A$
210 A$=B$
220 RETURN
```

Example 2:

```
10 DIM A$7
.
.
.
200 A$= XOR(ALL(FF)) ADDC HEX(01)
210 RETURN
```

In order to differentiate between positive and negative binary values, the programmer should ensure that one extra hex digit is reserved to indicate the sign of the number. This byte should be reserved to the left of the most significant hex digit of the largest binary value. The leftmost digit of each value should, therefore, be either "0" for positive values or "F" for negative values. Results can be left in this form until they are converted back from binary to decimal.

To convert these values, the programmer should first check the leftmost (high-order) digit to determine the sign of the value. If the value of the digit is F, the number is negative and the two's complement should be obtained (as described previously) before conversion. The numeric result of the conversion should then be multiplied by -1 to ensure the correct sign.

The following program illustrates several routines to convert binary values to decimal values (with or without the sign) and vice versa:

```
10 DIM A$7,B$7
.
.
.
90 REM % NUMERIC TO BINARY (UNSIGNED)
100 FOR I = LEN(STR(A$)) -1 TO 1 STEP -2
110 Q = INT(X/65536)
120 STR(A$,I,2) = BIN(X-65536*Q,2)
130 X=Q
140 NEXT I
150 IF I>1 THEN STR(A$,,1)=BIN(X)
160 RETURN
.
.
.
190 REM % TWO'S COMPLEMENT
200 B$ = ALL(00) SUBC A$
210 A$= B$
220 RETURN
.
.
.
290 REM % BINARY TO NUMERIC (UNSIGNED)
300 X=0
310 FOR I = 1 TO LEN(STR(A$))-1 STEP 2
320 X=X*65536 +VAL(STR(A$,I,2),2)
330 NEXT I
340 IF I+1 <LEN(STR(A$)) THEN X= X*256 +VAL(STR(A$,I+2))
350 RETURN
.
.
.
490 REM % SIGNED NUMERIC TO BINARY
500 X=INT(X)
510 IF X>=0 THEN 540
520 X=ABS(X): GOSUB 100: GOSUB 200
530 GOTO 550
540 GOSUB 100
.
.
.
590 REM % SIGNED BINARY TO NUMERIC
600 IF STR(A$,,1) <HEX(80) THEN 640
610 GOSUB 200: GOSUB 300
620 X =-X
630 GOTO 650
640 GOSUB 300
```

Decimal to Packed Decimal (BCD) Conversion and Ten's Complement Representation

To operate on decimal values with packed decimal (Binary Coded Decimal) arithmetic operators, the value must first be converted to the packed format. Converting a nonnegative integer to packed decimal form is accomplished most easily with the PACK statement:

```
10 DIM A$7
.
.
.
90 REM % NUMERIC TO BCD CONVERSION
100 PACK (##############) A$ FROM X
110 RETURN
```

If a negative number is to be represented, the ten's complement of the value must be obtained after conversion to BCD format. After first converting the absolute value of the decimal number to BCD format, determine the ten's complement of the BCD value in the following manner:

```
190 REM % TEN'S COMPLEMENT REPRESENTATION
200 B$ = ALL(00) DSC A$
210 A$ = B$
220 RETURN
```

Differentiation between positive and negative BCD values is accomplished in the same manner as for binary values. The programmer should reserve one extra digit to the left of the most significant digit of the largest BCD value. The value of this digit is then "0" for positive values and "9" for negative values. Results may be left in this form until conversion back to decimal format is required.

To convert to decimal format, the programmer should first check the high-order digit to determine its value. If the value is 9, the ten's complement of the value should be obtained (as previously described) before conversion. The resulting decimal value should then be multiplied by -1 to ensure the correct sign.

The following program illustrates routines for signed and unsigned conversion of decimal to BCD and BCD to decimal values:

```
10 DIM A$7, B$7
.
.
.
90 REM % NUMERIC TO BCD CONVERSION
100 PACK (##############) A$ FROM X
110 RETURN
.
.
.
190 REM % TEN'S COMPLEMENT REPRESENTATION
200 B$ = ALL(00) DSC A$
210 A$ = B$
220 RETURN
.
.
.
290 REM % BCD TO NUMERIC (UNSIGNED)
300 UNPACK (##############)A$ TO X
310 RETURN
.
.
.
```

```
490 REM % SIGNED NUMERIC TO BCD
500 X=INT(X)
510 IF X>=0 THEN 540
520 X=ABS(X): GOSUB 100: GOSUB 200
530 GOTO 550
540 GOSUB 100
.
.
.
590 REM % SIGNED BCD TO NUMERIC
600 IF STR(A$,,1)<HEX(80) THEN 640
610 GOSUB 200: GOSUB 300
620 X =-X
630 GOTO 650
640 GOSUB 300
.
.
.
```

6.2 GENERAL FORMS OF THE BINARY AND PACKED DECIMAL OPERATORS

General forms of the binary and packed decimal arithmetic operators ADD, SUB, DAC, and DSC are shown on the following pages, arranged alphabetically for ease of reference. Other operators which take alphanumeric arguments are described in Chapter 5.

ADD[C] Operator

General Form:

receiver = [...] ADD[C] operand [...]

Where:

receiver = alpha-variable [, alpha-variable] . . .

operand = { alpha-variable | literal-string | ALL | BIN }

Purpose:

The ADD operator is used to add a binary value to the binary value of an alpha-variable. ADD may be used only in the alpha-expression portion of an alphanumeric assignment statement. (See the discussion of alpha expressions and the alpha assignment statement in Chapter 5, section 5.8.) The binary value of the operand immediately following the ADD operator is added to the binary value of the receiver-variable, and the result is stored in the receiver-variable. For example, in the statement

```
:10 A$ = ADD B$
```

the binary value of B$ is added to the binary value of A$, and the result is assigned to A$.

The entire value in the receiver-variable, including any trailing spaces, is operated upon. If the operand is an alpha-variable, the entire value of the variable, including trailing spaces, is used. (Note that trailing spaces normally are not considered to be part of the value of an alphanumeric- variable.) Part of an alpha-variable can be operated on by using the STR function to specify a portion of the variable. For example:

```
:10 STR(A$,3,2) = ADD B$
```

operates only on the 3rd and 4th bytes of A$.

If "C" does *not* follow the ADD operator, the addition is carried out on a byte-by-byte basis from right to left with no carry propagation between characters. The last (rightmost) byte of the value of the operand is added to the last (rightmost) byte of the receiver-variable. The next-to-last byte of the operand is then added to the next-to-last byte of the receiver, and addition continues until the entire value in the receiver-variable is operated upon. For example:

```
:5 DIM A$2
:10 A$=HEX(0123)
:20 A$=ADD HEX(00FF)
:30 PRINT "RESULT = ";HEXOF(A$)
:RUN
RESULT = 0122
```

If "C" does follow ADD, the value of the operand is treated as a single binary number and added to the binary value of the receiver-variable with carry propagation between bytes. For example:

```
:5 DIM A$2
:10 A$=HEX(0123)
:20 A$=ADDC HEX(00FF)
:30 PRINT "RESULT = ";HEXOF(A$)
:RUN
RESULT = 0222
```

If the operand and the receiver are not the same length (i.e., if each contains a different number of bytes), the following rules apply:

1. The addition is right-justified, with leading zeros assumed for the shorter value.

2. The result is stored right-justified in the receiver-variable. If the answer is longer than the receiver-variable, the low-order portion of the value is stored, and high-order bytes which cannot be stored in the receiver-variable are truncated.

Examples of Valid Syntax:

```
10 A$=ADD HEX(FF)
20 A$=ADDC ALL(FF)
30 STR(A$,1,2)=B$ ADDC C$
```

DAC Operator

General Form:

receiver = [...] DAC operand [...]

Where:

receiver = alpha-variable [, alpha-variable] . . .

operand = { alpha-variable | literal-string | ALL | BIN }

Purpose:

The Decimal Add with Carry operator, DAC, performs decimal addition on a pair of unsigned packed decimal numbers. The value of the operand is added to the value of the receiver-variable, and the sum is assigned to the receiver-variable. DAC can be used only in the alpha-expression portion of an alphanumeric assignment statement. (See the discussion of alpha expressions and the alpha assignment statement in Chapter 5, section 5.8.)

Note that DAC does *not* check the values of the operands for valid packed decimal format. Since decimal addition with nonpacked decimal values produces undefined results, the VER function should be used to verify that the values of the operands are expressed in valid packed decimal format.

Decimal addition, like binary addition (ADD), is performed from right to left, starting with the low-order (rightmost) bytes of the receiver-variable and operand. Each value is treated as a single decimal number in the addition, and a carry is propagated automatically between bytes in the result.

The entire value of the receiver-variable is operated upon, including trailing spaces; similarly, if the operand is a variable, its entire contents (including trailing spaces) are used.

Example:

```
:10 DIM A$3, B$2
:20 A$=HEX(012345)
:30 B$=HEX(0101)
:40 A$=DAC B$
:50 PRINT HEXOF(A$)
:RUN
 012446
```

Examples of Valid Syntax:

```
10 A$ = DAC HEX(0001)
20 B$ = A$ DAC F$
30 C$ = STR(B$,1,2) DAC A$
```

DSC Operator

General Form:

receiver = [...] DSC operand [...]

Where:

receiver = alpha-variable [,alpha-variable] . . .

operand = { alpha-variable | literal-string | ALL | BIN }

Purpose:

The Decimal Subtract with Carry operator, DSC, performs decimal subtraction on a pair of unsigned packed decimal numbers. The value of the operand is subtracted from the value of the receiver-variable, and the result is assigned to the receiver-variable. DAC can be used only in the alpha-expression portion of an alphanumeric assignment statement. (See the discussion of alpha expressions and the alpha assignment statement in Chapter 5, section 5.8.)

Note that DSC does *not* check the values of the operands for valid packed decimal format. (Any characters other than hex digits 0-9 are illegal in a packed decimal number.) Since decimal subtraction with nonpacked decimal values yields undefined results, the values of the operands should be verified with the VER function.

Decimal subtraction is carried out byte-by-byte from right to left, starting with the rightmost (low-order) bytes of the receiver-variable and the operand. Each value is treated as a single decimal number, with automatic carry propagation between bytes.

The entire value of the receiver-variable is operated upon, including trailing spaces; similarly, if the operand is a variable, its entire value (including trailing spaces) is used.

Example:

```
100 DIM A$2, B$2, C$2
110 PACK (####) A$ FROM 100
120 PACK (####) B$ FROM 75
130 C$ = A$ DSC B$
140 PRINT HEXOF(C$)
:RUN
0025
```

The answer, 25, occupies the low-order byte of the receiver-variable, while the leading zeros indicate that the answer is positive. Suppose, however, that line 130 is changed as follows:

```
130 C$ = B$ DSC A$
```

Now if the program is rerun, the result is the ten's complement notation for -25:

```
:RUN
9975
```

Examples of Valid Syntax:

```
10 A$ = DSC B$
20 B$ = HEX(9999) DSC A$
30 D$ = B$ DSC HEX(01)
```

SUB[C] Operator

General Form:

receiver = [...] SUB[C] operand [...]

Where:

receiver = alpha-variable [,alpha-variable] . . .

operand = { alpha-variable | literal-string | ALL | BIN }

Purpose:

The SUB (Binary SUbtract) and SUBC (Binary SUbtract with Carry) operators perform binary subtraction on a pair of binary values. The binary value of the operand is subtracted from the binary value of the receiver-variable, and the difference is assigned to the receiver-variable.

SUB[C] can be used only in the alpha-expression portion of an alphanumeric assignment statement. (See the discussion of alpha expressions and the alpha assignment statement in Chapter 5, section 5.8.)

SUB treats each byte of the operand and receiver-variable as a separate value. Subtraction is performed on a character-by-character basis from right to left, starting with the rightmost byte of each field; *no* carry is propagated between characters. For example:

```
:10 DIM A$2, B$2, C$2
:20 A$ = HEX(0401)
:30 B$ = HEX(0202)
:40 C$ = A$ SUB B$: PRINT HEXOF (C$)
:RUN
 02FF
```

The SUBC operator treats the value of the operand as a single binary number and subtracts it from the binary number represented by the value of the receiver-variable, with a carry automatically propagated between bytes. (The SUBC operator is analogous to the ADDC operator.) As with SUB, the subtraction proceeds from right to left, starting with the low-order (rightmost) byte of each field. For example:

```
:10 DIM A$2, B$2, C$2
:20 A$ = HEX(0401)
:30 B$ = HEX(0202)
:40 C$ = A$ SUBC B$: PRINT HEXOF(C$)
:RUN
 01FF
```

The entire value of the receiver-variable, including trailing spaces, is operated upon. Similarly, if the operand is a variable, its entire value (including trailing spaces) is used. If only a portion of the value in a variable is to be operated upon, a STR function must be used to define that portion.

If the operand and the receiver are not the same length (i.e., if each contains a different number of bytes), the following rules apply:

1. The addition is right-justified, with leading zeros assumed for the shorter value.
2. The result is stored right-justified in the receiver-variable. If the answer is longer than the receiver-variable, the low-order portion of the value is stored, and high-order bytes which cannot be stored in the receiver-variable are truncated.

Example:

```
:10 DIM A$1, B$1, C$1
:20 A$ = HEX(08)
:30 B$ = HEX(04)
:40 C$ = A$ SUBC B$
:50 PRINT HEXOF(C$)
:RUN
 04
```

In this case, the answer is hex 4 (binary 0100), and the leading hex digit, 0, indicates that it is positive. Now, if line 40 in the program is changed as follows:

```
40 C$ = B$ SUBC A$
```

the result is the two's complement form of -4:

```
:RUN
FC
```

Examples of Valid Syntax:

```
10 A$ = SUB HEX(FF)
20 A$ = B$ SUBC ALL(FF)
30 STR(A$,1,2) = B$ SUBC C$
```

CHAPTER 7 THE SELECT STATEMENT

7.1 INTRODUCTION

The SELECT statement is perhaps the most versatile statement in the BASIC-2 language. SELECT is used to select device-addresses for I/O operations, to specify various options for mathematical operations (including the type of measure to be used in trig functions, whether results are to be rounded or truncated, and what type of response will be made to particular errors), and to select certain parameters for output devices. In addition, the SELECT statement is used to set, clear, and reset interrupts for designated devices. (This function of SELECT is described in Chapter 8.)

Because the SELECT statement has such a wide variety of uses and can accept such a broad range of parameters, it is the principal subject of this chapter. In particular, this chapter discusses:

- The SELECT statement, including its various functions and available parameters.
- The Device Table, a special area of memory used to store device-addresses and other information required to access and control external devices. The Device Table's contents can be modified by means of the SELECT statement.
- The ON...SELECT statement, a conditional SELECT statement in which the selection of specified parameters is determined by the value of a numeric expression or alpha-variable.
- The LIST DT command, a special form of the LIST command used to list the contents of the Device Table.

The SELECT ON/OFF statement, which is used to define interrupts and their priorities, is *not* discussed in this chapter; it is covered in Chapter 8, "Programmable Interrupt Feature."

SELECT

General Form:

SELECT select-parameter [,select-parameter...]

Where:

select-parameter =

```
D
R
G
ERROR [> error-code]
[NO] ROUND

P [digit]
LINE expression

CI device-address
INPUT device-address

CO device-address [(width)]
PRINT device-address [(width)]
LIST device-address [(width)]

PLOT device-address
TAPE device-address
DISK device-address

file-number device-address

ON [device-address [GOSUB line-number]]
ON CLEAR
OFF [device-address [GOSUB line-number]]
```

device-address = { /taa, | <alpha-variable> }

Where

t = one hex digit specifying the device-type

aa = two hex digits specifying the physical device address

alpha-variable = three-byte variable whose value must be three ASCII hexdigits representing the device type and address.

width = an expression < 256 specifying the maximum number of characters on a single line.

file-number = #n, where n = an integer or numeric- variable with a value >= 0 and < 16.

Purpose:
The SELECT statement is used for four purposes:

1. To select the desired math modes for arithmetic operations, including type of measure for trig functions, rounding or truncation of numeric results, and desired system response to specific math errors. (See section 7.2.)

2. To select output parameters for communicating with output devices. (See section 7.3.)

3. To select device-addresses for accessing specified devices with input/output statements and commands. (See section 7.4.)

4. To define, clear, and reset interrupts and their priorities. (See Chapter 8.)

7.2 MATH MODE SELECTION

Specifying Degrees, Radians, or Grads
Degree, radian, or gradian measure (360° = 400 grads) may be selected for the trig functions by using the "D", "R", or "G" select-parameters, respectively. For example, the statement

```
SELECT D
```

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

Selecting Rounding or Truncation
When performing arithmetic operations (+, -, *, /, ↑) and evaluating square roots (SQR function) and modulos (MOD function), results are rounded to 13 significant digits unless a SELECT NO ROUND statement has been executed. SELECT NO ROUND causes results to be truncated to 13 digits. Rounding can be restored by executing a SELECT ROUND statement. When the system is Master Initialized or CLEARed, rounding is automatically selected.

Computational Errors
Computational errors are those produced by the math package when performing an arithmetic operation or evaluating a function. Normally, when a computational error other than underflow occurs, program execution terminates and an error message is displayed. However, if the following statement has been executed:

```
SELECT ERROR > error-code
```

no error message will be displayed and program execution will continue with the values indicated in Table 7-1 below for all math errors whose error-codes are *less than or equal to* the specified error-code. Math errors whose error-codes are greater than the specified code result in program termination and an error message as usual.

Table 7-1.
SELECT ERROR Return Values

Error Code	Error Condition	Value Returned
C60	Underflow.	0
C61	Overflow.	±9.999999999999E+99
C62	Division by zero.	±9.999999999999E+99
C63	Zero divided by zero, or zero raised to the zeroth power.	0
C64	Zero raised to negative power.	+9.999999999999E+99
C65	Negative number raised to noninteger power.	ABS(X)^{Y}
C66	SQR of negative value.	SQR(ABS(X))
C67	LOG of zero.	-9.999999999999E+99
C68	LOG of negative value.	LOG(ABS(X))
C69	Argument too large for trig function.	0

The ERR function is set equal to the code of the last error that occurred. ERR is reset to a value of zero whenever a RUN or CLEAR command is executed, whenever the system is Master Initialized, or whenever ERR is referenced. For example:

```
10 X = ERR
20 IF X < > 0 THEN 100
```

Line 10 sets X = the error-code of the last computational error and then resets ERR to zero. Line 20 branches if an error has occurred.

When the system is Master Initialized or a CLEAR command is executed, all computational errors except underflow automatically cause program execution to terminate with an error message displayed. This condition is equivalent to SELECT ERROR >60 (where 60 is the lowest math error-code). Executing a SELECT ERROR statement with no parameters causes all math errors, including underflow, to terminate execution.

The SELECT ERROR statement and ERR function are further discussed in Chapter 9.

Default Math Modes

When the system is Master Initialized or CLEARed, the following modes of operation are automatically selected:

1. Radian measure for all trig functions.
2. Rounding to 13 significant digits for all math operations.
3. The display of the appropriate error message and halting of program execution if a computational error other than underflow occurs. (Underflow causes the program to proceed with a value of zero for the affected operation.)

7.3 OUTPUT PARAMETER SPECIFICATION

Three special output parameters can be specified in a SELECT statement to control the operation of one or more output devices such as printers, output writers, and the CRT:

- The "P" select-parameter is used to select a pause.
- The "LINE" select-parameter is used to select the maximum number of output lines for an output device.
- The "WIDTH" select-parameter is used to select the maximum line width for an output device in a particular class of output operations.

Selecting a Pause

The "P" select-parameter can be used to cause the system to pause for a specified period each time a carriage return is issued. A pause is most commonly used to slow down the rate at which output lines are displayed on the CRT, enabling the operator to easily read the output. When listing operations are performed on the CRT, the "S" parameter can be specified in a LIST command to automatically halt the listing when the screen is full; in this case, a pause is not necessary. One important use of the pause is during a program TRACE operation. Use of the SELECT P statement allows the operator to read debugging information more easily when TRACing a program.

The optional digit following "P" specifies the length of the pause in increments of 1/6th of a second. For example, the following statements implement the indicated pauses:

SELECT P1	Pause = 1/6 second
SELECT P6	Pause = 1 second
SELECT P0 (or P)	Pause = null

(SELECT P0 and SELECT P are equivalent statements.)

On the VP only, the pause applies to *all* classes of output operations sent to *any* output device. On the MVP, only the console CRT is affected by a SELECT P statement. Once selected, a pause is invoked whenever a carriage return code is issued in Program Mode or Immediate Mode.

A selected pause remains in effect until a different pause is selected or the pause is reset to null by any of the following operations:

- Execution of a SELECT P or SELECT P0 statement.
- Master Initialization of the system.
- RESET.
- Execution of a CLEAR or LOADRUN command.

Selecting the Number of Output Lines

The "LINE" select-parameter is used to specify the maximum number of lines to be displayed or listed on a CRT or printer for Console Output and LIST operations. "LINE" has *no* effect on output produced by a PRINT or PRINTUSING statement executed in Program Mode. In addition, "LINE" can be used to restrict cursor movement to a specified range of lines during INPUT operations.

"LINE" also defines the number of lines to be listed by each execution of a LIST command with the "S" parameter. A LIST S command automatically halts the listing after the number of lines specified by the "LINE" parameter are listed on the CRT or printer. To list the next block of lines, the operator must key RETURN.

When the system is Master Initialized, the "LINE" parameter defaults to the size of the primary CRT (address /005). Thus, for a 16 x 64 CRT, "LINE" defaults to 16; for a 24 x 80 CRT, "LINE" defaults to 24. "LINE" is also reset to the default value following the execution of a CLEAR, LOADRUN, or RESET command.

Selecting the Line Width

The maximum width of an output line for Console Output (CO), PRINT, and LIST operations can be specified with the optional "width" select-parameter. Normally, the system (or, in the case of Console Output, the user) issues a carriage return at the end of each output line. If the line exceeds the maximum line width of the output device, the system automatically issues a carriage return when the device's line width is exceeded. A maximum line width other than that of the device itself can be specified for a particular class of output operations with the "width" parameter. For example, the statement

```
SELECT CO/005 (50)
```

sets the maximum line width of the primary CRT (address /005) to 50 characters for Console Output operations. If a line entered during Console Output is shorter than 50 characters, the line width has no effect since the user issues a carriage return by keying RETURN at the end of the line. If a line has more than 50 characters, however, a carriage return is automatically issued after the 50th character is displayed, and the output continues on the next CRT line.

If the line width set by "width" is *longer* than the maximum line width of a specified output device, the resulting output will differ according to the characteristics of the particular device. For example, the statement

```
SELECT CO/005 (90)
```

sets the CRT line width at 90 characters for Console Output. If a line longer than the maximum line width of the CRT itself (64 or 80 characters, depending upon the model) is entered, the excess characters up to 90 will *wrap around* on the same CRT line. These excess characters will be displayed starting at the first column on the same line, overwriting existing characters on that line. A carriage return is issued only when either 90 characters have been output or the end of the line is reached, whichever comes first.

Most printers automatically generate carriage returns when their maximum line widths are exceeded, irrespective of the selected "width". Thus, in general, the printer automatically continues printing an overlong output line on the next printer line. Since some printers may not follow this procedure, however, widths which exceed the maximum line width of the printer should not be selected.

It must be emphasized that a selected "width" applies *only* to a specified output device when accessed during operations in a particular output class. Thus, the line widths selected with the SELECT CO statements in the examples above are operative only during Console Output operations; they would have no effect on the output of a LIST command or Program Mode PRINT statement. The statements

```
SELECT LIST/215 (100)
```

and

```
SELECT PRINT/005 (60)
```

could be used to specify line widths of 100 and 60 for LIST and PRINT operations, respectively.

It is possible to have different line widths selected for CO, PRINT, and LIST operations concurrently. When this situation occurs, a combination of operations in different output classes (e.g., Console Output, followed by PRINT) may produce unpredictable results, particularly on the CRT.

A line width of 0 has a special significance. If "width" is set to zero, the line width is disregarded, and *no* carriage return is issued until the end of the output line is reached, irrespective of its length. Thus, the statement

```
SELECT PRINT/005 (0)
```

causes the system to continue displaying PRINT output on the same CRT line until the end of the output line is reached or a carriage return code is explicitly issued by the program. The column counter used by the system to keep track of how many characters have been output and where the next character will be placed is *not* updated when a line width of zero is selected. This feature is, therefore, useful for moving the cursor on the CRT without altering the column count. (Note, however, that the TAB and AT functions *cannot* be used when a line width of zero is selected; see the discussion of the PRINT statement in Chapter 11, section 11.2.)

When the system is Master Initialized, the line width for all output classes (CO, PRINT, and LIST) is automatically set to the maximum line width of the primary CRT (address /005). For a 16 x 64 CRT, "width" defaults to 64; for a 24 x 80 CRT, "width" defaults to 80. The line width also is reset to its default value following execution of a CLEAR or LOADRUN command. RESET causes the CI and CO parameters to return to default (Master Initialization) values. CLEAR selects PRINT and LIST to be set to the current CO address and INPUT to be set to the current CI address. (On the MVP, the CI address must always be /001.)

The currently selected line widths for CO, PRINT, and LIST operations are stored in slots assigned to each of these I/O classes in the Device Table. See section 7.5 for a discussion of the Device Table.

7.4 I/O DEVICE SELECTION

A principal function of the SELECT statement is to assign the device-addresses of specified peripheral devices to the various classes of I/O operations.

The Classes of I/O Operations

All I/O operations performed by the system are divided into eight major classes:

1. **CI (Console Input)**

   Keyboard entry of programs, commands, and Immediate Mode lines.

2. **INPUT**

   Operator input for INPUT, LINPUT, and KEYIN statements.

3. **CO (Console Output)**

   Echo of characters entered from keyboard.

   Output from Immediate Mode PRINT and PRINTUSING statements.

   System error messages.

   STOP and END messages.

   TRACE output.

   HALT/STEP Key displays.

   INPUT and LINPUT messages.

4. **PRINT**

   Output from Program Mode PRINT and PRINTUSING statements.

5. **LIST**

   Output from LIST commands.

6. **PLOT**

   Output from PLOT statements.

7. **TAPE**

   Input and output for paper tape, magnetic tape, and certain card-reader operations as well as operations involving interface controllers. Certain I/O operations in this class use the following statements:

DATALOAD	DATASAVE
DATALOAD BT	DATASAVE BT
LOAD	SAVE
$GIO	$IF ON/OFF

8. **DISK**

   Input and output for all disk operations. (See the *2200VP/MVP Disk Reference Manual.*)

Device-Addresses

Each I/O device attached to the system is assigned a unique *device-address.* A device-address is composed of three hexadecimal digits: the first hex digit is called the *device-type* and is used to identify the particular class of I/O devices to which the device belongs. The next two hex digits constitute the *unit device-address,* which is used to electronically select the specified device for operation. A device-address should always be preceded by a slash (/) when specified. For example, the device-address of the Primary CRT is /005; this address is broken down by the system as follows:

```
          0  05
          ↑  ↑
device-type  unit device-address
```

The device-type is used by the system to identify the I/O class to which the device belongs and to specify certain control procedures to be used in communicating with that device. Different types of devices often require different control procedures to perform I/O operations. Device-types are explained in greater detail in section 7.7. The two-digit unit device-address corresponds to the actual address preset in the I/O controller board for each device attached to the CPU.

Although several devices on the system may have the same device-type (for example, the CRT and keyboard both have device-type "0"), every device must have a *unique* unit device-address. An exception, which applies only to the MVP, is that all CRT's, keyboards, and local printers have the same addresses (/005, /001, and /004, respectively). Since each partition can only access one set of these devices, there is no actual duplication; the 2236MXD terminal multiplexer determines which physical device is accessed at any given time.

Selecting Device-Addresses for I/O Operations

When an I/O operation is initiated, the system uses either the default device-address or the specified device-address to determine which I/O device is to be accessed. There are three ways to select a specific device-address for a particular I/O operation:

1. **Default Addresses (Primary I/O Devices)** — The system automatically provides default device-addresses for each class of I/O operations (see Table 7-2 below). The I/O devices associated with these default addresses are referred to as the *primary I/O devices*. If no device-address is selected or specified for a particular I/O operation, the default address for operations in that class is used automatically.

2. **The SELECT Statement** — The SELECT statement can be used to assign device-addresses for specified I/O classes. Once a device-address is selected for a particular I/O class, all operations in that class automatically use the selected address. (In some cases, the selected address can be overridden by a different address specified within the I/O instruction itself; see #3, "Direct Specification of the Device-Address," below.) For example, the statement

   ```
   SELECT LIST /215
   ```

   selects device-address /215 for all LIST operations. Use of the SELECT statement to select device-addresses is further explained in the next section.

3. **Direct Specification of the Device-Address** — Certain I/O instructions in the DISK and TAPE I/O classes permit a device-address to be directly specified in the instruction itself. The specified address always overrides the default address or the device-address selected for that I/O class with a SELECT statement. For example, the statement

   ```
   100 SAVE DCF/320, "PROG1"
   ```

   saves program PROG1 on the disk at address /320, irrespective of which device-address is currently selected for DISK operations.

System Default Device-Addresses

Each system has built-in default addresses for I/O devices in five of the eight I/O classes. These I/O devices are designated the *primary I/O devices* for the system. Whenever the system is Master Initialized, the default device-addresses are automatically selected for I/O operations. The primary I/O devices and their associated default device-addresses are listed in Table 7-2 below.

Table 7-2.
Default Addresses for Primary I/O Devices

I/O Operation Class	I/O Device Class	Primary I/O Device	Default Address
Console Input (CI)	Primary CI Device	Primary Keyboard	/001
INPUT	Primary CI Device	Primary Keyboard	/001
Console Output (CO)	Primary CO Device	Primary CRT	/005
PRINT	Primary CO Device	Primary CRT	/005
LIST	Primary CO Device	Primary CRT	/005
PLOT	Primary PLOT Device	Primary Plotter	/413
TAPE	Primary TAPE Device	Primary Tape Drive	/000
DISK	Primary DISK Device	Primary Disk Drive	/310*

If the system contains no additional I/O devices beyond the primary devices (e.g., if there is only one disk drive), then it is *never* necessary to SELECT device-addresses or specify them explicitly in a DISK or TAPE statement. The system will *automatically* use the default addresses and access the appropriate device for each I/O class. For example, disk operations use the DISK default address, /310, and access the Primary Disk. If additional devices such as a second disk drive or a plotter are added in any I/O class, however, device-address selection or, where possible, direct specification is required to identify which device is to be accessed.

Note that following Master Initialization, the default address for the Primary CI device (/001) is used for INPUT operations, and the default address for the Primary CO device (/005) is used for PRINT and LIST operations.

The Device Table

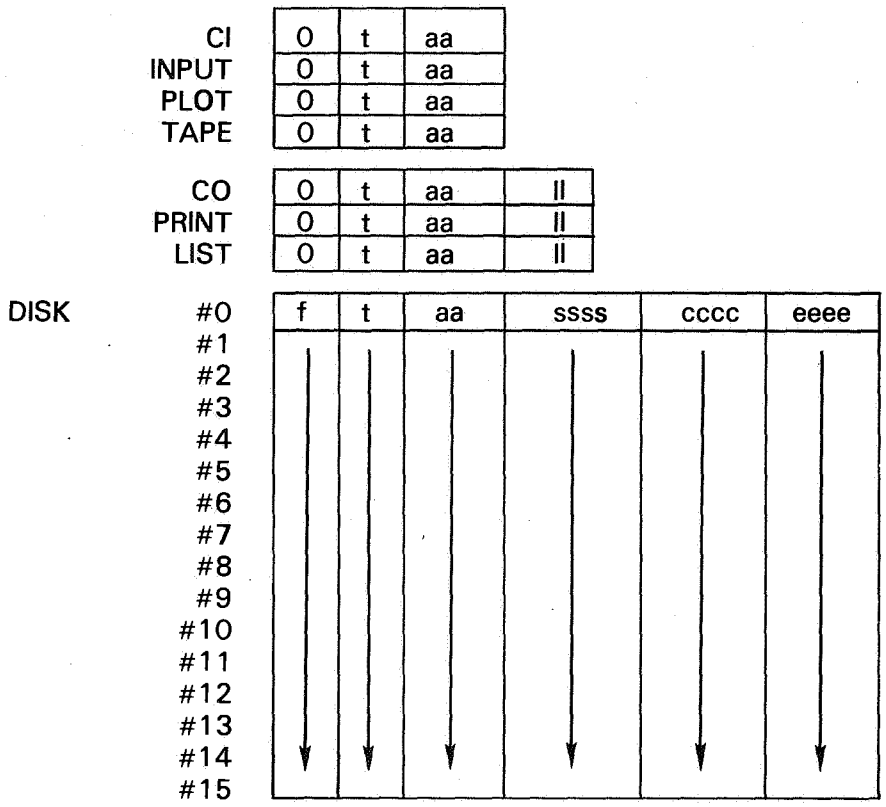
When the system is instructed to perform an I/O operation, the device-address of the device to be used for that operation is obtained from a special section of memory called the *Device Table*. For example, a statement such as

```
100 PRINT "ABCD"
```

would cause the system to check the Device Table for the device-address to be used for PRINT operations. Direct device-address specification in a statement overrides the Device-Table entry for the duration of that statement. In most I/O classes, addresses cannot be supplied in the individual I/O instructions; in these cases, the Device Table is the only source of device-addresses. On the MVP, a separate Device Table is maintained for each partition.

* On the MVP, the default DISK address is the address from which the OS was loaded.

The Device Table is composed of a number of rows or "slots". In general, each I/O class is assigned its own slot in the Device Table; either the currently selected or the default device-address for each I/O class is stored in its Device Table slot, along with certain other information. (The exception to this rule is the DISK I/O class, which has 16 slots in the Device Table. DISK operations use the Device Table to store a variety of information in addition to device-addresses. The uses of the Device Table in DISK operations are discussed in the *2200VP/MVP Disk Reference Manual*.) The format of the Device Table and the information stored in it are shown in Figure 7-1.



Where:
- f = file status
- t = device-type
- aa = unit device-address
- ll = line width
- ssss = starting sector address (disk only)
- cccc = current sector address (disk only)
- eeee = ending sector address (disk only)

Figure 7-1.
The Device Table in Memory, Showing
Format of Entries in Each Slot

When the system is Master Initialized, default addresses are placed in each I/O slot in the Device Table. In addition, default line widths are placed in the CO, PRINT, and LIST slots. The line width of the Primary CRT, address /005, is used as the default width. The remainder of the Device Table is set to zeros. Figure 7-2 shows the Device Table following Master Initialization. Note that device-addresses and line widths are given in hexadecimal, rather than decimal, notation.

	0	t	aa
CI	0	0	01
INPUT	0	0	01
PLOT	0	4	13
TAPE	0	0	00

	0	t	aa	ll
CO	0	0	05	50*
PRINT	0	0	05	50*
LIST	0	0	05	50*

		f	t	aa	ssss	cccc	eeee
DISK	#0	0**	3	10	0000	0000	0000
	#1	0	0	00	0000	0000	0000
	#2	0	0	00	0000	0000	0000
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	#15	0	0	00	0000	0000	0000

Figure 7-2.
The Device Table in Memory Following Master Initialization

Operations in each I/O class refer to the corresponding slot for that class in the Device Table. Thus, for example, Console Input operations use the CI slot to obtain a device-address, and PRINT operations use the PRINT slot to obtain a device-address and line width.

> **NOTE:**
>
> PRINT class operations include only the *Program Mode* execution of PRINT and PRINTUSING statements. Immediate Mode PRINT and PRINTUSING statements belong to the Console Output class and use the CO slot rather than the PRINT slot to obtain the device-address and line width.

* The line width is set to the width of the Primary CRT (address /005). If the Primary CRT is 24x80, line width = 80 (hex 50). If the Primary CRT is 16x64, line width = 64 (hex 40).

** On the 2200MVP, DISK is initially selected to the platter from which the Operating System was loaded.

Modifying Device Table Entries

Entries in the Device Table can be modified *explicitly* with a SELECT statement or *implicitly* with one of the following operations:

- Master Initialization.
- CLEAR and LOADRUN commands.
- RESET.

The use of the SELECT statement to explicitly change device-address and line-width parameters in a Device Table slot is covered in section 7.5. The methods used to implicitly modify the values in the Device Table are described in section 7.6.

7.5 EXPLICIT DEVICE TABLE MODIFICATION

Device Table entries for all I/O classes can be modified explicitly by executing a SELECT statement which specifies the I/O class or classes whose entries are to be modified and the new values for those entries. Note, however, that before a class of I/O operations can be assigned explicitly to a new device, the device-address of the new device must be known (see Appendix D).

On the MVP, the addresses of all devices to be used (other than the terminal which always reserves addresses /001, /005, and /004) must be specified in the Master Device Table at partition generation time (see Chapter 16, section 16.4). Attempting to access a device not so defined will result in error P48.

For example, to change the Console Output device from the Primary Console Output device (the Primary CRT, address /005) to another output device, a statement of the following form is used:

SELECT CO device-address (width)

An example of such a statement is:

SELECT CO/215(132)

This statement selects a line printer (device-address = /215) as the new Console Output device. The maximum line width to be used on the printer is set at 132 characters. Following execution of this statement, the CO slot in the Device Table looks like this:

	0	t	aa	ll
CO	0	2	15	84*

Figure 7-3.
The CO Slot in the Device Table Following Execution
of a SELECT CO/215 (132) Statement

The CRT can be reselected as the Console Output device with a statement of the form:

SELECT CO/005 (80)

This statement reselects the CRT as the Console Output device and resets the line width to 80 characters. Following execution of this statement, the CO slot in the Device Table contains the values shown in Figure 7-2.

*Hex 84 is equivalent to decimal 132.

NOTE:

If a line width is not specified for Console Output, PRINT, or LIST operations, the last line widths selected for these operations are used. Master Initialization sets the line widths to 80 characters for 24x80 CRT's or 64 characters for 16x64 CRT's.

The CI (Console Input) Select-Parameter

The CI select-parameter specifies the device-address to be used for all Console Input operations. Console Input includes the entry and editing of program text and Immediate Mode lines as well as system commands such as LOAD, CLEAR, and RUN. On the MVP, /001 is the only permissible CI address. Characters entered during a Console Input operation are automatically echoed to the currently selected Console Output device. The statement

```
SELECT CI/002
```

selects the keyboard with address /002 for Console Input operations by placing address /002 in the CI slot in the Device Table. Note that once a device other than the Primary Keyboard (address /001) is selected for Console Input, the Primary Keyboard *cannot* be used to enter program text or Immediate Mode lines or, more importantly, to issue any system commands.

The INPUT Select-Parameter

The INPUT select-parameter specifies the device-address to be used to enter data for INPUT, LINPUT, and KEYIN statements. For 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, and the values of X and Y must be keyed in on the keyboard selected for INPUT operations (address /002). INPUT and LINPUT on the MVP are legal only from the terminal keyboard (/001).

The CO (Console Output) Select-Parameter

The CO select-parameter is used to specify the device-address to be used for all Console Output operations and, optionally, to specify the line width for that device. Console Output includes output produced by the following operations:

- Echo of characters entered in Console Input operations from the CI device.
- Echo of characters entered in response to INPUT, LINPUT, and KEYIN requests from the INPUT device.
- Messages produced by the INPUT and LINPUT statements.
- Output of Immediate Mode PRINT and PRINTUSING statements.
- Output of TRACE operations.
- Output of HALT/STEP operations.

- System-generated error messages.
- STOP and END messages.

For example, the statement

```
SELECT CO/215 (120)
```

selects the printer with address /215 for Console Output operations by placing address /215 in the CO slot in the Device Table. The maximum line width is set at 120 characters.

All CO operations on the MVP except TRACE are always directed to the Primary CRT (/005), which has a line width of 80 characters. TRACE can be selected to output to another device (e.g., a printer).

The PRINT Select-Parameter

The PRINT select-parameter designates the output device on which all program output from PRINT and PRINTUSING statements is displayed and, optionally, specifies the maximum line width to be used for that device. For 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 printer with device-address /213 by placing address /213 in the PRINT slot in the Device Table. The line width is specified as 100 characters. The CRT can be reselected for programmed PRINT output with the following statement:

```
SELECT PRINT/005(80)
```

This statement reselects the CRT (address /005) as the device to which all PRINT and PRINTUSING output is directed and sets the maximum line width to 80 characters.

> **NOTE:**
>
> The output from PRINT and PRINTUSING statements executed in the Immediate Mode always appears on the Console Output device.

The LIST Select-Parameter

The LIST select-parameter designates the output device to be used for all program and cross-reference listings and, optionally, specifies the maximum line width to be used for LIST operations on that device. For example, the statement

```
SELECT LIST/215(132)
```

specifies that a printer (device-address = /215) is to be used for all listings by placing address /215 in the LIST slot in the Device Table. The maximum line width is specified as 132 characters.

The PLOT Select-Parameter

The PLOT select-parameter specifies the device-address to be used for output from PLOT statements. For example, the statement

```
SELECT PLOT /C13
```

selects the plotter at address /C13 for PLOT output. Note that the plotter with address /C13 is actually the Primary Plotter since /C13 has the same unit device-address component (13) as the default device-address /413. The "C" device-type used in place of the standard "4" device-type specifies the use of special control procedures which utilize binary plot vectors to position the plotter. Binary plot vectors can substantially increase the overall speed of a plotting operation on plotters which support them. In this example, the SELECT statement is used not to select a different output device for PLOT operations, but to specify a different control procedure to be used in plotting on the Primary Plotter.

The TAPE Select-Parameter

The TAPE select-parameter specifies the device-address to be used by any TAPE-class instruction which does not specify a device-address or file-number in the instruction itself. The TAPE class of I/O operations includes input and output operations initiated by the following instructions:

DATALOAD	$GIO
DATALOAD BT	$IF ON/OFF
DATASAVE	LOAD (other than from disk)
DATASAVE BT	SAVE (other than from disk)

These instructions are used to control a variety of I/O devices, including paper tape readers, nine-track magnetic tape drives, card readers, special interface controllers, and communications controllers. For example, the statement

```
100 SELECT TAPE/07B
```

selects the magnetic tape drive with address /07B for all I/O operations initiated by any of the above instructions by placing address /07B in the TAPE slot in the Device Table.

Note that TAPE-class instructions can specify a device-address directly as part of the instruction. In this case, the directly specified device-address always overrides the device-address stored in the TAPE Device Table slot. TAPE-class instructions can also specify a device-address *indirectly* by referencing a file-number from #0 to #15. In this case, the device-address stored in the Device Table slot identified by the corresponding file-number is used for the I/O operation. TAPE-class instructions used to control specific peripheral devices are, in general, discussed in the reference manuals for the individual devices. Exceptions to this rule are the $GIO and $IF ON/OFF statements, which are discussed in Chapter 15 of this manual. The use of file-numbers to reference device-addresses for DISK and TAPE instructions is described in the subsection "The 'File-Number' Select-Parameter."

The DISK Select-Parameter

The DISK select-parameter specifies the device-address to be used in any DISK instruction which does not directly specify a device-address or file-number. The complete DISK instruction set is documented in the *2200VP/MVP Disk Reference Manual.* For example, the statement

```
SELECT DISK /320
```

selects the disk drive with address /320 as the Primary Disk. The disk drive is then used automatically by any DISK instruction which does not specify a device-address or file-number. The above SELECT statement places address /320 in the DISK slot in the Device Table. Because the DISK slot is also

identified as the #0 slot in the Device Table, the statements are equivalent.

```
SELECT DISK/320
```

and

```
SELECT #0/320
```

To manipulate disk files with greater flexibility, the system provides two additional methods of identifying a desired disk unit in a DISK instruction. Many DISK instructions can specify the device-address directly within the instruction. This method of disk device-specification is independent of the Device Table. For example, the command

```
LIST DCT/350
```

lists the Catalog Index of the disk whose device-address is /350, *irrespective* of the address stored in the DISK Device Table slot.

A second method of specifying the device-address in a DISK instruction involves the use of file-numbers. When a file-number is included in a DISK instruction, the device-address stored in the Device Table slot identified by that file-number is used for the disk I/O operation. For example, the statement

```
100 LOAD DC T #5, "PAYROLL"
```

would use the device-address stored opposite file-number #5 in the Device Table to load the program "PAYROLL". This method is discussed in the following subsection, "The 'File-Number' Select-Parameter."

Note that the specification of a device-address or file-number directly in a DISK instruction does *not* alter the contents of the Device Table.

The "File-Number" Select-Parameter

In addition to individual slots for the various I/O classes, the Device Table also contains sixteen slots identified with the file-numbers #0 through #15 (see Figure 7-1). The #0 slot, which is the DISK I/O slot, contains the default address for the Primary Disk Drive and is generally used only by DISK instructions. The remaining slots (#1 — #15) may be used by both DISK and TAPE instructions, although their most common use is in disk operations.

Each file-number slot can contain, in addition to a device-address, several items of information not found in any of the other I/O slots (refer to Figure 7-1). This information includes:

- A file-status parameter, used in disk operations to report the status of a cataloged data file.
- Three sector-address parameters, used by the system to locate a cataloged disk file and point to the specified location within the file.

These special disk parameters are explained in the *2200VP/MVP Disk Reference Manual*. For the present discussion of the SELECT statement, only the device-address parameter in the file-number slot is relevant.

A SELECT statement can be used to store a device-address in one of the file-number slots in the Device Table exactly as it is used to store device-addresses in the other I/O slots. Thus, the statement

```
100 SELECT #3/320
```

places the device-address /320 in the #3 slot in the Device Table. Subsequently, a DISK instruction which references file-number #3 will use address /320 for its I/O operation. For example, the statement

```
150 DATALOAD DC OPEN F #3, "FILE"
```

opens the data file named "FILE" on the "F" platter of the disk unit whose address is /320.

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

```
10 SELECT #2/310, #3/330
20 DATALOAD DC OPEN T #2, "OLDFILE"
.
.
.
110 DATALOAD DC OPEN T #3, "NEW FILE"
```

Multiple Select-Parameters in a Single SELECT Statement

As line 10 in the foregoing example illustrates, it is legal to specify multiple select-parameters in one SELECT statement. Any combination of select-parameters may be specified in the same SELECT statement; the only provision is that individual parameters must be separated by commas. For example, the statement

```
50 SELECT D, NO ROUND, CO/215, #5/350
```

performs all of the following operations:

1. Selects degree measure for trig functions.
2. Selects truncation rather than rounding of numeric results.
3. Selects device-address /215 for Console Output operations.
4. Assigns device-address /350 to the #5 slot in the Device Table.

A series of select-parameters separated by commas as in statement 50 above is referred to as a *select-list*. The concept of a select-list is further explained in section 7.8, "Conditional Selection of Select-Parameters and Listing the Device Table."

7.6 IMPLICIT DEVICE TABLE MODIFICATION

The SELECT statement is used when the programmer wishes to *explicitly* modify one or more slots in the Device Table. The programmer should be aware, however, that there are certain system operations which modify entries in the Device Table as part of a more general function. This type of modification is referred to as *implicit* Device Table modification. It is performed whenever the system is Master Initialized, RESET is pressed, or a CLEAR or LOADRUN command is executed. Because each of these operations causes the Device Table to be modified in a *different* way, the programmer should be familiar with the specific effects of each operation.

Master Initialization

Master Initialization of the system clears the entire contents of the Device Table and then performs the following operations:

1. Sets the device-address in each I/O slot to the system default address for that I/O class (i.e., the address of the primary device for that class). The DISK default address is placed in the #0 slot.
2. Sets the line widths in the CO, PRINT, and LIST slots to the line width of the Primary CRT (address /005).
3. Sets the remainder of the #0 slot and all items in the #1 — #15 slots to zeros.

RESET

Whenever RESET is keyed, the CO and CI Device Table entries are automatically reset to the system default addresses for those classes. The following modifications are made to the Device Table:

1. The CO entry is set to the system default address for Console Output (/005). The line width is set to the line width of the Primary CRT (address /005).

2. The CI entry is set to the system default address for Console Input (/001).

3. LINE and "width" are set to the number of lines on the Primary CRT (address /005).

No other Device Table slots are affected by keying RESET.

CLEAR and LOADRUN

When a CLEAR command with *no* parameters is executed, the current CO device is selected for all character output operations, and the current CI device is selected for all character input operations. Note that the CO and CI entries themselves are *not* modified. (Because the LOADRUN command automatically issues a CLEAR command as part of its execution, its effect is identical to that of the CLEAR command.) The following modifications are made to the Device Table:

1. PRINT and LIST device-address entries are set to the current CO device-address. PRINT and LIST line widths are set to the current CO line width.

2. The INPUT device-address entry is set to the current CI device-address.

3. The #0 slot is zeroed out except for the DISK device-address, which is not altered.

4. The #1 — #15 slots are completely zeroed out.

Note that CLEAR does *not* affect the entries for CI, CO, PLOT, or TAPE, nor does it change the current address for DISK.

7.7 DEVICE-TYPES

The device-type digit of the device-address is used by the system to identify what type of device is being selected for an I/O operation. Since the various peripheral devices available on the system often require different control procedures to perform an input/output operation, the programmer must indicate to the system which type of I/O device is being used. The device-type digit informs the Operating System of the I/O class of the peripheral device, enabling the system to utilize the appropriate procedure during I/O.

For character printing and displaying operations (PRINT, LIST, and Console Output), the following device-types are normally used:

Type 0* = outputs a line-feed character (hex 0A) after each carriage return character (hex 0D) is output.

Type 2 = outputs a null character (hex 00) after each carriage return character (hex 0D) is output.

Type 7 = does not output any extra character after each carriage return character (hex 0D) is output.

*Type 4 can also be used for character output. It is equivalent to type 0 except that the automatic carriage return normally issued at the end of a line is suppressed.

For other I/O operations, the following device-types are generally used:

Type 0 = Character input (Console Input, INPUT, LINPUT, KEYIN)

Type 3,B,D = Disk operations

Type 4 = PLOT output

Type C = Vectored PLOT operations*

Type 6 = Paper tape reader and certain card reader operations.

7.8 CONDITIONAL SELECTION OF SELECT-PARAMETERS AND LISTING THE DEVICE TABLE

The ability to conditionally select a particular group of select-parameters during program execution can be extremely useful in many applications. This process involves the selection of a specified set of select-parameters from several alternatives. A response entered by the operator or a value computed by the program determines which set of parameters is selected. Conditional selection of select-parameters can be accomplished by writing several SELECT statements on different program lines and then using an ON/GOSUB or ON/GOTO statement to conditionally branch to a desired line. A more efficient and self-documenting technique, however, is provided by the ON/SELECT statement. The general form of the ON/SELECT statement is shown in section 7.9.

The ON/SELECT statement uses a numeric or alphanumeric value (the "select-value") to select a particular set of select-parameters, called a "select-list", from several specified select-lists. The select-value is used as an index to determine which of the specified select-lists will be selected. For example, in the statement

```
70 ON I SELECT #1/310; #1/B10; #1/350
```

the numeric-variable I contains the select-value, while #1/310, #1/B10, and #1/350 are the select-lists. In this case, if I=1, then SELECT #1/310 is executed; if I=2, then SELECT #1/B10 is executed; and if I=3, then SELECT #1/350 is executed. If I has a value other than 1, 2, or 3, *nothing* is selected (i.e., the Device Table is unchanged).

In this example, each select-list is composed of one select-parameter. As in a standard SELECT statement, however, multiple select-parameters may be specified in a select-list. When this situation occurs, the individual select-parameters within a select-list must be separated by commas. The select-lists themselves are separated by semicolons. The following distinction must be emphasized: "Multiple select-lists within an ON/SELECT statement are separated by semicolons, and individual select-parameters within a select-list are separated by commas." For example, the statement

```
100 ON J SELECT CO/005, PRINT/005, LIST/005; CO/215,
PRINT/215, LIST/215; CO/005, PRINT/215, LIST/216
```

contains three select-lists, each consisting of three select-parameters. In this case, if J=1, then CO, PRINT, and LIST operations are assigned to address /005; if J=2, CO, PRINT, and LIST are assigned to /215; and if J=3, CO is assigned to /005, PRINT is assigned to /215, and LIST is assigned to /216. For any other value of J, no selection is made.

The "null select-list" is a special select-list containing no select-parameters. It is defined by leading and trailing semicolons with nothing (or spaces) between them. The null select-list is used when no selection is to be made for a particular select-value. For example, the statement

```
200 ON I SELECT PRINT/005;;PRINT/215
```

*This device-type provides faster plotting than device-type 4.

selects PRINT operations to address /005 if I = 1 or to address /215 if I = 3. If I = 2 or if I is outside the range 1 — 3, no selection is made.

Note that if the select-value is a numeric expression, it is truncated before the ON/SELECT statement is evaluated. It is also possible to use an alphanumeric expression as a select-value; for a discussion of alpha select-values, see the discussion entitled "Advanced Use of the ON/SELECT Statement" in section 7.9.

7.9 GENERAL FORMS OF THE ON/SELECT AND LIST DT STATEMENTS

This section contains the general forms of both the ON/SELECT statement and the LIST DT form of the LIST command.

The contents of the Device Table can be listed on the CRT or an output device with the LIST DT command. LIST DT lists the contents of all Device Table slots in hexadecimal notation. The general form of the LIST DT command and the format of the Device Table listing are described in this section.

The use of the ON/SELECT statement is discussed in section 7.8.

ON/SELECT

General Form:

ON select-value SELECT select-list [; [select-list]]

Where:

select-value = { expression | alpha-variable }

select-list = select-parameter [,select-parameter. . .]

Purpose:

The ON/SELECT statement is a conditional SELECT statement in which the specific select-parameter assignment(s) made are determined by the value of an expression or alpha-variable, called the "select-value". The select-parameter assignments specified in the "nth" select-list are made, where "n" is the integer portion of the select-value. For example, the statement

```
10 ON X SELECT #0/310; #0/B10; #0/320
```

selects #0 to /310 when X=1, to /B10 when X=2, and to /320 when X=3. If the truncated value of the expression is less than one or greater than the number of select-lists, no assignments are made.

Each select-list consists of one or more select-parameters separated by commas (individual select-lists are separated by semicolons). When a select-list is selected, all the select-parameter assignments specified within that select-list are made. For example, the statement

```
10 ON X SELECT CO/215,PRINT/215,LIST/215; CO/005,
PRINT/005,LIST/005
```

selects CO, PRINT, and LIST operations to a line printer (address /215) if X=1. If X=2, these three output operations are selected to the CRT (address /005).

Null select-parameters in the ON statement imply that nothing is to be selected for the associated select-value. For example:

```
10 ON I SELECT PRINT/005;; PRINT/215
```

selects PRINT to address /005 if I=1 and PRINT to address /215 if I=3; nothing is selected if I=2 (that is, the current device specification for PRINT remains in effect).

Advanced Use of the ON/SELECT Statement

The select-value may be specified by an alpha-variable as well as by a numeric expression. If an alpha-variable is used, the binary value of the first character in the alpha-variable is used as the select-value. For example, if A$=HEX(02), execution of the statement

```
10 ON A$ SELECT DISK/310; DISK/320; DISK/330
```

would select DISK to address /320.

Examples of Valid Syntax:

```
10 ON I SELECT PRINT/005; PRINT/215
20 ON A$ SELECT R;D;G
30 ON (I+1)/2 SELECT #0/310; #0/320; #0/330
40 ON X SELECT CO/005,PRINT/005,#0/310;PRINT/215, LIST/215, #0/B10
```

LIST DT

General Form:

LIST [S] [title] DT

Where:

title = { literal-string / alpha-variable }

Purpose:

LIST DT displays the contents of the Device Table in hexadecimal notation. This statement is used primarily for debugging programs which reference the Device Table. The table is displayed in the following format:

```
CI          INPUT       PLOT    TAPE
0taa        0taa        0taa    0taa

 CO          PRINT       LIST
0taall00    0taall00    0taall00

# 0 ftaasssscccceeee   # 1 ftaasssscccceeee
# 2 ftaasssscccceeee   # 3 ftaasssscccceeee
# 4 ftaasssscccceeee   # 5 ftaasssscccceeee
# 6 ftaasssscccceeee   # 7 ftaasssscccceeee
# 8 ftaasssscccceeee   # 9 ftaasssscccceeee
#10 ftaasssscccceeee   #11 ftaasssscccceeee
#12 ftaasssscccceeee   #13 ftaasssscccceeee
#14 ftaasssscccceeee   #15 ftaasssscccceeee
```

Where:

0	=	nothing (filler)
f	=	file-status parameter (0 = data file not open, 1 = data file open on "F" platter, 2 = data file open on "R" platter)
t	=	device-type
aa	=	unit device-address
ll	=	line width
ssss	=	starting sector address*
cccc	=	current sector address*
eeee	=	ending sector address*

Note that the headings shown in the general format are not actually displayed in the listing (see the following example).

*For disk operations only (see the *2200VP/MVP Disk Reference Manual*).

On the MVP, LIST DT lists the Master Device Table according to the partitions' current device selections. Each entry in the Table is displayed as follows:

```
taa [-ppx]
```

Where:

- t = device-type (3 if disk; otherwise, zero)
- aa = physical (hardware) device-address
- pp = number of the partition using the device
- x = X if device is open exclusively for partition pp.
  0 if device is open by partition pp; otherwise, x equals a space.

The "S" and "title" parameters are common to all forms of the LIST command. Refer to the discussion of LIST in Chapter 10, section 10.2 for an explanation of their functions.

Example: LISTing the Contents of the Device Table

```
:LIST DT

0001 0001 0413 010A

00054000 00054000 02154000

031002AC03F3045D 0B10106511EC13AA
0320000000000000 0000000000000000
0000000000000000 0000000000000000
0000000000000000 0000000000000000
0000000000000000 0000000000000000
0000000000000000 0000000000000000
0000000000000000 0000000000000000
0000000000000000 0000000000000000
```

CHAPTER 8 PROGRAMMABLE INTERRUPT FEATURE

8.1 INTRODUCTION

The BASIC-2 language provides a limited I/O interrupt handling capability. The programmable interrupt automatically transfers program control to an interrupt processing subroutine when an interrupt condition occurs and subsequently returns control to the interruption point in the main program after the interrupt subroutine processing is finished. (The interrupt subroutine must conclude with a RETURN statement.) Because different devices can be assigned different priorities in the interrupt scheme, this type of interrupt is often called a "priority interrupt"; it is extremely useful for real-time instrument control as well as in applications where temporary interruption of program processing is required for operator inquiry or maximum printer throughput. However, because of the division of processing time among MVP partitions, the interrupt handling capability has only limited use on the MVP.

An interrupt condition is signalled from an I/O device when the device READY/BUSY condition is READY. While normal program processing continues, the defined interrupt devices are constantly polled for a READY signal. If a READY signal is received at one of the specified device-addresses, program execution is halted as soon as the currently executing statement is finished. (A program may be interrupted only upon completion of statement execution.) The current program location is saved by the system, and a subroutine branch is made to a specified line-number. The interrupt subroutine which begins at this line then performs the necessary processing. At the completion of subroutine processing, a RETURN statement sends the system back to the statement immediately following the last statement executed before the interrupt occurred, and main processing resumes at that point.

8.2 INTERRUPT PROGRAMMING

The four special forms of the SELECT statement which are used for interrupt control are listed below:

1. Interrupt/Priority Definition

   SELECT ON/device-address GOSUB line-number
   SELECT OFF/device-address GOSUB line-number

2. Enabling and Disabling Individual Interrupts

   SELECT ON/device-address
   SELECT OFF/device-address

3. General Inhibition and Reactivation of All Currently Enabled Interrupts

   SELECT ON
   SELECT OFF

4. Clearing All Currently Defined Interrupt Information for Redefinition

   SELECT ON CLEAR

The general form of the SELECT ON/OFF statement is shown in section 8.10.

A special form of the LIST statement also is provided to enable the programmer to examine the contents of the Interrupt Table at any point:

LIST I

The general form of the LIST I command appears in section 8.10.

8.3 INTERRUPT/PRIORITY DEFINITION

Interrupt processing for up to eight different devices is maintained in an internal system table called the Interrupt Table. Information stored for each interrupt includes the device-address of the device which is to initiate the interrupt, a subroutine line-number which specifies the starting point of the interrupt processing routine, and status information indicating whether the interrupt is currently active or not. Interrupt priority is established by the order in which the interrupts are defined in a program (the first interrupt defined has the highest priority, the second interrupt defined has the second-highest priority, etc.). For example:

```
10 SELECT OFF/017 GOSUB 100, ON/018 GOSUB 200, OFF/019 GOSUB 300
```

Statement 10 defines three interrupts for addresses /017, /018, and /019. The interrupt defined for address /017 has the highest priority, the interrupt defined for address /018 has the next- highest priority, and the interrupt defined for address /019 has the lowest priority. Also defined are the subroutine branch addresses associated with each interrupt (100, 200, and 300). If a READY signal is received from the device at address /018, for example, program execution continues at line 200 following completion of the current statement.

The initial status of an interrupt (either enabled or disabled) is defined by using the ON and OFF parameters. In statement 10 above, for example, the interrupt defined for device-address /017 is inactive. A READY signal at address /017 will not initiate an interrupt, although an interrupt is defined for that address. The second interrupt defined in statement 10 is for address /018; this interrupt is active, and a READY condition at address /018 will initiate an interrupt.

Once an interrupt has been defined, a second SELECT statement can be executed later in the program to change the interrupt subroutine address. For example, the subroutine address for device /018, originally defined as 200 in statement 10 above, can be changed with the following statement:

```
50 SELECT ON/018 GOSUB 400
```

An interrupt initiated at address /018 now will cause a branch to line 400. Note that changing the subroutine address in this way does *not* alter the originally defined interrupt priority.

8.4 ENABLING AND DISABLING INDIVIDUAL INTERRUPTS

Once an interrupt has been defined within a program, it may be enabled or disabled at any time by specifying its associated device-address in a SELECT ON or SELECT OFF statement. For example, the interrupt defined for address /017 in statement 10 above initially was disabled; it may be enabled at any subsequent point in the program with the following statement:

```
80 SELECT ON/017
```

Similarly, address /018, which initially was enabled in statement 10, may subsequently be disabled with the following statement:

```
90 SELECT OFF/018
```

Individual interrupts are referenced via their associated device-addresses. Once enabled with a SELECT ON statement, an interrupt will occur when its associated device becomes READY. Note that when a defined interrupt is enabled or reenabled at any point in a program, it automatically assumes its originally defined priority.

8.5 GENERAL INHIBITION AND REACTIVATION OF ALL CURRENTLY ENABLED INTERRUPTS

It is often useful to temporarily inhibit all currently enabled interrupts in a program. Such a suspension of interrupt capability is particularly desirable when variables, pointers, tables, or data records must be initialized or adjusted without being dynamically changed by interrupt routines during the process. A SELECT ON or SELECT OFF statement with no device-address specified will reactivate or inhibit all currently enabled interrupts:

```
SELECT OFF  (Inhibits all currently enabled interrupts)
SELECT ON   (Reactivates all currently enabled interrupts)
```

The general form of interrupt inhibit is executed independently of individual interrupt inhibits (i.e., SELECT OFF/XYY). Therefore, when the general interrupt inhibit is removed by SELECT ON, each individual interrupt assumes its previously defined status (ON or OFF).

8.6 CLEARING ALL CURRENTLY DEFINED INTERRUPT INFORMATION FOR REDEFINITION

Current interrupt information is automatically cleared from the Interrupt Table by system operations such as Master Initialization, CLEAR, RUN, RENUMBER, RESET, and program modification. However, it is also desirable to have a programmable capability to clear current interrupt information. This capability can be used either as a part of a program initialization procedure or to redefine interrupt priorities during program execution. The following statement is provided to clear all currently defined interrupt logic:

```
SELECT ON CLEAR
```

Following this statement, interrupts can be completely redefined by SELECT ON/OFF/taa GOSUB ... statements.

8.7 LISTING INTERRUPT STATUS

The LIST I command is provided to list the current contents of the Interrupt Table. Examination of the contents of the Interrupt Table can prove useful for program debugging. Information provided by LIST I includes the condition of the General Interrupt Inhibit/Enable, the currently executing subroutine if the system is currently processing an interrupt, and the status of each defined interrupt in order of priority. The general form of the LIST I command and the format of its output are shown in section 8.10.

8.8 INTERRUPT PROCESSING

When an interrupt condition (device READY for an interrupt-enabled device) occurs, program control is transferred to the interrupt processing subroutine specified in the corresponding SELECT ON/OFF /taa GOSUB... statement upon completion of current statement execution. Note that execution of a BASIC statement cannot be interrupted. If, for example, the system is awaiting operator entry in response to an INPUT request or is performing a disk MOVE or COPY operation, all interrupt activity is suspended until the statement completes execution. For this reason, the programmer must exercise care in designing a program to handle a device which requires immediate response. (KEYIN loops or single-character LINPUT should be used instead of INPUT, for example.)

The interrupt subroutine itself cannot be interrupted; all interrupt processing is suspended until a RETURN or RETURN CLEAR from the interrupt subroutine to the main program is executed. (The interrupt routine can, however, call subroutines.) SELECT ON statements executed in an interrupt subroutine do not take effect until the RETURN or RETURN CLEAR from the interrupt subroutine is executed. RETURN CLEAR removes the return address to the main program and terminates the subroutine without branching back to the main program. In general, all interrupt control SELECT statements can be legally executed within an interrupt subroutine. Since, however, no interrupts can take

effect until a RETURN from the interrupt subroutine is executed, interrupts made active are not effectively active until the RETURN occurs.

If more than one device interrupt is enabled, interrupt conditions are checked in the order in which the SELECT ON/OFF /taa GOSUB... parameters were originally executed. Thus, the device specified in the first SELECT ON/OFF /taa GOSUB... statement executed has the highest priority. Executing a SELECT OFF /taa statement and then reenabling the interrupt with another SELECT ON /taa statement does not change the priority of the device.

Interrupt priority can be changed by clearing all current SELECT ON/OFF /taa GOSUB... information from the system with a SELECT ON CLEAR statement and then reexecuting the SELECT ON/OFF /taa GOSUB... statement in the desired priority order. For example:

```
10 REM INITIAL PRIORITY (/017,/018,/019)
20 SELECT ON/017 GOSUB 100, ON/018 GOSUB 200, ON/019 GOSUB 300
.
.
.
80 REM CHANGE PRIORITY TO (/018,/019,/017)
85 SELECT ON CLEAR
90 SELECT ON/018 GOSUB 200, ON/019 GOSUB 300, ON/017 GOSUB 100
```

The SELECT ON CLEAR statement turns off all interrupt processing. Interrupts are not processed again until an appropriate SELECT ON/OFF /taa GOSUB... statement is executed.

Programming Example (VP only):

```
10  REM  INTERRUPT DEMONSTRATION
20  REM
30  REM  INTERRUPT DEFINITION.....
40  SELECT ON /001 GOSUB 100: REM KEYBOARD INTERRUPT ACTIVE
50  SELECT OFF/005 GOSUB 200: REM CRT INTERRUPT INACTIVE
60  REM .....
70  GOTO 70
72  REM    PROGRAM LOOPS AT LINE 70 UNTIL KEY IS PRESSED ON
74  REM    KEYBOARD. KEYBOARD INTERRUPT CAUSES SUBROUTINE
75  REM    BRANCH TO BE MADE TO LINE 100. THE INTERRUPT SUB-
76  REM    ROUTINE INPUTS THE CHARACTER FROM THE KEYBOARD,
77  REM    ENABLES THE CRT INTERRUPT SO THAT THE CHARACTER
78  REM    WILL BE DISPLAYED, AND RETURNS TO LINE 70. THE CRT
79  REM    INTERRUPTS LINE 70 AND DISPLAYS THE CHARACTER.
80  REM
100 REM KEYBOARD INTERRUPT SUBROUTINE (DEVICE /001) .....
110 KEYIN A$:          REM GET CHARACTER FROM KEYBOARD
120 SELECT ON /005:    REM ACTIVATE CRT INTERRUPT
130 RETURN:            REM RETURN TO LINE 70
140 REM
200 REM CRT INTERRUPT SUBROUTINE (DEVICE /005).....
210 PRINT STR(A$,,1);: REM DISPLAY CHARACTER
220 SELECT OFF/005:    REM TURN OFF CRT INTERRUPT
230 RETURN:            REM RETURN TO LINE 70
```

NOTE:

The CRT is generally too fast to require interrupt processing within a program; in a practical sense, it is always READY. Subroutine 200 in the example above is therefore meaningful only for purposes of illustration.

8.9 CONDITIONAL DEFINITION AND ENABLING OF INTERRUPTS(ON/SELECT)

The ON/OFF parameters and all associated interrupt parameters may be combined with other select-parameters in a single SELECT statement, although such an intermixing of select-parameters is likely to make the program confusing to document. More significantly, the ON/OFF parameters can be specified in an ON/SELECT statement to permit conditional definition and enabling of interrupts.

ON/SELECT is a conditional form of the SELECT statement in which a value, called a "select-value," is used to determine which one of a number of specified select-parameters will be selected. The following example illustrates the use of ON/SELECT to conditionally enable an interrupt for an operator-specified device:

```
40 SELECT OFF/017 GOSUB 3000, OFF/018 GOSUB 4000, OFF/019 GOSUB 5000
50 LINPUT "ENTER INTERRUPT DEVICE (1, 2, or 3)", N
60 ON N SELECT ON/017, ON/018, ON/019
```

In this example, interrupts for all devices are initially defined at line 40. At this point, however, all interrupts are disabled. Line 50 prompts the operator to specify which device will be used. The desired device is identified by specifying a number from 1 to 3, and this number is assigned to

numeric-variable N. At line 60, the value of N is used as a select-value to determine which interrupt will be enabled. If N=1, device /017 is enabled; if N=2, device /018 is enabled; and if N=3, device /019 is enabled. If N has a value less than 1 or greater than 3, nothing is enabled.

The ON/SELECT statement with interrupt parameters may be useful in situations where an operator or the program itself must have the ability to enable or disable specified interrupts. The general ON/SELECT statement is described in greater detail in Chapter 7, sections 7.8 and 7.9.

8.10 GENERAL FORMS OF THE INTERRUPT CONTROL INSTRUCTIONS

The general forms of the ON/SELECT statement and LIST I command are shown on the following pages.

SELECT ON/OFF

General Form:

SELECT { ON { [device-address [GOSUB line-number]] / CLEAR } / OFF [device-address [GOSUB line-number]] } [,...]

Purpose:

The SELECT ON/OFF statement is a form of the general SELECT statement containing special parameters used to define, redefine, and clear interrupts. A SELECT ON/OFF statement cannot be executed in Immediate Mode.

SELECT ON/OFF has four different functions:

1. Interrupt/Priority Definition

   Device-address and GOSUB parameters may be specified to identify the devices to be scanned for interrupts and to specify the location of the interrupt processing subroutine for each device:

   ```
   SELECT ON /device-address GOSUB line-number
   SELECT OFF /device-address GOSUB line-number
   ```

   A maximum of eight interrupts may be defined. The order in which interrupts are defined in the program determines the order in which incoming interrupts will be serviced (i.e., the interrupt priority). The ON and OFF parameters determine whether the interrupt is enabled or disabled at the time it is defined. Interrupts may be selectively enabled and disabled subsequently in the program without changing their initially defined priorities. For example, the statement

   ```
   50 SELECT ON/018 GOSUB 1000, OFF/019 GOSUB 1500
   ```

   defines interrupts for the devices with addresses /018 and /019 and assigns /018 the higher priority and /019 the lower priority. The interrupt for device /018 is initially enabled, while the interrupt for device /019 is initially disabled. If an interrupt is received from device /018, program execution is halted at the conclusion of the current statement, and processing branches to line 1000 to perform interrupt processing.

   Upon completion of the interrupt subroutine, execution resumes in the main program at the point of interruption. Because the interrupt for device-address /019 is disabled, an interrupt received from that device-address is ignored.

2. Enabling and Disabling Individual Interrupts

   A SELECT ON or SELECT OFF statement with only a device-address specified can be used to selectively enable or disable a previously defined interrupt. Statements of the form

   ```
   SELECT ON/device-address
   SELECT OFF/device-address
   ```

   are used for this purpose. For example, the statement

   ```
   100 SELECT ON/019
   ```

enables the interrupt for device /019 (previously defined in statement 50 above), while the statement

```
110 SELECT OFF/018
```

disables the interrupt for device /018 (also defined in line 50 above).

> **NOTE:**
>
> When an interrupt is enabled, it automatically assumes its originally defined priority.

3. General Inhibition and Reactivation of All Currently Enabled Interrupts

   All interrupts currently enabled can be temporarily inhibited with a statement of the form

   ```
   150 SELECT OFF
   ```

   Following execution of this statement, interrupts from currently enabled devices are ignored. Subsequently, all enabled interrupts can be reactivated with a statement of the form

   ```
   200 SELECT ON
   ```

   Note that this form of general interrupt inhibition/reactivation affects only those interrupts which are currently enabled. Disabled interrupts are *not* enabled by SELECT ON.

4. Clearing All Currently Defined Interrupt Information for Redefinition

   Information on all currently defined interrupts can be cleared from the Interrupt Table with a statement of the form

   ```
   250 SELECT ON CLEAR
   ```

   Following execution of this statement, a new set of interrupts could be defined with new priorities.

ON/OFF and associated interrupt parameters may be included with any other select-parameters in a SELECT statement. In addition, they may be specified in an ON/SELECT statement to permit conditional definition and enabling of interrupts. See Chapter 7, sections 7.8 and 7.9 for a discussion of the general SELECT statement and the ON/SELECT statement.

Examples of Valid Syntax:

```
10 SELECT ON/017 GOSUB 100, ON/018 GOSUB 150, OFF/019 GOSUB 200
20 SELECT OFF/017, OFF/018, ON/019
30 SELECT ON
40 SELECT ON CLEAR
```

LIST I Command

General Form:

LIST [S] [title] I

Where:

title = { literal-string / alpha-variable }

Purpose:

The LIST I command lists the current contents of the Interrupt Table in memory. The general format of the listing is as follows:

Item in Listing	Meaning
ON/OFF	Condition of general interrupt inhibition/ reactivation.
GOSUB or blank	GOSUB if currently in interrupt subroutine.
ON/OFF aa GOSUB xxxx ON/OFF aa GOSUB xxxx . . .	Parameters of each defined interrupt, including status (ON or OFF), device-address (aa), and interrupt sub-routine line-number (xxxx).

For example, consider the following program segment:

```
50 SELECT ON/017 GOSUB 1500, OFF/018 GOSUB 2500, ON/019 GOSUB 3000
.
.
.
200 SELECT OFF
.
.
.
```

If a LIST I command is executed immediately after line 200 in this program has been executed, the following listing is produced:

```
OFF
ON 17 GOSUB 1500
OFF 18 GOSUB 2500
ON 19 GOSUB 3000
```

The "S" and "title" parameters are general LIST command parameters. When displaying a listing on the CRT, "S" is used to list only a screenful of lines at a time. Since the Interrupt Table never occupies more lines than can be displayed on a standard CRT (unless the "LINE" parameter is set to fewer than the maximum number of displayable lines for the CRT), the "S" parameter has no practical purpose in a LIST I command. The "title" parameter is used to specify a title for printed listings. Both "S" and "title" are explained in the discussion of the general LIST command in Chapter 10, section 10.2.

Examples of Valid Syntax:

```
LIST I
LIST "INTERRUPT TABLE" I
```

CHAPTER 9 ERROR CONTROL FEATURES

9.1 INTRODUCTION

BASIC-2 provides an extensive set of error detection features designed to automatically detect and report a wide range of error conditions. The system checks each text line for various types of errors when the line is entered by the programmer. Further error checking is carried out during program resolution when the program in memory is resolved prior to execution. During program execution, the system automatically detects and reports any errors which arise.

When the system detects an error condition, it immediately terminates the current operation and displays an error message. If an error is discovered during text entry, the erroneous line is stored in memory. If an error is discovered during program resolution, resolution is terminated at that point. If an error occurs during program execution, the program is immediately terminated. In each case, the system displays the erroneous line and beneath it the message "↑ERR" followed by an error-code with the arrow pointing to the approximate position of the error.

Error-codes are two-digit numbers preceded by a letter prefix (e.g., A04). The letter identifies the particular class of errors to which the error belongs, while the two-digit number identifies the specific error condition. For example, an error commonly encountered during text entry is S11, "Missing Right Parenthesis". The "S" indicates a syntax error, and the "11" identifies the error uniquely as "Missing Right Parenthesis". There are seven classes of error conditions, and each is identified by a unique letter prefix in the error-code:

Class of Errors	Letter Prefix
Miscellaneous Errors	A
Syntax Errors	S
Program Errors	P
Computational Errors	C
Execution Errors	X
Disk Errors	D
I/O Errors	I

A complete list of the errors included in each class and their specific codes with detailed explanations of each error condition can be found in Appendix A. A summary listing of all error-codes is provided in Appendix H.

Note that the system stops error scanning when the first error is detected. Thus, if a line contains more than one error, only the first is detected and reported by the system.

9.2 NONRECOVERABLE AND RECOVERABLE ERRORS

Errors in the first three classes listed (Miscellaneous Errors, Syntax Errors, and Program Errors) cause the system to terminate the current operation and display an error message. The operator then must correct the error before proceeding with further operations. Errors of this kind are called "non-recoverable" errors and generally indicate incorrect syntax or program logic errors.

Errors in the four remaining classes (Computational Errors, Execution Errors, Disk Errors, and I/O Errors) typically occur during program execution. Errors which can be intercepted by the program before the system intervenes are called "recoverable" errors. BASIC-2 allows the programmer to respond to recoverable errors under program control without aborting the program or disrupting the display with an error message. Three BASIC-2 instructions are provided for intercepting and responding to recoverable errors: the SELECT ERROR statement, the ERR function, and the ERROR statement.

The SELECT ERROR statement provides a means of specifying which computational errors will be processed by the program and which will be handled with a system response. The ERR function is a numeric function which returns the error-code of the most recent error condition. When an error occurs, ERR can be used to examine the particular error-code and identify the error under program control. The ERROR statement, lastly, provides a versatile capability for initiating special error processing when an error is detected in a BASIC-2 statement. Both the ERR function and the ERROR statement can be used with all types of recoverable errors.

9.3 GENERAL FORMS OF THE ERROR CONTROL INSTRUCTIONS

The general forms of the SELECT ERROR statement, the ERR function, and the ERROR statement are shown on the following pages.

SELECT ERROR

General Form:

SELECT ERROR [> error-code]

Where:

error-code = any computational error-code (60-69)

Purpose:

The SELECT ERROR statement is used to suppress the normal system response to specified computational errors. Computational errors are those produced by the math package while performing an arithmetic operation or evaluating a function. Normally, the system responds to a computational error (other than underflow) by terminating program execution and displaying an error message. (Underflow normally does not terminate program execution.) However, a statement of the form

SELECT ERROR > error-code

suppresses the system error message for all computational errors whose error-codes are *less than or equal to* the error-code specified in the SELECT ERROR statement and permits program execution to continue with the values shown in Table 9-1 below. Computational errors whose error-codes are greater than the specified error-code cause program termination and generate a system error message.

Table 9-1.
SELECT ERROR Return Values

Error Code	Error Condition	Value Returned
C60	Underflow.	0
C61	Overflow.	±9.999999999999E+99
C62	Division by zero.	±9.999999999999E+99
C63	Zero divided by zero, or zero raised to the zeroth power.	0
C64	Zero raised to negative power.	+9.999999999999E+99
C65	Negative number raised to noninteger power.	ABS(X)↑Y
C66	SQR of negative value.	SQR(ABS(X))
C67	LOG of zero.	-9.999999999999E+99
C68	LOG of negative value.	LOG(ABS(X))
C69	Argument too large for trig function.	0

Only the numeric portion of the error-code is specified in a SELECT ERROR statement; the letter prefix must be omitted. For example, execution of the statement

```
SELECT ERROR > 65
```

causes the normal error response to be suppressed for errors C60-C65, while normal system error processing remains in effect for errors C66-C69. Note that the SELECT ERROR statement is used *only* for processing computational errors; it has no effect on errors outside the range C60-C69.

Computational errors suppressed with a SELECT ERROR statement cannot be handled with an ERROR statement (see the discussion of the ERROR statement in this section). Typically, the ERR function is used at the conclusion of a numeric processing routine to determine whether an error occurred during processing. A special error recovery routine can be invoked if an error condition is indicated by ERR (see the discussion of the ERR function in this section).

When the system is Master Initialized or a CLEAR or LOADRUN command is executed, normal system error response is selected for all computational errors *except underflow.* Master Initializing the system or executing a CLEAR or LOADRUN command has the same effect as executing the statement SELECT ERROR > 60.

A SELECT ERROR statement with no error-code, e.g.,

```
SELECT ERROR
```

causes *all* computational errors (including underflow) to terminate program execution with an error message.

ERR Function

General Form:
ERR

Purpose:

The ERR function is a numeric function which returns the error-code of the most recent error condition. Since ERR is assigned a new value whenever an error occurs, it always returns the error-code of the last error which occurred prior to referencing the ERR function. A reference to the ERR function automatically resets its value to zero. The ERR function does not discriminate between recoverable and nonrecoverable errors; it is set by any error condition. ERR returns the numeric portion of the error-code which uniquely defines each error. The letter prefix, which is used only to identify the error class, is stripped off. Thus, for example, following a division by zero (error-code C62), ERR = 62. If no error has occurred, ERR = 0.

The ERR function provides a convenient means of checking for the occurrence of an error within a program. It can be used with the SELECT ERROR statement to check for computational errors or with the ERROR statement to identify the type of recoverable error which has occurred.

A typical use of ERR is to check for errors at the conclusion of a numeric processing routine. For example:

```
100 SELECT ERROR > 69
        .
        .
        .
    (numeric processing)

250 X=ERR:IF X<>0 THEN PRINT "ERROR=";X
260 END
```

Line 250 concludes the numeric processing routine by checking for an error. The variable X is set equal to the error-code returned by ERR. If X=0, no errors have occurred and the program ends. If X<>0, an error has occurred and an error message is displayed with the error-code to inform the operator of the problem.

The ERR function can be used with the ERROR statement to identify a specific error. For example:

```
10 DATALOAD DC A$() : ERROR X=ERR: GOSUB'90
```

Line 10 uses the ERROR statement to check for an error following execution of the DATALOAD DC statement. If an error has occurred, X is set equal to the error-code, and the program branches to DEFFN'90 to process the error. Otherwise, normal execution continues at the next program line.

Because the ERR function will return the error-code of a nonrecoverable error, the operator can use ERR to recover from nonrecoverable errors after the program has terminated. The recovery procedure involves defining one of the Special Function Keys as a "help" key which is depressed by the operator when the program is terminated by a nonrecoverable error. Within the subroutine accessed by the Special Function Key, the value of ERR can be checked, and an appropriate recovery message is then displayed for the operator.

The ERR function is reset to zero whenever it is referenced. For example, following execution of the statement

```
X = ERR
```

ERR is automatically reset to a value of zero. ERR also is reset to zero whenever a RUN or CLEAR command is executed or the system is Master Initialized.

Examples of Valid Syntax:

```
:10 X = ERR: PRINT X
:20 DATALOAD DC #1,A$,B,N: ERROR X=ERR: GOSUB 250
:30 X=ERR: IF X <> 0 THEN 75
```

ERROR

General Form:

ERROR statement [:statement]...

Purpose:

The ERROR statement provides a convenient means of responding to recoverable errors under program control. If a recoverable error occurs in a BASIC-2 statement which is immediately followed by an ERROR statement, the normal system error response (i.e., program termination and error message) is suppressed, and program execution continues at the statement immediately following the word "ERROR". If no error occurred, execution continues at the next program line. If multiple statements follow ERROR on the same line, they are interpreted as part of the ERROR sequence and are executed in order (unless one statement causes a branch).

The ERROR statement cannot intercept nonrecoverable errors, nor can it intercept computational errors whose normal system response has been suppressed with a SELECT ERROR statement. For example, if the statement SELECT ERROR > 65 has been executed in a program, computational errors C60-C65 *cannot* be intercepted by an ERROR statement (see the discussion of the SELECT ERROR statement earlier in this section).

Whenever an error occurs, the ERR function is set to the corresponding error-code (see the discussion of the ERR function earlier in this section). ERR can be checked following an ERROR statement to identify the particular error which has occurred. For example:

```
100 DATASAVE DC #1, A$(), B$: ERROR X=ERR: GOSUB'100
110 GOTO 50
```

If an error is detected following execution of the DATASAVE DC statement on line 100, execution continues with the statements following ERROR. Otherwise, execution continues at the next program line.

The ERROR statement cannot be used in Immediate Mode.

Examples of Valid Syntax:

```
10 DATALOAD DC A(), B(): ERROR GOSUB'250
20 INPUT"ENTER X,Y COORDINATES",A1,B1: ERROR PRINT "ILLEGAL VALUE": GOTO 150
30 DATASAVE DC A$(): ERROR X=ERR: GOSUB 100
```

CHAPTER 10 SYSTEM COMMANDS

10.1 INTRODUCTION

System commands enable the operator to directly control system operations from the keyboard. In addition, special commands support an extensive set of debugging aids which enable the programmer to examine, modify, and document his BASIC-2 program. The available system commands are summarized below:

CLEAR	Clears from memory all (or only a specified portion) of program text and/or variables.
CONTINUE	Continues program execution after it has been halted by executing a STOP statement or depressing the HALT/STEP Key.
HALT/STEP	Temporarily halts program execution; normal execution can be resumed by keying CONTINUE, or the operator can step through the program one statement at a time by repeatedly keying HALT/STEP.
LIST	Lists a program or specified portion of a program on the CRT display or a printer.
LIST V	Provides a cross-reference listing of some or all variables defined in the resident program.
LIST #	Provides a cross-reference listing of some or all line-numbers referenced in the resident program.
LIST '	Provides a cross-reference listing of one or all marked subroutines referenced in the resident program. (See DEFFN', Chapter 11, section 11.2.)
LIST T	Provides a cross-reference listing of all program lines containing a specified character string.
LIST DT	Lists the contents of the Device Table in memory. (See Chapter 7, section 7.8.)
LIST I	Lists the contents of the Interrupt Table in memory. (See Chapter 8, section 8.10.)
LOAD	Loads a user program from disk into memory. (See the *Wang BASIC-2 Disk Reference Manual.*)

10.2 GENERAL FORMS OF THE SYSTEM COMMANDS

General forms of the BASIC-2 system commands (with the exceptions noted in section 10.1) are shown on the following pages, arranged alphabetically for ease of reference.

LOAD RUN	Loads a program from disk into memory and automatically runs the program. (See the *Wang BASIC-2 Disk Reference Manual.*)
RENUMBER	Renumbers all or some of the program with the specified starting line number and the increment.
RESET	Terminates program execution, resets all I/O devices, and returns system control to the keyboard.
RUN	Begins execution of the user program in memory.
SAVE	Saves a user program on disk. (See the *2200VP/MVP Disk Reference Manual.*)
STMT NUMBER	Automatically generates a program line number incremented 10 greater than the highest current line number. Available on certain keyboards only.
TRACE	Enables the programmer to trace through program execution.

The disk commands LIST DC, LOAD, LOAD RUN, and SAVE are covered in the *2200VP/MVP Disk Reference Manual.*

CLEAR

General Form:

```
CLEAR [ P [line-number] [,[line-number]]
      [ V
      [ N
```

Purpose:

The CLEAR command reinitializes the program text and variable areas in user memory. CLEAR with no parameters removes all user program text and variables from memory and automatically selects the current console devices for the following I/O operations:

PRINT, PRINTUSING, and LIST operations — Selected to the current Console Output device. (Both the device-address and line width currently selected for CO operations are assigned to PRINT, PRINTUSING, and LIST operations.)

KEYIN, INPUT, and LINPUT operations — Selected to the current Console Input device.

CLEAR also turns off Pause and Trace Modes and clears the Interrupt Table and subroutine stacks in memory. The #1 — #15 entries in the Device Table are completely zeroed, as are the Start-, End-, and Current-sector-address parameters in slot #0. The default disk address in slot #0 is *not* altered, nor are the entries for PLOT and TAPE. (See Chapter 7, section 7.4 for a description of the format and contents of the Device Table.) On the MVP, all devices currently OPEN for the partition executing the CLEAR command are CLOSED (see Chapter 16, section 16.4 for information on OPENing devices).

CLEAR V removes all common and noncommon variables from memory, but does *not* clear the program text area or subroutine stacks. CLEAR V also does not affect the Device and Interrupt Tables.

CLEAR N removes all noncommon variables from memory. The names and values of common variables are not changed, nor are the program text area or subroutine stacks affected. The Device and Interrupt Tables are *not* altered.

CLEAR P removes program text from user memory without disturbing any variables. The CLEAR P command has several forms: CLEAR P with no line-numbers deletes *all* user program text from memory; CLEAR P with two line-numbers deletes the two specified lines and all intervening lines; CLEAR P with the first line-number specified and the second omitted (e.g., CLEAR P 100) clears all remaining lines from the specified line to the highest numbered line; and CLEAR P with the second line-number specified and the first omitted (e.g., CLEAR P,500) clears all lines from the lowest numbered line up to and including the specified line. All forms of CLEAR P also clear the Interrupt Table and subroutine stacks; however, CLEAR P does *not* alter the Device Table.

Examples of Valid Syntax:

```
CLEAR
CLEAR V
CLEAR N
CLEAR P
CLEAR P 10,20
CLEAR P 10
CLEAR P,100
```

CONTINUE

General Form:

CONTINUE

Purpose:

The CONTINUE command is used to continue program execution after a program has been halted by a STOP statement or by keying HALT/STEP. Program execution CONTINUEs at the program statement immediately following the last-executed statement. If multiple commands are used with CONTINUE in an Immediate Mode line, CONTINUE must be the *last* command on the line.

Restrictions:

Program execution *cannot* be CONTINUEd if one of the following conditions exists:

1. A Memory or Stack Overflow error has occurred.
2. A variable has been entered which was not previously defined.
3. A CLEAR V or CLEAR N command has been executed.
4. Program text has been modified by execution of a CLEAR, CLEAR P, or RENUMBER command or by entry of a new program line.
5. RESET has been keyed.
6. The program is unresolved (i.e., program execution was initiated with a Special Function Key without first resolving the program by executing a RUN command).

Examples of Valid Syntax:

```
CONTINUE
$RELEASE TERMINAL: CONTINUE
```

HALT/STEP Key

General Form:
HALT/STEP Key

Purpose:

The HALT/STEP Key is used to HALT program execution or a listing operation. HALT/STEP also can be used to STEP through program execution statement by statement. The HALT/STEP Key can be used to perform the following functions:

1. If a program is executing, the HALT/STEP Key stops execution after the completion of the currently executing statement. Keying CONTINUE enables program execution to resume with the next statement and continue with subsequent statements.

2. If a LISTing is in progress, the HALT/STEP Key stops the listing after the current line has been listed and terminates the LIST operation.

3. The HALT/STEP Key can be used to step through the execution of a program. If program execution was terminated by executing a STOP statement or depressing the HALT/STEP Key, depressing the HALT/STEP Key again causes the next program statement to be displayed and executed. After this statement has been executed, program execution is again HALTed. Whenever HALT/STEP is subsequently keyed , the next statement is displayed and executed. In multiple-statement lines, those statements which have already been executed are not listed; however, the colons separating these statements are displayed to indicate the presence of preceding statements in the line. The GOTO statement can be used in Immediate Mode to begin stepping execution at a particular line-number if the program already has been resolved with RUN (see the discussion of the GOTO statement in Chapter 11, section 11.2). HALT/STEP also may be used to step through a program in Trace Mode if the operator wishes to examine the values of variables. (See the discussion of the TRACE command later in this section.)

Restrictions:

It is *not* possible to step through program execution with HALT/STEP if one of the following conditions exists:

1. A text or table overflow error has occurred.

2. A variable has been entered that was not previously 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 or by the entry of a new program line.

5. RESET has been keyed.

6. The program is unresolved (i.e., program execution was initiated with a Special Function Key before the program was resolved by first executing a RUN command).

Example:

Assume the following program is in memory (the program must have been RUN, perhaps with a STOP statement prior to line 90):

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

HALT/STEP can be used to step through the program starting at line 110. TRACE is turned on so that variables receiving new values are displayed. The instructions which must be followed to step through the program in Trace Mode appear below in the left-hand column. The BASIC statements which must be input and the displays which appear on the CRT when the HALT/STEP Key is depressed are presented in the right-hand column.

Operator Action	**CRT Display Following Action**
Turn Trace Mode on.	:TRACE
Start stepping at line 110.	:GOTO 110
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
Touch HALT/STEP Key.	: 110:: GOTO 30 TRANSFER TO 30 :_

The LIST command is a particularly powerful and versatile debugging tool which enables the programmer to list and cross-reference a program and then examine the contents of certain system tables used by the program. The general form of the LIST command with all possible parameters is shown below:

```
              ┌ [D] [line-number]       [,[line-number]]      ┐
              │ V [variable-name]       [,[variable-name]]    │
LIST [S] [title] # [line-number]        [,[line-number]]      │
              │ '  [integer]                                  │
              │   { literal-string  }   [ ,literal-string  ]  │
              │ T {                 }   [                  ]  │
              │   { alpha-variable  }   [ ,alpha-variable  ]  │
              │ DT                                            │
              │ I                        [ file-number     ]  │
              │ DC  platter              [ /disk-address,  ]  │
              └                                               ┘
```

Because each form of the LIST command performs a different function, each is treated as a separate command with its own general form. The two most specialized forms of LIST — LIST DT and LIST I — are treated in Chapters 7, section 7.9 and 8, section 8.10, respectively.

When the system is Master Initialized, the CRT (address /005) is initially selected for LIST operations. Other printing devices may be selected for listing with a SELECT LIST statement (see the discussion of the SELECT statement in Chapter 7, section 7.3). Note that execution of a CLEAR or LOADRUN command automatically reselects LIST operations to the current CO device (see the discussion of the CLEAR command earlier in this section).

LIST

General Form:

LIST [S] [title] [D] [start-line-number] [,[end-line-number]]

Where:

title = { literal-string / alpha-variable }

start-line-number = line-number at which LISTing begins (default = lowest line-number)

end-line-number = line-number at which LISTing stops (default = highest line-number)

Purpose:

The LIST command instructs the system to display program text in memory in line-number sequence. All lines in the specified range are LISTed. If no line-numbers are specified in the LIST command, the entire program in memory is LISTed. If one line-number without a comma follows LIST, only that line is displayed. If the first line-number is omitted and the second specified (indicated by a comma preceding the line-number, e.g., LIST ,200), all lines from the lowest numbered line through the second line-number are listed. If the second line-number is omitted and the first specified (indicated by a comma following the line-number, e.g., LIST 10,), listing begins at the first line-number and continues through the highest numbered line. The LIST command is programmable.

If the "D" parameter is included in the LIST command, multiple-statement lines are displayed in decompressed form (i.e., one statement on each CRT or printer line). Each line-number is output as a four-digit number and, if necessary, padded with leading zeroes (e.g., 0010) when listed with LIST D. The use of leading zeroes (which normally are not included in a line-number listing) permits uniform alignment of listed lines.

The "S" parameter is a special feature for the CRT display which permits the program to be listed in steps. If the "S" parameter is included in the LIST command, program listing stops when the screen is full. To continue listing, RETURN must be keyed. Note that for nonstandard CRT's, the number of lines on the CRT is specified by a SELECT LINE statement. The "S" parameter is ignored in Program Mode.

Keying HALT/STEP during the listing of a program stops the listing after the current statement has been listed. However, the listing cannot be continued from that point. Alternatively, the user can execute a SELECT P statement to control the rate at which a program is listed on the CRT. When a SELECT P statement is executed, the system pauses for an interval ranging from 1/6 to 1 1/2 seconds after each line is listed.

The optional "title" parameter is a convenient means of identifying a hardcopy program listing. If a title is included in the LIST command, the system performs the following actions:

1. Issues a top-of-form (HEX(0C)) code.
2. Prints the highlighted title.
3. Skips a line.
4. Prints the program listing.

On a printer, these actions cause the title to be printed at the top of a new page in expanded print, followed by a blank line and the program listing. On a CRT, the title is displayed before the program listing. Depending on the model of the terminal, this title may be highlighted by high intensity blinking, unlerline, or reverse video. Consult the terminal manual for a discussion of HEX(0E).

A special form of the REM statement can be used to list program titles and subtitles in expanded print if the program is listed with a LIST D command. This special REM statement has the following general form:

REM %[↑] title/comment

The "%" following REM indicates that the following comment is a title which must be listed on a separate line in expanded print. LIST D issues a carriage return, skips a line after printing "REM%", and prints the specified title (highlighted) on a separate line. The system then skips another line and resumes the decompressed listing. For example, the statements

```
100 REM % TITLE
110 X=Y: Z=W
```

are printed as

```
0100 REM %

  TITLE (highlighted)

0110 X=Y
     : Z=W
```

when the program is listed with LIST D. If the up-arrow (↑) parameter is specified following REM%, the system issues a top-of-form code, prints the title in expanded print at the top of a new page, and then skips a line before continuing the listing. Note that the up-arrow (↑) must *immediately* follow the percent sign (%), with *no* intervening spaces.

NOTE:

The REM% statement has special significance *only* for programs LISTed with LIST D. If the "D" parameter is omitted, a REM% statement is treated as a normal REM in the listing.

The END, GOTO, LOAD, and RETURN instructions also cause the system to skip a line when listing programs under LIST D.

Examples:

Assume that the following program lines are in memory:

```
10 REM 1
20 REM 2.1: REM 2.2: REM 2.3
30 REM 3
40 REM % TITLE
50 REM 5
60 REM 6
70 REM 7
80 REM 8
90 REM 9
```

1. **LISTing the entire program after clearing the screen.**

```
:LIST HEX (03)
10 REM 1
20 REM 2.1: REM 2.2: REM 2.3
30 REM 3
40 REM % TITLE
50 REM 5
60 REM 6
70 REM 7
80 REM 8
90 REM 9
```

2. **LISTing lines 30 through 60, inclusive.**

```
:LIST 30,60
30 REM 3
40 REM % TITLE
50 REM 5
60 REM 6
```

3. **LISTing a single line.**

```
:LIST 70
70 REM 7
```

4. **LISTing from line 70 through the highest numbered line.**

```
:LIST 70,
70 REM 7
80 REM 8
90 REM 9
```

5. **LISTing from the lowest numbered line through line 40.**

```
:LIST,40
10 REM 1
20 REM 2.1: REM 2.2: REM 2.3
30 REM 3
40 REM % TITLE
```

6. **Decompressed LISTing of lines 10 through 30, inclusive.**

```
:LIST D 10,30
0010 REM 1
0020 REM 2.1
     : REM 2.2
     : REM 2.3
0030 REM 3
```

7. Decompressed LISTing of the entire program.

```
:LIST D
0010 REM 1
0020 REM 2.1
     : REM 2.2
     : REM 2.3
0030 REM 3
0040 REM %

 TITLE (highlighted)

0050 REM 5
0060 REM 6
0070 REM 7
0080 REM 8
0090 REM 9
```

LIST V

General Form:

LIST [S] [title] V [variable-name] [,[variable-name]]

Where:

variable-name =

letter [digit]	for numeric-scalars
letter [digit]$	for alpha-scalars
letter [digit](	for numeric-arrays
letter [digit]$(	for alpha-arrays

title =

alpha-variable
literal-string

Purpose:

The LIST V command generates a cross-reference listing of all references to the specified variables within the current program. The LIST V command is programmable.

NOTE:

The program *must* be free of syntax errors.

If no variable-names are specified following LIST V, the cross-reference listing is performed for all variables in the program. If a single variable is specified (e.g., LIST V A$), references to that variable only are listed.

Specifying a range of variables (e.g., LIST V X,Y) causes all references to variables of the same type within that range, inclusive, to be listed. Four types of variables are available: numeric-scalar, alpha-scalar, numeric-array, and alpha-array. Both variables specified in the LIST V command must be the same type. If the first variable is omitted (indicated by a comma preceding the second variable, e.g., LIST V ,H), the second variable determines the type to be listed, and references to all variables of that type up to and including the specified second variable are listed. If the second variable is omitted (indicated by a comma following the first variable, e.g., LIST V X,), all references to variables of the type specified by the first variable are listed, beginning with the first variable.

The "S" parameter is a special feature for the CRT terminal which permits the variable references to be listed in steps. If the "S" parameter is included in the LIST V command, listing stops when the screen is full. To continue listing, the operator must key RETURN. Note that for nonstandard CRT's, the number of lines on the CRT is specified by a SELECT LINE statement. The "S" parameter is ignored in Program Mode.

Keying HALT/STEP during the listing of a variable cross-reference stops the listing after the current line is finished. However, the listing cannot be continued from that point. Alternatively, the operator may slow down listing on the CRT by invoking a pause ranging from 1/6 to 1 1/2 seconds with a SELECT P statement. In this case, the system pauses for the specified interval after each line is listed.

The optional "title" parameter is a convenient means of identifying a hardcopy cross-reference listing. If a title is included in the LIST V command, the system performs the following actions:

1. Issues a top-of-form (HEX(0C)) code.
2. Prints the highlighted title.
3. Skips a line.
4. Prints the listing.

On a printer, these actions cause the title to be printed at the top of a new page in expanded print, followed by a blank line and the cross-reference listing. On a CRT, the title is displayed before the the cross-reference listing.

Examples 1-4 (LISTing Variable References):

Assume the following program lines are in memory:

```
10 DIM A(3),B$80,C$(2,3),D9$(81),N(40)
20 A1=5:X,X1,Y=0.3
30 A(1) = A1*Y/2
40 B$ = D9$(1)
```

1. LISTing all variables.

```
:LISTV
A(      —  0010 0030
A1      —  0020 0030
B$      —  0010 0040
C$(     —  0010
D9$(    —  0010 0040
N(      —  0010
X       —  0020
X1      —  0020
Y       —  0020 0030
```

2. LISTing a single variable.

```
:LISTV A1
A1      —  0020 0030
```

3. LISTing a range of numeric-scalar-variables.

```
:LISTV X,Y
X       —  0020
X1      —  0020
Y       —  0020 0030
```

4. LISTing all numeric-scalar-variables.

```
:LISTV A,
A1      —  0020 0030
X       —  0020
X1      —  0020
Y       —  0020 0030
```

LIST#

General Form:

LIST [S] [title] # [line-number] [, [line-number]]

Where:

title = { literal-string / alpha-variable }

Purpose:

The LIST# command generates a cross-reference listing of all references to the specified line-numbers within the current program. If no line-numbers are specified, the entire program is cross-referenced. If only one line-number follows LIST#, only references to that line are displayed. If two line-numbers follow LIST#, all references from the first through the second line-number, inclusive, are listed. If the first line-number is omitted (indicated by a comma preceding the second line-number, e.g., LIST#,3000), the listing begins at the lowest numbered line and terminates with line 3000. If the second line-number is omitted (indicated by a comma following the first line-number, e.g., LIST# 100,), the listing begins with line 100 and terminates at the highest numbered line. Line-numbers which are referenced but do not exist are enclosed in parentheses in the listing. The LIST# command is programmable.

The "S" parameter is a special feature for the CRT display which permits the line-number references to be listed in steps. If the "S" parameter is included in the LIST# command, listing stops when the screen is full. To continue listing, RETURN must be keyed. Note that for nonstandard CRT's, the number of lines on the CRT is specified by a SELECT LINE statement. The "S" parameter is ignored in Program Mode.

Keying HALT/STEP during the listing of a line-number cross-reference stops the listing after all references to the current line-number have been listed. However, the listing cannot be continued. Alternatively, the operator may slow down listing on the CRT by invoking a pause ranging from 1/6 to 1 1/2 seconds with a SELECT P statement. In this case, the system pauses for the specified interval after each line is displayed.

The optional "title" parameter is a convenient means of identifying a hardcopy cross-reference listing. If a title is included in the LIST# command, the system performs the following actions:

1. Issues a top-of-form (HEX(0C)) code.
2. Prints the highlighted title.
3. Skips a line.
4. Prints the listing.

On a printer, these actions cause the title to be printed at the top of a new page in expanded print, followed by a blank line and the cross-reference listing. On a CRT, the title is displayed before the cross-reference listing.

Examples:

Assume the following program lines are in memory:

```
10 GOTO 100
20 IF I=3 THEN 90:IF J=4 THEN 100
30 GOSUB 200
40 KEYIN A$ 50, 300: GOTO 40
50 PRINT A$:GOTO 40
90 X=2.7
100 Y=Z/X
110 GOTO 500
.
.
.
200 REM SUBROUTINE
.
.
.
290 RETURN
300 END
```

1. **LISTing all line-number references.**

```
:LIST#
 0040    —   0040 0050
 0050    —   0040
 0090    —   0020
 0100    —   0010 0020
 0200    —   0030
 0300    —   0040
(0500)   —   0110
```

2. **LISTing all references to a specified line-number.**

```
:LIST # 40
 0040    —   0040 0050
```

3. **LISTing all references to line-numbers within a specified range.**

```
:LIST#90,250
 0090    —   0020
 0100    —   0010 0020
 0200    —   0030
```

LIST'

General Form:

LIST [S] [title] ' [integer]

Where:

title = { literal-string / alpha-variable }

Purpose:

This form of the LIST command creates a cross-reference listing for the specified DEFFN' subroutines in the program. If no integer is specified in the LIST' command, a cross-reference for all DEFFN' subroutines ('0-'255) referenced and/or defined in the program is produced. If an integer is specified, that particular DEFFN' is cross-referenced. The line in which each DEFFN' subroutine is defined and all references to the subroutine in GOSUB' statements throughout the program are included in the listing. If a referenced DEFFN' is not defined in the program, the system prints four question marks (????) instead of a line-number. The LIST' command is programmable.

The "S" parameter is a special feature for the CRT terminal which permits the cross-reference listing to proceed in steps. If the "S" parameter is included in the LIST' command, listing stops when the screen is full. To continue listing, RETURN must be keyed. Note that for nonstandard CRT's, the number of lines on the CRT is specified by a SELECT LINE statement. The "S" parameter is ignored in Program Mode.

Keying HALT/STEP during a cross-reference listing of DEFFN' subroutines stops the listing after all references to the current DEFFN' have been listed. However, the listing cannot be continued. Alternatively, the user can execute a SELECT P statement to control the rate at which a program is listed on the CRT. When a SELECT P statement is executed, the system pauses for a specified interval after each line is listed. The SELECT P statement permits the user to select a pause ranging from 1/6 to 1 1/2 seconds in length.

The optional "title" parameter is a convenient means of identifying a hardcopy cross-reference listing. If a title is included in the LIST' command, the system performs the following actions:

1. Issues a top-of-form (HEX(0C)) code.
2. Prints the highlighted title.
3. Skips a line.
4. Prints the listing.

On a printer, these actions cause the title to be printed at the top of a new page in expanded print, followed by a blank line and the cross-reference listing. On a CRT, the title is displayed before the cross-reference listing.

When the system is Master Initialized, the CRT is initially selected for all LIST operations. Other printing devices may be selected for listing with a SELECT LIST statement. (See the discussion of the SELECT statement in Chapter 7, section 7.4.)

Examples:

Assume the following program lines are in memory:

```
:10 DEFFN'15 "TEXT"
:20 GOSUB'0: GOSUB'1
.
.
.
:200 DEFFN'0
:210 GOSUB'1
.
.
.
:900 DEFFN'1
:910 GOSUB'2
.
.
```

1. LISTing all DEFFN' subroutine definitions and references.

```
:LIST'
0200 DEFFN' 0
        — 0020
0900 DEFFN' 1
        — 0020 0210
???? DEFFN' 2
        — 0910
0010 DEFFN' 15
```

2. LISTing a specified DEFFN' subroutine definition and all references to that subroutine.

```
:LIST '01
0900 DEFFN' 1
        — 0020 0210
```

LIST T

General Form:

LIST [S] [title] T { literal-string / alpha-variable } [,literal-string / ,alpha-variable] ...

Where:

title = { literal-string / alpha-variable }

Purpose:

The LIST T command generates a cross-reference listing of all program text lines that contain a specified string of characters. The string to be sought can be specified as a literal string or as the value of an alpha-variable. Space characters are ignored in the string being sought and in the program text. Including more than one literal string and/or alpha-variable in the LIST T argument list enables the system to search for more than one character string. The LIST T command is programmable.

The "S" parameter is a special feature for the CRT display which permits the cross-reference listing of the specified character strings to proceed in steps. If the "S" parameter is included in the LIST T command, listing stops when the screen is full. To continue listing, RETURN must be keyed. Note that for nonstandard CRT's, the number of lines on the CRT is specified by a SELECT LINE statement. The "S" parameter is ignored in Program Mode.

Keying HALT/STEP during the cross-reference listing of a specified character string stops the listing after all references to that string are listed. However, the listing cannot be continued. Alternatively, the operator may slow down listing on the CRT by invoking a pause ranging from 1/6 to 1 1/2 seconds with a SELECT P statement. In this case, the system pauses for the specified interval after each line is displayed.

The optional "title" parameter is a convenient means of identifying a hardcopy cross-reference listing. If a title is included in the LIST T command, the system performs the following actions:

1. Issues a top-of-form (HEX(0C)) code.
2. Prints the highlighted title.
3. Skips a line.
4. Prints the listing.

On a printer, these actions cause the title to be printed at the top of each page in expanded print, followed by a blank line and the cross-reference listing. On a CRT, the title is displayed before the cross-reference listing.

When the system is Master Initialized, the CRT is initially selected for LIST operations. Other printing devices may be selected for listing with a SELECT LIST statement. (See the discussion of the SELECT statement in Chapter 7, section 7.4.)

Examples:

Consider the following program in memory:

```
10 REM DISK SELECT IS ON LINE 20
20 SELECT DISK /320, #3 /B20
30 DATA LOAD DC OPEN T#3,"PEANUT"
 : DSKIP #3,1
40 LOAD F#3,"BUTTER"
50 REM LIST T IGNORES S PA  C  E S
```

```
:LIST T "SELECT"
"SELECT"
     —  0010 0020
```

```
:LIST T "#3"
"#3"
     — 0020 0030 0040
```

```
:A$="SP A C E S":LIST T A$
"SP A C E S"
     — 0050
```

```
:LIST T "A","B","C","Z"
"A"
     — 0030 0040 0050
"B"
     — 0020 0040
"C"
     — 0010 0020 0030 0050
"Z"
```

RENUMBER

General Form:

RENUMBER [line#1] [-line#2] [TO line#3] [STEP s]

Where:

line#1 = first line-number to be renumbered (default = lowest program line-number)

line#2 = last line-number to be renumbered (default = highest program line-number)

line#3 = new starting line-number (default = STEP value)

s = STEP value, an integer such that 0 < s < 100 (default = 10)

Purpose:

The RENUMBER command renumbers a program in memory. All program lines from line#1 to line#2, inclusive, are renumbered to start at the line-number specified as line#3, and each line-number is incremented by the specified STEP value. If no STEP value is specified, a default STEP value of 10 is used. All references to line-numbers within the renumbered program (e.g., in GOTO, GOSUB, PRINTUSING, etc., statements) also are automatically modified in accordance with the new numbering scheme.

If both line#1 and line#2 are specified, all lines between and including the two specified line-numbers are renumbered. If necessary, the renumbered lines are automatically moved to the appropriate place in the program. If only line#1 is specified, all lines starting with the specified line are renumbered. If only line#2 is specified (indicated by a hyphen preceding line#2, e.g., RENUMBER -500 TO 50), all lines starting with the lowest line up to the specified line-number, inclusive, are renumbered.

RENUMBER can be used to insert a section of program text between two program lines. However, if the renumbered program section will not completely fit between the two specified lines, RENUMBER is not executed and an error message is displayed.

For example, consider the following program:

```
10 DIM A$30
20 GOTO 500
500 A$=" ": GOSUB 800
510 PRINT A$: STOP#
800 REM READ SUBROUTINE
810 READ A$
820 RETURN
900 DATA "CAT"
```

The following RENUMBER command could be executed:

```
RENUMBER 800-820 TO 50
```

The renumbered program now appears below:

```
10 DIM A$30
20 GOTO 500
50 REM READ SUBROUTINE
60 READ A$
70 RETURN
500 A$=" ": GOSUB 50
510 PRINT A$: STOP#
900 DATA "CAT"
```

Note that lines 800-820 have been renumbered to lines 50-70 *and moved to the appropriate location in the program.* Note also that the GOSUB reference in line 500 has been changed from line 800 to line 50.

Examples of Valid Syntax:

```
:RENUMBER
:RENUMBER -1000 TO 10
:RENUMBER 100-200 TO 1000 STEP 5
:RENUMBER 100 TO 1000 STEP 10
:RENUMBER 100-200 TO 500
:RENUMBER 100 STEP 20
:RENUMBER TO 500 STEP 5
:RENUMBER STEP 20
```

RESET

General Form:
RESET Key

Purpose:
RESET immediately stops program listing or execution, clears the CRT screen, resets all I/O devices, and returns control to the keyboard. Program text in memory is not CLEARed by the RESET operation, and all variables retain their current values. Specifically, keying RESET performs the following functions:

1. Terminates program execution or listing.
2. Resets all I/O devices. On the MVP, only devices in use by the partition executing the RESET command are reset.
3. CLOSEs all devices currently OPEN (hogged) by this program. (See the discussion of $CLOSE in Chapter 16, section 16.10 and the discussion of disk hogging in the *Wang BASIC-2 Disk Reference Manual.*)
4. Clears the Interrupt Table (see the discussion of the Interrupt feature in Chapter 8).
5. Clears the internal stacks of all FOR/NEXT loop and subroutine information.
6. Turns off Trace and Pause Modes if they are selected.
7. On the MVP, sets the text pointer to the originating partition if global text was being executed (see the discussion of global partitions in Chapter 16, section 16.9).
8. Selects the Primary Keyboard (address /001) and CRT (address /005) for Console Input and Console Output operations, respectively.
9. Clears the CRT screen, displays the READY message, and returns control to the keyboard.

In general, RESET should be used only as a last alternative when a problem is encountered. RESET should be used to terminate program execution only if HALT/STEP fails since the program cannot be CONTINUEd following RESET. RESET should never be used to terminate execution during a disk write operation because invalid data could be stored on the disk.

If the system has undergone a malfunction which cannot be corrected by RESET, it may be possible to correct the problem by Master Initializing the system. This operation, however, erases all program text and data stored in user memory.

RUN

General Form:

RUN [line-number [,statement-number]]

Purpose:

The RUN command resolves and initiates execution of the user's program. A RUN command with no parameters first clears all noncommon variables from memory (common variables are not disturbed) and then initiates program resolution. Before the program can be executed, it must first be resolved. During program resolution, the program is scanned for syntax errors, and all references to variables and line-numbers are verified.

If a line-number is specified in the RUN command, the program is resolved and execution begins at the specified program line. However, noncommon variables are not CLEARed from memory, and all variables retain their current values. Thus, a HALTed program can be CONTINUEd from a specified line with the current data values by entering RUN followed by the appropriate line-number. Program execution may not be restarted in the middle of a FOR/NEXT loop or a subroutine.

Program execution can be started at a particular statement within a multistatement program line by including the statement-number after the line-number in the RUN command. Statements on a program line are numbered sequentially from left to right, starting at 1.

Program resolution is performed automatically by the RUN command prior to initiating program execution. Resolution involves the following operations:

1. Space is reserved for all referenced variables. Numeric-variables initially are set to zero, while alphanumeric-variables initially are set to all blanks.

2. The DATA pointer used by the READ statement is initialized to the first DATA value in the first DATA statement.

3. A complete sequential scan of the program verifies that (1) all referenced line-numbers (e.g., in GOTO, GOSUB, or ON/GOTO/GOSUB statements) exist, (2) all common variables are defined in the program before any noncommon variables are referenced, and (3) all referenced array-variables are defined in a DIM or COM statement.

If program resolution is completed with no errors, RUN immediately begins execution of the program from the lowest numbered line or the specified starting line. If an error is detected during resolution, the program is not executed and an error is signalled. (A more detailed description of program resolution is provided in Chapter 2, section 2.2.)

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 and to verify that all referenced lines exist and the text has no syntax errors. After a program has been RUN once, program execution may be restarted with a Special Function Key (which does not carry out program resolution prior to beginning execution).

Examples of Valid Syntax

```
RUN
RUN 30
RUN 100,3
```

Special Function Key

General Form:
Special Function Key

Purpose:
There are 16 Special Function Keys available on most system keyboards. In conjunction with the SHIFT Key, these keys provide access from the keyboard to 32 program subroutines or text entry definitions. Lowercase Special Function Keys are numbered '0-'15. Uppercase Special Function Keys, which are accessed by depressing SHIFT in conjunction with the SF Key, are numbered '16-'31. The additional Special Function Key on certain 2236D-type terminals is labelled "FN", and it provides Special Function '126 (lowercase) and '127 (SHIFTed). The Special Function Keys '0-'31 function differently depending upon whether the system is in Text Entry Mode or Edit Mode. In Text Entry Mode, they can be used to access user-defined subroutines or text strings. While in Edit Mode, the Special Function Keys (with the exception of the FN Key on 2236D-type terminals) cannot perform their user-defined functions, and the rightmost keys perform system-defined Edit functions. Edit Mode is discussed in Chapter 3, section 3.3.

When a Special Function Key is depressed in Text Entry Mode, the system searches for a DEFFN' statement with a corresponding number in the program stored in memory. If the DEFFN' statement identifies a subroutine, subroutine execution is initiated automatically. If the DEFFN' is used to define a character string for text entry, the defined string is placed in the Work Buffer and displayed on the CRT. Thus, depressing Special Function Key '2 causes execution of the subroutine identified by a DEFFN'2 statement or the entry of the text string defined in a DEFFN'2 statement. If there is no corresponding DEFFN' statement for the depressed SF Key, the audio alarm beeps and no further action is taken. Although a Special Function Key which accesses a DEFFN' subroutine will initiate program execution, it does *not* cause program resolution. Since an unresolved program typically will not execute successfully, every program should be resolved by running it with a RUN command prior to executing it via a Special Function Key. Note also that an unresolved program cannot be STEPped or CONTINUEd, even if a STOP statement or the HALT/STEP Key is used to halt execution. Any attempt to resume execution of a HALTed or STOPped program which is not resolved results in an error.

Text Entry

If a Special Function Key is defined for text entry, depressing the key causes the character string defined by the associated DEFFN' statement to be displayed on the CRT and become part of the current text line in the Work Buffer (see the discussion of the DEFFN' statement in Chapter 11, section 11.2). Suppose, for example, Special Function Key '2 is defined in the following statement:

```
:10 DEFFN'2 "HEX("
```

Depressing Special Function Key '2 after the following statement has been entered:

```
:20 PRINT
```

results in

```
:20 PRINT HEX(_
```

Thus, pressing Special Function Key '2 is now equivalent to keying in "H", "E", "X", and "(".

Subroutine Entry

If a Special Function Key is defined for DEFFN' subroutine entry, the subroutine can be entered either manually by depressing the appropriate Special Function Key or under program control from a GOSUB' statement within a program. Arguments which are to be passed to a manually entered subroutine must be keyed in (separated by commas) immediately before the Special Function Key is depressed. If the operator does not enter a sufficient number of arguments to satisfy all variables in the DEFFN' variable list, the system displays a question mark (?) and awaits the entry of the remaining arguments. If an illegal value is entered as an argument, the system signals an error and displays a question mark. The corrected value and all values *following* the corrected value must then be reentered. All values preceding the illegal value are accepted and need not be reentered. For subroutines executed under program control, arguments must be included as parameters in the GOSUB' statement. The number of arguments passed must equal the number of variables in the DEFFN' statement identifying the subroutine. When a RETURN statement is executed, control is passed back either to the keyboard for manually entered subroutines or to the program statement immediately following the GOSUB' statement for subroutines executed under program control.

Example 1:

Assume the following program has been entered:

```
:10 STOP
:20 DEFFN '3
:30 PRINT "THIS IS SUBROUTINE '3"
:40 RETURN
```

Then keying in RUN resolves the program:

```
:RUN

STOP
:
```

and depressing Special Function Key '3 causes execution of the subroutine and the following message to be displayed:

```
THIS IS SUBROUTINE '3
:
```

Example 2:

Define Special Function Key '0 to calculate the value of the following expression:

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

Assume that the following program has already been entered:

```
:0 STOP
: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
:RUN

STOP
```

Then, execute the subroutine for the following values of X and Y:

X=.092 and Y=-.32

Solution:

Manual Entry	**Program Entry**
Enter .092, -.32 Depress SF Key '0	:100 GOSUB'0(.092,-.32) :110 STOP# :RUN 100
CRT Display:	CRT Display:
:.092,-.32 X= .092 Y=-.32 Z=-5.507152 :_	X= .092 Y=-.32 Z=-5.507152 STOP 110 :_

The Special Function Keys can be used with DEFFN' subroutines to provide a number of entry points at which to begin program execution as well as to branch to specified points within an executing program. Because, however, the system stores subroutine return information in a fixed-length system stack, Special Function Keys should not be used repeatedly to access subroutines unless one of the following conditions is met:

1. Subroutine execution terminates with a RETURN statement, which returns control to the keyboard and deletes the subroutine return information from the stacks.

2. Subroutine return information is removed from the stacks by executing a RETURN CLEAR statement at the end of each DEFFN' subroutine (see the discussion of the RETURN CLEAR statement in Chapter 11, section 11.2).

3. RESET is keyed prior to depressing the Special Function Key. (RESET clears the stacks.)

Failure to observe at least one of these precautions eventually will result in a Stack Overflow error (Error A04).

STMT NUMBER Key

General Form:
STMT NUMBER Key

Purpose:

The STMT NUMBER Key automatically displays the line-number of the next program line to be entered. The line-number generated is 10 more than the highest line-number currently in memory. If the highest line-number is greater than 9989, the STMT NUMBER Key does not produce a new line-number.

A line-number also may be entered manually, using the numeric entry keys. Line-numbers must be integers from 0 to 9999, inclusive.

Program lines may be entered in any order. They are usually numbered in increments of ten so that additional lines can be easily inserted between existing lines. The system keeps program lines in numerical order regardless of the order of entry.

Example:

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 _
```

TRACE

General Form:
TRACE [OFF]

Purpose:
The TRACE statement produces a trace of important operations in the execution of a BASIC-2 program. Trace Mode is turned on when a TRACE statement is executed either within a program or in Immediate Mode and turned off by executing a TRACE OFF statement. While the system is in Trace Mode, output is produced from the following program operations:

1. Assignment (LET) statements and FOR/NEXT loops.
2. Program branches (GOTO, GOSUB, RETURN, interrupt GOSUB, etc.).

Note that Immediate Mode lines are *not* included in a TRACE.

Output from Assignment Statements and FOR/NEXT Loops
Whenever a variable receives a new value in an assignment statement or during the execution of a FOR/NEXT loop, both the variable name and the new value are output by TRACE. The output adheres to the following conventions:

1. The equal sign (=) in an assignment statement is replaced in TRACE output by a back arrow (←) character. (This character prints as an underline on the high-speed printer.) The back arrow is more mnemonic than an equal sign since the contents of the variable are not necessarily identical to the value displayed in the TRACE output (see conventions 2 and 3 below).

Examples:

```
:10 A$ = "ABCDEF"
:RUN

A$ ← ABCDEF   (CRT Display)
     or
A$ _ABCDEF    (High-Speed Printer)

:5 DIM B$(5)
:10 A,B,C = 41.2
:20 STR(A$,1,3) = "ABC"
:30 B$(3) = "THREE"
:RUN

A ← B ← C ← 41.2
STR(A$ ← ABC
B$( ← THREE
```

2. Nonprintable and control characters (HEX(00)-HEX(0F)) in literal strings are displayed and/or printed as decimal-point characters (.). This technique prevents TRACE output from destroying current information on the CRT display.

Example:

```
:10 A$ = HEX(41030A42)
:RUN

A$ ← A..B
```

3. Only the first 16 characters of an alpha value are displayed.

Example:

```
:10 A$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
:RUN

A$ ← ABCDEFGHIJKLMNOP
```

4. The termination of a FOR/NEXT loop is indicated with the words "NEXT END".

Example:

```
:5 DIM A(5)
:10 FOR I=1 TO 3
:20 A(I) = I
:30 NEXT I
:RUN

I ← 1
A( ← 1
I ← 2
TRANSFER TO 20
A( ← 2
I ← 3
TRANSFER TO 20
A( ← 3
I ← NEXT END
```

Output from Program Branches

Whenever a transfer is made within a program, TRACE outputs the words "TRANSFER TO", followed by the line-number of the transfer point.

Example:

```
:10 GOSUB 500
:20 GOSUB'45
:30 GOTO 600
.
.
.
:500 REM SUBROUTINE
:510 RETURN
:550 DEFFN'45:REM MARKED SUBROUTINE
:560 RETURN
:600 REM
:RUN

TRANSFER TO 500
TRANSFER TO 20
TRANSFER TO 550
TRANSFER TO 30
TRANSFER TO 600
```

CHAPTER 11 GENERAL-PURPOSE BASIC-2 STATEMENTS

11.1 INTRODUCTION

BASIC-2 contains most of the general-purpose statements commonly regarded as standard features of the BASIC language. These include statements which perform such fundamental program operations as assignment, loop control, conditional branching, subroutine definition and access, keyboard data entry, and generation of printed output. In addition, Wang has added a number of language extensions and enhancements designed to better support each of these areas. The general-purpose statements available in BASIC-2 are summarized below:

COM	Defines one or more variables as common, enabling them to pass common data between overlayed program modules.
COM CLEAR	Redefines designated common variables as noncommon or noncommon variables as common.
DATA	Provides data values used by a corresponding READ statement.
DEFFN	Defines a function of one variable for use in a program. The function so defined is specified by the programmer.
DEFFN'	Defines the beginning of a DEFFN' subroutine. The subroutine can be accessed in a program by executing a GOSUB' statement or from the keyboard by depressing a Special Function Key. Arguments can be passed to the subroutine.
DIM	Defines one or more noncommon variables.
END	Terminates execution of a BASIC program; returns the amount of free space in memory when executed.
FN	Accesses a user-defined function defined with DEFFN, passes a single argument to the function, and receives the returned value.
FOR. . .TO,NEXT	Defines the boundaries of a loop and determines how many times the loop is executed.
GOSUB	Initiates a branch to the first line of a subroutine.
GOSUB'	Initiates a branch to a specified DEFFN' subroutine. Can pass one or more arguments to the subroutine.
GOTO	Branches to a specified line-number.

IF...THEN	In its simple form, tests a single condition and executes the statement following THEN if the condition is true; otherwise, execution continues at the next statement. In its complex form, provides for testing multiple conditions in a single statement.
Image (%)	Defines a format for printed output generated by the PRINTUSING or PRINTUSING TO statements.
INPUT	Accepts entered data during program execution. (Primarily for operator entry from the keyboard; displays an optional prompt to request data.)
KEYIN	Inputs a single character from a keyboard-like device.
LET(Assignment)	Assigns the value of a numeric expression to one or more numeric-variables or assigns an alphanumeric character string to one or more alpha-variables.
LINPUT	Recalls the value of an alphanumeric-variable to the display for editing or data entry. Restricts cursor movement to defined limits in the display.
MAT COPY	Copies all or a portion of the value of one alpha-variable to a second alpha-variable.
MAT MOVE	Moves specified elements of one array to a second array in a specified order. Optionally, converts numeric data into sort format and vice versa.
MAT SEARCH	Locates all substrings in an alpha-variable which satisfy a given relation to a specified value.
ON/GOSUB, ON/GOTO	Causes a branch to a particular line depending upon the value of a specified expression. The ON statement is a computed or conditional GOSUB or GOTO statement.
PRINT	Evaluates and prints the value of a numeric expression, alpha-variable, or character string. Values are printed in a system-defined format.
PRINT AT	Moves the CRT cursor to a designated position in the CRT display.
PRINT HEXOF	Prints the hexadecimal equivalent of an alphanumeric value.
PRINT TAB	Tabs to a specified column in a print line. (Used for both CRT and printers.)
PRINTUSING	Prints an output line on the CRT or printer. Values are printed in a format defined by the programmer using an image specification.
PRINTUSING TO	Same as PRINTUSING except formatted print output is stored in a specified alphanumeric-variable for future processing instead of being printed.
READ	Reads data values from a designated DATA statement.
REM	Provides a means of inserting comments anywhere in a program.
RESTORE	Resets the DATA pointer used by READ to a specified value in a DATA statement.
RETURN	Returns program execution to the main program following completion of a subroutine.

RETURN CLEAR	A dummy RETURN statement which clears subroutine return information from the internal stacks without causing a branch back to the main program.
STOP	Terminates program execution and displays an optional message and/or the line-number of the STOP statement. Execution can be resumed with HALT/STEP or CONTINUE.

11.2 GENERAL FORMS OF THE GENERAL-PURPOSE STATEMENTS

The general forms of all statements described in section 11.1 are shown on the following pages, arranged alphabetically for ease of reference. Each general form conforms to a standard meta-language, which indicates various options, requirements, and features of these general forms. The rules for this meta-language are discussed in Appendix B, "BASIC-2 Terminology and Rules of Syntax." It is recommended that the reader become familiar with this appendix before consulting the general forms of the BASIC-2 statements.

COM

General Form:

COM com-element [,com-element] . . .

Where:

com-element = { numeric-scalar-variable | numeric-array-name (dim1[,dim2]) | alpha-array-name (dim1[,dim2])[length] | alpha-scalar-variable [length] }

dim1, dim2 = dimensions (positive integers or numeric-scalar-variables) such that 1 <= dim1, dim2 <= 255 for two-dimensional arrays; 1 <= dim1 <= 65535 for one-dimensional arrays.

length = positive integer or numeric-scalar-variable such that 1 <= length <= 124.

Purpose:

The COM statement, like the DIM statement, is used to define variables within a BASIC program. Unlike variables defined by a DIM statement, however, variables defined by a COM statement are protected against destruction when a new program is LOADed into memory or the current program is run. Common variables can, therefore, be used to pass common data between successive program module overlays.

When a program is run, currently defined common variables and their contents are not disturbed, while all noncommon variables are cleared from memory. (Noncommon variables include all variables not explicitly defined in a COM statement.) Common variables are cleared from memory only by executing a CLEAR, CLEAR V, or LOADRUN command or by Master Initialization.

Since common variables must be defined before any noncommon variables are defined or referenced, COM statements usually are assigned low line-numbers in a program.

The COM statement is processed during program resolution, which occurs immediately prior to program execution. Program resolution is initiated by executing either a RUN command or a LOAD statement under program control. During program execution, a COM statement is ignored. During program resolution, the entire program is scanned for variable references, and space is reserved for all variables which have not been previously defined. When a variable is initially defined in a COM statement, it is assigned space in a section of memory identified as "common." If a noncommon variable has been defined prior to a common variable, an error is signalled and program resolution halts. If a common variable is encountered which has already been defined in the current program or a previous program, its dimensions must be identical to the previously defined dimensions; otherwise, an error is signalled. Scalar-variables do not need to be defined in a DIM or COM statement. If reference is made to an undefined scalar-variable, the system automatically reserves space in the noncommon section of memory (undefined alphanumeric-scalars are allotted 16 bytes). Note that unless a scalar-variable is defined in a COM statement, it is assumed to be noncommon. Array-variables, by contrast, must be defined in a DIM or COM statement. Reference to an undefined array-variable results in an error.

If a set of common variables is to be used in several sequentially run programs, the COM statements do not have to appear in any program except the first. All variables defined as common retain their original dimensions and current values in all subsequent programs. COM statements may, however, be included in subsequent programs for documentation purposes. In each program, the specified dimensions for any previously defined variables must be identical to the original dimensions. The contents of such redefined common variables are not altered by redefinition. New common variables also may be defined at the beginning of any subsequent program.

Advanced Use of Scalar-Variables in the COM Statement

The dimensions and lengths of variables in COM statements can be specified by numeric-scalar-variables. The numeric-scalar-variables used for this purpose must have legal values for dimensions and lengths when the COM statement is processed during program resolution. When a RUN command without a line-number or a program overlay (LOAD statement) is executed, noncommon variables are set to zero; thus, scalar-variables used to specify the dimensions and lengths of variables in COM statements generally should be defined as common variables.

This feature is particularly useful in applications where array sizes must be dynamically determined at the time of program execution. For example, the first program module in a system may determine the array dimensions that are to be used:

```
5 REM THIS IS MODULE 1
10 COM X,Y
20 INPUT "DIMENSIONS",X,Y
30 LOAD "MODULE2"
```

The second module might use those dimensions for the array definition:

```
5 REM THIS IS MODULE 2
10 COM N(X,Y)
    .
    .
    .
```

Examples of Valid Syntax:

```
10 COM A(10),B(3,3),C
20 COM D$(20),E$(2,3)100,F1$64
30 COM N(500)
40 COM N(R,C),A$(X,Y)L
```

COM CLEAR

General Form:

COM CLEAR [scalar-variable | array-designator]

Purpose:

The COM CLEAR statement is used to define as noncommon some or all previously defined common variables or to define as common some or all previously defined noncommon variables. When the COM CLEAR statement is executed, the values of the variables specified in the statement are not changed. The COM CLEAR statement merely changes the status of variables from common to noncommon (or vice versa) by moving the Common Variable Pointer up or down in the Variable Table in memory; it does *not* actually clear any variables from memory. The actual procedure followed by COM CLEAR in changing a variable's status is described in detail in Chapter 2, section 2.2.

If no variable-name is specified in a COM CLEAR statement, all currently defined common variables are redefined as noncommon variables. A subsequent RUN command or program overlay (LOAD statement) will clear all noncommon variables from memory.

If a common variable is specified in a COM CLEAR statement, the specified common variable and all common variables defined *after* it in the current program are made noncommon variables; all common variables which have been defined in the program *prior* to the specified variable or which have been defined in previous program modules remain common.

If a noncommon variable-name or array-designator is specified in a COM CLEAR statement, all noncommon variables defined in the program *prior* to the specified variable are made common; the specified variable itself and all variables defined *after* it remain noncommon.

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

If the variable specified in a COM CLEAR statement is not defined, an error is signalled.

Unlike COM, COM CLEAR is an executable statement; it is performed in sequence during normal program execution rather than once at program resolution (as in the case of COM).

Example 1:

```
:10 COM CLEAR
```

Redefines all previously defined common variables as noncommon.

Example 2:

```
:110 COM A,B,X,Y(10)
      .
      .
      .
:200 COM CLEAR X
```

A and B remain common; X and Y( ) redefined as noncommon.

Example 3:

```
:10 COM X(10,10)
:20 A = B + C
:30 COM CLEAR B
```

Redefines A as a common variable; B and C remain noncommon; X( ) remains common.

DATA

General Form:

DATA n [,n] . . .]

Where:

n = number or literal string

Purpose:

The DATA statement supplies 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) in the DATA statement value list are obtained and assigned to variable(s) in the READ statement variable list. The values in a DATA statement must be specified in the order in which they are to be used, and individual values must be separated by commas. If a program contains several DATA statements, they are used in line-number sequence. Numeric-variables in a READ statement must be assigned legal BASIC numbers from a DATA statement, and alphanumeric-variables must be assigned literal strings.

The RESTORE statement allows the system to reset the current DATA statement pointer so that DATA statement values can be reused. (See the discussion of the RESTORE statement found later in this section.)

The DATA statement may not be used in Immediate Mode.

Example:

```
:10 READ W
:20 PRINT W, W↑2
:30 GOTO 10
:40 DATA 5, 8.26, -687, 22
:RUN
 5                    25
 8.26                 68.2276
-687                  471969
 22                   484
10 READ W
        ↑ERR X70 (insufficient data)
```

In the above example, the four values listed in the DATA statement are sequentially used by the READ statement and printed. When a fifth value is requested, an error is displayed since all DATA statement values have been used. The program must be written to ensure that either no more data is read than is available or the ERROR statement is used to recover from insufficient data errors. For example, line 10 above could be changed to:

```
:10 READ W :ERROR END
```

Examples of Valid Syntax:

```
10 DATA 4, 3, 5, 6
20 DATA 6.56E+45, -644.543
30 DATA "BOSTON,MASS","SMITH",12.2
40 DATA HEX(0A), "ABC"
```

DEFFN

General Form:

DEFFN a(v) = expression

Where:

a = a letter or digit which identifies the function

v = a numeric-scalar-variable

Purpose:

The "DEFine FuNction" statement, DEFFN, enables the programmer to define a function of one variable within a program. The new function may be defined with any legal numeric expression. The defined function can be referenced from other points in the program with an FN function: the value of an argument specified in the FN function is automatically passed to the DEFFN statement for evaluation, and the result is returned to the FN function. A user-defined function may be used in a program wherever system-defined numeric functions are legal. (See Chapter 4, section 4.8 for information on the available system-defined numeric functions.)

Three parameters are required in a DEFFN statement:

1. A letter (A-Z) or digit (0-9) which serves as the "function name."
2. A numeric-scalar-variable which serves as a "dummy variable."
3. The expression which defines the new function.

The letter or digit specified as the function name is used to uniquely identify the defined function when it is referenced with an FN function. The variable specified as a dummy variable is used as a placeholder only, indicating where in the DEFFN expression the argument value of the FN function is to be used. The dummy variable plays no functional role in the evaluation of the DEFFN expression, and its contents are not altered by this operation. For example, in the statement

```
:10 DEFFN A(X) = X↑2-X
```

"A" is the user-defined function name, and "X" is the dummy variable.

Two parameters are required in an FN function which references a user-defined function:

1. A letter or digit which represents the function name.
2. An argument whose value is to be passed to the defined function for evaluation.

The argument specified in an FN function may be any legal numeric expression. The expression is first evaluated by FN, and the resulting argument value is passed to the specified DEFFN statement. The DEFFN expression is then evaluated, using the argument value in place of each occurrence of the dummy variable, and the result is returned to the FN function. For example, in the statement

```
:50 V = FNA(C*2)
```

"A" is the function name which identifies the defined function being referenced, and "C*2" is the argument whose value will replace the dummy variable in the DEFFN expression.

Example:

The following brief routine illustrates the use of a user-defined function:

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

When line 30 is executed, the system first evaluates the expression (C*2). The result, 6, is the argument value passed to the defined function named "A". Every occurrence of the dummy variable "X" in defined function "A" (line 10) is replaced with the argument value, 6: DEFFN A(6) = 6↑2-6 = 30. The result, 30, is then returned to the referencing FN function, where it may be assigned to a variable, printed, or used as an argument in a larger expression.

A defined function may not refer to itself, but it may refer to other defined functions. For example:

```
:10 DEFFN 1(A) = A↑2-A
:20 DEFFN 2(A) = A + FN1(A)
```

A maximum of five defined functions may be "nested" within a single DEFFN statement in this manner.

Two functions may not refer to each other (producing, in effect, an endless loop). For example, the following pair of statements is illegal:

```
:10 DEFFN 1(A) = FN2(A)
:20 DEFFN 2(A) = FN1(A)
```

The DEFFN statement is not executed when encountered during the normal sequence of execution; its execution is controlled exclusively by reference from an FN function. For this reason, a DEFFN statement can appear anywhere in a program without regard for where references to the function may occur. Neither a DEFFN statement nor an FN function may be used in Immediate Mode.

Examples of Valid Syntax:

```
10 DEFFN A(C) = (3*A) - 8*C
20 DEFFN 1(A3) = (EXP(A3) - EXP(-A3))/2
30 DEFFN A(C) = FNB(C)*FNC(C)
40 S = FNA(2*X)
50 PRINT (X + FN1(A3/2))↑2
```

DEFFN'
Keyboard Text Entry Definition

General Form:

DEFFN' integer literal-string] . . .

Where:
0 <= integer <= 255

Purpose:
The DEFFN' statement has two purposes, the first of which is described here:

1. To define a character string to be supplied when a Special Function Key is used for keyboard text entry.

2. To identify the beginning of a subroutine that can be called by depressing a Special Function Key or executing a GOSUB' statement. See the discussion of DEFFN' (Subroutine Entry Point) later in this section.

The DEFFN' statement must be the first statement on the 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 represents one of the keyboard Special Function Keys. All 2200VP/MVP keyboards have at least 32 Special Function Keys numbered '0-'31. Some keyboards have an additional key labelled "FN" that is Special Function Key '126 when unshifted and '127 when shifted. When the corresponding Special Function Key is depressed, the literal string defined in the DEFFN' statement is displayed and becomes part of the text line currently in the input buffer.
For example, statement 100 defines Special Function Key '12 as the character string "HEX(":

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

Depressing Special Function Key '12 after the following text has been keyed in

```
:200 PRINT
```

produces the following line:

```
:200 PRINT HEX(
```

More than one literal string can be specified in the same DEFFN' statement; individual literal strings must be separated by semicolons. For example, line 100 defines Special Function Key '05 as the character string "BOSTON, MASS." followed by a carriage return (HEX (0D)):

```
100 DEFFN' 05 "BOSTON, MASS."; HEX(0D)
```

In this case, the carriage return code causes the line to be executed as soon as Special Function Key '05 is depressed; there is no need for the operator to key RETURN.
The keyboard can be customized by defining the Special Function Keys as characters that do not appear on the keyboard. Keyboard customization is achieved by using the hex form of a literal string to specify the codes for the special characters. A carriage return (HEX (0D)) code always terminates the definition; characters following a carriage return in a DEFFN' statement are ignored. Note that certain hex codes have a special meaning to the system and cannot be used to represent characters. Hex codes FA-FF are reserved for system use and cannot be used in program statements or Immediate

Mode lines. A Special Function Key defined with one of these codes is ignored if depressed during Console Input. When entering data in response to an INPUT request, the hex codes for double quotation marks, the comma, and the carriage return have a special significance. For LINPUT operations, only the carriage return code has a special significance.

Examples of Valid Syntax:

```
10 DEFFN'1 "REWIND"
20 DEFFN'31 HEX(5C)
30 DEFFN'02 "40 HRS.";HEX(0D)
```

DEFFN'
Subroutine Entry Point

General Form:

DEFFN' integer [(variable [,variable]. . .)]

Where:
0 <= integer <= 255

Purpose:
The DEFFN' statement has two purposes, the second of which is described here:

1. To define a character string to be supplied when a Special Function Key is used for keyboard text entry. See the discussion of DEFFN' (Keyboard Text Entry Definition) found earlier in this section.

2. To identify the beginning of a subroutine that can be called by depressing a Special Function Key or executing a GOSUB' statement.

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

DEFFN' Subroutine Entry Definition
The DEFFN' statement, followed by an integer and an optional variable list enclosed in parentheses, is used to indicate the beginning of a subroutine. The DEFFN' subroutine may be entered from the program via a GOSUB' statement (see the discussion of GOSUB' later in this section) or from the keyboard by depressing the appropriate Special Function Key. 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. The keyboard model being used determines which DEFFN' subroutines can be accessed from the keyboard. Execution of the program then begins at the corresponding DEFFN' statement. (For example, if Special Function Key '02 is depressed, program execution begins at the DEFFN'2 statement.)
When a RETURN statement is encountered in the subroutine, control is passed back to the program statement immediately following the last-executed GOSUB' statement or back to the keyboard (Console Input Mode) if entry was made by depressing a Special Function Key.

Example:

```
10 GOSUB '2
20 FOR I=1 TO 20
     .
     .
     .
100 DEFFN '2
     .      }
     .      } subroutine
     .      }
190 RETURN
```

Executing line 10 causes a transfer to line 100; when the RETURN statement is executed, control is passed back to line 20. If the system was waiting for keyboard entry and Special Function Key '02 was depressed, program execution would commence at line 100, and upon execution of the RETURN statement, execution would terminate and the system would return control to the keyboard.

Passing Arguments to Subroutines

The DEFFN' statement may optionally include a variable list. The variables in the list receive the values of arguments passed to the subroutine from a GOSUB' statement or from the keyboard if the DEFFN' statement is accessed from the keyboard with a Special Function Key. If a GOSUB' subroutine call is made within a program, the arguments must be listed (enclosed in parentheses and separated by commas) in the GOSUB' statement (see the discussion of GOSUB' later in this section). If the number of arguments passed to the subroutine does not equal the number of variables in the DEFFN' variable list or if values and variables do not match by type (numeric to numeric, alpha to alpha), an error is signalled.

Example:

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

If the subroutine call is made from the keyboard with a Special Function Key, the arguments to be passed can be typed in prior to depressing the Special Function Key. For example, data could be entered from the keyboard and passed to the receiving variable list in the DEFFN' statement in line 200 above in the following way:

```
:1.2, 3.24, JOHN (depress Special Function Key '02)
```

If the number of values entered is not sufficient to satisfy all variables in the DEFFN' variable list, the system displays a question mark (?) and waits for the operator to enter the remaining arguments. If an illegal value is entered (i.e., numeric value for alpha-variable or alpha literal string for numeric-variable), the system accepts any legal arguments up to the illegal value, then signals an error, displays a "?", and waits for the operator to enter the remaining arguments. The operator must first reenter a new value in place of the illegal value, then enter any succeeding values, and finally key RETURN. (This procedure is identical to the error response procedure for INPUT; see the discussion of the INPUT statement later in this section.) For example:

```
(Depress Special Function Key '02)
? 1.2, 3.24 (RETURN)
? JOHN (RETURN)
```

The DEFFN' statement does not permit the specification of a message to be displayed when data is requested. For this reason, it is usually more convenient to request data from the keyboard in a prompted fashion using an INPUT or LINPUT statement. These methods of requesting data involve writing the INPUT or LINPUT statement in a subroutine and using a DEFFN' statement to identify the subroutine. For example:

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

In the above example, depressing SF Key '04 initiates execution of the subroutine starting at line 100 (DEFFN' 4), which immediately causes the INPUT statement at line 110 to be executed. Note that when program execution is initiated via a Special Function Key, no program resolution is performed. For this reason, the program should be run initially with a RUN command to ensure that a proper resolution phase is carried out; execution of an unresolved program may result in program errors.

The DEFFN' statement may be used with the Special Function Keys to provide a number of entry points to begin execution of a program. Because, however, the system stores DEFFN' subroutine return information in a fixed-length system stack, subroutines should not be accessed repeatedly from the keyboard unless one of the following conditions is met:

1. The subroutine terminates with a RETURN statement, which deletes return information for that subroutine from the stacks.
2. The subroutine return information is removed from the stacks by executing a RETURN CLEAR statement.
3. RESET is keyed prior to depressing the Special Function Key. (RESET clears the stacks.)

Failure to meet at least one of these conditions will eventually cause a Stack Overflow error.

DIM

General Form:

DIM dim-element [,dim-element] . . .

Where:

dim-element = { numeric-scalar-variable
numeric-array-name (dim1 [,dim2])
alpha-array-name (dim1 [,dim2])[length]
alpha-scalar-variable [length] }

dim1, dim2 = dimensions (positive integers or numeric-scalar-variables) such that 1 <= dim1, dim2 <= 255 for two-dimensional arrays; 1 <= dim1 <= 65535 for one-dimensional arrays.

length = positive integer or numeric-scalar-variable such that 1 <= length <= 124.

Purpose:

The DIM statement reserves space for noncommon variables which are to be referenced in a program. A single DIM statement can be used to reserve space for more than one variable by separating the successive dim-elements with commas. The dimensions of arrays and lengths of alphanumeric-variables or array elements are normally specified by positive integers; zero is not allowed. Numeric-scalar-variables, however, can also be used to specify array dimensions and the lengths of array elements and alpha-variables. If a length is not specified for an alpha-variable, it is assumed to be 16 bytes. All array-variables must be defined in DIM or COM statements before any reference to the array can occur in a program. Scalar-variables need not be defined in DIM or COM statements; however, alpha-scalars are automatically assigned a length of 16 characters if not otherwise defined in a DIM or COM statement.

The DIM statement is processed during program resolution, after a RUN command is entered. Before program execution begins, the entire program in memory is scanned sequentially for variable occurrences. When a DIM or COM statement is encountered, space is reserved for each specified variable if the variable has not already been defined. If the variable has previously been defined, an error results unless the dimensions and length are identical to those specified in the previous definition. If a scalar-variable which has not been defined is encountered in the program, space is automatically reserved for it; the length for alpha-scalars is assumed to be 16 bytes. If a reference to an array occurs outside a DIM or COM statement, an error results if the array has not been defined or if the dimensions do not agree with those specified in the previous definition.

Examples:

10 DIM N(45)	Reserves space for a one-dimensional numeric-array with 45 elements.
20 DIM A(8,10)	Reserves space for a two-dimensional numeric-array with 8 rows and 10 columns (80 elements).
30 DIM K(35),L(3),M(8,7)	Reserves space for two one-dimensional arrays and one two-dimensional array.
40 DIM A$32	Reserves space for an alpha-scalar-variable with a length = 32.

50 DIM B$(4,4)10	Reserves space for a two-dimensional alpha-array with 16 elements; the length of each element = 10.
60 DIM C8$(50)	Reserves space for a one-dimensional alpha-array with 50 elements; the length of each element = 16 (default length).

Advanced Use of Scalar-Variables in the DIM Statement

The dimensions and lengths of variables in DIM statements can be specified by numeric-scalar-variables. The numeric-scalar-variables must have legal values for dimensions and lengths at the time the DIM statement is processed during program resolution. When a RUN command without a line-number or a program overlay (LOAD statement) is executed, all noncommon variables are cleared from memory; thus, scalar-variables used to specify dimensions and lengths of variables in DIM statements generally should be defined as common variables (see the discussion of the COM statement earlier in this section).

This feature permits array sizes to be dynamically altered by the operator at run time if the first program module in the system is used to define the array sizes for succeeding modules. For example, the first program module may determine the array dimensions that are to be used:

```
5 REM THIS IS MODULE 1
10 COM X,Y
20 INPUT "DIMENSIONS",X,Y
30 LOAD "MODULE2"
```

The second module might use those dimensions for the array definition:

```
5 REM THIS IS MODULE 2
10 DIM N(X,Y)
20 . . .
```

Examples of Valid Syntax:

```
10 DIM N(5,10), A$(2,3)64
20 DIM X,A$,B$17
30 DIM M$(X,Y)L, Q(R,C)
```

END

General Form:

END

Purpose:

The END statement is an optional program statement indicating the end of a BASIC program's job flow. It need not be the last executable statement in a program, however, and more than one END statement may be used in a program. When the END statement is encountered in a program, it terminates program execution and displays the amount of free space currently available in user memory.

The END statement also can be used in Immediate Mode. When a program has been keyed into the system, an END statement may be entered without a line-number (Immediate Mode) to obtain the amount of free space currently available in user memory. Since the END statement occupies four bytes, the amount of free space displayed is actually four bytes less than the amount of free space currently available in user memory. An END statement executed under program control flushes the system stacks (Operator Stack and Value Stack), while an END statement executed in Immediate Mode does not affect the stacks.

Example:

```
:END

END PROGRAM
FREE SPACE=2379
:
```

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

1. When a program is keyed in or loaded from a peripheral device following a CLEAR command, the free space displayed by the END statement in Immediate Mode reflects only the space occupied by the program text and the Immediate Mode line since variable space has not yet been allocated.

2. After the program has been executed once, the free space displayed after either an Immediate Mode END or an END statement executed under program control reflects the total space occupied by the program and all variables.

NOTE:

BASIC-2 also provides a system function, the SPACE function, which returns a free space value. Unlike END, however, SPACE returns a free space value which includes the space occupied by the Value Stack. Refer to Chapter 2, section 2.5 for a comparison of the free space values returned by SPACE and END.

Examples of Valid Syntax:

```
999 END
END
```

FOR...TO

General Form:

FOR counter = expression-1 TO expression-2 [STEP expression]

Where:
counter = numeric-scalar-variable

Purpose:

The FOR. . .TO statement is used with its companion, the NEXT statement, to construct a loop. FOR. . .TO marks the beginning of the loop and defines the loop parameters. NEXT marks the end of the loop (see the discussion of the NEXT statement later in this section). The statements between FOR. . .TO and NEXT are executed repeatedly until the exit value of the loop is reached.

FOR. . .TO defines the loop parameters and stores them in a system stack in memory. The numeric-scalar-variable specified as the counter-variable serves as the loop counter and is automatically incremented or decremented each time through the loop. The initial value of the counter-variable is specified in expression-1. Expression-2 specifies the exit value for the loop; when the value of the counter either exceeds or falls below the exit value, the loop is terminated. Optionally, a STEP value may be specified which defines the amount by which the counter is to be incremented or decremented each time through the loop. The expression used for the STEP value may be positive or negative. If no STEP value is specified, a default value of +1 is used.

Each time through the loop, the STEP value is added to the value of the counter, and the result is compared with the exit value. The type of comparison made is determined by the sign of the STEP value. If the STEP value is positive, the counter is tested to determine whether it exceeds the exit value; if the STEP value is negative, the counter is tested to determine whether it is less than the exit value. When the counter either exceeds or falls below the exit value, the loop is terminated and the normal sequence of execution resumes with the statement immediately following NEXT. Otherwise, the loop is reexecuted.

Although the FOR. . .TO statement is required to set up the loop parameters initially, it is executed only once when the loop begins and does not form a part of the loop itself. The subsequent loop operations, including the tasks of incrementing the counter, comparing it with the exit value, and reexecuting the loop, are controlled by the NEXT statement.

The relationships between the initial counter value, the exit value, and the STEP increment in a FOR. . .TO statement are significant. For example, in the statement

```
50 FOR I=1 TO 5 STEP -1
```

the negative STEP increment instructs the NEXT statement to test for a counter value less than the exit value. In this case, however, the loop begins with a counter value less than the exit value. Following the first execution of the loop, therefore, the termination condition will be satisfied, and the loop will be prematurely ended after a single execution.

Several special conditions should be noted regarding the use of FOR. . .TO/NEXT loops:

1. It is illegal to branch into the middle of a FOR. . .TO/NEXT loop from elsewhere in a program. Since the loop parameters are defined by FOR. . .TO, the execution of a NEXT statement for which no corresponding FOR. . .TO statement has been executed results in an error.

2. It is legal to branch out of a FOR. . .TO/NEXT loop before the loop is terminated; however, a problem will arise if this operation is repeated often within a program. The loop parameters defined in a FOR...TO statement are stored in a system stack in memory. When a loop termi-

nates in the normal manner, this information is cleared from the stacks. If the loop is not terminated, the information remains in the stacks; thus, repeatedly branching out of an unterminated loop causes the loop parameter information to accumulate in the stacks, eventually producing a stack overflow. In addition to normal loop termination, there are two other ways to clear loop information from the stacks. For loops contained within a subroutine, execution of the subroutine RETURN or RETURN CLEAR statement automatically clears the parameters of all loops within the subroutine. For nested loops, execution of the NEXT statement of an outer loop automatically clears the parameters of all inner loops. Once the parameters of a loop have been cleared from the stacks, execution of a NEXT statement in that loop is illegal.

3. It is legal to branch out of a loop with a RETURN statement. Execution of the RETURN statement automatically clears all loop parameters from the system stacks.

4. There is no practical limit to the number of loops which can be nested in a program.

5. A loop with a STEP value of zero is executed only once.

Examples:

1. A STEP value of 1 is used if not specified.

```
10 FOR X = 1 TO 16
20 PRINT X, SQR(X)
30 NEXT X
```

2. The STEP value can be negative as well as positive.

```
10 FOR I = 10 TO 8 STEP -.2
20 PRINT I, LOG(I)
30 NEXT I
```

3. Termination of an outer loop properly terminates the inner loop.

```
10 FOR I = 1 TO 4
20 FOR J = 1 TO 6
30 READ A(I,J)
40 IF A(I,J) = 9999 THEN 60
50 NEXT J
60 NEXT I
```

4. RETURN properly terminates loops contained within subroutines.

```
10 X = 3: Y = 100
20 GOSUB 100
30 END
100 S = 0
110 FOR B1 = X TO Y
120 S = S+B1
130 IF S>1000 THEN 150
140 NEXT B1
150 RETURN
```

5. Loops can be terminated by setting the counter to the final value and then executing a NEXT statement.

```
10 INPUT X
20 S = 1
30 FOR I = 2 TO X
40 S = S*I
50 IF X<=S THEN 100
60 NEXT I
70 PRINT S
80 GO TO 10
100 I = X: NEXT I: GOTO 10
```

GOSUB

General Form:

GOSUB line-number

Purpose:

The GOSUB statement is used to transfer program execution to the first program line of a subroutine. (GOSUB is illegal in Immediate Mode.) The designated program line may contain any BASIC statement, including a REM statement. The subroutine normally ends with a RETURN statement which returns program execution to the statement immediately following the last-executed GOSUB. The transfer back to the main program occurs as soon as a RETURN statement is encountered. Statements which follow RETURN on the same program line are ignored.

A subroutine may be executed within a larger subroutine. This technique is called "nesting" of subroutines. There is no practical limit to the number of levels to which subroutines can be nested within a program.

Subroutine return information is stored in a pair of system stacks in memory. When a RETURN statement is executed, return information for the corresponding subroutine is cleared from the stacks. If no RETURN is executed, however, the information is not cleared. Thus, repeated entry to subroutines without executing RETURN statements causes the return information to accumulate in the stacks. If this situation is repeated frequently, an overflow error will eventually occur. Subroutine return information can be cleared without causing a return to the main program by executing a RETURN CLEAR statement (see the discussion of the RETURN CLEAR statement later in this section).

Examples:

1. Subroutine

```
10 GOSUB 30
20 PRINT X: STOP
30 REM THIS IS A SUBROUTINE
40 INPUT A,B        } subroutine
50 X = A↑2+B↑2      }
60 RETURN: REM END OF SUBROUTINE
```

2. Nested Subroutines

```
10 GOSUB 100
20 PRINT "RESULT=";R
30 END
100 REM THIS IS A SUBROUTINE
110 GOSUB 200          }
120 R = SIN (X)        }
130 GOSUB 200          } subroutine
140 R = R + COS(X)     }
150 RETURN             }
200 REM THIS IS A NESTED SUBROUTINE
210 INPUT "VALUE",X                    nested subroutine
220 RETURN: REM END OF NESTED SUBROUTINE
```

GOSUB'

General Form:

GOSUB' integer [(subroutine-argument [,subroutine-argument]. . .)]

Where:

0 <= integer <= 255

subroutine-argument = { literal-string | alpha-variable | numeric-expression }

Purpose:

The GOSUB' statement is used to transfer program execution 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 the discussion of the DEFFN' statement earlier in this section). When a GOSUB' statement is executed, program execution is transferred to the DEFFN' statement having an integer identical to the one specified in the GOSUB' statement (e.g., GOSUB'6 transfers execution to the DEFFN'6 statement). Within the subroutine, normal execution continues until a subroutine RETURN statement is executed, at which point program execution is transferred back to the statement immediately following the last-executed GOSUB'.

Like unmarked subroutines (see the discussion of the GOSUB statement found earlier in this section), marked subroutines can be nested within a program so that a marked subroutine can be called from within another subroutine. There is no practical limit to the number of subroutines which can be nested in a program. The use of GOSUB' is not permitted in Immediate Mode, but the Special Function Keys can be used to access corresponding subroutines from the keyboard (see the discussion of the DEFFN' statement found earlier in this section).

Return information for a marked subroutine is stored in a pair of system stacks in memory and is cleared from the stacks only when a RETURN statement for that subroutine is executed. Repeatedly accessing marked subroutines which do not end with a RETURN statement causes return information to accumulate in the stacks, eventually causing an overflow error. Return information can be cleared from the stacks without effecting a transfer back to the main program by executing a RETURN CLEAR statement (see the discussion of the RETURN CLEAR statement found later in this section).

Example:

```
:10 GOSUB'7
:20 STOP
:30 DEFFN'7: REM START OF SUBROUTINE
:40 PRINT "THIS IS SUBROUTINE '7"
:50 RETURN: REM END OF SUBROUTINE
:RUN
THIS IS SUBROUTINE '7
STOP
```

Passing Arguments to Subroutines

Unlike a normal GOSUB, a GOSUB' statement can contain arguments whose values are passed to variables in the corresponding DEFFN' statement. The values of numeric expressions, literal strings, or alphanumeric-variables are assigned sequentially to the variables in a DEFFN' statement (see the discussion of the DEFFN' statement found earlier in this section). Alphanumeric values must be assigned to alphanumeric-variables, and numeric values to numeric-variables.

Example:

```
:25 GOSUB'12 ("JOHN", 12.4, 3*2+7)
:30 STOP
:100 DEFFN'12 (A$,B,C(2))
:110 PRINT A$,B,C(2)
:120 RETURN
:RUN
JOHN                 12.4                 13
STOP
```

GOTO

General Form:

GOTO line-number

Purpose:

The GOTO statement is used to transfer program execution to a designated location in the program. When a GOTO statement is executed, program execution is transferred to the first statement on the program line having the specified line-number. GOTO does not perform a subroutine call, however, and does not save the location of the next sequential statement prior to branching. Therefore, it is not possible to return to the statement immediately following GOTO by executing a RETURN.

The GOTO statement also can be used in Immediate Mode to enable the operator to begin stepping through program execution from a particular line-number. However, the program must have been resolved previously by executing a RUN command. Execution of the GOTO statement sets the system at the specified line, but program execution does not take place until the HALT/STEP Key is touched or CONTINUE is keyed followed by RETURN.

Example:

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

END PROGRAM
FREE SPACE = 3841
```

IF...THEN

General Form (Simple):

IF operand-1 { < | <= | = | >= | > | <> } operand-2 THEN line-number

Where:

operand = { literal-string | alpha-variable | numeric-expression }

Purpose:

In its simple form, the IF...THEN statement causes a branch to a designated program line if a specified condition is true. If the specified condition is false, no branch is taken and the normal sequence of execution continues with the next statement.

The condition to be tested by the IF...THEN statement is stated as a relation between two operands. The pair of operands being compared may be numeric or alphanumeric; however, a numeric operand cannot be compared with an alphanumeric operand.

The relations which can be tested by an IF...THEN statement are explained below (numeric operands are used for illustration).

Relation Tested	Explanation of Relation
A<B	A less than B
A<=B	A less than or equal to B
A=B	A equal to B
A>=B	A greater than or equal to B
A>B	A greater than B
A<>B	A not equal to B

Comparison of two alphanumeric operands proceeds character-by-character, from left to right. The comparison of any two characters is based upon the respective values of their hexadecimal codes. (For example, the hex code for an "A" is 41 and the hex code for a "B" is 42. Thus, the relation "A"<"B" is true, while "A">"B" is false.) This character-by-character comparison continues until unequal characters are found. The first pair of unequal characters determines the relationship between the two values. If no unequal characters are found, the two values are equal. In general, trailing spaces in an alpha-variable are not included in the comparison. Thus, "YES" = "YES " is true. However, if two character strings of unequal length are compared, the shorter string is automatically extended to the length of the longer string with one or more space characters (HEX (20)), and these characters are then used in the comparison. For example, if "ABCD" is compared to "ABC", the shorter string is padded with a space character, yielding "ABC ". The first pair of unequal characters in this case are "D" and "space." The relation "ABCD" > "ABC " is true since the hex code for "D" (HEX (44)) is greater than that of the space character (HEX (20)).

In its simple form, the IF...THEN statement is not legal in Immediate Mode. Other forms of the IF...THEN statement may, however, be used in Immediate Mode. (See the discussion of the complex form of the IF...THEN statement later in this section.)

Examples of Valid Syntax:

```
10 IF A>B THEN 35
20 IF A$ = "YES" THEN 100
30 IF A$=HEX(8082) THEN 200
40 IF X(1) < .001 THEN 350
50 IF STR(A$,1,3) <> B$(1) THEN 500
```

IF...THEN

General Form (Complex):

IF condition THEN { line-number / statement } [:ELSE statement]

Where:

condition = one or more relations separated by the logical operators AND, OR, or XOR.

Purpose:

The complex form of the IF...THEN statement permits multiple relations to be tested in a single statement. The types of relations which may be tested are listed in the discussion of the simple form of the IF...THEN statement earlier in this section. Relations are separated by the logical operators AND, OR, and XOR. If AND separates two relations, the pair of relations is true if and only if *both* simple relations are true. If OR separates two relations, the pair of relations is true if and only if *at least one* simple relation is true. If XOR separates two relations, the pair of relations is true if and only if *exactly one* of the simple relations (but not both) is true. Relations are processed from left to right.

Examples:

```
10 IF A > B AND B < C THEN 100
20 IF A$=B$ OR A$=C$ THEN 100
30 IF A<1 XOR B>0 THEN 100
```

Any BASIC statement may be used after the word "THEN" except a DEFFN, DATA, or Image (%) statement. If a line-number follows the word "THEN", the expression "THEN line-number" is interpreted as "GOTO line-number"

Examples:

```
10 IF A=B THEN PRINT "ERROR"
20 IF A$=B$ THEN X=1
```

The IF...THEN statement also may be followed by an ELSE clause. The word "ELSE" may be followed by any statement except a DEFFN, DATA, or Image (%) statement. The ELSE clause is executed if and only if the IF condition is false. For example, the statement

```
20 IF A > B THEN PRINT "OK": ELSE PRINT "ERROR"
```

prints "OK" if A>B and "ERROR" if A<=B.

Note that an ELSE clause may contain only one statement. Any statements following the ELSE clause are not considered part of the IF...THEN statement and are executed in normal program sequence regardless of the validity of the IF condition. If the IF...THEN statement or ELSE clause initiates a branch to another program line, however, statements following the ELSE clause are not executed in normal execution sequence. For example, in the statement below, X is set = 5 regardless of whether Z=1 or not:

```
IF Z=1 THEN Y=2: ELSE Y=0: X=5
```

Examples of Valid Syntax:

```
10 IF A>B OR C>B THEN Y=1
20 IF A$=T$ AND B$=C$ THEN 100
30 IF A$="T" OR B$="X" THEN PRINT "ERROR": ELSE GOTO 100
40 IF A>B AND A<C AND B<>0 THEN A=B*C
```

IF END THEN

General Form:

IF END THEN { line-number / statement } [:ELSE statement]

Purpose:

The IF END THEN statement is used to test for the presence of an end-of-file record (data trailer record) when reading data records from a disk file and to either branch to a specified line or execute a specified statement when the end-of-file record is read.

When an end-of-file record is read with a DATALOAD DC or DATALOAD DA statement, the system sets the end-of-file flag. Subsequently, the status of this flag can be checked with an IF END THEN statement. If the flag is set (indicating that an end-of-file record has been read), the IF END THEN statement resets the flag (turns it off) and either transfers program execution to the specified line-number or executes the specified statement following the word "THEN." If the flag is not set (indicating that a normal data record has been read), the IF END THEN statement causes no action, and program execution continues at the next statement or with the optional ELSE clause.

The end-of-file flag is set (turned on) when an end-of-file record is read from a data file by a DATALOAD DC or DATALOAD DA statement. (The values of subsequent variables in the DATALOAD DC/DA variable list are not changed when an end-of-file record is read.) Subsequently, the flag is turned off either by an IF END THEN statement or another DATALOAD DC/DA statement. The RUN and CLEAR commands also turn off the end-of-file flag.

If the ELSE clause is included in an IF END THEN statement, it is executed if and only if the end-of-file flag is off.

Programming Example:

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

Examples of Valid Syntax:

```
10 IF END THEN X=7
20 IF END THEN N=0: ELSE N=1
30 IF END THEN 1000
40 IF END THEN GOSUB' 250
```

Image (%)

General Form:

%[character-string] [format-specification] . . .

Where:

character-string = a string of alphanumeric characters which does not contain a "#" character.

format-specification = [+ / −] [$] [#[,]. . .] [.] [#. . .] [↑↑↑↑] [+ / − / ++ / −−]

where:

at least one "#" is required. Both leading and trailing signs may not be present.

Purpose:

The Image (%) statement is used with a PRINTUSING statement to provide a format-specification for formatted output. The Image statement is identified with a percent (%) character, which appears as the first character in the line. The Image statement contains text to be printed, along with the format-specifications used to format print-elements contained in the PRINTUSING statement. For a complete discussion of the use of images, see the discussion of the PRINTUSING statement later in this section.

An Image statement may have any printable characters inserted before and after the format-specifications. Each format-specification contains at least one digit-selector character (#) as well as any of the following special symbols: ($), (+), (−), (.), (++), and (−−). Commas (,) may be embedded in the integer portion of a format-specification after the first "#" character but before the decimal point (.) or up-arrow symbols (↑↑↑↑). Either a leading or a trailing sign may be specified in the format-specification, but *not both.*

The Image statement must be the only statement on the statement line; colons are interpreted as part of the image rather than as statement separators.

Examples of Valid Syntax:

```
10 %CODE NO. = ####COMPOSITION=## ###
20 %####UNITS AT $#,###.## PER UNIT
30 %+#.##
```

INPUT

General Form:

INPUT [literal-string [,]] variable [, variable]. . .

Purpose:

The INPUT statement allows the operator to supply data during the execution of a program. Data values entered by the operator are assigned sequentially to the variables specified in the INPUT variable list. Optionally, a message may be included in the INPUT statement which prompts the operator to enter the required data. For example, either of the following statements will enable the operator to enter a pair of numeric values and assign them to the variables A and B:

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

When the system encounters an INPUT statement, it displays the optional input message (in this case, "VALUE OF A,B"), followed by a question mark (?). If no message is specified, only the question mark is displayed. The system then 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 entering data in response to an INPUT request is the Console Input device unless another device has been specified by using the SELECT INPUT statement (see the discussion of the SELECT statement in Chapter 7, section 7.4).

Values entered by the operator are assigned sequentially to the variables in the INPUT statement variable list. If more than one value is entered on a line, individual values must be separated by commas. Alternatively, each value may be entered on a separate line. Several lines may be used to enter the required INPUT data. If there is a system-detected error in the entered data, the values 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 keying RETURN without entering any data. This action causes the system to terminate execution of the INPUT statement (remaining variables in the INPUT variable list remain unchanged) and proceed immediately to the next statement following INPUT.

When alphanumeric data is entered, 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 in a character string, the string must be enclosed in quotes.

Example 1:

```
:10 INPUT X
:RUN
?12.2 (RETURN)
```

Example 2:

```
:20 INPUT "X,Y",X,Y
:RUN
X,Y? 1.1, 2.3 (RETURN)
```

Example 3:

```
:20 INPUT "MORE INFORMATION" A$
:30 IF A$="NO" THEN 50
:40 INPUT "ADDRESS",B$
:RUN
MORE INFORMATION? YES (RETURN)
ADDRESS? "BOSTON, MASS" (RETURN)
```

Advanced Use of the Special Function Keys in INPUT Mode

Special Function Keys may be used in conjunction with the INPUT statement. If a Special Function Key has been defined for text entry (see the discussion of the DEFFN' statement earlier in this section) and the system is awaiting input, depressing the Special Function Key will cause the character string associated with that key to be entered. For example:

```
:10 DEFFN'01 "HOBBITS"
:20 INPUT A$
:RUN
?
Depressing Special Function Key '01
will cause "HOBBITS" to be entered:
? HOBBITS
```

If the Special Function Key is defined to call a DEFFN' subroutine (see the discussion of the DEFFN' statement earlier in this section) and the system is awaiting input, depressing the Special Function Key causes the specified subroutine to be executed. When the subroutine RETURN is encountered, a branch is made back to the INPUT statement, which is then executed again. All INPUT values must now be entered again. INPUT values entered prior to depressing the SF Key are lost and must be reentered. Subroutine return information is stored in a system stack and cleared when a RETURN is executed for that subroutine. Thus, repetitive subroutine entries via Special Function Keys should not be made unless the subroutine RETURN is always executed or the return information is removed by executing a RETURN CLEAR statement. Failure to clear the subroutine return information eventually will cause a Stack Overflow error (ERR A04).

For example, the following program enters and stores a series of numbers. When Special Function Key '02 is depressed, the numbers 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:T=T+A(I):NEXT I
:100 PRINT "TOTAL =";T
:110 N=1
:120 RETURN
:RUN
AMOUNT?#7
AMOUNT?#5
AMOUNT?#11
AMOUNT?      (Depress Special Function Key '02)
TOTAL = 23
AMOUNT?
```

KEYIN

General Form:

1. KEYIN [device-address, / file-number,] alpha-variable [,,line-number]

2. KEYIN [device-address, / file-number,] alpha-variable, line-number, line-number

Where:

device-address = /taa, where t = one hex digit specifying the device-type and aa = two hex digits specifying the unit device-address.

file-number = #n, where n = an integer or numeric-variable whose integer value ranges from 0 to 15.

Purpose:

The KEYIN statement is used to receive a single character from an input device. KEYIN provides a convenient mechanism for scanning several input devices or for receiving and editing keyed-in information on a character-by-character basis.

The input device from which KEYIN receives data is specified directly with a device-address (/taa) or indirectly with a file-number (#n). If neither a device-address nor a file-number is specified, the device currently selected for INPUT operations is used by KEYIN.

General form 1 of the KEYIN statement contains either no line-number or a single line-number preceded by two commas. This form of the KEYIN statement waits for a character to be input from the specified input device. When using an MVP, general form 1 is preferred since it does not waste CPU time by repeatedly polling the input device. The program waits until the input device sends a character. When a character is received, it is stored in the alpha-variable, and execution resumes at the next statement. Including the optional line-number enables KEYIN to distinguish between regular characters and codes generated by the Special Function Keys. If a standard character is received, execution continues at the next statement; if a Special Function Key code is received, execution is transferred to the line specified by the line-number.

General form 2 of the KEYIN statement checks a specified input device once. If a character is ready to be input from the device, KEYIN receives it into the alpha-variable and takes the appropriate action. The action taken by KEYIN depends upon whether the character received is a standard character or a Special Function Key code. If a standard character is received, KEYIN causes a branch to the first line-number; if a Special Function Key code is received, KEYIN branches to the second line-number. If no character is ready to be transferred, KEYIN terminates and execution continues at the next statement.

"Dead Key" Operation

Certain VP/MVP keyboards support "dead key" operations for underlining and accenting characters. (Dead keys are those which do not advance the cursor when depressed.) On some keyboards, the dead keys operate "blindly", i.e., when an accent or underline key is depressed, no code is sent to the CPU and no output is displayed on the screen. If the next character entered may be accented or underlined, the code representing the accented or underlined character is sent to the CPU.

Other keyboards have visible accent and underline operations. On these keyboards, the accent or underline code is sent to the CPU preceded by a HEX(FF) code flagging the next code as a special character. The KEYIN program may wish to display the accent or underline and move the cursor back under the accent by issuing a backspace code, HEX(08). The next character received would be combined with the accent or underline in order to emulate the operation of accents and underlines with CI, INPUT, and LINPUT. If the BACKSPACE Key is pressed immediately after an accent or underline key, a HEX(FF) code precedes the backspace code, HEX(08), when both codes are sent to the CPU.

Foreign Character Codes

Foreign characters whose codes are HEX(80) and above are sent to the CPU with the Special Function (ENDI) bit set in order to distinguish them from text atoms, which are codes that represent BASIC-2 keywords. Foreign character codes will thus cause KEYIN to branch to the line-number specified for the Special Function Keys.

Examples of Valid Syntax:

General Form 1

```
10 KEYIN A$
20 KEYIN #3, A$,,100
30 KEYIN /002, A$
```

General Form 2

```
10 KEYIN A$, 100, 200
20 KEYIN #3, A$, 100, 200
```

LET (Assignment)

General Form:

[LET] numeric-variable [,numeric-variable]. . . = numeric-expression

or

[LET] alpha-variable [,alpha-variable]. . . = alpha expression

Where:

alpha expression = { alpha-operand [alpha-operator alpha-operand]. . . }
{ alpha-operand [& alpha-operand]. . . }

Purpose:

The assignment (LET) statement directs the system to evaluate the expression on the right-hand side of the equal sign and then assign the result to the variable or variables specified on the left-hand side of the equal sign. If multiple variables appear to the left of the equal sign, they must be separated by commas. An error results if a numeric value is assigned to an alphanumeric-variable or if an alphanumeric value is assigned to a numeric-variable. The word LET is optional in an assignment statement. If it is omitted, its presence is assumed. The assignment statement also is discussed in Chapter 1, section 1.9. Numeric expressions are described in Chapter 4, section 4.5, and alpha expressions are described in Chapter 5, section 5.8.

Examples of Valid Syntax:

```
10 LET J=3
20 X,Y,Z = P+15/2+SIN(P-2.5)
30 A$(3)=B$
40 C$,D$(2)="ABCDE"
50 X = A*B↑C
60 LET A$=HEX(0000)
70 STR(B$,1,3) = C$
80 A$, B$, STR(C$(1),3) = STR(E$,N,M)
90 A$( ) = A$ & B$
100 C$ = A$ AND B$ OR D$
```

LINPUT

General Form:

LINPUT [literal [,]] [?] [–] alpha-variable

Purpose:

The LINPUT (LINe-inPUT) statement allows an operator to enter and edit alphanumeric data, including leading spaces, quotes, and commas, directly into a receiving alpha-variable. The specified alpha-variable serves as an input buffer into which the characters are stored directly as they are entered, character by character. The size of the receiving alpha-variable in bytes may not exceed the total number of characters which can be displayed on the CRT of the VP or 480 characters if the MVP is used.

The LINPUT statement has two distinct forms, distinguished by the presence of an optional question mark.

LINPUT Without Question Mark

When a LINPUT statement without the question mark (?) is executed, the following events take place:

1. The optional message (literal) in the LINPUT statement is displayed, followed by an asterisk (on the VP) or a blank with blinking cursor (on the MVP).
2. The system is placed in Edit Mode, and all Edit Keys are activated.
3. The entire current contents of the specified alpha-variable are recalled and displayed on the screen beginning immediately after the current cursor position.
4. The CRT cursor is positioned at the *first character* of the recalled value.

The operator may enter data or edit the current contents of the alpha-variable directly. As characters are inserted, deleted, or replaced in the display, these changes are simultaneously made in the variable itself so that the display always matches exactly the contents of the alpha-variable. Alternatively, on the VP only, the operator may switch from Edit Mode to Text Entry Mode by depressing the EDIT Key. The operator may then insert data directly into the variable.

Leading spaces and special characters such as quotes and commas, which do not function as terminators, can be entered. The entered character string is terminated by keying RETURN.

LINPUT with Question Mark

If a "?" is included in the LINPUT statement, the LINPUT operation begins in Text Entry Mode rather than Edit Mode, and on the VP, the cursor is positioned *after the last nonblank character* in the field rather than at the beginning of the field. On the MVP, the cursor is positioned at the beginning of the field. LINPUT? is useful when Special Function Keys are used to control field operations because the Edit Key need not be depressed prior to depressing the Special Function Key in order to enter the user subroutine. For example:

LINPUT Without "?"

```
:10 DIM A$5
:20 A$="ABC"
:30 LINPUT "ENTER VALUE"A$
:RUN
VP-   ENTER VALUE*A B C
MVP-  ENTER VALUE A B C
            (Blinking Cursor)
```

LINPUT With "?"

```
:10 DIM A$5
:20 A$="ABC"
:30 LINPUT "ENTER VALUE"?A$
:RUN
VP-  ENTER VALUE*A B C_
MVP- ENTER VALUE A B C
            (Steady Cursor)
```

General Features of LINPUT

During a LINPUT operation, cursor movement is restricted to the region of the CRT display occupied by the value of the receiving alpha-variable. Cursor movement beyond the limits of the alpha-variable is inhibited. (If the operator attempts to enter data beyond the end of the LINPUT field, the audio alarm beeps and the characters are not accepted.) This feature of the LINPUT operation enables the display to be formatted easily for data entry applications.

Although the contents of the receiving alpha-variable are directly altered as data is keyed in, up to 256 bytes of data originally stored in the alpha-variable may be recovered before RETURN has been keyed. Recovery of originally stored data may occur because the original contents of the alpha-variable are copied to a work space area of memory immediately upon execution of the LINPUT statement. The original contents can be returned to the alpha-variable and displayed by touching LINE ERASE and then depressing the RECALL Key. If the system is not in Edit Mode, however, the original contents of the alpha-variable can not be recovered. This procedure must be performed before keying RETURN. Once RETURN is keyed, the original contents of the alpha-variable are cleared from the work area and irretrievably lost.

If a minus sign (−) is specified immediately before the alpha-variable, each character of the alpha-variable is underlined in the display. The actual contents of the alpha-variable in memory are not altered; when RETURN is keyed, the underline is removed from each character in the LINPUT field. The underline feature is particularly useful for generating pseudospace characters which can be used to identify fields to be filled by the operator. This feature is available only on systems with an underlining CRT.

A Special Function Key can be depressed in response to a LINPUT request. If the Special Function Key has been defined for text entry, its associated text string is displayed and inserted into the alpha-variable. If the Special Function Key is defined for subroutine access, execution of the corresponding subroutine is initiated. However, *no* parameters can be passed to a subroutine accessed in this manner. When the subroutine RETURN statement is executed, program control is returned to the statement immediately following the LINPUT statement. (Note that a LINPUT operation differs significantly from an INPUT operation since INPUT allows parameters to be passed to the subroutine and returns control to the INPUT statement itself rather than the next sequential statement when the subroutine RETURN is executed.)

LINPUT, in conjunction with the PRINT AT and the VER functions, provides convenient tools for implementing data entry routines. LINPUT provides a significant advantage over INPUT for data entry applications because the typing of invalid data in response to a LINPUT request does not elicit a system error message. Data is accepted and stored in the alpha-variable exactly as it is entered by the operator. Subsequently, the data can be verified and corrected under program control by using the NUM, VER, and POS functions and the CONVERT statement.

Examples:

```
        :5 DIM A$10
        :10 A$ = " "
        :20 LINPUT "MESSAGE", A$
        :RUN
  VP-   MESSAGE*
 MVP-   MESSAGE _

        :20 LINPUT "MESSAGE", -A$
        :RUN
  VP-   MESSAGE*_ _ _ _ _ _ _ _ _ _
 MVP-   MESSAGE _ _ _ _ _ _ _ _ _ _
              (Blinking cursor at field beginning)
```

MAT COPY

General Form:

MAT COPY [−] source-alpha-variable TO [−] output-alpha-variable

Purpose:

The MAT COPY statement enables the programmer to construct new character strings from one or more existing character strings. MAT COPY transfers data from the source-variable (or a portion of the source-variable specified with a STRing function) to the output-variable (or a portion of the output-variable specified with a STRing function). Each variable is treated as a single contiguous character string. Data is transferred byte by byte until the output-variable or the specified portion of the output-variable is filled. If the amount of data to be copied is not sufficient to fill the specified portion of the output-variable, the remaining bytes are filled with spaces. The source- and output-variables may be the same variable.

If the source-variable is preceded by a minus sign (−), data is transferred in reverse order and stored left-justified in the output-variable (see example 3 below). Similarly, if a minus sign (−) precedes the output-variable, the data is stored in reverse order right-justified in the output-variable (see example 2 below). When minus signs precede both the source-variable and the output-variable, data is copied in the order specified in the source-variable and stored right-justified in the output-variable (see example 4 below).

Examples:

To clarify these various options, consider the examples below in which the following dimensions are assumed for the input-variable A$() and the output-variable B$():

```
DIM A$(1,5)1, B$(1,7)1
```

Further, the following value is assumed for A$():

A$() =	A	B	C	D	E

1. Duplication of Alpha-Variable, Left-Justified

```
MAT COPY A$() TO B$()
```

B$() =	A	B	C	D	E	space	space

2. Reversal of Alpha-Variable, Right-Justified

```
MAT COPY A$() TO -B$()
```

B$() =	space	space	E	D	C	B	A

3. Reversal of Alpha-Variable, Left-Justified

```
MAT COPY -A$() TO B$()
```

B$() =	E	D	C	B	A	space	space

4. Duplication of Alpha-Variable, Right-Justified

MAT COPY -A$() TO -B$()

B$() =

space	space	A	B	C	D	E

Examples of Valid Syntax:

```
10 MAT COPY A$() TO A$()
20 MAT COPY -W$ TO B$()
30 MAT COPY STR(A$(),10,50) TO -W$
40 MAT COPY -STR(W$,5,10) TO -STR(A$,20)
```

MAT MOVE

General Form (Simple):

MAT MOVE move-array [,locator-array] [,n] TO receiver-array

Where:

move-array = { alpha-array-designator [(x[,y])] / numeric-array-designator }

locator-array = { alpha-array-designator / alpha-array-element }

receiver-array = { alpha-array-element [(x[,y])] / numeric-array-element / alpha-array-designator [(x[,y])] / numeric-array-designator }

n = the counter-variable, a numeric-scalar-variable specifying the maximum number of elements to be moved. When the move is complete, the number of elements actually moved is returned to this variable.

(x,y) = designates a field within each alpha-array-element.

x = expression specifying the starting position of the field.

y = expression specifying the number of characters in the field. If y is not specified, its value is assumed to equal the number of remaining characters in the element.

Purpose:

In its "simple form," the MAT MOVE statement is used to transfer data element-by-element from one array to another. Optionally, a "complex form" of the MAT MOVE statement can be used to convert numeric data from Wang internal numeric format to alphanumeric data in sort format and to restore data from sort format to numeric format as part of the transfer operation. However, this feature of MAT MOVE is used only for sorting operations and is documented under the complex form of MAT MOVE in Chapter 14, section 14.2.

If no locator-array is specified, MAT MOVE moves data directly from the move-array, beginning with the first move-array-element, to the receiver-array, beginning at the specified receiver-array-element. If no receiver-array-element is specified, the moved data is stored beginning at the first element of the receiver-array. For example, assume that two arrays A$() and B$() are defined:

```
10 DIM A$(3), B$(3)
```

The statement

```
20 MAT MOVE A$() TO B$()
```

is equivalent to the following statements:

```
20 B$(1) = A$(1): B$(2) = A$(2): B$(3) = A$(3)
```

If the locator-array is specified, data is moved indirectly from the move-array in the order specified by the subscripts in the locator-array, starting with the subscript in the specified locator-array-element. If no locator-array-element is specified, data is moved from the move-array to the receiver-array, beginning with the data element specified by the subscript of the first element in the locator-array. The locator-array is usually produced by a MAT SORT or MAT MERGE statement, and it specifies the order in which values are to be placed in the receiver-array following a sort or merge operation. (See the discussion of locator-arrays in Chapter 14, section 14.1.)

It is possible to transfer a specified field of each element from an alphanumeric move-array to an alphanumeric receiver-array rather than transfer entire elements. Specified fields of each element are transferred by defining the starting location of the field to be moved and, optionally, its length in bytes. For example, the statement

```
20 MAT MOVE A$()(10,5) TO B$()(4,5)
```

specifies that a five-character substring starting at the tenth character position of each element of A$() is to be moved to a five-character field starting at the fourth character position in each element of B$(). If a length value is not specified, the substring is assumed to extend to the end of each element. If elements or defined fields in elements of the receiver-array are longer than the values transferred from the move-array, each transferred value is extended with trailing spaces to the length of the receiving field. If the receiving fields are shorter than the transferred values, the values are truncated to the lengths of the receiving fields.

Data is transferred into the receiver-array row by row. MAT MOVE continues to transfer data until one of the following conditions occurs:

1. The end of the move-array is reached for direct data transfer.
2. The end of the array of subscripts (locator-array) is reached for indirect data transfer.
3. A binary "0000" is found in the array of subscripts.
4. The number of elements specified by the counter-variable (the numeric-scalar-variable preceding "TO") has been moved.
5. The receiver-array has been filled.

When MAT MOVE has finished data transfer, a count of the number of elements moved is returned to the counter-variable if it was specified.

Dimensional Requirements

Move-Array	— any one- or two-dimensional, alpha- or numeric-array.
Locator-Array	— an alpha-array with elements of length 2.
Receiver-Array	— any one- or two-dimensional, alpha- or numeric-array.

Subscripts in the locator-array are binary values which are two bytes in length. If the move-array is a two-dimensional array, the first byte of the subscript specifies the row subscript, and the second byte specifies the column subscript. If the move-array is a one-dimensional array, the subscript is interpreted as a two-byte binary value (from 1 to 65535) identifying an element.

Example:

Suppose an array of data D$( ) and an associated array of subscripts S$( ) exist in memory as shown below. The problem is to move the data from D$( ) to E$( ) in the order specified by S$( ).

D$( ) =

	1	2	3	4
1	B	A	C	D
2	C	B	E	A
3	A	F	A	E

S$( ) =

0102	0303	0301	0204
0202	0101	0103	0201
0104	0203	0304	0302

Execution of the statements

```
100 DIM E$(3,4)1
110 MAT MOVE D$( ), S$( ) TO E$( )
```

produces an array E$( ) with the values shown below:

E$( ) =

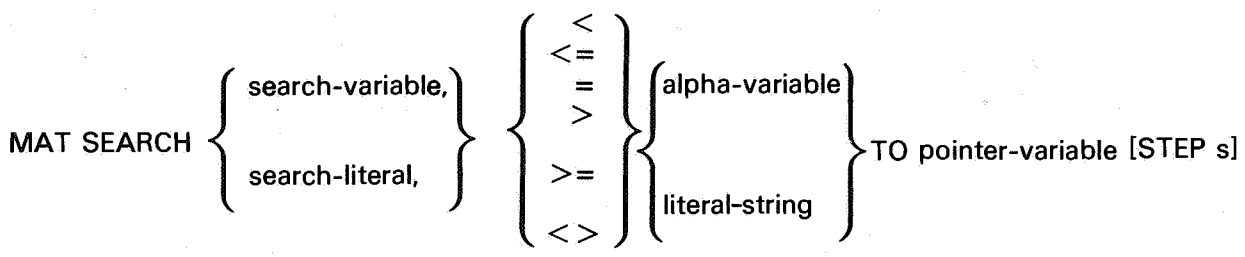
A	A	A	A
B	B	C	C
D	E	E	F

Examples of Valid Syntax:

```
10 MAT MOVE A$( ) TO B$(1,1)
20 MAT MOVE A$( )(5,3), L$( ) TO B$( )
30 MAT MOVE A( ), L$(3,1), X TO B(Y)
40 MAT MOVE A( ), L$( ) TO B( )
50 MAT MOVE A$( ), L$(X) TO B$(E)(3,4)
```

MAT SEARCH

General Form:



s = a numeric expression (1 <= s < 65536).

Purpose:

The MAT SEARCH statement scans the search-variable or search-literal for substrings that satisfy the given relationship to the value of a specified alpha-variable or literal string. The positions of substrings which satisfy the given relationship are placed into the pointer-variable in the order in which they are found. Each position is stored as a two-byte binary value (or "pointer") specifying the position of the first character of the substring relative to the beginning of the search-variable (or portion of the variable if a STRing function is used) or search-literal. If there is any space remaining in the pointer-variable after the search is complete, a binary "0000" is inserted as the next two bytes following the last pointer; the remainder of the pointer-variable is unchanged. If a pointer-variable is not large enough to accept all the pointers to substrings satisfying the given relationship, the search ends when the pointer-variable is full.

MAT SEARCH performs a global search in which the entire search-variable or search-literal, including trailing spaces, is treated as a single contiguous character string. Element boundaries are ignored if the search-variable is an alpha-array-variable. The current length of the alpha-variable or literal string in the specified relation determines the length of the substrings checked in the search-variable or search-literal.

Note that trailing spaces are not usually considered to be part of alpha values. Hence, if trailing spaces or values that end in HEX(20) are to be checked for, the STRing function must be used to specify the exact number of characters to be checked. For example,

```
100 MAT SEARCH A$() = STR(Z$, 1, 5) TO B$()
```

specifies that substrings in A$() which are five characters in length and equal to the first five characters of Z$ are to be sought.

The ability to specify a search-literal rather than a search-variable is useful when a fixed table must be searched for variable data. In this case, the use of a literal string saves assigning the data to another array and produces code that is more self-explanatory than if an array had been used. For example:

```
10 DIM A$3, C$2
50 LINPUT "DEVICE ADDRESS", A$
   :MAT SEARCH "310B10320B20350B50360B60", = STR(A$,,3)
    TO C$ STEP 3
   :ON 1 + VAL(C$,2)/3 SELECT #N/310; #N/B10; #N/320;
   #N/B20; #N/350; #N/B50; #N/360; #N/B60
   :ELSE GOTO 50
.
.
.
```

This example accepts a three-byte device address from the operator, verifies it against a list of valid addresses, and then selects the requested device. If the address is invalid, the program branches back to repeat the request.

Normally, MAT SEARCH first determines if the substring starting at the first character in the search-variable or search-literal satisfies the given relationship to the alpha-variable or literal string. If so, the position is stored in the pointer-variable, and the substring starting at the second character is checked, and so on. However, if a STEP parameter "s" is specified, substrings starting at the first position, the (1 + s) position, the (1 + 2s) position, and so on, are checked. The MAT SEARCH operation ends when the pointer-variable is full or when the number of characters in the search-variable remaining to be checked is less than the length of the alpha-variable or literal string in the relation.

Examples:

Assume that the array G$( ) is defined with the values shown below. All occurrences of the value "A" must be found in G$( ):

G$( ) =

	1	2	3	4
1	B	A	C	D
2	C	B	E	A
3	A	F	A	E

The following statements could be used to search G$( ) for all occurrences of "A":

```
10 DIM P$(2,6)2
100 P$( ) = ALL(FF)
110 MAT SEARCH G$( ), = "A" TO P$( )
```

In this case, G$( ) is the search-variable and P$( ) is the pointer-variable. Execution of line 110 causes P$( ) to receive pointers to all occurrences of "A" in G$( ). The resulting contents of P$( ) are as follows:

P$( ) =

	1	2	3	4	5	6
1	0002	0008	0009	000B	0000	FFFF
2	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF

The search-variable G$( ) is scanned row by row, and the location of each character "A" is specified with reference to the first character of G$( ). For example, element (2,4) in G$( ) contains an "A". In P$( ), the location of this character is expressed as HEX(0008), indicating that it is the eighth character in G$( ), counting from the first character (element 1,1). Element boundaries are ignored in this process. Note that the value HEX(0000) is inserted in the element following the last valid pointer stored in P$( ). The remaining bytes of P$( ) are filled with HEX(FFFF) because P$( ) was initialized to this value in line 100 prior to beginning the MAT SEARCH operation.

Examples of Valid Syntax:

```
10 DIM A$100, B$5, S$(10)3, W$(50)2, W1$50
20 MAT SEARCH S$( ), = "XX" TO W1$
30 MAT SEARCH A$, = HEX(0102) TO W1$
40 MAT SEARCH STR(S$( ),5,20), < B$ TO W1$ STEP 5
50 MAT SEARCH A$,<>STR(S$( ),18) TO STR(W$( ), 30, 20)
60 MAT SEARCH "LDASTAADC", = STR(Z$,1,3) TO B$( ) STEP N
70 MAT SEARCH HEX(0020413749), = B$ TO C$
```

NEXT

General Form:

NEXT counter-variable [,counter-variable]...

Where:

counter-variable = numeric-scalar-variable specified as counter in companion FOR. . .TO statement.

Purpose:

The NEXT statement must be the last statement in a loop initiated by a FOR. . .TO statement (see the discussion of the FOR. . .TO statement earlier in this section). The counter-variable(s) specified in the NEXT statement must correspond to the counter-variable(s) defined in one or more preceding FOR. . .TO statements.

The NEXT statement adds the STEP increment to the current value of the counter-variable. If no STEP increment was explicitly specified in the FOR. . .TO statement, an increment of 1 is assumed. The result is then compared to the exit value specified in the corresponding FOR. . .TO statement. If the STEP value is positive, the counter value is tested to determine whether it exceeds the exit value; if the STEP value is negative, the counter value is tested to determine whether it is less than the exit value. If the tested condition is not met, the loop is reexecuted. If the tested condition is met, the loop is terminated, and execution continues with the statement following NEXT or with the first statement of the next outer loop.

Nested loops may be ended with a single NEXT statement. For example:

```
NEXT I,J,K
```

is equivalent to

```
NEXT I: NEXT J: NEXT K
```

In Immediate Mode, a NEXT statement and its corresponding FOR. . .TO statement must both appear in the same statement line.

Examples of Valid Syntax:

```
10 FOR M=2 TO N-1 STEP 30: J(M)=X(M)↑2
20 NEXT M
30 FOR X=8 TO 16 STEP 4
40 FOR A=2 TO 6 STEP 2
50 LET B(A,X) = B(X,A)
60 NEXT A
70 NEXT X
```

Nested Loops

ON/GOTO
ON/GOSUB

General Form:

ON { expression / alpha-variable } { GOSUB / GOTO } [,[line-number,]] . . . line-number : ELSE statement

Where:

expression = a numeric expression whose truncated integer value is the branch value.

alpha-variable = an alpha-variable whose first byte represents (in binary) the branch value.

line-no = a line-number to which a branch is to be made, depending upon the branch value.

Purpose:

The ON statement is a computed or conditional GOTO or GOSUB statement (see the discussion of the GOTO and GOSUB statements earlier in this section). 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, the statement

```
ON I GOSUB 100,200,300
```

would cause a transfer to line 200 in the program. If the truncated value of I is either less than 1, greater than the number of line-numbers specified, or points to a null line-number, no branch is taken. Null line-numbers are indicated by consecutive commas in the list of line-numbers and imply that no branch is to be made for the associated branch value. In each of these cases, the ELSE clause is executed, if specified; otherwise, the next sequential statement is executed. For example:

```
100 ON I GOTO 10,20,,40: ELSE GOSUB 1000
```

This statement will branch to line 10, 20, or 40 if I = 1, 2, or 4, respectively. For all other values of I, the ELSE clause will cause a branch to the subroutine beginning at line 1000.

Advanced Use of Alpha-Variables in the ON Statement

The branch value may be specified by an alpha-variable as well as by a numeric expression. If an alpha-variable follows the word "ON," the binary value of the first byte of the value of the alpha-variable is used as the branch value. For example, if A$=HEX(02), execution of the statement

```
50 ON A$ GOTO 100, 200, 300
```

would cause a transfer to line 200.

Examples of Valid Syntax:

```
10 ON I GOSUB 10, 15, 100, 900 :ELSE GOSUB'20
20 ON 3*J-1 GOSUB 100, 200, 300, 400
30 ON X GOTO,,200,300
```

PRINT

General Form:

PRINT [print-element] [{ , / ; } [print-element]] ...

Where:

print-element = { numeric-expression / alpha-variable / literal-string / AT( ) / BOX ( ) / HEXOF( ) / TAB( ) }

Purpose:

The PRINT statement prints the values of the specified print-element(s) on a designated output device in a system-defined format. In Program Mode, output from a PRINT statement is sent to the output device currently selected for PRINT operations. In Immediate Mode, PRINT output is sent to the currently selected Console Output device. The use of the Console Output device for Immediate Mode PRINT output is a convenient debugging feature which enables an operator to halt program execution and then examine the results of Immediate Mode PRINT statements on the CRT even while programmed PRINT output is selected to a printer. (See Chapter 7, section 7.3 for a more detailed discussion of selecting PRINT and CO devices.) The PRINT statement may contain both numeric and alphanumeric print-elements as well as the special PRINT functions AT( ), HEXOF( ), and TAB( ). These functions are described under separate headings later in this chapter.

Alphanumeric Print-Elements

An alphanumeric print-element may consist of a literal string in quotes, a HEX literal, or an alpha-variable. A literal string is printed exactly as it appears within quotation marks, including trailing spaces. The quotation marks themselves are not printed. For example, the statement

```
:10 PRINT "ABCD"
```

produces the following output when the program is run:

```
:RUN
ABCD
```

A HEX literal can be used to specify special characters not found on the keyboard as well as system control codes. For example, the statement

```
:10 PRINT HEX(03);
```

clears the CRT screen when executed because HEX(03) is the code used to home the cursor and clear the screen. If an alpha-variable is specified as a print-element, the value of the variable is printed; however, trailing spaces in the variable are ignored and only the "current length" of the variable is used. Trailing spaces are printed only if they fall within the range defined by a STR( ) function. For example, the statements

```
:10 A$ = "ABC"
:20 PRINT A$; "DEF"
:30 PRINT STR(A$,1,5); "DEF"
```

produce the following output when the program is run:

```
:RUN
ABCDEF
ABC  DEF
```

Note that in the second case, two trailing spaces from A$ are printed since they are included within the five-character range defined by the STR( ) function.

Numeric Print-Elements

Numeric print-elements may be constants, numeric-variables, or numeric expressions. If an expression is specified, it is evaluated by PRINT and the result is printed. (The values of variables are never altered by a PRINT statement.) A numeric value is printed in either of two formats, depending upon its magnitude. In general, a value which can be expressed exactly in 13 or fewer digits is printed in the normal numeric format defined below, while a value requiring more than 13 digits is printed in scientific notation format. A numeric value is always printed with a leading sign and a trailing space. A leading minus sign is printed for negative values, and a leading space is printed for nonnegative values.

1. Normal Numeric Format

   A numeric value is printed in normal format (leading minus sign or blank, one to 13 digits plus decimal point, and trailing space) if it satisfies either of the following conditions:

   a) Its absolute value is in the following range:

   $$0.1 <= |\text{value}| < 10^{+13}$$

   b) Its absolute value is less than .1, but it can be expressed exactly in 13 or fewer digits.

   For example, the statements

   ```
   :10 A = .005: B = 9999999999999: C = -55
   :20 PRINT A
   :30 PRINT B
   :40 PRINT C
   ```

   produce the following output when RUN is keyed:

   ```
   :RUN
    .005
    9999999999999
   -55
   ```

Leading and trailing zeroes are ignored, and the decimal point is not printed if the value is an integer. Numeric values printed in normal numeric format occupy a minimum of three character positions in the print line (a leading minus sign or blank, one digit, and a trailing blank) and a maximum of 16 character positions (a leading minus sign or blank, a maximum of 13 digits, a decimal point, and a trailing blank).

2. Scientific Notation Format

A numeric value is printed in scientific notation format if its absolute value does not correspond to either of the conditions specified above for printing in normal numeric format.

A value printed in scientific notation format always occupies exactly 16 character positions in the print line. These character positions contain a leading minus sign or space, one integer digit, a decimal point, exactly eight decimal digits, the letter "E", the sign of the exponent, a two-digit exponent, and a trailing space. Trailing zeroes are printed in the numeric value, and a leading zero is printed in the exponent. The sign of the exponent is always explicit (i.e., the sign of a nonnegative exponent prints as a "+" rather than a space). For example, the statements

```
:10 A = -2.56E18: B=1E13
:20 PRINT A
:30 PRINT B
:40 PRINT 275634*112913534
:50 PRINT 5/75
```

produce the following output when RUN is keyed:

```
:RUN
-2.56000000E+18
 1.00000000E+13
 3.11228090E+13
 6.66666666E-02
```

NOTE:

A maximum of nine significant digits can be obtained when a value is printed in scientific notation using the PRINT statement. If the full 13 digits of precision are required, the value must be output by using the PRINTUSING statement (see the discussion of the PRINTUSING statement later in this section).

The two factors which determine the format of numeric values printed with the PRINT statement are the number of digits in the numeric value and the magnitude of these digits. Apart from ensuring that values fall within a given range, therefore, the programmer can not control the format of printed output generated with the PRINT statement. If greater format control is desired, the PRINTUSING statement should be used instead of the PRINT statement. The output format of values printed with PRINTUSING can be defined explicitly by the programmer.

PRINT-ELEMENT SEPARATORS

Multiple print-elements in a PRINT statement must be separated by one of the element separators: the comma (,) or the semicolon (;). Successive print-elements not separated by an element separator violate the syntax rules for the PRINT statement and will produce an error. In addition to its role as an element separator, the comma functions as a format control character (see the discussion below entitled "The Comma Used as an Element Separator and Format Control Character").

The Semicolon Used as an Element Separator

A semicolon may be used to separate successive print-elements. In general, the semicolon serves only to satisfy the syntax requirement for an element separator. It causes no additional spaces to be inserted between print-elements; the only spaces that appear between two successive print-elements separated by a semicolon are those output as part of the value itself. Numeric values always are printed with a leading minus sign or space and a trailing space. For example, the statements

```
:10 A = 1234: B = 5678
:20 PRINT "ABC";"DEF"
:30 PRINT A;B
:40 PRINT A;"ABC";B;"DEF"
```

produce the following output when RUN is keyed:

```
:RUN
ABCDEF
 1234  5678
 1234 ABC 5678 DEF
```

Note that the pair of alphanumeric values output by line 20 are printed without any intervening spaces, while the numeric values output by lines 30 and 40 are each accompanied by leading (since both are positive) and trailing spaces.

There is one special case in which the semicolon has a functional meaning in a PRINT statement. If a semicolon follows the last print-element in a PRINT statement, it suppresses the carriage return normally generated by the system at the end of the PRINT line. For example, the statements

```
:10 PRINT "ABC";
:20 PRINT "DEF"
```

produce the following output when RUN is keyed:

```
:RUN
ABCDEF
```

Here the carriage return which normally would be issued at the end of line 10 is suppressed by the trailing semicolon so that output from both lines 10 and 20 appears on the same PRINT line.

In general, a value which does not fit completely into the remaining space at the end of a PRINT line is not continued on the next line, but is moved in its entirety to the beginning of the following line. The exceptions to this rule are character strings defined in HEX literals and HEXOF functions. These character strings are split if they overlap from one line to the next.

The Comma Used as an Element Separator and Format Control Character

The comma serves a dual function as an element separator and a format control character in a PRINT statement. The appearance of a comma between two print-elements signals to the system that the values are to be printed in "zoned format." In zoned format, each output line is divided into a number of zones, and each zone is 16 characters in width. The total number of zones in an output line depends upon the selected line width of the output device. A 24x80 CRT, for example, is divided into five zones (0-15, 16-31, 32-47, 48-63, and 64-79), while a 16x64 CRT is divided into four zones.

A comma preceding a print-element signals that the value is to be printed starting at the beginning of the next zone. If the CRT cursor or print-element is not currently positioned at the beginning of a zone, spaces are automatically printed until the beginning of the next zone is reached. The print-element is printed starting at that point. If the value of a print-element exceeds the width of the last

zone in the output line, the system issues a carriage return and prints the value, starting at the beginning of the first zone on the next line. For example, the statement

```
:10 PRINT 12*2,"ABC"
```

produces the following output:

```
:RUN
 24             ABC
```

The PRINT statement normally issues a carriage return code when it terminates execution after the last print-element has been output. A comma can be used at the end of a PRINT line to suppress this carriage return. The trailing comma causes the system to print spaces to the start of the next zone and then output the first print-element of the next PRINT statement starting at that zone rather than at the beginning of the first zone on the next line. For example, the statements

```
:10 N=50: P=100
:20 PRINT "N=";N,
:30 PRINT "P=";P
```

produce the following output:

```
:RUN
N= 50                   P= 100
```

Here the trailing comma after "N" in line 20 suppresses the carriage return which would normally be issued at that point; consequently, the output from line 30 appears in the next zone on the same output line as the output from line 20.

For most output devices, the system maintains a column count and automatically issues a carriage return when the end of an output line is reached. This carriage return is independent of the carriage return issued by the PRINT statement and is not suppressed by a trailing comma. For example, the statements

```
:10 FOR I = 1 TO 8
:20 PRINT I↑2,
:30 NEXT I
```

produce the following output on a 24x80 CRT:

```
:RUN
1             4             9             16            25
36            49            64
                            254
```

A leading comma in a PRINT statement (e.g., PRINT, A$) causes the system to space over to the start of the next zone before outputting the first print-element. Multiple comma separators can be used to space print-elements more than one zone apart. Each comma causes the succeeding print-element to be shifted one zone to the right in the output line. Spaces are printed to position the CRT cursor or print-element to the start of the specified zone. The AT and TAB functions can be used in a PRINT statement to control more precisely the format of a PRINT line. (See the discussion of the PRINT AT and PRINT TAB functions found later in this section.)

A PRINT statement with no print-elements or element separators issues a carriage return and advances the CRT cursor or paper one line.

Reducing the Rate of PRINT Output on the CRT with a SELECT P Statement

The length of time during which PRINT output is displayed on the CRT can be extended by invoking a pause with a SELECT P statement. SELECT P permits the user to select a pause ranging from 1/6 second to 1 1/2 seconds, in increments of 1/6 second. Thus, the statement SELECT P1 invokes a

pause of 1/6 second, while the statement SELECT P6 invokes a pause of one second. The pause is invoked each time a carriage return is output. SELECT P or SELECT P0 deactivates the pause, as do RESET, CLEAR, and Master Initialization.

Use of Different Device-Types with a PRINT Statement

The system issues a carriage return code (HEX (0D)) to signal the termination of an output line. A column count is maintained internally, and a carriage return is issued automatically when the output line is filled. It is frequently convenient, however, to terminate an output line before it is completely filled. The system, therefore, provides the following means for issuing carriage returns:

- A carriage return is automatically issued when the last print-element in a PRINT statement has been output unless the PRINT statement ends with a trailing comma or semicolon.
- A carriage return is issued whenever a HEX (0D) character is encountered in a character string or HEX literal.

Once an output line is terminated with a carriage return code, the output device normally should perform a line feed to advance to the next output line. Some output devices (such as the printers and output writers) automatically perform a line feed whenever they receive a carriage return signalling line termination. Others (such as the CRT) must be explicitly instructed to perform a line feed after each carriage return with a line feed code (HEX (0A)).

The mechanism used to tell the system whether or not it should issue a line feed following each carriage return for a particular device is the device-type in the output device-address. The device-type is the first hexadecimal digit in a three-digit device-address. In the device-address /005, for example, the device-type is 0; in device-address /215, the device-type is 2. In general, there are three different device-types used in output device-addresses: 0, 2, and 7. Each device-type has a particular significance:

Type 0 — Instructs the system to automatically output a line feed code (HEX (0A)) following each carriage return. (Used with the CRT.)

Type 2 — Instructs the system to output a null character (HEX(00)) instead of a line feed code following each carriage return. (Used with printers and output writers.)

Type 7 — Instructs the system to output no extra characters following each carriage return. (For most applications, device-type 7 is equivalent to device-type 2.)

The importance of the device-type becomes apparent when a device-type and an output device are mismatched. For example, the use of device-type 2 with the CRT (as in SELECT PRINT /205) can create printing problems. Since device-type 2 does not automatically output a line feed and the CRT does not generate its own, the cursor is not advanced to the next CRT line following a carriage return. A subsequent PRINT statement, therefore, prints its output on the same CRT line, overwriting the previous output line. Similarly, the use of device-type 0 with a printer (as in SELECT PRINT /015) produces double-spaced output since device-type 0 automatically sends a line feed code after each carriage return in addition to the line feed supplied by the printer itself.

The special condition in which carriage return codes (HEX(0D)) are specified in a HEX literal when device-type 0 is used must also be explained. Normally, device-type 0 automatically adds a line feed code following any carriage return code. A carriage return code within a HEX literal constitutes the sole exception to this rule. No line feed or HEX(00) character is added by the system following a carriage return code when it is specified as part of the value of a HEX literal, even if device-type 0 has been selected.

Line Width

The maximum width of an output line can be defined for each output device in a SELECT statement. The line width is enclosed in parentheses immediately following the output device-address. For example, the statement

```
10 SELECT PRINT/215(132)
```

selects the line printer for PRINT operations and sets a maximum line width of 132 columns. As a PRINT line is output, the system maintains a column count. When the count exceeds 132, a carriage return is automatically issued, terminating the line. A line can be terminated, however, before the entire 132 characters have been output. This occurs when the last print-element has been output by a PRINT statement with no trailing punctuation or a carriage return code (HEX(0D)) is encountered in a HEX character string.

Note that the line width is selected independently for each class of output operations. Thus, the same output device may receive a different line width for different operations. For example:

```
50 SELECT PRINT/215(132), LIST/215(80)
```

In this case, the line printer uses a line width of 132 columns for PRINT output and a line width of 80 columns for LIST output.

A line width of zero has a special significance to the system. If the line width = 0, the system does *not* maintain a column count and does *not* issue an automatic carriage return when the line width is exceeded. In this special case, therefore, a carriage return is issued *only* when the last print-element in a PRINT statement is output or a carriage return code (HEX(0D)) is encountered in a character string. (Occurrence of an automatic line feed when a device's carriage width is exceeded is device-dependent. Most printers do issue an automatic line feed; however, CRT's wrap around on the same line.) The zero line width is used to manipulate the CRT cursor since it permits the programmer to move the cursor without updating the column count and to explicitly control when and where a carriage return will be issued. (Note that because no column count is maintained, the TAB and AT functions cannot be used with a line width of zero.)

PRINT AT Function

General Form:

AT (expression-1, expression-2 [,[expression-3]])

Where:

expression-1 = CRT row
expression-2 = CRT column
expression-3 = number of characters to be erased

Purpose:

The AT function is used within a PRINT statement to position the CRT cursor to any location on the screen and to optionally erase all or part of the screen from that point onward, including the character located at the cursor position. The cursor location is specified by giving its row (expression-1) and column (expression-2) positions. Rows are numbered 0 — m and columns are numbered 0 — n, where "m" is one less than the number of lines on the CRT display, and "n" is one less than the number of character positions in a line. For example, the statement

```
PRINT AT (3,5);A$;AT(3,30);X
```

displays the value of A$ in row 3 beginning at column 5 and the value of X in row 3 beginning at column 30. (Note that the AT function cannot be used if the line width is selected to 0.)

Expression-3 specifies the number of characters to be erased on the CRT screen. The CRT cursor is positioned according to the first two expressions, and then the specified number of characters are erased, beginning with the character located at the cursor position. If the number of characters to be erased is greater than the number of characters on the current CRT line, characters on the next line are also erased. For example, the statement

```
PRINT AT (3,32,2);
```

positions the cursor in row 3 at column 32, and the characters in columns 32 and 33 are erased.

If a comma follows expression-2 but expression-3 is omitted, the CRT screen is erased from the specified position to the end of the screen. For example, the statement

```
PRINT AT(8,0,)
```

erases lines 8 — 23 of a 24x80 CRT screen (assuming SELECT LINE = 24). The number of lines on the CRT is determined by the current value of SELECT LINE; the default value for SELECT LINE is 24 for a 24 x 80 CRT and 16 for a 16 x 64 CRT.

Examples of Valid Syntax:

```
10 PRINT AT(5,10);X
20 PRINT AT(X,Y);A$;AT(X+1,Z);B$
30 PRINT AT(3,20,1);X,Y
40 PRINT AT(8,0,);
```

PRINT BOX Function

General Form:

BOX (height, width)

Where:

height = a numeric expression specifying the height in lines of the box.
width = a numeric expression specifying the width in character positions of the box.

Purpose:

The BOX function, which is legal only within a PRINT statement, is used to draw or erase a box of specified dimensions on a CRT with box graphics capability. The current cursor position is used by the BOX function as the upper-left corner of the box; the BOX function does not move the cursor. If the height of the box exceeds the number of lines available on the screen (after the current cursor position), the screen will scroll the needed number of lines. If the width of the box exceeds the available screen width, the box will "wrap around" on the same line as it began.

The height expression specifies the number of CRT lines the box will occupy; the width expression indicates the number of character positions to be occupied. For example, the statement

```
PRINT BOX(2,5)
```

will draw a box two lines in height and five character positions in width, beginning at the current cursor position. The signs of the expressions determine whether the box will be drawn or erased. If the signs of both values are positive, a box is drawn; if both are negative a box is erased. For example, the statement

```
PRINT BOX(-2,-5)
```

will cause a box like the one drawn in the previous example to be erased. If the two expressions have different signs, an error P34 occurs. Both expressions are truncated.

If one expression is zero, the BOX function will draw or erase a line of the dimensions specified. For example, a statement of the form

```
PRINT BOX(0,5)
```

will draw a horizontal line five characters long, and the statement

```
PRINT BOX(-2,0)
```

will erase a vertical line two lines high. Using a zero expression and a negative expression in the same function does not violate the corresponding sign rule.

Examples of Valid Syntax

```
10 PRINT BOX(4,7)
20 PRINT BOX(-4,-9)
30 PRINT BOX(0,3)
40 PRINT BOX(-12,0)
```

PRINT HEXOF Function

General Form:

HEXOF (literal-string | alpha-variable)

Purpose:

The HEXOF function, which is legal only within a PRINT statement, is used to print the value of an alpha-variable or literal string in hexadecimal notation. Trailing spaces, HEX(20), in the value of the alpha-variable are printed. If the printed value of the alpha-variable exceeds one line on the CRT display or printer, it is automatically continued on the next line or lines. The line width for PRINT operations can be set to the desired width by using a SELECT PRINT statement; this feature can be used to format more precisely the output from lengthy print-elements.

Example:

```
:10 A$ = "ABC"
:20 PRINT "HEX VALUE OF A$ = "; HEXOF(A$)
:RUN
HEX VALUE OF A$ = 414243202020202020202020202020
```

Examples of Valid Syntax:

```
10 PRINT HEXOF(STR(A$,X,Y));
20 PRINT HEXOF(A$( ))
```

PRINT TAB Function

General Form:

TAB (expression)

Where:

0 < = value of expression < 256

Purpose:

The TAB function, which is legal only within a PRINT statement, is used to produce tabulated formatting of printed output. TAB positions the CRT cursor or print-element to a specified column in the output line by inserting spaces in the line up to the specified column. On the CRT, all intervening values displayed on the line are erased. (Note that the AT function, which also can be used to position the cursor in the CRT display, does not output spaces when moving the cursor and, therefore, does not erase intervening output.) For example, the statement PRINT TAB(17) causes the system to print spaces in the line up to column 17.

Columns are numbered 0 — n, where "n" is one less than the line width of the screen or printer. The value of the expression in the TAB function is computed, and the integer portion specifies the TAB column. The system then inserts spaces in the output line up to that column position. If the specified column has already been passed, the TAB is ignored. If the TAB value is greater than the defined width of the output line, the CRT cursor or print-element is positioned at the beginning of the next line. TAB values greater than 255 are illegal.

Example:

```
:10 FOR I = 0 TO 5
:20 PRINT TAB (I); "X";I
:30 NEXT I
:RUN
X 0
 X 1
  X 2
   X 3
    X 4
     X 5
```

PRINTUSING

General Form:

PRINTUSING image-spec [{ , / ; } print-element]. . .[;]

Where:

image-spec = { line-number of Image statement / alpha-variable containing image / literal-string specifying image }

print-element = { expression / alpha-variable / literal-string }

Purpose:

The PRINTUSING statement is used to create formatted print output of numeric and alphanumeric data in a user-defined format. The formatted print line created by the PRINTUSING statement is printed on the output device currently selected for PRINT operations in Program Mode or on the device currently selected for CO operations in Immediate Mode (see the discussion of the SELECT statement in Chapter 7, section 7.3).

The format of the output print line is defined by the programmer in a PRINTUSING "image." The image can be specified in one of three ways:

1. As a literal string specified in the PRINTUSING statement, e.g.,

```
:100 PRINTUSING "##.##", N
```

2. As the value of an alpha-variable specified in the PRINTUSING statement, e.g.,

```
:100 A$ = "##.##"
:110 PRINTUSING A$, N
```

3. As a separate Image statement whose line-number is specified in the PRINTUSING statement, e.g.,

```
:100 %##.##
:110 PRINTUSING 100, N
```

The third form of the PRINTUSING statement may not be used in Immediate Mode. (See the definition of the General Form of the Image (%) statement found earlier in this section.)

If the image is specified as a separate Image statement or as the value of an alpha-variable, the same image can be used by several PRINTUSING statements.

A PRINTUSING image consists of any combination of character strings and format-specifications. Character strings in the image represent constant data such as headings and labels; they are transferred to the output print line exactly as they appear in the image. Format-specifications supply the output format for print-elements in the PRINTUSING statement. The value of each numeric print-element in the PRINTUSING statement is edited into the output print line according to the format defined in a corresponding format-specification in the image.

A format-specification consists of at least one "#" character, called a "digit-selector character," and optionally one or more of the following special characters: "+", "−", ".", ",", "$", "↑↑↑↑", "−−", and "++". Digit-selector characters in a format-specification are replaced in the print line by

digits from the corresponding numeric print-element in the PRINTUSING statement. Special characters are edited into the print line according to the rules enumerated below in the subsection "Numeric Print-Elements." For example, the statements

```
:90 N = 55.25
:100 PRINTUSING "$##.##", N
```

produce the following print line:

```
$55.25
```

The format-specification has the following general form:

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

A format-specification may not contain embedded spaces or other alpha characters except the designated special characters. It must contain at least one digit-selector ("#") character.

Note that although the image provides character strings and format-specifications for the print line, it is not altered or destroyed by the execution of a PRINTUSING statement. The same image can, therefore, be used as often as desired.

Print-elements edited into the print line may be either numeric or alphanumeric.

Numeric Print-Elements

Three types of formats are available for numeric print-elements:

Integer Format	—	###
Fixed-Point Format	—	##.##
Exponential Format	—	##.#↑↑↑↑

The following rules govern the editing of numeric print-elements into an output print line:

1. If an Integer Format is specified, the integer portion of the value is edited into the print line, and the decimal portion, if any, is truncated. If the value to be printed is shorter than the integer format (i.e., the format contains more digit-selector characters than the value has digits), the value is padded with leading spaces. If the value is too large for the format (i.e., the value has more digits than the format has digit-selectors), the format-specification is edited into the print line in place of the value.

2. If a Fixed-Point Format is specified, both the integer and the decimal portions of the value are edited into the print line. The decimal portion is truncated or extended with trailing zeroes to fit the format. The integer portion is extended with leading spaces if it is smaller than the format; if the integer portion is too large for the format, the format-specification is edited into the print line in place of the value.

3. If an Exponential Format is specified (signified with four up-arrows, ↑↑↑↑), the value is edited as specified by the format. The most significant digit replaces the leftmost digit-selector, leading spaces are not used, and the exponent is edited into the format following the value. The exponent is printed in the form "E±dd," where "dd" are the digits of the exponent. If the value is too large for the format, the format-specification itself is edited into the print line in place of the value.

4. If the format begins with a plus sign (+) and the value of the numeric print-element N lies outside the range $-1 < N < 1$, the sign of the value (+ or −) is edited into the print line

immediately preceding the first significant digit unless a dollar sign ($) is also included in the format-specification. If a dollar sign is also specified in the format-specification, the sign of the numeric print-element is edited into the print line immediately preceding the dollar sign. If the format begins with a plus sign and the value of the numeric print-element N lies within the range $-1 < N < 1$, the sign of the value is edited into the print line immediately preceding the leading zero which appears to the left of the decimal point. When a dollar sign also appears in the format-specification for numeric print-elements whose values fall within the range $-1 < N < 1$, the sign of the value is always edited into the print line immediately preceding the dollar sign.

5. If the format begins with a minus sign (−) and the value of the numeric print-element N is less than or equal to (−1), a minus sign is edited into the print line immediately preceding the first significant digit. If the value of the numeric print-element N lies within the range $-1 < N < 0$, a minus sign is edited into the print line immediately preceding the leading zero which appears to the left of the decimal point. If a dollar sign ($) is also included in the format-specification, however, the minus sign is edited into the print line immediately preceding the dollar sign for all negative values. For positive values, a leading space is edited into the print line if the format-specification begins with a minus sign.

6. If no sign is specified in the format, no sign or space is edited into the print line, and the absolute value of the numeric print-element is output.

7. If the format contains a dollar sign ($), the dollar sign is edited into the print line preceding the first significant digit. If both a dollar sign and a leading plus or minus sign (+ or −) are specified, the dollar sign is edited into the print line between the leading sign (or space) and the first significant digit.

8. If the format contains either a comma (,) or a decimal point (.), the specified character is edited into the print line in the corresponding location.

9. If the format ends with a trailing plus sign (+), the true sign of the value (+ or −) is edited into the print line following the last digit.

10. If the format ends with a trailing minus (−) sign, either a minus sign (for negative values) or a space (for positive values) is edited into the print line following the last digit.

11. If the format ends with a pair of trailing plus signs (++), two space characters are edited into the print line at the end of nonnegative values, and the characters "CR" are edited into the line at the end of negative values.

12. If the format ends with a pair of trailing minus signs (−−), the characters "DB" are edited into the print line at the end of negative values, and two spaces are edited into the print line at the end of nonnegative values.

NOTE:

A format-specification can have only one true sign. If both leading and trailing signs are specified, the leading sign is interpreted as the true sign, while the trailing sign (or signs) is generally regarded as an alpha character and edited into the print line exactly as it appears in the image. An exception to this rule may occur when a second format-specification follows the first with no intervening spaces. In that case, the trailing sign (or second sign in a "++" or "−−" pair) in the first format-specification is interpreted as the true sign of the second format-specification if it immediately precedes the first digit-selector or dollar sign in the second format-specification.

Alphanumeric Print-Elements

The value of an alphanumeric print-element is edited into the print line by replacing each character in the format-specification with a character from the alpha value. Generally, only digit-selector characters (#) are used in the format-specification for alpha values. However, no distinction is made between special characters and digit-selector characters in the format. Each format character is replaced by a character from the alpha print-element. The alpha character string is left-justified in the format field and either truncated or extended with trailing spaces to fit the format. For example:

```
:100 A$ = "+##.##"
:110 PRINTUSING A$, "ABCDEFGHI"
:120 PRINTUSING A$, "ABC"
:RUN
ABCDEF
ABC
```

Multiple Format-Specifications

Multiple format-specifications can be included within a single image. A format-specification is terminated by a space character or by any alphanumeric character other than a digit-selector character (#) or one of the designated special characters. Thus, multiple format-specifications can be established in which spaces or intervening character strings serve as labels.

Values from the PRINTUSING print-element list are paired sequentially with format-specifications in the image and are edited into the corresponding positions in the print line. Alphanumeric character strings in the image are edited into corresponding positions in the print line exactly as they appear in the image.

After a formatted print-element is output, the character string (if any) immediately following the corresponding format-specification in the image is output. If there are no more print-elements, the system then issues a carriage return code, which terminates the output line unless the last print-element is followed by a trailing semicolon. (See the discussion entitled "Suppression of the Carriage Return Code," which appears immediately below.)

If there are fewer print-elements than format-specifications, the print line is terminated when the character string (if any) following the last-used format-specification has been edited into the line. The remainder of the image, beginning with the first unused format-specification, is ignored. For example:

```
:100 A$ = "### ABC ### DEF ###"
:110 PRINTUSING A$, 123, 456
:RUN
123 ABC 456 DEF
```

If there are more print-elements specified in the PRINTUSING statement than format-specifications in the image, the PRINTUSING statement reuses the image in the following manner until all print-elements have been output. When the character string (if any) following the last format-specification in the image has been used, a carriage return code is issued, terminating the print line. If there are additional print-elements, a second print line is constructed by reusing the same image. This process continues until all print-elements have been output. For example:

```
:100 A$ = "### ABC ###"
:110 PRINTUSING A$, 123, 456, 789
:RUN
123 ABC 456
789 ABC
```

Suppression of the Carriage Return Code

The carriage return code normally issued to end a print line when all print-elements have been formatted or all format-specifications have been used can be suppressed with a semicolon. The semicolon is used instead of a comma to separate print-elements at the point where a carriage return would normally be issued. This feature is useful for repeatedly reusing a single image with a series of print-elements. For example:

```
:100 A$ ="### "
:110 PRINTUSING A$, 123; 456; 789
:RUN
123 456 789
```

Regardless of the number of format-specifications and print-elements used by a PRINTUSING statement, PRINTUSING normally generates a carriage return upon completing statement execution. This carriage return code can be suppressed by a trailing semicolon. When a trailing semicolon is used, a carriage return is issued only when the length of the print line exceeds the selected line width of the output device (CRT or printer).

Selecting a Pause and the Meaning of Device-Types

The rate at which PRINTUSING output is generated on the CRT or printer can be slowed by invoking a pause with the SELECT PAUSE statement. Refer to the discussion of the PRINT statement earlier in this section and to the discussion of the PAUSE select-parameter in Chapter 7, section 7.3 for a description of the pause.

The use of different device-types can also affect the format of PRINTUSING output. Refer to the discussion of the PRINT statement earlier in this section and to the discussion of device-types in Chapter 7, section 7.7 for a full explanation of the significance of device-type selection on printed or displayed output.

Examples:

1.

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

2.
```
:100 A$ = "COEFF = +.###↑↑↑↑ ERROR = -##↑↑↑↑"
:110 PRINTUSING A$, 2.13E-5, 2.3E-9
:RUN
COEFF = +.213E-04 ERROR =    23E-10
```

3.
```
:100 PRINTUSING 200
:200 %  PROFIT AND LOSS STATEMENT
:RUN
 PROFIT AND LOSS STATEMENT
```

4.
```
:100 PRINTUSING "+#.##", 317.23
:RUN
+#.##   (Value too large for format)
```

5.
```
:50 A$ = "J. SMITH"
:60 T = 70565.32
:100 PRINTUSING 200, A$, T
:200 %  SALESMAN ########      TOTAL SALES $##,###.##
:RUN
 SALESMAN J. SMITH      TOTAL SALES $70,565.32
```

6.
```
:100 A$ = "ACCT. NO.            AMOUNT"
:110 B$ = "      #####          ###,###.##"
:120 PRINTUSING A$
:130 PRINTUSING B$, M(1),N(1),M(2),N(2),M(3),N(3)
:RUN
ACCT. NO.            AMOUNT
      101             14,512.01
      105             45,002.56
      112              6,751.02
```

PRINTUSING TO

General Form:

PRINTUSING TO alpha-variable, image-spec [{ ' / ; } print-element]. . .[;]

Where:

image-spec = { line-number of Image statement / alpha-variable containing image / literal-string specifying image }

print-element = { expression / alpha-variable / literal-string }

Purpose:

The PRINTUSING TO statement is used to store formatted print output in an alpha-variable. The program subsequently can process the output when it is convenient. The PRINTUSING TO statement is identical to the PRINTUSING statement except that the formatted print line is stored in an alpha-variable rather than sent directly to an output device (see the discussion of the PRINTUSING statement found earlier in this section).

The following rules govern the storage of formatted output in an alpha-variable when the PRINTUSING TO statement is used:

1. The first two bytes of the alpha-variable are reserved for use as a binary count of the number of characters stored in the variable with the PRINTUSING TO statement. The count indicates to the PRINTUSING TO statement the position in the variable at which to begin storing a formatted print line.

2. If an alphanumeric array-designator is used as the receiving variable, array-element boundaries are ignored and information is contiguously packed.

3. The count is automatically updated by the PRINTUSING TO statement whenever a new print line is stored in the variable. The count must be initialized to binary zero by the application program before the PRINTUSING TO statement is first executed and whenever the alpha-variable is to be reused from the beginning. Successive PRINTUSING TO statements progressively fill the receiving variable in a packed fashion and update the count.

4. The formatted output line stored in the alpha-variable is identical to a print line generated on an output device by PRINTUSING with a selected device-type of 7 and a line width = 0 (see the PRINT statement for a discussion of device-types and line width). When the PRINTUSING TO statement is used, no extra character (line feed or null) is inserted following a carriage return, and the automatic carriage return normally issued when the print line exceeds the maximum line width is suppressed. Other device-types and line widths may be used with a PRINT statement when the line is ultimately printed.

5. If the formatted print line will not completely fit in the alpha-variable, it is truncated and the count is set to the maximum value.

Example:

```
:100 A$ = "VALUE = ##.##"
:110 B$ = ALL(00):REM INITIALIZE VARIABLE
:120 PRINTUSING TO B$, A$, 12.23
:130 PRINT "COUNT="; VAL(B$,2)
:140 PRINT STR(B$,3,VAL(B$,2))
:RUN
COUNT = 14
VALUE  = 12.23
```

READ

General Form:

READ variable [,variable]. . .

Purpose:

A READ statement causes the next available element in a DATA list, which contains the values listed in a DATA statement in the program, to be assigned sequentially to the corresponding variable in the READ list. This assignment process continues until all variables in the READ list have received values or until all elements in the DATA list have been used. The READ variable list can include both numeric- and alphanumeric-variables. Each variable in the READ list must reference the corresponding type of data, however, or an error will result. Numeric-variables must reference numeric data and alphanumeric-variables must reference alphanumeric data.

READ statements are used in conjunction with DATA statements. The first-executed READ statement searches the program for the first DATA statement and assigns values from the DATA value list sequentially to the variables in the READ variable list. If a READ statement contains more variables than there are values in a DATA statement, the system searches for a second DATA statement in line-number sequence. If there are no more DATA statements in the program, an error message is displayed and the program is terminated. If a READ statement contains fewer variables than there are values in a DATA statement value list, the next READ statement begins with the first unused value in the DATA statement. READ statements may not be used in Immediate Mode.

The RESTORE statement can be used to reset the DATA list pointer, thus allowing values in a DATA list to be reused (see the discussion of the RESTORE statement found later in this section).

NOTE:

DATA statements may be placed anywhere in a program. Multiple DATA statements must be specified in the order in which their values are to be used in the program.

Examples:

1.
```
:100 READ A,B,C
:200 DATA 4, 315, -3.98
```

2.
```
:100 READ A$, N, B1$ (3)
:200 DATA "ABCDE", 27, "XYZ"
```

3.
```
: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 except colons, which indicate the end of the statement.

Purpose:

The REM statement is used solely as a documenting tool by the programmer to insert comments or explanatory remarks into a program. Since the REM statement is a nonexecutable statement which is ignored when encountered during program execution, it may be inserted anywhere in a program.

Two special forms of the REM statement have significance only when a program listing is printed with the LISTD command. REM% causes the text string to be printed (highlighted or in expanded print) on a separate line; the printer then skips a line before resuming the listing. REM%↑ causes the system to issue a top-of-form, print (highlighted or in expanded print) the text string on a separate, and then skip a line before resuming the listing. When the REM%↑ form of the REM statement is used, the up-arrow (↑) must *immediately* follow the percent sign (%) without *any* intervening spaces. REM% and REM%↑ thus provide a convenient means of inserting headings and subheadings into a program listing. These special features of the REM statement are significant only when the program is listed with the LISTD command; otherwise, REM% and REM%↑ are treated as ordinary REMs.

Examples of Valid Syntax:

```
10 REM SUBROUTINE
20 REM FACTOR
30 REM THE NUMBER MUST BE LESS THAN 1
40 REM% FICA COMPUTATION SUBROUTINE
50 REM%↑ FILE OPEN ROUTINE
```

RESTORE [LINE]

General Form:

```
RESTORE [LINE line-number [,expression]]
        [expression                     ]
```

Where:
1 <= expression < 65536

Purpose:

The RESTORE statement allows the repetitive use of DATA statement values by READ statements. When the RESTORE statement is encountered in a program, the system resets the DATA pointer to the specified DATA value. A subsequent READ statement will read DATA values beginning with the specified value.

If the RESTORE statement is followed by an expression, the system resets the DATA pointer to the "n"th DATA value in the program, where "n" is the integer portion of the expression. If the RESTORE LINE form is used and the optional expression is specified, the system resets the DATA pointer to the "n"th DATA value on the specified program line, where "n" is the integer portion of the expression following the line-number. If the optional expression in the RESTORE LINE statement is omitted, the system resets the DATA pointer to the first DATA value in the first DATA statement on the specified line. If there is no DATA statement on the specified line, the system searches for the first line containing a DATA statement after the specified line and then sets the DATA pointer accordingly. If the RESTORE statement is followed by neither a line-number nor an expression, the system resets the DATA pointer to the first DATA value in the first DATA statement in the program.

On the MVP, the RESTORE statement sets the DATA pointer to the specified DATA value in the partition whose program text is currently being executed. The text currently being executed may be assigned to a global partition (see Chapter 16, section 16.7).

NOTE:

RESTORE 0 is equivalent to RESTORE 1.

Examples of Valid Syntax:

```
10 RESTORE
20 RESTORE 5
30 RESTORE (X-Y)/2
40 RESTORE LINE 100
50 RESTORE LINE 100, 3
```

RETURN

General Form:
RETURN

Purpose:

The RETURN statement is used at the end of a subroutine to return program execution to the statement immediately following the last-executed GOSUB or GOSUB' statement. The RETURN statement also automatically clears the subroutine return parameters from the internal stacks along with the loop parameters of any FOR. . .TO/NEXT loops contained within the subroutine.

If entry was made to a marked subroutine via a Special Function Key on the keyboard, the RETURN statement terminates program execution and returns control either to the keyboard or an interrupted INPUT statement. The INPUT statement is then reexecuted from the beginning. If the Special Function Key was depressed in response to a LINPUT statement, control is returned to the statement immediately following LINPUT.

Repetitive entries to subroutines without executing a RETURN statement cause return information to accumulate in the system stacks in memory, eventually resulting in a Stack Overflow error (ERR A04). Subroutine return information can be cleared from the stacks without returning to the main program by executing a RETURN CLEAR statement (see the discussion of the RETURN CLEAR statement later in this section).

Examples:

1. RETURN Used To Recover From an Unmarked Subroutine

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

2. RETURN Used To Recover From a Marked Subroutine

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

Advanced Use of the RETURN Statement in a FOR/NEXT Loop

The RETURN statement can be used to terminate incomplete FOR/NEXT loops that were initiated after the transfer to the subroutine occurred. For example:

```
10 GOSUB 100
.
.
.
100 REM START OF SUBROUTINE
110 FOR I = 1 to 10
.
.
.
```

```
150 IF A(I)=0 THEN RETURN
.
.
.
190 NEXT I
```

If the RETURN statement is executed before the FOR/NEXT loop has terminated, the loop information is removed from the internal stacks along with the subroutine parameters; executing a NEXT I statement after the RETURN statement has been executed will cause an error.

RETURN CLEAR

General Form:

RETURN CLEAR [ALL]

Purpose:

The RETURN CLEAR statement is a dummy RETURN statement which is used to clear subroutine return address and FOR/NEXT loop information from the internal stacks without returning to either the keyboard or the main program. Execution of a RETURN CLEAR statement causes subroutine return address information from the last-executed subroutine call to be removed from the internal stacks. However, the program does not branch back to the statement immediately following the last-executed GOSUB or GOSUB' statement in the main program; instead, the normal sequence of execution continues with the statement following the RETURN CLEAR statement. If the ALL parameter is specified in a RETURN CLEAR statement, all subroutine return address and FOR/NEXT loop information is cleared from the internal stacks.

The RETURN CLEAR statement is used when a program exits from a subroutine without returning to either the keyboard or the main program. This statement is particularly advantageous when using Special Function Keys to start program execution from the keyboard at a desired point. Program execution can be initiated via a Special Function Key either in Console Input Mode or in response to an INPUT or LINPUT statement. When a keyboard Special Function Key is depressed to initiate program execution, a subroutine branch is made to the appropriate DEFFN' statement, and program execution then begins. A subsequently executed RETURN statement would either cause the system to return to the Console Input Mode, repeat the INPUT request, or return to the statement following the LINPUT statement. In some cases, however, the user may wish to start and continue a program with a Special Function Key without returning to the keyboard or the main program. The RETURN CLEAR statement can then be used to clear the subroutine parameters from the internal stacks without returning from the subroutine. If no return is made from a subroutine and the RETURN CLEAR statement is not used, the subroutine return information remains in the internal stacks. Repeated subroutine calls without returning cause this information to accumulate in the stacks until the capacity of one or both stacks is exceeded and a Stack Overflow error is signalled. This problem can be prevented by the use of the RETURN CLEAR statement.

Examples:

```
10 DEFFN' 15: RETURN CLEAR
20 RETURN CLEAR
```

STOP

General Form:

STOP [literal] [#]

Purpose:

The STOP statement terminates program execution. A program may contain more than one STOP statement. When a STOP statement is encountered, the system prints STOP followed by the optional literal string if one is specified and the line-number of the STOP statement if a "#" follows STOP. To continue program execution at the statement immediately following the STOP statement, a CONTINUE command may be entered or the HALT/STEP Key may be used to step through program execution one statement at a time.

Examples of Valid Syntax:

```
10 STOP
20 STOP "MOUNT DATA DISK"
30 STOP #
40 STOP "ERROR" #
```

CHAPTER 12 DATA CONVERSION STATEMENTS

12.1 INTRODUCTION

The BASIC-2 language provides an extensive set of instructions designed specifically to simplify the conversion of data from one format to another. These instructions permit the user to interpret information in a foreign format or pack data into a more efficient format for storage or transmission. The statements included in this special data conversion instruction set are summarized below:

CONVERT	Converts a numeric value to an ASCII alphanumeric character string representation of the number and vice versa.
$FORMAT	Provides a means of mnemonically defining a format-specification for the $PACK and $UNPACK operations.
HEXPACK, HEXUNPACK	HEXPACK converts an ASCII character string of hexadecimal digits into the binary equivalent of those digits. HEXUNPACK reverses the operation.
PACK, UNPACK	PACK converts numeric values to Wang packed decimal format. UNPACK reverses the operation.
$PACK, $UNPACK	Perform packing and unpacking of data in a variety of user-specifiable formats.
ROTATE	Rotates the bits of a single character or string of characters from one to seven places to the left or right.
$TRAN	Utilizes a table-lookup technique to provide high-speed character conversion.

In addition to the statements covered in this chapter, other BASIC-2 instructions which may be useful in data conversion operations include the Boolean operators AND, OR, XOR, and BOOL (discussed in Chapter 5), the alphanumeric functions BIN and VAL (also covered in Chapter 5), the binary arithmetic operators ADD and SUB (discussed in Chapter 6), and the MAT MOVE statement (discussed in Chapter 14).

12.2 GENERAL FORMS OF THE DATA CONVERSION STATEMENTS

General forms of the data conversion statements described in section 12.1 are shown on the following pages, arranged alphabetically for ease of reference.

CONVERT

General Form:

1. CONVERT alpha-variable TO numeric-variable
2. CONVERT expression TO alpha-variable, (format)

Where:

format = { [+ / −] [$][#[,]]. . .[.][#]. . .[↑↑↑↑] [+ / − / ++ / −−] }
{ alpha-variable containing format }

NOTES:

1. At least one # is required in a format.
2. The maximum number of characters in a format is 254.
3. Leading and trailing signs may not both occur in the same format.
4. Spaces in the format are ignored.

Purpose:

The CONVERT statement is used either to convert a numeric value to an ASCII alphanumeric character string representative of the number or vice versa. Two forms of the statement are provided:

Form 1: Alpha-To-Numeric Conversion

Form 1 of the CONVERT statement converts the number represented by the ASCII characters in the specified alpha-variable to a numeric value. For example:

```
:10 A$="1234"
:20 CONVERT A$ TO X
:30 PRINT "X =";X
:RUN
X = 1234
```

ERR X75 is signalled if the ASCII character string in the specified alpha-variable is not a legitimate BASIC representation of a number (see Chapter 4). This error can be processed by following the CONVERT statement with an appropriate error clause (see the discussion of the ERROR statement in Chapter 9, section 9.3).

Part of an alpha-variable can be converted to a numeric value by using the STR function. For example:

```
:10 A$="ABC12.45DEF"
:20 CONVERT STR(A$,4,5) TO X
:30 PRINT "X =";X
:RUN
X = 12.45
```

Alpha-to-numeric conversion is particularly useful when numeric data is read from a peripheral device in a format not directly compatible with numeric input or when a code conversion is first necessary. Such a conversion is also useful when it is desirable to validate keyed-in numeric data under program control. In the latter case, numeric data can be received into an alpha-variable with a LINPUT statement and then tested for validity with the NUM or VER function before being converted to numeric format.

Examples of Alpha-to-Numeric Conversion:

```
:10 CONVERT A$ TO X

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

Form 2: Numeric-to-Alpha Conversion

Form 2 of the CONVERT statement converts the numeric value of the specified expression to an ASCII character string according to the image specified. The alphanumeric character string is then stored in the specified alpha-variable.

The image specifies precisely how the numeric value is to be converted. Each character in the image corresponds to a character in the resultant character string. The image is composed of digit-selector characters (#) to signify digits and optionally "+","−","++","−−",".",and "↑" characters to specify sign (leading or trailing), decimal point, and exponential 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 format starts or ends with a plus sign (+), the real sign of the value (+ or −) is edited into the character string at the specified position (beginning or end).

2. If the format starts or ends with a minus sign (−), a blank for positive values or a minus sign for negative values is edited into the character string at the specified position (beginning or end).

3. If the format ends with a "++" sign, two spaces for nonnegative values or the characters "CR" for negative values are edited into the end of the character string.

4. If the format ends with two minus signs (−−), two spaces for nonnegative values or the characters "DB" for negative values are edited into the end of the character string.

5. If no sign is specified in the format, no sign is edited into the character string, i.e., the absolute value of the specified expression is output.

6. If the format contains a dollar sign ($), comma (,), or decimal point (.), the specified character is edited into the character string at the corresponding position.

7. If the format is fixed point (i.e., no "↑" 's specified), the numeric value is edited into the character string as a fixed-point number. The fractional portion is automatically truncated or extended with trailing zeroes, and the integer portion is padded with leading zeroes to fit the format-specification. An error is signalled if the integer portion exceeds the format-specification.

8. If the format is exponential (i.e., four "↑" 's specified), the value is edited into the character string in exponential format. The value is scaled to fit the specified format (leading zeroes are not used to pad the integer portion). The exponent is edited into the character string in the form "E±dd".

> **NOTE:**
>
> Exponential format is indicated with exactly four "↑" 's (i.e., ↑↑↑↑); the specification of any other number of "↑" 's produces an error upon entry.

Examples of Numeric-to-Alpha Conversion:

Statements	**Results**
:10 X = 12.345	
:20 CONVERT X TO A$,(###)	A$ = "012"
:30 CONVERT X TO A$,(+##.##)	A$ = "+12.34"
:40 CONVERT X TO A$,(-#.#↑↑↑↑)	A$ = " 1.2E+01"
:50 B$ = "#,###+"	
:60 CONVERT 100*X TO A$,(B$)	A$ = "1,234+"
:70 B$ = "$##.##++"	
:80 CONVERT-X TO A$,(B$)	A$ = "$12.34CR"

Programming Examples:

Example 1: Asterisk Filling

```
:10 CONVERT 1.23 TO A$, (+####.##)
:20 PRINT "RESULT OF CONVERT: ";A$
:30 $TRAN (STR(A$,2,POS(STR(A$,2)< >"0")),"*0")R
:40 PRINT "    ASTERISK FILLED:  ";A$
:RUN
RESULT OF CONVERT: +0001.23
     ASTERISK FILLED: +***1.23
```

Example 2: Removing Leading Zeroes

```
:10 CONVERT 1.23 TO A$, (+#####.##)
:20 PRINT "                RESULT OF CONVERT: ";A$
:30 STR(A$,2)=STR(A$,POS(STR(A$,2)< >"0")+1)
:40 PRINT "WITH LEADING ZEROES REMOVED: ";A$
:RUN
            RESULT OF CONVERT: +0001.23
WITH LEADING ZEROES REMOVED: +1.23
```

NOTE:

In some cases, numeric data converted to alphanumeric format with Form 2 of the CONVERT statement may cause an error when converted back to numeric format with Form 1. Specifically, the following characters are invalid in numeric format:

1. dollar sign
2. trailing plus or minus sign
3. debit (DR) and credit (CR) signs
4. comma

$FORMAT

General Form:

$FORMAT alpha-variable = field-specification [,field-spec]...

Where: field-specification =

SKIPxxx	(skip field)
Fxxx	(ASCII free format)
Ixxx[.dd]	(ASCII integer format)
Dxxx[.dd]	(IBM display format)
Uxxx[.dd]	(IBM USASCII-8 format)
P+xxx[.dd]	(IBM packed decimal format)
Pxxx[.dd]	(unsigned packed decimal format)
Axxx	(alphanumeric format)

xxx = field width (0 < xxx < 256)
dd = implied decimal position (0 < = dd < 16)

Purpose:

The $FORMAT statement is a special form of the assignment statement which provides a mnemonic means of creating a format-specification for the field form of the $PACK and $UNPACK statements. The format-specification variable defined in a $FORMAT statement can then be used in a subsequent $PACK or $UNPACK operation.

Examples of Valid Syntax:

```
10 $FORMAT F$ = I3, P8.2, A10
20 $FORMAT F1$ = F13, SKIP5, A9
30 $PACK (F=F$) B$( ) FROM X,Y,A$
40 $UNPACK (F=F1$)B$( ) TO N,P$(1)
```

HEXPACK

General Form:

HEXPACK alpha-variable-1 FROM alpha-variable-2

Purpose:

The HEXPACK statement converts an ASCII character string of hexadecimal digits into the binary equivalent of those hex digits. Hexadecimal digits entered from the keyboard are always entered as ASCII characters. If a hexadecimal value is to be used in binary operations such as addition or subtraction, it must first be converted from ASCII code to its true binary equivalent. For example, the hex digit "A" has a binary value of 1010. However, this hex digit is represented by an ASCII character "A", which has a binary value of 01000001. The HEXPACK statement can be used to convert the binary value of the ASCII character "A" into the binary value of the hexadecimal digit "A" and to store this value in a receiving alpha-variable. For example:

```
:10 DIM A$1,B$1
:20 A$="A"
:30 HEXPACK B$ FROM A$
:40 PRINT"A$=   ";HEXOF(A$)
:50 PRINT"B$=   ";HEXOF(B$)
:RUN
A$=  41
B$=  0A
```

Alpha-variable-2 contains the ASCII character string of hexadecimal digits. Each pair of ASCII characters is converted to one byte of the corresponding binary value. Thus, alpha-variable-2 normally is twice as long as alpha-variable-1. Only certain ASCII characters constitute legal representations of hexadecimal digits. These include the characters 0-9 and A-F as well as the special characters ":", ";", "<", "=", ">", and "?". These characters are converted to binary values and are listed in Table 12-1 below.

Table 12-1.
Binary Values for HEXPACK Characters

ASCII Character	Binary Value
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A or :	1010
B or ;	1011
C or <	1100
D or =	1101
E or >	1110
F or ?	1111

If the value of alpha-variable-2 contains any characters other than the ASCII characters listed in Table 12-1, including embedded spaces, an error message is displayed and program execution is terminated. Trailing spaces within the alpha-variable are not regarded as part of its value, however, and do not generate an error.

Alpha-variable-1 receives the converted binary value. Since each pair of characters in the value of alpha-variable-2 is converted to a one-byte binary value in alpha-variable-1, alpha-variable-1 should have at least half as many bytes as alpha-variable-2. If alpha-variable-1 is too short to contain the entire converted binary value, an error is signalled and program execution halts. If alpha-variable-1 is longer than the converted binary value, the binary value is left-justified and the remaining bytes of alpha-variable-1 are not modified.

Example 1:

```
:10 DIM P$2, U$4
:20 LINPUT "VALUE TO BE PACKED:"U$
:30 HEXPACK P$ FROM U$
:40 PRINT HEXOF(P$)
:RUN
VALUE TO BE PACKED:12C9
12C9
```

NOTE:

Line 30 in the preceding example is equivalent to:

```
30 P$ = HEX(12C9)
```

The availability of the special characters ":" (HEX(3A)) through "?" (HEX(3F)) to represent hex digits A-F (1010-1111) enables the HEXPACK statement to recognize any ASCII code with a high-order "3" digit (HEX(30) through HEX(3F)) as a legitimate representation of a hexadecimal digit. This fact makes it easy to transform any code into an acceptable representation of a hex digit and hence to perform operations such as packing the low-order hex digits (low-order four bits) from a string of hexadecimal digits. The technique is illustrated in Example 2.

Example 2:

```
:10 DIM P$2,U$4
:20 V$ = HEX(01020C09)
:30 V$ = OR ALL(30)
:40 HEXPACK P$ FROM V$
:50 PRINT HEXOF(P$)
:RUN
12C9
```

Example 3:

```
:10 REM *** RIGHT-JUSTIFIED HEXPACK ***
:20 DIM P$8
:30 INPUT "UNPACKED HEX",U$:                    REM ENTER UNPACKED HEX
:40 U1$=ALL("0"): STR(U1$,17-LEN(U$))=U$: REM RIGHT-JUSTIFY
                                                UNPACKED HEX
```

```
:50 HEXPACK P$ FROM U1$:                    REM PACK HEX
:60 PRINT "HEXOF(P$) = ";HEXOF(P$)
:RUN
UNPACKED HEX? 123
HEXOF(P$) = 0000000000000123
```

Examples of Valid Syntax:

```
10 HEXPACK A$ FROM B$
20 HEXPACK STR(A$,1,3) FROM STR(B$,7)
30 HEXPACK A$() FROM B$()
```

HEXUNPACK

General Form:

HEXUNPACK alpha-variable-1 TO alpha-variable-2

Purpose:

The HEXUNPACK statement converts the binary value of alpha-variable-1 to a string of ASCII hexadecimal characters which represent that value. The resulting characters are stored in alpha-variable-2.

The HEXUNPACK statement functions similarly to the PRINT HEXOF statement, with the difference that HEXUNPACK stores the resulting character string in an alpha-variable rather than outputting it to the CRT or printer. The value of alpha-variable-1 is processed left to right in groups of four bits. Each group of four bits is converted to an ASCII character which is the hexadecimal representation of the four-bit binary value. For example, since the hexadecimal representation of binary 0001 is "1", this binary value is converted to an ASCII "1" character. Because each four bits (half-byte) of alpha-variable-1 is converted to a full byte (one character) in alpha-variable-2, alpha-variable-2 must be at least twice as long as alpha-variable-1. If alpha-variable-2 is too short to contain the entire character string which results from unpacking the binary value in alpha-variable-1, an error is signalled and program execution halts. If alpha-variable-2 is longer than the converted character string, the character string is left-justified within the variable and the remaining bytes are not modified.

Programming Example:

```
:10 DIM P$2, U$4
:20 P$ = HEX(12C9)
:30 HEXUNPACK P$ TO U$
:40 PRINT U$
:RUN
12C9
```

Examples of Valid Syntax:

```
10 HEXUNPACK A$ TO B$
20 HEXUNPACK STR(A$,5) TO STR(B$,1,4)
30 HEXUNPACK A$() TO B$()
```

PACK

General Form:

PACK (image) alpha-variable FROM { numeric-array / expression }

Where: image = [+ / −] [#]. . . [.] [#]. . . [↑↑↑↑] or alpha-variable containing image

length of image < 255

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 according to the format specified by the image and then stored sequentially in the specified alphanumeric-variable. Array-variables are filled sequentially row by row from the beginning of the first array-element until all numeric data has been stored. An entire numeric-array can be packed by specifying a numeric array designator (e.g., N( )). An error will result if the receiving alpha-variable is not large enough to store all the numeric values to be packed.

The image is composed of digit-selector 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 "#" character in the image.
2. If a sign ("+" or "−") is specified, it occupies the high-order half-byte. A single hex digit is used to represent both the sign of the number and the sign of the exponent for exponential images. This hex digit can assume the following values:
   - 0 if both number and exponent are positive.
   - 1 if number is negative and exponent is positive.
   - 8 if number is positive and exponent is negative.
   - 9 if both number and exponent are negative.
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 the discussion of the UNPACK statement later in this section), the decimal point position must be specified in the image.

5. The packed numeric value is left-justified in the alpha-variable, with the sign digit (if specified) occupying the high-order half-byte, followed by the number in packed decimal format. The exponent occupies the two low-order half-bytes (if specified). The packed value always requires a whole number of bytes, even if the image calls for other than a whole number. For example, the image "###" calls for 1 1/2 bytes, but 2 bytes are required. In such cases, the value of the unused half-byte (the low-order half-byte) is not altered by the PACK operation.

6. If the image is expressed in fixed-point format, the numeric value is edited as a fixed-point number, truncating or extending with zeros any fraction and inserting leading zeros for non-significant integer digits according to the image specification. An error results if the number of integer digits exceeds the format-specification.

7. If the image is expressed in exponential format, the numeric value is edited as an exponential number. The value is scaled as specified by the image (there are no leading zeros). The exponent occupies one byte and is stored as the two low-order hex digits in the packed value.

Examples of Storage Requirements:

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

Examples of Valid Syntax:

```
10 PACK (###) A$ FROM X
20 F$ = "+##.##"
30 PACK (F$) B$ FROM N2
40 PACK (####) STR(A$,4,2) FROM N(1)
50 C$(2) = "-#.###↑↑↑↑"
60 PACK (C$(2)) A$() FROM N()
70 PACK (##.##) A1$() FROM X,Y,N(),M()
```

$PACK

General Form:

$PACK[(F|D)=(alpha-variable|literal-string)] alpha-variable FROM (literal-string|expression|variable|array)[,(literal-string|expression|variable|array)]...

Purpose:

The $PACK statement is used to store data in a buffer in a specified format. The values of the variables and arrays following the word "FROM" are sequentially read, formatted, and placed in the buffer. Values are taken from arrays element-by-element and row-by-row. An error results if the buffer is too small to hold all the values specified. The buffer is the alpha-variable immediately preceding the word "FROM". For example:

```
$PACK B$() FROM X, A$, Y(), B$()
     buffer to          variables
  receive data      supplying data
```

Three forms of $PACK are available:

1. Delimiter Form (specified by "D=" parameter)

   The delimiter form of $PACK separates the data values with a specified delimiter character when the values are stored in the buffer.

2. Field Form (specified by "F=" parameter)

   The field form of $PACK stores data values in the buffer in fields of specified lengths (i.e., each data value occupies a specified number of bytes).

3. Internal Form (assumed if neither the "D=" nor the "F=" parameter is specified)

   The internal form of $PACK stores data in the standard Wang 2200 disk record format.

Examples of Valid Syntax:

```
$PACK A$ FROM X
$PACK STR(B$,3,8) FROM X, A$
$PACK (F = X$) B$() FROM X()
$PACK (D = D$) B1$() FROM B$(), X, Y, A$1
$PACK (F = F$()) C$ FROM A$, STR(B$,3,4)
$PACK (D = STR(D$, 4)) B$() FROM X, Y, Z(1)
```

The Delimiter Form of the $PACK Statement

The delimiter form of $PACK separates data values with a specified delimiter character when the values are stored in the buffer. The values of the variables and arrays in the variable list are sequentially read and placed into the buffer. The packing terminates when all the specified values have been stored.

The first two bytes of the alpha-variable or literal string following "D=" are the delimiter specification. For example:

```
$PACK (D = A$) B$( ) FROM X( )
          ↑
      delimiter
    specification
      variable

$PACK (D = HEX(002C)) A$( ) FROM N( )
          ⌣
    delimiter specification
      expressed as hex
          literal
```

The first byte of the delimiter specification must be either hex 00, hex 01, hex 02, or hex 03; however, this byte is ignored by $PACK and is used only when unpacking data with the $UNPACK statement. The second byte is the delimiter character, which separates individual values in the buffer.

Buffer format:

data	D	data	D	data ...	D	data

where: D = delimiter character

Data format:

1. Alphanumeric Values

C	C	...	C

where: C is one character of the value to be packed.

If the value to be packed is specified as the value of an alpha-variable, trailing spaces are considered part of the value. Thus, the length of the character string stored in the buffer is equal to the defined length of the alphanumeric-variable from which the string was extracted. Array-elements are stored as separate values, with intervening delimiter characters.

2. Numeric Values

Format 1:

s	d	d	...	d	.	d	d	...	d

Format 2:

s	d	.	d	d	d	d	d	d	d	d	E	±	d	d

sign — mantissa — exponent

where: s = sign (space if value >= 0 or minus sign (–) if value < 0)
d = ASCII digit

Format 1 is used if the value is greater than or equal to 10^{-1} and less than 10^{+13} or if the value is less than 10^{-1} but can be expressed in 13 or fewer digits. *Format 2* is used in all other cases. Numeric values are stored as ASCII characters in the formats above.

> **NOTE:**
>
> Formats 1 and 2 are the same formats in which numeric values are printed by a PRINT statement.

Example 1:

This program packs the values of X, A$, and Y into B$; values are separated by commas.

```
:100 DIM B$30,A$5
:110 A$ = "ABC" : X = -12 : Y = 4.56E-18
:120 D$ = HEX(002C)
:130 $PACK (D = D$) B$ FROM X, A$, Y
:140 PRINT "B$="; B$
:RUN
B$ = -12,ABC , 4.56000000E-18
```

The Field Form of the $PACK Statement

The field form of $PACK stores data values in the buffer in specified fields (i.e., each value occupies a specified number of characters). The values of the variables and arrays in the variable list are sequentially read and placed in the buffer. The packing terminates when all the specified values have been stored.

The alpha-variable or literal string following the parameter "F=" contains the field specifications for the buffer. Each field specification consists of two bytes. The first byte specifies the type of field, and the second byte specifies the field width (i.e., the number of characters in that field). The following two examples illustrate that the field specifications for a buffer may be contained within an alphanumeric-variable or expressed as a hexadecimal literal string:

```
$PACK (F = F$) B$() FROM X()
```

field specification variable

```
$PACK (F = HEX(1008)) B$() FROM X()
```

field specification
specified as hex literal

If the first byte of the field specification is HEX(00), the corresponding field in the buffer is skipped. Alphanumeric fields are indicated by specifying HEX(A0) as the first byte of the field specification. Several types of numeric fields are permitted; numeric data is indicated by specifying a hex digit from 1 to 6 as the first hex digit of the first byte in the field specification. Each of the digits 1-6 identifies a

different numeric format (see Table 12-2 below). The second hex digit specifies the implied decimal position in binary; the decimal point is assumed to be the specified number of digits from the right-hand side of the field. For example, the value +123.45 is stored as +12345, and the implied decimal point position is 2. An error will result if a numeric value is packed into an alphanumeric field or if an alphanumeric value is packed into a numeric field.

Table 12-2.
Valid Field Specifications

Numeric Fields	Meaning
00xx	skip field
10xx	ASCII free format
2dxx	ASCII integer format
3dxx	IBM display format
4dxx	IBM USASCII — 8 format
5dxx	IBM packed decimal format
6dxx	unsigned packed decimal format
A0xx	alphanumeric field

where: xx = field width in binary (xx > 0)
d = implied decimal position in binary

A separate field specification must be supplied for every variable or array in the variable list. All elements in an array utilize the field specification specified for that array. For example, the following statement:

```
$PACK (F = F$) B$( ) FROM A$, B( ), C$
```

requires three field specifications. If F$ = HEX(A0081006A010), then

A008 is the field specification for A$.
1006 is the field specification for each element in the array B( ).
A010 is the field specification for C$.

A field specification may also be defined mnemonically in a $FORMAT statement (see the discussion of the $FORMAT statement earlier in this section). This technique permits the programmer to utilize simple mnemonic codes rather than hex codes for field specifications. For example, the field specifications defined above in F$ could be defined as follows in a $FORMAT statement:

```
$FORMAT F$ = A8, F6, A16
```

Buffer format:

field 1	field 2	field 3	. . .	field n

Data format:

1. Alphanumeric Fields (A0xx)

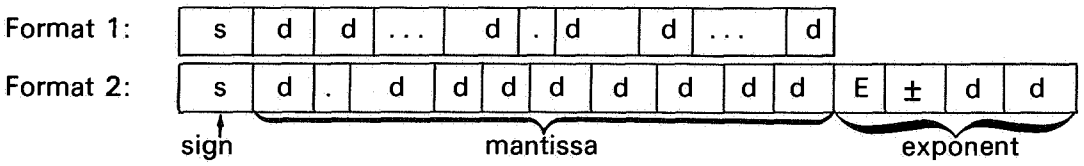
C	C	. . .	C

where: C is one character of the value to be packed.

If the value is shorter than the length of the field, the value is left-justified in the field and the remainder of the field is filled with spaces. If the value is too long, it is truncated to fit within the field.

2. Numeric Fields

a) ASCII free format (10xx)



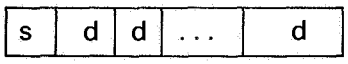
where: s = sign (space if value >= 0 or minus sign (–) if value < 0)
d = ASCII digit

Format 1 is used if the value is greater than or equal to 10^{-1} and less than 10^{+13} or if the value is less than 10^{-1} but can be expressed in fewer than 13 digits. *Format 2* is used in all other cases.

Numeric values are stored as ASCII characters in one of the formats illustrated above. Note that Formats 1 and 2 are the same formats used by the PRINT statement when printing numeric values. If the value to be stored is shorter than the length of the field, the value is left-justified in the field and the remainder of the field is filled with spaces. If the value is too large to fit in the field, it is truncated to the length of the field.

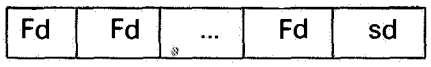
For the numeric formats illustrated below, the following rule applies: If the value is shorter than the length of the field, leading zeroes are inserted. If the value is too long, an error results.

b) ASCII integer format (2dxx)



where: s = sign (ASCII + or –), required
d = ASCII digit

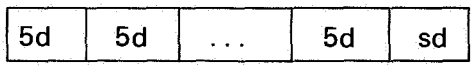
c) IBM display format (3dxx)



(hexadecimal notation)

where: s = sign (C=+, D=–)
d = digit (0-9)

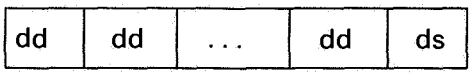
d) IBM USASCII-8 format (4dxx)



(hexadecimal notation)

where: s = sign (A=+, B=–)
d = digit (0-9)

e) IBM packed decimal format (5dxx)



(hexadecimal notation)

where: s = sign (C=+, D=–)
d = digit (0-9)

f) Unsigned packed decimal format (6dxx)

dd	dd	. . .	dd	dd

(hexadecimal notation)

where: d = digit (0-9)

Decimal arithmetic may be performed on unsigned packed decimal numbers. (See the discussion of the DAC and DSC operators in Chapter 6, section 6.2.)

Example 1:

Suppose a buffer B$ is to have three fields (one alpha and two numeric), each five characters long. The values of A$, X, and Y are to be packed into B$.

```
:10 DIM B$15
:20 A$ = "ABC": X = -12: Y = +1.2345
:30 F$ = HEX(A00520051005)
:40 $PACK (F = F$) B$ FROM A$, X, Y
```

B$ = "ABC -0012 1.23 "

Example 2:

Suppose X = 12.345

a)
```
:10 F$ = HEX(100A)
:20 $PACK (F = F$) B$ FROM X
```

Results in B$ = "12.345"

b)
```
:10 F$ = HEX(240A)
:20 $PACK (F = F$) B$ FROM X
```

Results in B$ = "+000123450"

Note that there is an implied decimal point 4 digits from the right end of the field.

c)
```
:10 F$ = HEX(3305)
:20 $PACK (F = F$) B$ FROM X
```

Results in B$ = HEX(F1F2F3F4C5)

d)
```
:10 F$ = HEX(4206)
:20 $PACK (F = F$) B$ FROM X
```

Results in B$ = HEX(5050515253A4)

e)
```
:10 F$ = HEX(5506)
:20 $PACK (F = F$) B$ FROM X
```

Results in B$ = HEX(00001234500C)

f)
```
:10 F$ = HEX(A010)
:20 $PACK (F = F$) B$ FROM X
```

Results in an error (ERR X76) because a numeric value is being packed into an alphanumeric field.

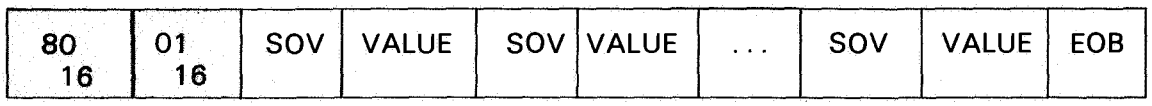
g) :10 F$ = HEX(2304)
:20 $PACK (F = F$) B$ FROM X

Results in an error (ERR X71) because the receiving field is too small to hold the value.

The Internal Form of the $PACK Statement

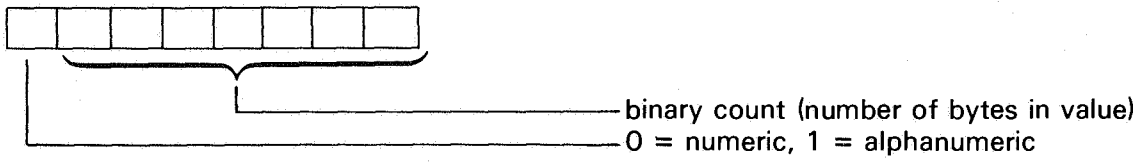
The internal form of $PACK stores data in the standard Wang 2200 disk record format. This is the format used by the DATASAVE DC statement when saving a data record on disk. The values of the variables and arrays in the variable list are sequentially packed into the buffer. The packing terminates when all values have been packed.

Standard Wang 2200 Record Format (buffer format)



Control Bytes

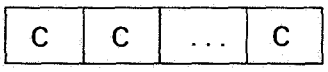
1. SOV (Start-Of-Value) character precedes each data value in the record and indicates whether the value is numeric or alphanumeric and the length of the value.



2. EOB (End-Of-Block, HEX(FD)) character indicates the end of valid data in the record.

3. Data format:

   a) Alphanumeric Values

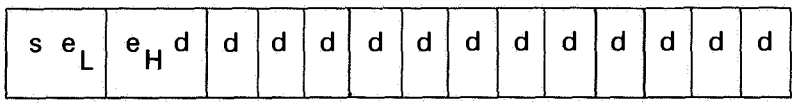


   where: C is any character of the alphanumeric value to be packed.

   Trailing spaces are included after the actual value so that the length of the entire value stored is the same as the defined length of the alphanumeric-variable.

   b) Numeric Values

   Numeric values are stored in Wang Internal Numeric Format.



(8 bytes)

where:

s = sign:
- 0 if mantissa +, exponent +
- 1 if mantissa −, exponent +
- 8 if mantissa +, exponent −
- 9 if mantissa −, exponent −

e_L e_H = exponent (2 digits)

d = mantissa digit (always 13)

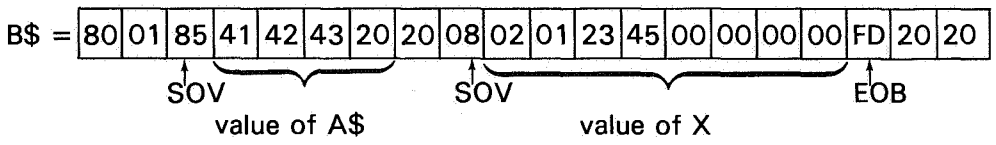
Numeric values are normalized (i.e., leading zeroes are eliminated). All digits must be BCD.

Example 1:

The values of A$ and X are packed into a buffer B$ in internal format with the following routine:

```
:10 DIM B$20, A$5
:20 A$ = "ABC" : X = 123.45
:30 $PACK B$ FROM A$, X
```

The resulting buffer appears as follows:



ROTATE

General Form:

ROTATE [C] (alpha-variable, expression)

Where:

-8 <= expression < 9

Purpose:

The ROTATE statement, if not followed by a "C", rotates the bits of each character in the value of the alphanumeric-variable the specified number of bits. All characters in the alphanumeric-variable are operated on, including trailing spaces. The number of bits to rotate in each character is specified by the integer portion of the value of the expression. If the value is positive, the bits of each character are rotated to the left; the high-order bits of each replace the low-order bits. If the value is negative, the bits are rotated to the right; the low-order bits replace the high-order bits of each character. For example:

```
:10 DIM A$2
:20 A$ = HEX(8142)
:30 ROTATE (A$,1)
:40 PRINT HEXOF (A$)
:50 ROTATE (A$,-3)
:60 PRINT HEXOF (A$)
:RUN
0384
6090
```

If ROTATE is followed by "C", the entire value of the alpha-variable is rotated the specified number of bits (i.e., the entire value is treated as a single bit string). If the value of the expression is positive, the rotation proceeds to the left; a negative value causes a rotation to the right. For example:

```
:10 DIM A$2
:20 A$ = HEX(8142)
:30 ROTATEC (A$,1)
:40 PRINT HEXOF (A$)
:50 ROTATEC (A$,-3)
:60 PRINT HEXOF (A$)
:RUN
0285
A050
```

Note that ROTATEC with a value of 8 rotates the value one character to the left; a value of -8 rotates the value one character to the right. For example:

```
:10 DIM A$5
:20 A$ = "ABCDE"
:30 ROTATEC (A$,8)
:40 PRINT A$
:50 A$="ABCDE"
:60 ROTATEC(A$,-8)
:70 PRINT A$
:RUN
BCDEA
EABCD
```

Examples of Valid Syntax:

```
10 ROTATE (A$,3)
20 ROTATE (B$( ),X)
30 ROTATEC(STR(C$,X,Y), 2*Z)
40 ROTATEC(A$,-8)
```

$TRAN

General Form:

$TRAN (arg-1, arg-2) [mask] [R]

Where:

arg-1 = alpha-variable

arg-2 = alpha-variable or literal-string

mask = two hex digits

Purpose:

The TRANslate instruction, $TRAN, is used primarily to perform code conversion (for example, to convert EBCDIC to ASCII). Each byte from arg-1 is replaced by a byte determined by arg-2 according to a systematic table-lookup procedure. The $TRAN instruction may be used whenever byte substitutions are needed to perform operations such as hex conversions, character verification, and initialization.

If a mask is specified, each character of arg-1 is logically ANDed with the specified mask character prior to the translation.

The general form of the $TRAN statement represents two different table-lookup procedures for data conversion operations. The desired procedure is identified by the inclusion or omission of the parameter "R". If the "R" parameter is specified in a $TRAN statement, the arg-2 component of the statement must represent a list of "to-from" translation characters. The list must consist of pairs of bytes to be interpreted as follows:

a) The second byte in each pair of bytes is a "translate from" character, i.e., the even-numbered bytes in an arg-2 list define the set of characters to be translated.

b) The first byte in each pair of bytes is a "translate to" character corresponding to a particular "from" character, i.e., the odd-numbered bytes in an arg-2 list define the replacement characters for a translation operation. A "to-from" pair of blank characters denotes the end of the translation table.

During execution of a $TRAN statement with the "R" parameter specified, the following events occur:

1. The next successive arg-1 byte is masked if a mask is specified.

2. The masked (or original) byte is looked up in the arg-2 list by comparing the byte with each successive "translate from" character in the list.

3. As soon as a match is found, the corresponding replacement character (the preceding "translate to" character) is returned to the arg-1 byte position currently being processed.

4. If no match is found, the masked (or original) byte is returned to the arg-1 byte position currently being processed.

Alternatively, if the "R" parameter is omitted in a $TRAN statement, the arg-2 component of the statement must specify the translation table as a set of consecutive characters. When "R" is not specified, the sequential position of each character in a translation table is extremely important since

the table-lookup procedure is a "displacement" procedure executed as follows:

1. The next successive arg-1 byte is masked if a mask is specified.

2. The equivalent decimal system value of the masked (or original) byte is calculated by treating the byte (the two-hexdigit-code) as an 8-bit binary number rather than an ASCII character, e.g., in the ASCII character set used by Wang systems, an unmasked uppercase "G" is represented by HEX (47) = binary 01000111 = decimal 71.

3. The decimal system value from step two becomes the "displacement" used to locate the proper translation character in the arg-2 table. The displacement can be defined as the movement of an imaginary pointer which points to the first position in the table for a zero displacement and moves to the (m+1)th position for a displacement of "m". For example, since the decimal system value of an ASCII "G" is 71, its translation character must appear in position 72 in an arg-2 table if no mask is used in the translation operation. (See example 2, which follows.)

4. After a translation character is located in the arg-2 table, the character is stored in the arg-1 byte position currently being processed.

5. If the translation table is too short for the required displacement in step three, the masked (or original) byte is returned to the arg-1 position currently being processed.

Example 1: Translating Selected Codes

Assuming data has been stored in A$ by program logic executed prior to line 100, the following sequence replaces each HEX(11) code in A$ by a HEX(0D) code and replaces each HEX(07) code by a HEX(00) code:

```
100 $TRAN (A$, HEX(0D110007))R
```

Example 2: Extracting the Low-Order Hex Digit From ASCII Codes

The following program sequence extracts and prints the low-order hex digit from each ASCII character stored in X$:

```
:5 DIM X$4
:10 T$="0123456789ABCDEF"
:20 X$="*GO#"
:30 PRINT HEXOF(X$)
:40 $TRAN (X$,T$) 0F
:50 PRINT X$
:RUN
2A474F23
A7F3
```

In line 10, a table containing the characters corresponding to the sixteen hexadecimal symbols (called hex digits for convenience) is assigned to T$. In line 20, a sample set of ASCII characters is stored in X$ to demonstrate the effect of the actual $TRAN operation when the program is run. Upon execution, line 30 prints the hexadecimal notation for each byte stored in X$ prior to the translation operations. In line 40, the mask (HEX(0F)= binary 00001111) replaces the four high-order bits in each byte stored in X$ with zeros before calculating the displacement for the table-lookup procedure.

Originally, the first byte of X$ is an asterisk character whose code is HEX(2A) (= binary 00101010). After the logical AND operation with the mask (HEX(0F)), the result is HEX(0A) (= binary 00001010). The decimal equivalent of the resulting code is 10, which becomes the displacement for the table-lookup procedure. The displacement moves the invisible pointer from the first to the eleventh position in the T$ table, where the character "A" is located; hence, the character "A" replaces the asterisk as the first byte in X$. The process continues with the second byte of X$, which is translated from the character "G" to the character "7" by the same procedure, etc.

UNPACK

General Form:

UNPACK (image) alpha-variable TO numeric-variable [,numer-variable] . . .

Where:

image = [±] [#]. . . [.] [#]. . . [↑↑↑↑] or alpha-variable containing image

image length < 255

Purpose:

The UNPACK statement is used to unpack numeric data originally packed by a PACK statement. Starting at the beginning of the specified alphanumeric variable or array, packed numeric data is unpacked, converted to internal format, and then stored in the specified numeric variables or arrays. The format of the packed data is specified by the image (see the discussion of the PACK statement earlier in this section); thus, the same image used originally to pack the data should be used in the UNPACK statement. Values are sequentially unpacked and assigned to the receiving numeric variables or arrays; the number of receiving variables determines the number of values to be unpacked. An error is signalled if there are not enough packed values in the alpha-variable to fill all the receiving numeric-variables.

Examples of Valid Syntax:

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

$UNPACK

General Form:

$UNPACK [({ F / D } = { alpha-variable / literal-string })] alpha-variable TO { variable / array } [,{ variable / array }]...

Purpose:

The $UNPACK statement is used to extract data from a buffer and store the data values in specified variables. Data values are sequentially read from the buffer and stored in the variables and arrays following the word "TO". Arrays are filled element by element and row by row; each array-element receives one data value. The buffer is the alpha-variable preceding the word "TO". For example:

```
$UNPACK A$() TO X, A$, Y(), B$()
          ↑
buffer containing data      variables to receive data
```

Three forms of $UNPACK are available:

1. Delimiter Form (specified by the "D=" parameter)

   The delimiter form unpacks data which are separated by a specified delimiter character.

2. Field Form (specified by the "F=" parameter)

   The field form unpacks data which are separated into fields in which each data value occupies a specified number of bytes.

3. Internal Form (assumed if neither the "D=" nor the "F=" parameter is specified)

   Data stored in the standard Wang 2200 disk record format can be unpacked by the internal form of $UNPACK.

Examples of Valid Syntax:

```
$UNPACK A$ TO X
$UNPACK STR(A$,5) TO X, B$, Y
$UNPACK (F = F$) A$() TO X()
$UNPACK (D = D$) A$() TO B$()
$UNPACK (F = A$()) B1$ TO A$, B$, STR(C$,3,2)
$UNPACK (D = STR(Q$,3,2)) X$() TO X, Y, Z(1,2)
```

The Delimiter Form of the $UNPACK Statement

The delimiter form of $UNPACK is used to unpack data values which are separated by a specified delimiter character. Data values are sequentially read from the buffer and stored in the variables following the word "TO". The unpacking terminates when either the buffer is empty or the entire variable list has been satisfied.

The first two bytes of the alpha-variable or literal string following the "D=" parameter serve as the delimiter specification. For example:

```
$UNPACK (D = A$) B$( ) TO X, Y, Z
```

delimiter specification variable

```
$UNPACK (D = HEX(002C)) B$( ) TO X, Y, Z
```

delimiter specification specified as hex literal

The first byte of the delimiter specification is an atom defining certain unpacking rules. If there is insufficient data in the buffer to satisfy all the receiver-variables, the atom specifies whether program execution will terminate and an error message will be displayed or whether the remaining variables will simply retain their current values. In addition, if there is more than one delimiter character between data values in the buffer, the atom specifies whether successive delimiters will be ignored or whether variables in the variable list will be skipped and allowed to retain their current values. The second byte is the delimiter character itself (i.e., the character that separates each pair of data values in the buffer). Valid delimiter specifications (described in hexadecimal notation) appear in Table 12-3 which follows.

Table 12-3.
Valid Delimiter Specifications

Specification	Meaning
00xx	Error if insufficient data for all variables in list. Skip variables in list if successive delimiters occur between data values in the buffer.
01xx	Ignore remaining variables if insufficient data. Skip variables in list if successive delimiters occur between data values in the buffer.
02xx	Error if insufficient data for all variables in list. Ignore successive delimiters.
03xx	Ignore remaining variables if insufficient data. Ignore successive delimiters.

where: xx = delimiter character

Buffer format:

data	D	data	D	data	D	D	D	data	. . .	D	data

where: D = delimiter character

Data format:

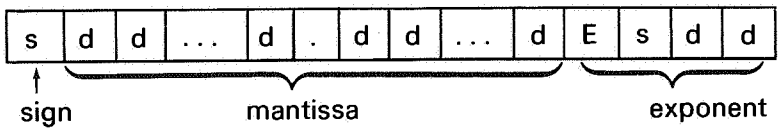
1. Alphanumeric Values

C	C	. . .	C

where: C is any character except a delimiter.

2. Numeric Values (ASCII free format)

The data can be any valid BASIC representation of a number; spaces are ignored (see Chapter 4).



where: s = sign (ASCII + or −), optional
d = ASCII digit
1 < = number of mantissa digits < = 13
decimal point optional
exponent optional (one or two digits)

NOTE:

A delimiter of space, plus sign (+), minus sign (−), digit, decimal point, or "E" could create confusion with arbitrary numeric data.

Example 1:

Suppose B$ contains five numeric data values separated by commas. The following routine will unpack these values and store them in the variables A, B, C, D, and E.

B$ = "123,-12345,0,+5,.009"

```
:100 D$ = HEX(002C)
:110 $UNPACK (D = D$) B$ TO A, B, C, D, E
:120 PRINT A; B; C; D; E
:RUN
 123  -12345    0    5    .009
```

Example 2:

Suppose B$ contains three alphanumeric values separated by spaces.

B$ = "ABC DEF GHI"

a) :100 D$ = HEX(0320)
:110 $UNPACK (D = D$) B$ TO W$, X$, Y$, Z$

Results in W$ = "ABC"
X$ = "DEF"
Y$ = "GHI"
Z$ = unchanged

b) :100 D$ = HEX(0220)
:110 $UNPACK (D = D$) B$ TO W$, X$, Y$, Z$

Results in error (ERR X76) because there are not enough values to satisfy all the variables in the list.

c) :100 D$ = HEX(0020)
:110 $UNPACK (D = D$) B$ TO W$, X$, Y$, Z$

Results in W$ = "ABC"
X$ = "DEF"
Y$ = "GHI"
Z$ = unchanged

The Field Form of the $UNPACK Statement

The field form of $UNPACK is used to unpack data which is divided into fields so that each data value occupies a specified number of characters. Data values are sequentially read from the buffer and stored in the variables following the word "TO". The unpacking terminates either when the buffer is empty or the entire variable list has been satisfied.

The alpha-variable or literal string following the "F=" parameter provides the field specification for the buffer. Each field specification is two bytes in length. The first byte specifies the type of field, and the second byte specifies the field width (i.e., the number of characters in that field).

```
$UNPACK (F = F$) B$( ) TO X, Y, Z
          ↗
field specification variable
```

```
$UNPACK (F = HEX(1008)) B$( ) TO X, Y, Z
          ↗
field specification
specified as hex literal
```

If the first byte of the field specification is HEX(00), the corresponding field in the buffer is skipped. Alphanumeric fields are indicated by specifying HEX(A0) as the first byte of the field specification. Several types of numeric fields are permitted; numeric data is indicated by specifying a hex digit from 1 to 6 as the first hex digit of the first byte in the field specification. Each of the digits 1-6 identifies a unique numeric format (see Table 12-4 below). The second digit specifies the implied decimal position in binary; the decimal point is assumed to be the specified number of digits from the right-hand side of the field. For example, if a field contains the value +12345 and an implied decimal position of 2 is specified, the value unpacked would be +123.45. An error will result if a numeric field is unpacked into an alphanumeric-variable or if an alphanumeric field is unpacked into a numeric-variable.

Table 12-4.
Valid Field Specifications

Numeric Fields	Meaning
00xx	skip field
10xx	ASCII free format
2dxx	ASCII integer format
3dxx	IBM display format
4dxx	IBM USASCII - 8 format
5dxx	IBM packed decimal format
6dxx	unsigned packed decimal format
A0xx	alphanumeric field

where: xx = field width in binary (xx > 0)
d = implied decimal position in binary

A separate field specification must be specified for every variable or array in the variable list. For example, the following statement:

```
$UNPACK (F = F$) B$( ) TO A$, B( ), C$
```

requires three field specifications. If F$ = HEX(A0081006A010), then

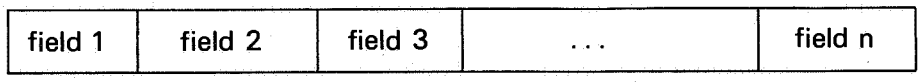
A008 is the field specification for A$.
1006 is the field specification for each element in the array B( ).
A010 is the field specification for C$.

It is also possible to define a field specification mnemonically in a $FORMAT statement. $FORMAT permits the use of simple mnemonics instead of hex codes to specify field formats. For example, the field specification defined for F$ above could be defined as follows in a $FORMAT statement:

```
$FORMAT F$ = A8, F6, A16
```

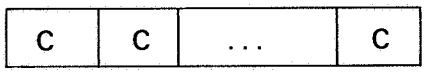
See the discussion of the $FORMAT statement for the meanings of the mnemonics used.

Buffer format:



Data format:

1. Alphanumeric Fields (A0xx)

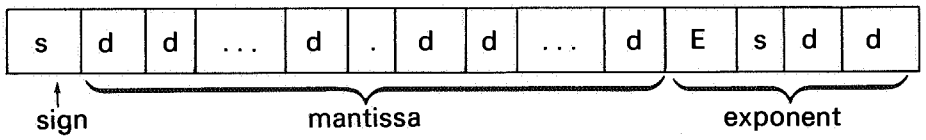


where: C is any character in an alphanumeric field to be unpacked.

2. Numeric Fields

a) ASCII free format (10xx)

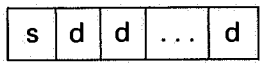
The data can be any valid BASIC representation of a number; spaces are ignored (see Chapter 4).



where: s = sign (ASCII + or −), optional
d = ASCII digit
1 < number of mantissa digits < = 13
decimal point optional
exponent optional (one or two digits)

For each of the following formats (b-f), each number may have up to 13 digits of precision. If there are more than 13 significant digits in a numeric value, the exponent of the number is appropriately adjusted. Leading zeros are ignored.

b) ASCII integer format (2dxx)



where: s = sign (ASCII + or −), required
d = ASCII digit

c) IBM display format (3dxx)

Fd	Fd	. . .	Fd	sd

(hexadecimal notation)

where: s = sign (C=+, D= −)*
d = digit (0-9)

d) IBM USASCII-8 format (4dxx)

5d	5d	. . .	5d	sd

(hexadecimal notation)

where: s = sign (A=+, B= −)*
d = digit (0-9)

e) IBM packed decimal format (5dxx)

dd	dd	. . .	dd	ds

(hexadecimal notation)

where: s = sign (C=+, D= −)*
d = digit (0-9)

*The $UNPACK statement considers B or D to be minus (−); any other hex digit is considered to be plus (+).

f) Unsigned packed decimal format (6dxx)

dd	dd	. . .	dd	dd

where: d = digit (0-9)

Decimal addition and subtraction can be performed on unsigned packed decimal numbers. (See the discussion of the DAC and DSC operators in Chapter 6, section 6.2.)

Example 1:

Suppose B$ contains the following data:

B$ = "+12345678901234567890"

a)
```
:100 F$ = HEX(A005)
:110 $UNPACK (F = F$) B$ TO A$
```

Results in A$ = "+1234"

b)
```
:100 F$ = HEX(1010)
:110 $UNPACK (F = F$) B$ TO X
```

Results in an error (ERR X75) because B$ contains more than 13 digits.

c)
```
:100 F$ = HEX(2015)
:110 $UNPACK (F = F$) B$ TO X
```

Results in X = 1.234567890123E19

d)
```
:100 F$ = HEX(2206)
:110 $UNPACK (F = F$) B$ TO X
```

Results in X = 123.45

Example 2:

```
:10 B$ = HEX(F1F2F3D4)
:20 F$ = HEX(3304)
:30 $UNPACK (F = F$) B$ TO X
```

Results in X = -1.234

Example 3:

```
:10 B$ = HEX(51525354A5)
:20 F$ = HEX(4005)
:30 $UNPACK (F = F$) B$ TO X
```

Results in X = 12345

Example 4:

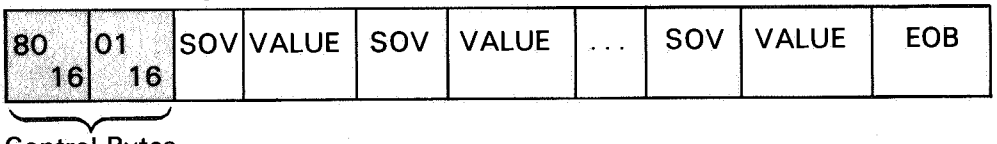
```
:10 B$ = HEX(000012345C)
:20 F$ = HEX(5105)
:30 $UNPACK (F = F$) B$ TO X
```

Results in X = 1234.5

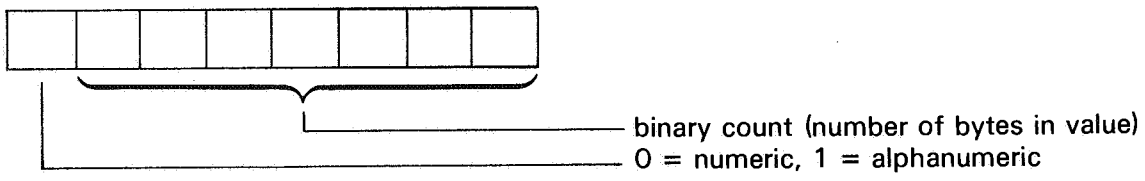
The Internal Form of the $UNPACK Statement

Data stored in the standard Wang 2200 disk record format can be unpacked by the internal form of $UNPACK. Data records that have been saved on a disk platter by either the DATASAVE DC or the DATASAVE DA statement are stored in this format. Data values are sequentially read from the buffer and stored in the variables following the word "TO". The unpacking terminates when the buffer is empty and the EOB (End-of-Block) character is encountered or when the entire variable list has been satisfied. An error will result if a numeric value is unpacked into an alphanumeric-variable or if an alphanumeric value is unpacked into a numeric-variable.

Standard Wang 2200 Record Format (buffer format)



1. SOV (Start-of-Value) character precedes each data value in the record and indicates whether the value is numeric or alphanumeric and the length of the value.



2. EOB (End-of-Block, HEX(FD)) character indicates the end of valid data in the record.

3. $UNPACK ignores the first two control bytes.

4. Data format:

   a. Alphanumeric Values

C	C	...	C

   where: C is any character of the alphanumeric value to be unpacked.

   b. Numeric Values

   Numeric values must be in Wang Internal Numeric Format.

s e L	e d H	d d	d d	d d	d d	d d	d d

(8 bytes)

   where: s = sign: 0 if mantissa +, exponent +
   1 if mantissa −, exponent +
   8 if mantissa +, exponent −
   9 if mantissa −, exponent −

   $e_L e_H$ = exponent (2 digits)

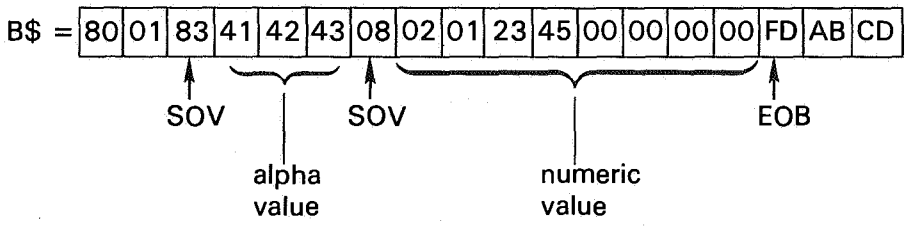
   d = mantissa digit (always 13)

   Leading zeroes in numeric values are eliminated. All digits must be BCD.

> **NOTE:**
>
> If the numeric values are invalid, the results of the conversion performed by $UNPACK are undefined.

Example:

Suppose B$ contains one alphanumeric value and one numeric value. The contents of B$ are to be unpacked into the variables A$ and X.




```
:100 $UNPACK B$ TO A$, X
:110 PRINT A$; X
:RUN
ABC 123.45
```

CHAPTER 13 MATH MATRIX STATEMENTS

13.1 INTRODUCTION

The math matrix statements provide 14 built-in matrix operations, summarized in Table 13-1 below.

Table 13-1.
Matrix Operations

Operation	Description	Example
MAT addition	array = array + array	MAT A = B+C
MAT CON	each element of array = 1	MAT A = CON
MAT equality	array = array	MAT A = B
MAT IDN	matrix = identity matrix	MAT A = IDN
MAT INPUT	receive array-elements from keyboard	MAT INPUT A,B$
MAT INV,d,n	matrix = inverse of matrix d = determinant of matrix n = normalized determinant	MAT A = INV (B),D,N
MAT multiplication	array = array x array	MAT A = B*C
MAT PRINT	print elements of array	MAT PRINT A,B$
MAT READ	array = DATA values	MAT READ A,B$
MAT REDIM	redimension array	MAT REDIM A(X,Y)
MAT scalar multiplication	array = constant x array	MAT A = (3)*B
MAT subtraction	array = array − array	MAT A = B−C
MAT TRN	array = transpose of array	MAT A = TRN(B)
MAT ZER	each element of array = 0	MAT A = ZER

Matrix operations are performed on numeric-arrays according to the rules of linear algebra and can be used to solve systems of nonsingular homogeneous linear equations. Inversion of matrices can be done in significantly shorter time than is possible with BASIC programs. MAT operations on alphanumeric-arrays can be used for simple and rapid I/O (Input/Output) and printing of alphanumeric material.

13.2 ARRAY DIMENSIONING

Both numeric- and alphanumeric-arrays can be manipulated with the matrix statements. The rules of the BASIC language require that each array be dimensioned. An array is normally dimensioned in a DIM or COM statement before being used in a MAT statement. If an array is not dimensioned by DIM or COM, it is automatically dimensioned to be a 10 x 10 matrix when referenced in a MAT statement. In an alphanumeric-array, the maximum length of each element is set equal to 16 bytes.

13.3 ARRAY REDIMENSIONING

The dimensions of an array can be changed explicitly during the execution of MAT statements by supplying the new dimensions, enclosed in parentheses, following the array-name in any of the following MAT statements:

```
MAT CON
MAT IDN
MAT INPUT
MAT READ
MAT REDIM
MAT ZER
```

For example, assume the array A( ) was initially dimensioned as a 10 X 8 array. The statement

```
MAT A = CON(5,5)
```

causes A( ) to be redimensioned as a 5 X 5 array.

Arrays can also be redimensioned implicitly. For example:

```
10 DIM A(10,10),B(2,2),C(2,2)
20 . . .
30 . . .
40 MAT A=B+C
```

The array A( ) is redimensioned at statement 40 from a 10 x 10 array to a 2 x 2 array.

For alphanumeric-arrays, the maximum length of each element can be changed by specifying the new length after the dimension specification. For example:

```
MAT REDIM A$(2,3)10
```

This statement redimensions the array A$( ) to be two rows by three columns with the maximum length of each element in the array equal to 10 bytes.

13.4 REDIMENSIONING RULES

1. When explicit or implicit redimensioning occurs, the space which the newly dimensioned array requires can not exceed the amount of space available in the original array. For numeric-arrays, the total number of elements must remain constant or decrease. For alphanumeric-arrays, the total number of characters (bytes) must also remain constant or decrease.

2. A vector (a singly subscripted array) cannot be redimensioned as a matrix (a doubly subscripted array) nor can a matrix be redimensioned as a vector.

3. Caution must be exercised when using DIM or COM statements in programs where redimensioning occurs. When a program is overlayed, the entire program is resolved (see Chapter 2, section 2.2). At this time, any existing common variable which is explicitly DIMed or COMed must agree in size and shape with its original specifications. In particular, arrays that have been redimensioned must usually be restored to their original size before overlaying. The preceding caution also applies if a program is to be RUN again after stopping it without clearing the common variables.

13.5 GENERAL FORMS OF THE MATH MATRIX STATEMENTS

General forms of the 14 math matrix statements summarized in Table 13-1 are shown on the following pages, arranged alphabetically for ease of reference.

MAT + (MAT addition)

General Form:

MAT c = a + b

Where:

c, a, and b are numeric-array-names.

Purpose:

This statement adds two matrices or vectors of the same dimensions. The sum is then stored in array c, which can appear on both sides of the equation. Array c is redimensioned to have the same dimensions as arrays a and b. An error occurs if the dimensions of a and b are not the same.

If arrays A( ) and B( ) each have N rows and M columns, then the statement

```
10 MAT C = A + B
```

is equivalent to the following statements:

```
10 FOR I = 1 TO N: FOR J = 1 TO M
20 C(I,J) = A(I,J) + B(I,J)
30 NEXT J,I
```

For example, if arrays A( ) and B( ) have the following values:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & .5 & 2 \\ 1 & -1 & -2 \end{bmatrix}$$

then MAT C = A + B results in

$$C = \begin{bmatrix} 1 & 2.5 & 5 \\ 5 & 4 & 4 \end{bmatrix}$$

Examples of Valid Syntax:

```
10 MAT A = A + D
20 MAT E = F + G
30 MAT A = A + A
```

MAT CON (MAT CONstant)

General Form:

MAT c = CON [(dim1[,dim2])]

Where:

c is a numeric-array-name, and dim1 and dim2 are expressions specifying new dimensions.

For one-dimensional arrays:
1 <= dim1 < 65536

For two-dimensional arrays:
1 <= dim1, dim2 < 256

Purpose:

This statement sets all elements of the specified array equal to one. Use of the dimension expressions (dim1 and dim2) causes the matrix to be redimensioned to the specified size. The number of elements in the redimensioned array must be less than or equal to the number of elements in the array when originally dimensioned.

If A( ) is a matrix with N rows and M columns, the statement

```
10 MAT A = CON
```

is equivalent to the following statements:

```
10 FOR I = 1 TO N: FOR J = 1 TO M
20 A(I,J) = 1
30 NEXT J,I
```

Examples of Valid Syntax:

```
10 MAT A = CON(10)
20 MAT C = CON(5,7)
30 MAT B = CON(5*Q,S)
40 MAT A = CON
```

MAT = (MAT equality)

General Form:

MAT c = a

Where:

c and a are numeric-array-names.

Purpose:

This statement replaces each element of array c with the corresponding element of array a. Array c is automatically redimensioned to conform to the dimensions of array a.

If arrays A( ) and B( ) are matrices with N rows and M columns, the statement

```
10 MAT A = B
```

is equivalent to the following statements:

```
10 FOR I = 1 TO N: FOR J = 1 TO M
20 A(I,J) = B(I,J)
30 NEXT J,I
```

For example, if the following contents of A( ) and B( ) are assumed:

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \end{bmatrix}$$

then the statement MAT A = B causes A( ) to assume the following values:

$$A = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \end{bmatrix}$$

Examples of Valid Syntax:

```
10 MAT A=B
20 MAT C=D
30 MAT F=E
```

MAT IDN (MAT identity)

General Form:

MAT c = IDN [(dim1,dim2)]

Where:

c is a numeric-array-name, and dim1,dim2 are expressions specifying new dimensions (1 <= dim1, dim2 < 256).

Purpose:

This statement causes the specified matrix to assume the form of the identity matrix. If the specified matrix is not a square matrix, an error message is displayed and execution is terminated. Use of the dimension expressions dim1 and dim2 causes the matrix to be redimensioned to the specified size. The number of elements in the redimensioned array must be less than or equal to the number of elements in the array when originally dimensioned, and the new dimensions (dim1 and dim2) must be equal.

If array C( ) has N rows and columns, the statement

```
10 MAT C = IDN
```

is equivalent to the following statements:

```
10 FOR I = 1 TO N: FOR J = 1 TO N
20 IF I = J THEN C(I, J) = 1: ELSE C(I, J) = 0
30 NEXT J,I
```

For example, if C( ) is a matrix with four rows and four columns, the statement MAT C = IDN causes C( ) to assume the following value:

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Examples of Valid Syntax:

```
10 MAT A = IDN(4,4)
20 MAT B = IDN
30 MAT C = IDN(X,Y)
```

MAT INPUT

General Form:

MAT INPUT array-name [(dim1[,dim2]) [length]] [,...]

Where:

dim1, dim2 = expressions specifying new dimensions.
For one-dimensional arrays:
1 <= dim1 < 65536

For two-dimensional arrays:
1 <= dim1, dim2 < 256

length = expression specifying the maximum length of each alpha-array-element (1 <= length < 125). The default length of each element is 16.

Purpose:

The MAT INPUT statement allows the user to supply values from the keyboard for an array during execution of a program. When the system encounters a MAT INPUT statement, it displays a question mark (?) and waits for the user to supply values for the array(s) specified in the MAT INPUT statement. The dimensions of the array(s) are those which were last specified in the program by a COM, DIM, or MAT statement unless new dimensions (dim1, dim2) are supplied. The maximum length for alphanumeric-array-elements can be specified by including the length after the dimension specification; if no length is specified, a default value of 16 bytes is used.

Entered values are assigned to an array row by row until the array is filled. If more than one value is entered on a line, the values must be separated by commas. Alphanumeric data containing leading spaces or commas can be entered by enclosing the character string in quotes (" "). Several lines can be used to enter the required data, and excess data is ignored. If there is a system-detected error in the entered data, the data must be reentered, beginning with the erroneous value. All values preceding the error are accepted. Input data must be compatible with the array (i.e., numeric data must be entered for numeric-arrays and alphanumeric literal strings must be entered for alphanumeric-arrays). Entering no data on an input line (i.e., only touching the RETURN key to enter a carriage return) causes the system to terminate the MAT INPUT statement and ignore the remaining elements of the array currently being filled.

Example 1: Numeric-Arrays

```
5 DIM A(2),B(3),C(3,4)
10 MAT INPUT A,B(2),C(2,3)
```

After this program is run, the following values might be entered in response to the MAT INPUT request:

```
?-3, -5, .612, .41
```

If RETURN is keyed at this point, these values are assigned to array-elements A(1), A(2), B(1), and B(2). Because array C( ) remains to be filled, the request is repeated. The following values might be entered:

```
?-6.4, -5.6, 98
```

When RETURN is keyed, these values are assigned to array-elements C(1,1), C(1,2), and C(1,3). If RETURN is keyed a second time without entering further values, the MAT INPUT statement is terminated with the remaining elements of C( ) unfilled. The resulting values of arrays A( ), B( ), and C( ) are as follows:

$$A = \begin{bmatrix} -3 \\ -5 \end{bmatrix} \qquad B = \begin{bmatrix} .612 \\ .41 \end{bmatrix} \qquad C = \begin{bmatrix} -6.4 & -5.6 & 98 \\ 0 & 0 & 0 \end{bmatrix}$$

Example 2: Alphanumeric-Arrays

```
10 DIM C$(2), A$(4)4
100 MAT INPUT A$(4)3,C$
```

When line 100 is executed, MAT INPUT displays a question mark, and the operator can enter values for A$(1) - A$(4) and C$(1) - C$(2). Keying RETURN terminates the input operation.

Examples of Valid Syntax:

```
10 MAT INPUT A
20 MAT INPUT A$
30 MAT INPUT B(2,3),C
40 MAT INPUT B$(6)10,C,N
```

MAT INV (MAT INVerse)

General Form:

MAT c = INV(a) [,[d][,n]]

Where:

c and a are numeric-array-names.
d = a numeric-variable which equals the value of the determinant of array a.
n = a numeric-variable which equals the value of the normalized determinant of array a.

Purpose:

This statement causes matrix c to be replaced by the inverse of matrix a. Array c can appear on both sides of the equation. Matrix c is redimensioned to have the same dimensions as matrix a. Unless matrix a is a square matrix, an error message is displayed and program execution is terminated.

BASIC-2 performs matrix inversion by the Gauss-Jordan Elimination method done in place with maximum pivot strategy. The determinant is calculated during the inversion. If specified, the normalized determinant is also calculated.

After inversion, the variable d (if specified) equals the value of the determinant of matrix a. If a second variable is specified following the determinant variable, it receives the normalized determinant of matrix a.

Several types of errors can occur during matrix inversion:

1. Singular matrix
2. Computational errors (overflow or underflow)
3. Round-off error accumulation
4. Ill-conditioned matrix

Singular Matrices

If a matrix is singular (i.e., has no inverse), the determinant of that matrix is zero. If a determinant variable is not included in the MAT INV statement, the system will stop with an ERR X72. If a determinant variable is included in the MAT INV statement, it is set equal to zero and program execution continues; the resultant matrix contains meaningless values. The program must check the value of the determinant after each inversion.

Computational Errors

Computational errors can occur during the calculation of the inverse. Generally, it is best to let underflow default to zero; however, overflow should be detected (i.e., the programmer should SELECT ERROR > 60). If overflow is detected, the matrix being inverted can be scaled down by dividing each element by some constant before the inversion is performed.

Round-Off Error

Round-off error accumulation results from the successive calculations performed during the inversion of the matrix. By utilizing the maximum pivot strategy, BASIC-2 reduces this type of error; however, some loss of precision is inevitable, especially with large matrices. Round-off error can be detected by calculating the residuals defined by the equation below:

$$R = I - A * C$$

where: I = identity matrix
A = matrix being inverted
C = computed inverse

For an exact inverse, each element of R will be zero. If C is a close approximation of the inverse, each element of R will be small.

Ill-Conditioned Matrices

There are matrices for which the residual does not provide a reasonable measure of the accuracy of the resultant inverse. Such matrices are said to be ill-conditioned. Loss of accuracy is not due to round-off error accumulation or the algorithm chosen to perform the inversion, but rather to the nature of the data itself. Typically, the size of the determinant is used to detect ill-conditioned matrices, and small determinants usually indicate potential problems. However, for an accurate measurement of the condition of the matrix, the determinant must be normalized relative to the matrix being inverted; thus, BASIC-2 provides a normalized determinant. The normalized determinant for a matrix A is defined below:

$$\text{if } A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

$$\text{and } \alpha_K = \sqrt{a_{K1}^2 + a_{K2}^2 + \cdots + a_{Kn}^2}$$

$$\text{then NORM } |A| = \begin{bmatrix} a_{11}/\alpha_1 & a_{12}/\alpha_1 & \cdots & a_{1n}/\alpha_1 \\ a_{12}/\alpha_2 & a_{22}/\alpha_2 & \cdots & a_{2n}/\alpha_2 \\ \vdots & & & \vdots \\ a_{n1}/\alpha_n & a_{n2}/\alpha_n & \cdots & a_{nn}/\alpha_n \end{bmatrix}$$

$$= \frac{|A|}{\alpha_1 \alpha_2 \cdots \alpha_n}$$

If the normalized determinant of matrix A is small relative to one, then A is nearly singular and ill-conditioning can be expected.

Example: Inverting a 4 x 4 Matrix

The following program accepts values from the keyboard for a 4 x 4 matrix and then computes the inverse of the matrix, the value of the determinant, and the value of the normalized determinant:

```
10 DIM A(4,4)
20 PRINT "ENTER ELEMENTS OF A 4X4 MATRIX"
30 MAT INPUT A
40 MAT B=INV(A),D,N
50 MAT PRINT B
60 PRINT
70 PRINT "DETERMINANT =";D
80 PRINT "NORM        =";N
```

```
:RUN
ENTER ELEMENTS OF A 4X4 MATRIX
? 0,2,4,8
? 0,0,1,0
? 1,0,0,1
? 4,8,16,32

-1              0              0              .25
-3.5            -2             -4             1
 0              1              0              0
 1              0              1              -.25

DETERMINANT   =-8
NORM          =-1.67365481E-02
```

Examples of Valid Syntax:

```
10 MAT A = INV(B)
20 MAT Z1 = INV(P),D
30 MAT B = INV(B),D,N
```

MAT* (MAT multiplication)

General Form:

MAT c = a * b

Where:

c, a, and b are numeric-array-names.

Purpose:

Array a is multiplied by array b, and the product is stored in array c. Array c cannot appear on both sides of the equation. If the number of columns in array a does not equal the number of rows in array b, an error message is printed and execution is terminated. The resulting dimensions of array c are determined by the number of rows in array a and the number of columns in array b.

If array A( ) has N rows and M columns and array B( ) has M rows and P columns, the statement

```
10 MAT C = A * B
```

causes array C( ) to have N rows and P columns. This statement is equivalent to the following statements:

```
10 FOR I = 1 TO N: FOR J = 1 TO P
20 C(I,J) = 0
30 FOR K = 1 TO M
40 C(I,J) = C(I,J) + A(I,K) * B(K,J)
50 NEXT K,J,I
```

For example, if the following values are assumed for arrays A( ) and B( ):

$$A = \begin{bmatrix} 0 & 1 & 4 \\ 7 & 7 & 7 \end{bmatrix} \qquad B = \begin{bmatrix} 5 & 1 & 0 & 4 \\ 4 & 1 & 0 & 4 \\ 3 & 4 & 3 & 4 \end{bmatrix}$$

then the statement MAT C = A*B yields the following results in array C( ):

$$C = \begin{bmatrix} 16 & 17 & 12 & 20 \\ 84 & 42 & 21 & 84 \end{bmatrix}$$

If the rows and columns are not compatible, an error message is displayed. For example:

```
10 DIM A(2,2),B(4,4)
20 MAT C = A * B
           ↑ERR P45
```

Examples of Valid Syntax:

```
10 MAT G = E * F
20 MAT C = A * B
```

MAT PRINT

General Form:

MAT PRINT array-name [{,;}array-name]...[,;]

Purpose:

The MAT PRINT statement prints arrays in the order given in the statement. Each array is printed row by row. All the elements of a row are printed on as many lines as required. The first element of a row always starts at the beginning of a new print line. An array is printed in zoned format unless the array-name is followed by a semicolon, in which case elements are printed consecutively without additional intervening spaces. (See the discussion of the PRINT statement in Chapter 11, section 11.2.) For alphanumeric-arrays, the zone length is set equal to the maximum length defined for each array-element, which is not always 16 characters. A vector, which is a one-dimensional array, is printed as a column vector.

Example: The MAT PRINT Statement Used in a Program

This program accepts as input nine alphanumeric literal strings, which have a maximum length of 16 characters each, and prints them as a 3 x 3 array.

```
:10 DIM Z$ (3,3)
:20 MAT INPUT Z$
:30 MAT PRINT Z$;
:RUN
?1,2,3,4,5,6,7,8,9

 123
 456
 789
```

Examples of Valid Syntax:

```
10 MAT PRINT A;B,C$
20 MAT PRINT A,B$
30 MAT PRINT N,
40 MAT PRINT A$;
```

MAT READ

General Form:

MAT READ array-name [(dim1[,dim2]) [length]] [,. . .]

Where:

dim1, dim2 = expressions specifying new dimensions.
For one-dimensional arrays:
1 < = dim1 < 65536

For two-dimensional arrays:
1 < = dim1, dim2 < 256

length = expression specifying the maximum length of each alpha-array-element (1 <= length < 125). The default length of each element is 16.

Purpose:

The MAT READ statement is used to assign values contained in DATA statements to array-variables without referencing each element of the array individually. The MAT READ statement causes the referenced arrays to be filled sequentially with the values available from the DATA statement(s). Each array is filled row by row. Values are retrieved from a DATA statement in the order specified in the MAT READ statement. If a MAT READ statement requires more values than are available in the referenced DATA statement, the system searches for the next sequential DATA statement. If the program contains no further DATA statements, an error is signalled and execution is terminated.

Alphanumeric-arrays also can be specified in the MAT READ statement. The values specified in the DATA statement must be compatible with the array (i.e., numeric values must be specified for numeric-arrays and alphanumeric literal strings must be specified for alphanumeric-arrays).

The dimensions of the array(s) are those which were last specified in the program by a COM, DIM, or MAT statement unless new dimensions are specified. The maximum length for alphanumeric-array-elements can be specified by including the length after the dimension specification; if no length is specified, a default value of 16 bytes is used.

Example: The MAT READ Statement Used in a Program

```
:10 DIM N(2,3), A$(4,4)8
:20 MAT READ N,A$(3,3)2
:30 MAT PRINT N;A$,
:40 DATA 1,2,3,4,5,6
:50 DATA "A","B","C","D","E","F","G","H","I"
:RUN
 1   2   3
 4   5   6

ABC
DEF
GHI
```

Examples of Valid Syntax:

```
10 MAT READ N(5,4)
20 MAT READ A$(R,C)L
30 MAT READ A,B,Z$
```

MAT REDIM (MAT REDIMension)

General Form:

MAT REDIM array-name (dim1[,dim2])[length] [,. . .]

Where:

dim1, dim2 = expressions specifying new dimensions.
For one-dimensional arrays:
1 <= dim1 < 65536

For two-dimensional arrays:
1 <= dim1, dim2 < 256

length = expression specifying the maximum length of each alpha-array-element (1 <= length < 125). The default length of each element is 16.

Purpose:

The MAT REDIM statement redimensions the specified arrays. The new dimension(s) are enclosed in parentheses immediately following the array-name. The maximum length of each element in an alphanumeric-array can be specified by including the length following the dimension specification. If a maximum length is not specified, a default length of 16 is assumed. A vector (one-dimensional array) cannot be redimensioned as a matrix (two-dimensional array), and a matrix cannot be redimensioned as a vector. A redimensioned array cannot be larger than the originally defined array. Any attempt to produce a redimensioned array which is larger than the originally defined array yields error P59.

Example:

```
10 DIM A(3,3)
20 MAT INPUT A
30 PRINT "ORIGINAL ARRAY"
40 MAT PRINT A;
50 MAT REDIM A(2,2)
60 PRINT "REDIMENSIONED ARRAY"
70 MAT PRINT A;
```

If the following values are entered in response to line 20:

```
?9,8,7,6,5,4,3,2,1
```

then lines 40 and 70 produce the following output:

```
ORIGINAL ARRAY

9   8   7
6   5   4
3   2   1

REDIMENSIONED ARRAY

9   8
7   6
```

Examples of Valid Syntax:

```
10 MAT REDIM A(4,5)
20 MAT REDIM B$(20)8,C1$(3*X,4),D$(8)
```

MAT( )* (MAT scalar multiplication)

General Form:

MAT c = (k) * a

Where:

c and a are numeric-array-names and k is a numeric-expression.

Purpose:

This statement causes each element of array a to be multiplied by the value of expression k. The product is stored in array c, which can appear on both sides of the equation. Array c is redimensioned to the same dimensions as array a.

If array A( ) has N rows and M columns, the statement

```
10 MAT C = (F1↑2)*A
```

is equivalent to the following statements:

```
10 FOR I = 1 TO N: FOR J = 1 TO M
20 C(I,J) = (F1↑2) *A(I,J)
30 NEXT J,I
```

For example, if the expression (F1↑2)=5 and the array A( ) has the following contents:

$$A = \begin{bmatrix} 5 & 3 & 1 \\ 2 & 2 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

then the statement MAT C=(F1↑2)*A yields the following result in C:

$$C = \begin{bmatrix} 25 & 15 & 5 \\ 10 & 10 & 10 \\ 5 & 5 & 5 \end{bmatrix}$$

Examples of Valid Syntax:

```
10 MAT C = (SIN(X))*A
20 MAT D = (X+Y↑2)*A
30 MAT A = (5)*A
```

MAT – (MAT subtraction)

General Form:

MAT c = a – b

Where:

a, b, and c are numeric-array-names.

Purpose:

This statement subtracts array b from array a and stores the difference between each pair of elements in the corresponding element of array c. Array c can appear on both sides of the equation. An error message is displayed and execution is terminated if the dimensions of a and b are not the same. Array c is redimensioned to have the same dimensions as arrays a and b.

If arrays A( ) and B( ) have N rows and M columns, then the statement

```
10 MAT C = A – B
```

is equivalent to the following statements:

```
10 FOR I = 1 TO N: FOR J = 1 TO M
20 C(I,J) = A(I,J) – B(I,J)
30 NEXT J, I
```

For example, if arrays A( ) and B( ) have the following contents:

$$A = \begin{bmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \\ 3 & 3 & 3 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}$$

then the statement MAT C = A-B yields the following result in C( ):

$$C = \begin{bmatrix} 2 & 2 & 2 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Examples of Valid Syntax:

```
10 DIM A(6,3),B(6,3),C(6,3),D(4),E(4)
20 MAT C = A – B
30 MAT C = A – C
40 MAT D = D – E
```

MAT TRN (MAT transpose)

General Form:

MAT c = TRN(a)

Where:

a and c are numeric-array-names.

Purpose:

This statement causes array c to be replaced by the transpose of array a. Array c is redimensioned to the same dimensions as the transpose of array a. Array c cannot appear on both sides of the equation.

If array A( ) has N rows and M columns, then the statement

```
10 MAT C = TRN(A)
```

creates an array C( ) which has M rows and N columns. This statement is equivalent to the following statements:

```
10 FOR I = 1 TO M: FOR J = 1 TO N
20 C(I,J) = A(J,I)
30 NEXT J,I
```

For example, if A( ) is a 3 X 2 array with the following contents:

$$A = \begin{bmatrix} 9 & 8 \\ 6 & 5 \\ 3 & 2 \end{bmatrix}$$

then the statement MAT C = TRN(A) yields the following result in C( ):

$$C = \begin{bmatrix} 9 & 6 & 3 \\ 8 & 5 & 2 \end{bmatrix}$$

Example of Valid Syntax:

```
10 MAT C = TRN(A)
```

MAT ZER (MAT ZERo)

General Form:

MAT c = ZER [(dim1[,dim2])]

Where:

c is a numeric-array-name, and dim1 and dim2 are expressions specifying new dimensions.

For one-dimensional arrays:
1 <= dim1 < 65536

For two-dimensional arrays:
1 <= dim1, dim2 < 256

Purpose:

This statement sets all elements of the specified array equal to zero. If new dimensions (dim1, dim2) are specified, the array is redimensioned to the specified dimensions. The number of elements in the redimensioned array must be less than or equal to the number of elements in the array when originally dimensioned.

For an array A( ) with N columns and M rows, the statement

```
10 MAT A = ZER
```

is equivalent to the following statements:

```
10 FOR I = 1 TO N: FOR J = 1 TO M
20 A(I,J) = 0
30 NEXT J,I
```

Examples of Valid Syntax:

```
10 MAT C = ZER(5,2)
20 MAT B = ZER
30 MAT A = ZER(F,T+2)
40 MAT D = ZER(20)
```

CHAPTER 14 SORT STATEMENTS

14.1 INTRODUCTION

The BASIC-2 language provides a unique built-in sorting capability with the MAT SORT, MAT MERGE, and MAT MOVE statements. The functions performed by these statements are summarized below:

- MAT SORT — used to sort data in an alpha-array. MAT SORT scans the "sort-array," which contains the data, and produces the "locator-array," which contains the subscripts of the elements of the sort-array in order of ascending data value.
- MAT MERGE — used to merge data from two or more sorted input files into a sorted output file. MAT MERGE first scans the "merge-array" in which each row contains data from one of the input files and then creates the "locator-array," which contains the subscripts of merge-array elements in order of ascending data value.
- MAT MOVE — used to move data from one array (the "move-array") to a second array (the "receiver-array"). The order in which elements are moved can be specified by the subscripts in a locator-array, which is created by either MAT SORT or MAT MERGE. The MAT MOVE statement creates a sorted output array from an input array based on the locator-array created by MAT SORT or MAT MERGE. Optionally, MAT MOVE also converts numeric data to sort format or restores data from sort format to numeric format.

Neither the MAT SORT nor the MAT MERGE statement causes any rearrangement of data in the original array. Instead, each statement produces as its output a locator-array which contains the subscripts of elements in the data array arranged in order of ascending value. The MAT MOVE statement is used to move data from the original data array to an output data array. The order in which the array subscripts appear in the locator-array determines the order in which data is moved. When this operation is completed, the output array contains the data from the original array, arranged in sorted sequence.

This process is portrayed graphically in Figure 14-1 below. Imagine that the input array is a one-dimensional array of five elements, with each element containing a single letter:

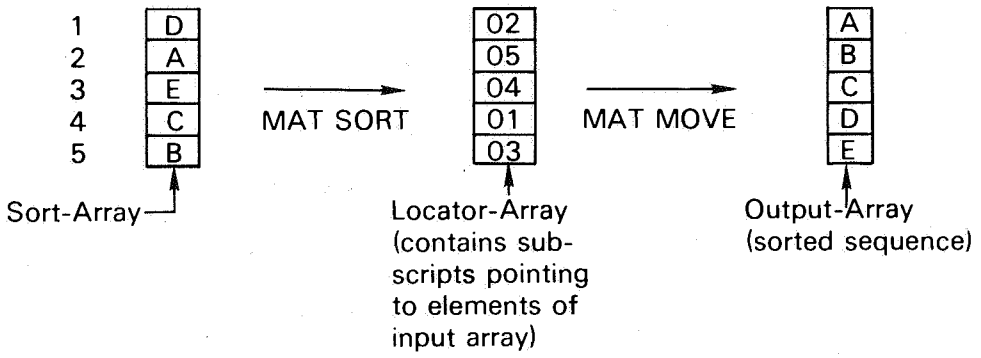


Figure 14-1.
Simplified Sort Sequence

The purpose of the MAT MERGE statement is to merge two or more sorted files into a single large sorted file. MAT MERGE is required when the input file to be sorted is larger than the available memory. In this case, the file cannot be completely sorted in one pass; it must be sorted in segments, and the segments must then be merged into a single output file. Typically, the segments are saved on disk in several temporary files. When the entire original file has been sorted in this manner, the result is a number of sorted "subfiles" which must be merged into a single output file. In a merge, data from each sorted input file is read into an array called the "merge-array." Each row of the merge-array serves as a "merge buffer" for one of the input files. MAT MERGE then scans the merge-array and produces an array of subscripts (the locator-array) which points to elements of the merge-array in sorted sequence. Using the MAT MOVE statement, the merged data must then be moved from the merge-array to an output array and saved in the output file on disk. This process continues until all data from all input files has been merged into the output file.

The merging process is illustrated in highly simplified form in Figure 14-2:

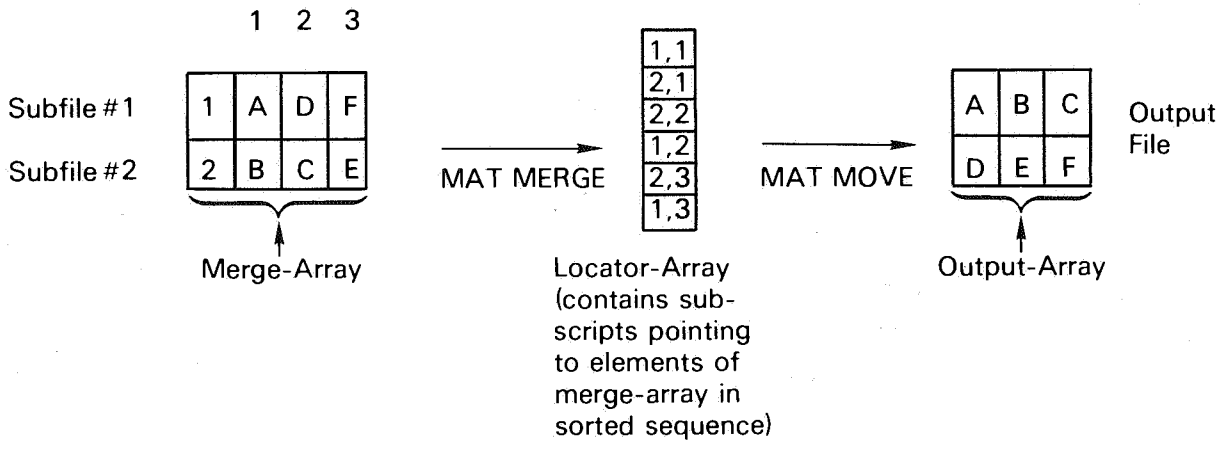


Figure 14-2.
Simplified Merge Sequence

Sorting Numeric Data

Numeric data must be converted from internal numeric format to sort format before it can be sorted with MAT SORT or MAT MERGE. Conversely, after it has been sorted, the numeric data must be restored to internal numeric format before any computation can be performed with it. If the input array and output array are different types of arrays, MAT MOVE automatically converts data from numeric to sort format and restores data from sort to numeric format when the data is moved. Numeric data in sort format is stored as alphanumeric data; thus, if the input array (called the move-array) in a MAT MOVE statement is numeric and the output array (called the receiver-array) is alphanumeric, the numeric data is automatically converted to sort format as it is transferred into the alphanumeric-array. This operation must be performed prior to sorting the numeric data. Conversely, if the input array is alphanumeric and the output array numeric, the data is automatically restored from sort format to internal numeric format as it is transferred to the numeric-array. This operation normally is performed after a file of numeric data has been sorted. If both the input array and output array are the same type (i.e., both numeric or both alphanumeric), MAT MOVE simply carries out the data transfer and no conversion is performed.

Representation of Array Subscripts in the Locator-Array

The subscripts stored in the locator-array by MAT SORT and MAT MERGE (and used by MAT MOVE) are two-byte binary values which identify elements in the input array. In BASIC-2, array subscripts are always represented in binary as two-byte values. Their meanings are different, however, depending upon whether the array is one-dimensional or two-dimensional.

A one-dimensional array has one column with one or more rows (see Chapter 2, section 2.3). Thus, the subscript for each element is a two-byte binary value identifying the row occupied by that element. For example, the statement DIM N(5) dimensions a one-dimensional array with five elements. This array consists of one column with five rows (one element per row). The subscript for element N(1) would be expressed in hex notation as 0001. The subscript for N(5) would be 0005. Since the maximum value of a two-byte binary number is 65,535 in decimal notation, a one-dimensional array can theoretically have up to 65,535 elements.

A two-dimensional array has one or more rows and one or more columns. For example, the statement DIM N(5,3) dimensions a two-dimensional array with five rows and three columns. In this case, the two-byte binary subscript is divided into two separate values: the first byte identifies the row, and the second byte identifies the column in which the element is located. Thus, the subscript for element N(1,1) would be 0101 (hex) and the subscript for element N(4,3) would be 0403. Because the maximum value of a one-byte binary number is 255 in decimal notation, a two-dimensional array can have up to 255 rows and 255 columns.

14.2 GENERAL FORMS OF THE SORT STATEMENTS

General forms of the sort statements MAT MERGE, MAT MOVE, and MAT SORT are shown on the following pages.

MAT MERGE

General Form:

MAT MERGE merge-array[(x[,y])] TO control-var, work-var, loc-array

Where:

merge-array = a two-dimensional alpha-array. x,y optionally define a field within each element of the merge-array:

- x = an expression whose truncated value specifies the starting position of the field within each element.
- y = an expression whose truncated value specifies the length of the field in bytes. If this expression is omitted, the field is assumed to occupy the remainder of the element.

control-var = the control-variable, an alpha-variable used to store merge status information.

work-var = the work-variable, an alpha-variable used by the system as work space.

loc-array = the locator-array, an alpha-array with elements of length two used to store subscripts.

Purpose:

The MAT MERGE statement performs the merge operation required to combine two or more sorted input files into a single sorted output file. MAT MERGE operates on four different alphanumeric-variables:

1. The *merge-array* is a two-dimensional alphanumeric-array containing data to be merged. The number of rows in the merge-array must equal the number of input files to be merged. Each row in the merge-array serves as the "merge-buffer" for an input file. The number of columns (1 to 254) in the merge-array (i.e., the number of elements per row) is dictated by available memory space.

2. The *control-variable* is used to store information on the status of the merge following each execution of MAT MERGE. (Typically, a merge requires several executions of MAT MERGE.) The control-variable is an alpha-variable with at least one more byte than there are rows in the merge-array. If the merge-array has n rows, the control-variable must have at least n+1 bytes.

3. The *work-variable* is used as work space by the system when performing a MAT MERGE and assumes no role in the operation of the application program. It is an alpha-variable with at least twice as many bytes as the merge-array has rows.

4. The *locator-array* is used by MAT MERGE to store the subscripts of elements in the merge-array. This array represents the output of a MAT MERGE operation. It must be an alpha-array whose elements are two bytes in length. In general, the larger the size of the locator-array, the more efficient the merge. The locator-array should have at least as many elements as there are columns in the merge-array.

Each row of the merge-array serves as a merge-buffer for one of the input files. Data must be loaded into these rows from the input files by the application program. After scanning and comparing the data in the merge-array, MAT MERGE stores the subscripts of merge-array elements in the locator-array in order of increasing data value. Thus, the first subscript placed in the locator-array is that of the lowest valued element in the merge-array; the next subscript is that of the next-higher data value, etc. If the merge terminates before the locator-array is filled with subscripts, MAT MERGE places a HEX(0000) code in the locator-array element immediately following the last subscript. Note that MAT MERGE does not actually move any data in the merge-array itself during this process.

The merge can be performed on the basis of a defined field within each element rather than the entire element if the field expressions are specified following the merge-array-designator. The first field expression identifies the starting location of the field to be used in each element. The second field expression specifies the field length in bytes. If the second expression is omitted, the field is assumed to extend to the end of the element. For example, the following statement

```
100 MAT MERGE A$() (5,12) TO B$(), C$(), D$()
```

causes the merge comparison to be made on a 12-byte field in each element of A$(), starting at the fifth byte of the element.

The process of scanning the merge-array and storing subscripts in the locator-array terminates when one of the following conditions is met:

1. the locator-array is filled with subscripts, or
2. a row of the merge-array has been completely "emptied" (i.e., all elements in that row have been merged).

The application program must now process the merged data and, if necessary, replenish the empty row in the merge-array. Processing the merged data normally involves executing MAT MOVE to move the merged data elements into an output buffer. Replenishing a row of the merge-array usually involves DATALOADing data from the appropriate input file to an input buffer and copying the data from the input buffer to the empty merge-array row with a LET statement. (The reason for using an input buffer as an intermediate step is explained in the following section.)

The Role of the Control-Variable

A merge operation typically requires a series of passes with MAT MERGE executed once on each pass. Following each execution of MAT MERGE, the application program must determine why the MAT MERGE operation terminated and then take appropriate action. The status of the merge following each execution of MAT MERGE is determined by examining the control-variable.

If the merge-array has n rows, then the control-variable has n+1 bytes. The first n bytes of the control-variable contain pointers to elements in the corresponding rows of the merge-array. The n+1st byte of the control-variable contains a status code which is set by MAT MERGE following statement execution. This code can have the following values:

Code	Meaning
Hex 00	Locator-array full
Hex 01 — Hex FF	Row number of empty row in merge-array

When MAT MERGE terminates, each of the first n bytes of the control-variable contains a pointer to the next element to be scanned in the corresponding row of the merge-array on the next execution of MAT MERGE. Initially, the merge scan should begin with the first element in each row. The application program must therefore initialize the first n bytes of the control-variable to HEX(01) prior to beginning the merge. (See Figure 14-3 which appears below.) This initialization procedure is easily performed with the ALL function. Since the n+1st byte of the control-variable is used for the termination status code, its initial value is not important.

Merge-Array

1	A	B	Q	T	X
2	C	D	E	F	U
Rows 3	G	H	I	J	K
4	L	M	R	V	Y
5	N	O	P	S	W

Control-Variable

1	01	Pointer to First Element in Merge-Array Row One
2	01	.
3	01	.
4	01	Pointer to First Element in Merge-Array Row Four
5	01	.
6	01	Status Code (Initial Value Not Important)

Figure 14-3.
Control-Variable Prior to Beginning MAT MERGE

On each subsequent execution of MAT MERGE, the pointers in the control-variable are automatically updated to point to the next element to be scanned in each row. Whenever MAT MERGE terminates, therefore, these pointers indicate the position of the next element to be scanned in each row at the start of the next execution of MAT MERGE.

An exception to this procedure occurs when MAT MERGE terminates with one of the merge-array rows empty. In that case, the corresponding element of the control-variable receives a HEX(FF) code, and the status code (n+1st byte) is set to point to the empty row. This condition is illustrated below in Figure 14-4:

Merge-Array

1	A	B	**Q**	T	X
2	C	D	E	F	**U**
Rows 3	G	H	I	J	K
4	L	M	**R**	V	Y
5	N	O	P	**S**	W

Control-Variable

1	03	Points to Next Element to Be Scanned in Row One
2	05	.
3	FF	Indicates Empty Row
4	03	.
5	04	.
6	03	Status Code Contains Row Number of Empty Row

Figure 14-4.
Control-Variable Following Termination
of MAT MERGE Due to Empty Row

There are, therefore, two distinct courses of action open to the application program following each execution of MAT MERGE:

(1) If the n+1st byte of the control-variable = HEX(00) following MAT MERGE execution, statement execution terminated because the locator-array was filled. The program must MAT MOVE the merged data from the merge-array to an output buffer using the locator-array and, if the output buffer is full, store it in the output file. Then MAT MERGE must be reexecuted to resume merging the unmerged data in the merge-array.

(2) If the n+1st byte of the control-variable = HEX(01) to HEX(FF) following MAT MERGE execution, MAT MERGE terminated with an empty row. The program must MAT MOVE the merged data to an output buffer as in case (1) above. Before reexecuting MAT MERGE, however, the program must replenish the empty row in the merge-array (if there is more data) and reset the corresponding pointer in the control-variable.

> **NOTE:**
>
> MAT MERGE reuses the locator-array from the beginning each time it is executed, destroying the subscripts stored there by the previous MAT MERGE operation. Data merged on one pass, therefore, must always be MAT MOVEd into an output buffer prior to reexecuting MAT MERGE on the next pass.

Replenishing Empty Rows in the Merge-Array

In order to replenish an empty row in the merge-array, the application program must first load data from the appropriate input file into the empty row and then reset the pointer in the corresponding control-variable element to point to the first element to be scanned in the replenished row.

In some cases it is possible to simply DATALOAD from an input file directly into its row in the merge-array whenever that row becomes empty. In general, however, this technique cannot be used because it cannot accommodate the situation in which insufficient data remains in the file to completely fill the row. If the row cannot be filled, the data must be right-justified within the row by using a LET STR(...) = STR(...) statement. The recommended procedure is therefore to first DATALOAD from the input file to an input buffer and then transfer the data from the input buffer to the appropriate row of the merge-array, right-justified.

When a row of the merge-array is emptied, MAT MERGE places a HEX(FF) in the corresponding element of the control-variable. When the row is refilled, the application program must replace the HEX(FF) code with a pointer to the first element in the refilled row. If the row is completely refilled, the pointer is reset to HEX(01). If the row cannot be completely filled with data, however, the data must be right-justified and the pointer must be set to point to the first valid data element in the row. For example, if the merge-array has ten elements per row, but only five values remain to be read from the input file, these five values occupy elements 6-10, right-justified in the row. Elements 1-5 of this row contain invalid data and should not participate in the next merge pass. The pointer to this row in the control-variable must therefore be set to HEX(06), instructing the next scan to begin with the sixth element in the row.

When there is no more data remaining in an input file to replenish its row, the appropriate pointer in the control-variable is left as HEX(FF). A HEX(FF) code instructs MAT MERGE to ignore the corresponding row on the next merge pass.

When MAT MERGE discovers that all elements of the control-variable are set to HEX(FF), it places a two-byte HEX(0000) code in the first element of the locator-array and terminates. The merge operation is then complete.

Example 1: A Three-File Merge

In the following example, data from three sorted input files, INPUT1, INPUT2, and INPUT3, is read into a merge-array and merged using MAT MERGE. The following arrays are used by MAT MERGE:

M$( ) — merge-array
C$( ) — control-variable
W$( ) — work-variable
L$( ) — locator-array

The statement used to dimension these arrays is shown below:

```
10 DIM M$(3,6)1,C$(4)1,W$(3)2,L$(12)2
```

Note that the merge-array M$( ) has three rows (since there are three input files) and six columns. Each element is one byte in length to simplify the example. The control-variable C$( ) has one more element than M$( ) has rows, and each element in the control-variable is one byte in length. The work-variable W$( ), however, has exactly as many elements as M$( ) has rows, and each element is two bytes in length. The locator-array L$( ) has 12 elements, each two bytes in length.

Before the merge can begin, the control-variable must be initialized to point to the first element in each row of the merge-array:

```
50 C$( ) = ALL(01)
```

The initial contents of C$( ) appear below in Figure 14-5:

	C$( )	
1	01	← Pointer to Element 1 in Row 1 of Merge-Array
2	01	← Pointer to Element 1 in Row 2 of Merge-Array
3	01	← Pointer to Element 1 in Row 3 of Merge-Array
4	01	← Status Code

Figure 14-5.
Initial Value of Control-Variable C$( )

For simplicity, the example in Figure 14-6 assumes that the data to be sorted are the characters of the alphabet. After data from each file has been DATALOADed into an input buffer and transferred into the merge-array, M$( ) has the following contents:

Columns

		1	2	3	4	5	6	
	1	A	D	H	I	L	P	← Merge-Buffer for INPUT1
Rows	2	B	C	F	J	M	Q	← Merge-Buffer for INPUT2
	3	E	G	K	N	O	R	← Merge-Buffer for INPUT3

Figure 14-6.
Merge-Array M$( )

At this point, the merge can be executed:

```
60 MAT MERGE M$() TO C$(), W$(), L$()
```

MAT MERGE scans M$() and places the subscripts of merge-array elements into the locator-array L$() in order of ascending data value. Thus, the subscript of "A" is the first one stored in L$(), and the subscript of "B" is the second one stored, etc. MAT MERGE also rewrites the first three elements of the control-variable C$() to point to the next element in each row of M$() to be scanned on the next merge pass. Since the merge terminates when the locator-array is full, the status code in the fourth element of C$() receives a HEX(00) code. The contents of C$() and L$() following execution of statement 60 are shown in Figure 14-7 below:

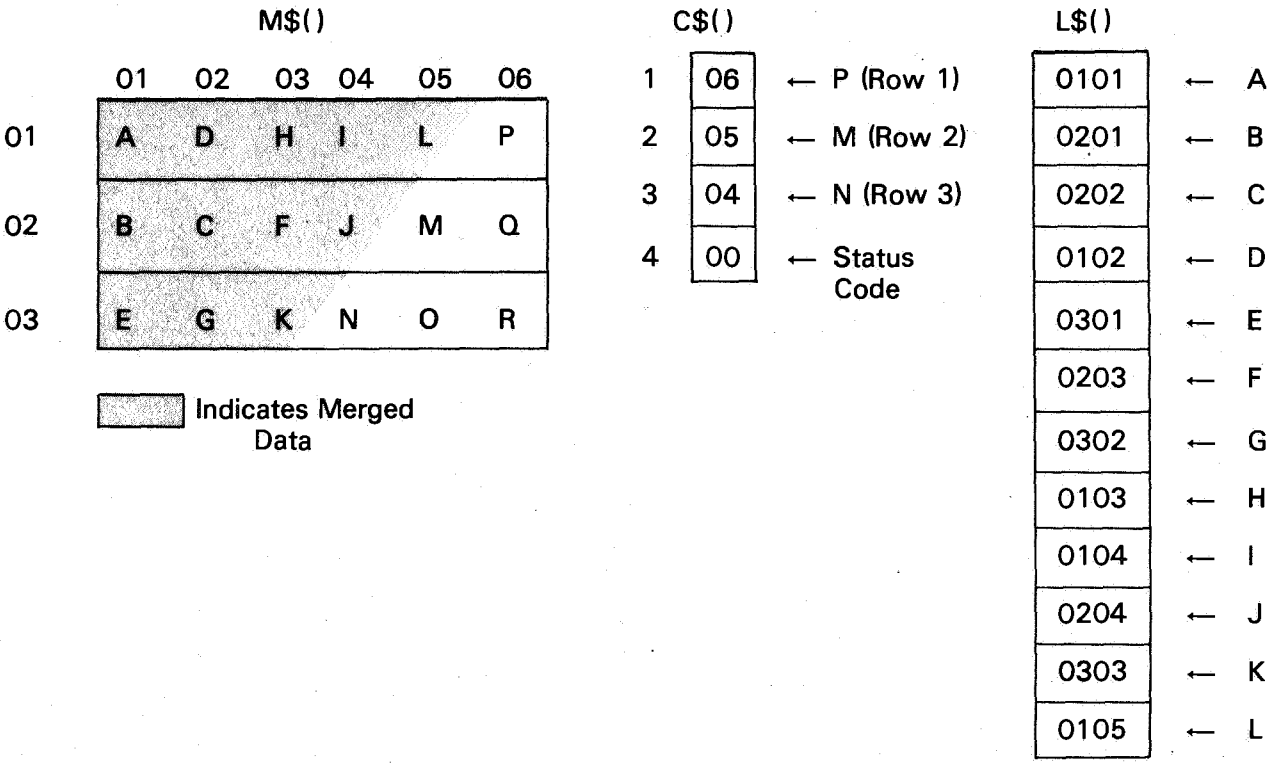


Figure 14-7.
Contents of Control-Variable C$() and
Locator-Array L$() Following MAT MERGE

A complete program which performs the three-file merge described in this example on the input files INPUT1, INPUT2, and INPUT3 is shown in Figure 14-8. To represent the typical situation more closely, the length of each sort value in this program is eight bytes rather than one. Every record in the three input files contains 50 values which are each eight bytes in length. The merged data is stored in an output file called OUTPUT.

Suppose there are three sorted data files on disk: INPUT 1, INPUT 2, and INPUT 3, each with 50 elements of length 8 / logical record. The following program merges these files into a fourth file called OUTPUT:

```
10 REM ARRAY DEFINITIONS .....
20 DIM M$(3,50)8:REM MERGE-ARRAY
30 DIM I$(50)8:    REM INPUT BUFFER
40 DIM O$(50)8:    REM OUTPUT BUFFER
50 DIM C$(4)1:     REM CONTROL-VARIABLE
60 DIM W$(3)2:     REM WORK-VARIABLE
70 DIM L$(50)2:    REM LOCATOR-ARRAY
80 REM OPEN THE DATA FILES ON DISK.....
90 SELECT #1/310, #2/310, #3/310, #4/310
100 DATA LOAD DC OPEN T #1, "INPUT1"
110 DATA LOAD DC OPEN T #2, "INPUT2"
120 DATA LOAD DC OPEN T #3, "INPUT3"
130 DATA LOAD DC OPEN T #4, "OUTPUT"
140 REM FILL THE MERGE-ARRAY.....
150 FOR I = 1 TO 3
160 GOSUB '40(I)
170 NEXT I
200 M=1
210 REM MERGE .....
220 MAT MERGE M$() TO C$(), W$(), L$()
230 IF L$(1)=HEX(0000) THEN 500: REM EXIT IF DONE
240 REM MOVE THE MERGED DATA TO THE OUTPUT BUFFER.....
250 S=1
260 N=50
270 MAT MOVE M$(), L$(S), N TO O$(M)
280 M=M+N: REM M = NO. OF ELEMENTS IN OUTPUT BUFFER
290 REM PUT DATA INTO OUTPUT FILE IF OUTPUT BUFFER FULL.....
300 IF M< =50 THEN 350
310 DATA SAVE DC #4, O$()
320 M=1
330 S=N+1
340 IF S< 51 THEN 260: REM BRANCH IF MORE DATA TO MOVE
350 REM CHECK MERGE TERMINATION FLAG .....
360 T=VAL(C$(4))
370 IF T=0 THEN 220: REM BRANCH IF NO ROWS OF M$() EMPTY
380 GOSUB '40(T): REM REFILL EMPTY ROW OF M$()
390 GOTO 220
400 REM .....
410 DEFFN'40(R): REM    READ THE NEXT BLOCK FROM SPECIFIED
420 REM                 INPUT FILE AND PUT INTO MERGE-ARRAY
430 DATA LOAD DC #R, I$()
440 IF END THEN 480
450 STR(M$(C),(R-1)*400+1,400) = I$()
460 C$(R)=HEX(01): REM RESET APPROPRIATE CONTROL-VARIABLE ELEMENT
470 RETURN
480 C$(R) = HEX(FF): REM ROW EMPTY
490 RETURN
500 END
```

Figure 14-8.
Program and Flow Diagram for Three-File MAT MERGE

Examples of Valid Syntax:

```
10 MAT MERGE A$() TO C$(), W1$(), L$()
20 MAT MERGE B$() (1,5) TO C$,W$,N$()
30 MAT MERGE M$() TO X$(), Y$,Z$()
```

MAT MOVE

General Form:

MAT MOVE move-array [,locator-array] [,n] TO receiver-array

Where:

move-array = { move-alpha-array-desig [(x[,y])] | move-numeric-array-desig }

locator-array = { locator-array-desig | locator-array-element }

receiver-array = { receiver-alpha-array-element [(x[,y])] | receiver-numeric-array-element | receiver-numeric-array-desig | receiver-alpha-array-desig [(x[,y])] }

n = a numeric-scalar-variable representing the maximum number of elements to be moved.

(x,y) = optional field-designators defining a field within each alpha-array-element such that
x = expression specifying the starting position of the field.
y = expression specifying the number of characters in the field (assumes remainder of element if not specified).

Purpose:

The MAT MOVE statement is used to transfer data element-by-element from one array to another. In its complex form, MAT MOVE also can be used to automatically convert numeric data in Wang internal format to alphanumeric data in sort format and conversely, to restore alpha data in sort format to numeric data in Wang internal format. Conversion to or from sort format is performed automatically if the move-array and receiver-array are different types of arrays. If both the move-array and the receiver-array are the same type, the data is simply moved and conversion does not occur. (See chart below.)

Move-Array	Receiver-Array	Implied Function
Alpha	Alpha	Data transfer only.
Alpha	Numeric	Data transfer, restore data to internal numeric format.
Numeric	Alpha	Data transfer, convert data to sort format.
Numeric	Numeric	Data transfer only.

Note that MAT MOVE does not perform a validity check on alpha data which is being restored to Wang internal numeric format. Care must be taken to insure that system errors do not result from improperly generated numeric data.

If no locator-array is specified, data is transferred sequentially from the move-array, starting with the first element, into sequential elements of the receiver-array, beginning with the specified receiver-array-element. The elements of the receiver-array are filled with data row by row. Specifying a receiver-array-designator (e.g., R$( )) indicates that data is to be transferred into the receiver-array starting at the first element. If either the move-array, the receiver-array, or both are alphanumeric, data may be moved to and from designated fields of each element.

A locator-array may be specified in addition to the move-array and receiver-array. The locator-array contains a series of two-byte array subscripts which point to elements in the move-array. (Typically, the locator-array is created by a MAT SORT statement and contains the move-array subscripts in sorted sequence.) If a locator-array is specified in the MAT MOVE statement, data is transferred from elements of the move-array to the receiver-array in the order in which the move-array subscripts appear in the locator-array. Subscripts are taken from the locator-array starting at the specified element; if a locator-array-designator is used, the starting element is assumed to be the first.

MAT MOVE continues transferring data from the move-array to the receiver-array until one of the following conditions is met:

1. The end of the move-array is reached and no locator-array is specified.
2. The end of the array of subscripts (locator-array) is reached.
3. A binary "0000" is found in the locator-array (see the discussion of MAT SORT which follows this section).
4. The number of elements specified by the counter-variable has been moved.
5. The receiver-array has been filled.

When MAT MOVE has finished data transfer, a count of the number of elements moved is returned to the counter-variable, if it is specified.

Table 14-1 which follows outlines the dimensional requirements for the various arrays involved in a MAT MOVE statement.

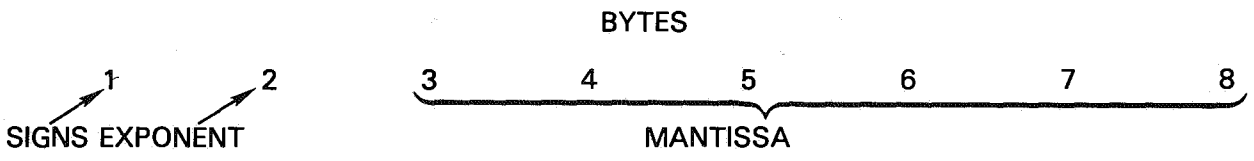
Table 14-1.
Dimensional Requirements

Move-Array	-	any one-or two-dimensional, alpha- or numeric-array.
Locator-Array	-	an alpha-array with elements of length two.
Receiver-Array	-	any one-or two-dimensional, alpha- or numeric-array.

Each subscript in the locator-array consists of two bytes. If the move-array is two-dimensional, the first byte of the subscript specifies the row subscript, and the second byte specifies the column subscript. If the move-array is one-dimensional, the subscript of each element is expressed as a two-byte binary number.

Numeric to Alpha Conversion: Wang Sort Format

When the MAT MOVE statement is used to convert numeric data to alphanumeric data, the data must first be converted to Wang Sort Format. Wang Internal Format does not facilitate sorting of data; therefore, format conversion is automatically accomplished before individual elements are MOVEd and SORTed. Wang Sort Format organizes a number within the receiver-array in the following manner:



The first four bits of the alpha-array-element are used to represent the signs of the mantissa and exponent. Table 14-2 below summarizes the decimal value of these bits for each case of signing.

Table 14-2.
Values of Sign Bits and Their Meanings

Value (Decimal)	Meaning
9	mantissa and exponent both positive
8	mantissa positive and exponent negative
1	mantissa and exponent both negative
0	mantissa negative and exponent positive

The next eight bits are used to represent the high-and low-order digits of the exponent. These digits are given in either decimal or decimal complement form, depending upon the signs of the value. Table 14-3 below summarizes when the decimal and decimal complement forms are used to express the digits of the exponent.

Table 14-3.
Decimal and Decimal Complement Forms

Form	Condition for Use
Decimal	mantissa and exponent both positive
Complemented	mantissa positive and exponent negative
Decimal	mantissa and exponent both negative
Complemented	mantissa negative and exponent positive

Table 14-3 illustrates that the exponential digits are given as their decimal complements if the signs of the mantissa and exponent differ.

The remaining bytes of the alpha-array-element are used to specify the digits of the mantissa. These digits are given in decimal form if the sign of the mantissa is positive or in decimal complement form if the sign of the mantissa is negative. BASIC-2 allows a maximum of 13 digits to be specified in the mantissa. A full 13-digit number requires eight bytes when stored in sort format. If, however, the elements of the receiver-alpha-array specified in a MAT MOVE statement are other than eight bytes in length, the number is truncated or padded with spaces as required. The length must be at least two bytes since the second byte contains the most significant digit of the number.

Example: Using a Locator-Array in a MAT MOVE Statement To Create a Sorted Output Array

Assume a two-dimensional array of data D$( ) with the values shown below and an associated locator-array S$( ) created in a MAT SORT operation. The data must be moved from D$( ) to a receiver-array E$( ) in the order specified by S$( ), creating an output array E$( ) which contains the data from D$( ) in sorted order.

D$() =

	1	2	3	4
1	B	A	C	D
2	C	B	E	A
3	A	F	A	E

S$() =

0102	0303	0301	0204
0202	0101	0103	0201
0104	0203	0304	0302

Figure 14-9.
Contents of Array D$() and the Locator-Array S$()

Execution of the following statements:

```
100 DIM E$(3,4)1
110 MAT MOVE D$(), S$(1) TO E$(1)
```

produces an array E$() with the values shown below:

E$() =

A	A	A	A
B	B	C	C
D	E	E	F

Examples of Valid Syntax:

```
10 MAT MOVE A$() TO B$(1,1)
20 MAT MOVE A$() (5,3), L$(1) TO B(1)
30 MAT MOVE A(), L$(3,1), X TO B$(Y)
40 MAT MOVE A(), W TO B(1,5) (2,5)
50 MAT MOVE A$() TO N()
60 MAT MOVE A$(), N TO B$()
```

MAT SORT

General Form:

MAT SORT sort-array TO work-variable, locator-array

Where:

sort-array =	sort-array-desig[(x[,y])]
sort-array-desig =	an alpha-array-designator (e.g., A$( )) containing the data to be sorted.
(x,y) =	optional field-designators which define a field within each element of the sort-array; x and y are expressions such that: 1. The truncated value of x specifies the starting position of the field. 2. The truncated value of y specifies the number of characters in the field (remainder of element assumed if y is not specified).
work-variable =	an alpha-variable used by the system as a temporary storage area.
locator-array =	an alpha-array with elements of length 2 used to contain subscripts of elements in the sort-array in sorted sequence.

Purpose:

The MAT SORT statement creates a locator-array containing subscripts arranged according to the ascending order of data values in the sort-array. An entire element or a defined field within each element may be used as the sort value. Once the locator-array has been created by MAT SORT, it can be used to create a sorted output array with MAT MOVE, which will move elements from the sort-array to a specified receiver-array in the order specified by the locator-array.

All subscripts in the locator-array are expressed as two-byte binary values. If the sort-array is two-dimensional, the first byte is the row subscript and the second byte is the column subscript. If the sort-array is one-dimensional, each element is represented as a two-byte binary value. (See the subsection entitled "Representation of Array Subscripts in the Locator-Array" at the conclusion of section 14.1.)

A work-variable must be defined by the programmer for the use of the system in performing the sort. This work-variable is not used by the BASIC-2 program.

Table 14-4 details the dimensional requirements for the variables used in a MAT SORT statement.

Table 14-4.
Dimensional Requirements

Work-Variable	—	must have at least twice as many bytes as there are elements in the sort-array.
Locator-Array	—	must have at least as many elements as the sort-array, and each element must be two bytes in length. If the locator-array has more elements than the sort-array, the first element of the locator-array which does not receive a subscript from the sort-array receives a two-byte value of binary zero [HEX(0000)]. The remaining elements of the locator-array are unchanged.

If the sort-array contains the same value more than once, the duplicate values may not occur in the same order as in the sort-array when they are sorted.

Examples: Using the MAT SORT Statement To Sort Data in an Alpha-Array

Assume that S$( ) is the sort-array. Array S$( ) is a matrix of 12 elements containing the data values shown below:

Columns

Rows	1	2	3	4
1	B	A	C	G
2	E	L	C	F
3	B	F	H	O

Figure 14-10.
Contents of Sort-Array S$( )

The sorting procedure is as follows:

1. First, a work-variable W$ and a locator-array L$( ) must be dimensioned:

   100 DIM W$24, L$(3,4)2

   Note that the work-variable contains twice as many bytes (24) as there are elements in the sort-array (12). The locator-array contains the same number of elements as the sort-array, with each element two bytes in length.

2. Next, a MAT SORT statement is executed:

   110 MAT SORT S$( ) TO W$, L$( )

   Execution of this statement causes the locator-array L$( ) to be filled with subscripts specifying the order of the sorted values from S$( ). The resultant locator-array L$( ) is shown below.

		Columns			
		1	2	3	4
Rows	1	0102	0101	0301	0203
	2	0103	0201	0302	0204
	3	0104	0303	0202	0304

Figure 14-11
Locator-Array L$( ) Following the MAT SORT Operation

The subscripts in L$( ) point to elements in S$( ) in ascending order. For example, the lowest value in S$( ) is the character "A," stored in element S$(1,2). Its subscript (1,2) is stored in the first element of the locator-array as a two-byte binary value. The first byte of this value identifies the row in which the data value is located, and the second byte identifies the column. In hexadecimal notation, this subscript is 0102. The next-higher value, the character "B," occurs twice in S$( ) in elements (1,1) and (3,1). These subscripts (HEX(0101) and (0301)) are stored in the next two elements of L$( ). The sort proceeds in this way, with the order of the subscripts in the locator-array reflecting the ascending order of sort values in the sort-array.

Examples of Valid Syntax:

```
10 MAT SORT A$( ) TO W$, S$( )
20 MAT SORT G$( ) (6,10) TO F$, L1$( )
30 MAT SORT B$( ) (5,3) TO W$, B1$( )
```

CHAPTER 15 GENERAL I/O STATEMENTS

15.1 INTRODUCTION

BASIC-2 provides the user with unique control over and access to I/O devices. The $IF ON/OFF statement provides the user with the means to determine when a given peripheral device is ready to supply data or be sent data. The user's program may be designed to perform another task if the device is not in the desired state, thus saving time. The $GIO statement allows the user to control directly the sequence of signals sent to a peripheral device, allowing devices not supported by other BASIC-2 I/O statements to be used. These two I/O statements provide the BASIC-2 programmer with increased flexibility and permit more efficient programming.

15.2 CONSIDERATIONS FOR THE USE OF $GIO

There are certain circumstances in which Wang-supplied I/O routines are not sufficient to satisfy user I/O requirements. For example, a user may wish to attach a non-Wang peripheral device to a 2200VP/MVP via an interface controller board such as the Model 2250. In this case, no I/O routine exists for the device and it is necessary for the user to write one.

The BASIC-2 statement $GIO enables the user to write such routines using a series of microcommands which control the electronic signals on the I/O bus. Various microcommand sequences can be written to read data from devices and perform other I/O functions. Data transfer rates up to 100,000 characters per second can be achieved in a user-written $GIO routine.

Certain microcommands may cause unforeseen errors due to the internal structure of Wang peripherals. Because this structure is beyond the scope of this manual, the user should not write $GIO routines for Wang-supported devices. For several such devices (e.g., disk, nine-track tape), there exist Wang-written $GIO microcommand sequences. When these microcommand sequences are available for a device, they are provided in the appropriate reference manual and must be used rather than user-written routines.

The programmer should also be familiar with the I/O bus, which is discussed in the next section of this chapter, as well as BASIC-2 programming before attempting to write a $GIO routine. In general, $GIO should be used only when necessary. The programmer should also verify carefully the accuracy of his routines to avoid unrecoverable errors.

15.3 THE BUS, SIMPLIFIED

The I/O bus is the path along which data travels between the CPU and the peripherals. Although data can travel in either direction on the I/O bus, only one peripheral participates in data transfer since only one device is enabled at any moment. Peripheral devices are enabled using an *Address Bus Strobe* (ABS), which transmits the address of the desired peripheral along the bus. The controller whose address corresponds to the one which is sent is enabled; all others are disabled at that time. Another ABS can be used to change the enabled device at some other point during program execution.

Data is output from the CPU to a device selected with an ABS via one of two methods, an *Output Bus Strobe* (OBS) or a *Control Bus Strobe* (CBS). When executing a $GIO routine to output data, an OBS should be used unless the programmer is certain that a CBS is necessary. While both strobes carry data from the CPU to a device, most controllers interpret them differently. Thus, the Output Bus Strobe and the Control Bus Strobe are not generally interchangeable, and the programmer must be familiar with the particular controller before using a CBS.

Before the CPU sends a byte, the device must be in a state where data reception is possible. When a device is *ready,* the controller puts this information on the bus and a transfer of data can take place. A *WR* (Wait for Ready) at the beginning of a microcommand assures the programmer that the device is ready before an attempt to send data is made. It is essential that the WR command be used unless the programmer is certain that communication with the device can proceed without this precaution. By not waiting for device ready, it is possible to send a byte over the bus while the device is not able to receive it; the byte is then lost. On the MVP, the WR command must be used since methods of communication with peripherals that function on the VP fail on a multiuser system. A programmer using the MVP must understand the constraints which problems such as contention for shared devices and breakpoints impose upon use of the $GIO routine. The use of the $OPEN and $CLOSE commands to hog and release devices is recommended while using $GIO. In addition, several microcommands cannot be used on the MVP. These microcommands are indicated with asterisks in the $GIO tables found at the end of this chapter.

When data is received by the CPU from a device, it is accompanied by an *Input Bus Strobe* (IBS). This signal indicates to the CPU that there is data coming in from a peripheral device. When using $GIO input microcommands, a WR command should always precede the remainder of the command unless the programmer determines that there is reason to do otherwise. Although a breakpoint may occur during a WR when programming on the MVP, no data is lost. Without the WR, however, a breakpoint may occur before the byte is sent, resulting in a timeout error (I92) and a lost byte. With the WR command, the CPU is assured of receiving the byte within 8 ms.

In the accompanying tables, all recommended I/O-type microcommands are indicated by boldface hexcodes. Unless sufficient reason dictates otherwise, these are the only I/O-type microcommands that should be used.

15.4 GENERAL FORMS OF THE GENERAL I/O STATEMENTS

The general forms of the $IF ON/OFF and $GIO statements appear on the following pages.

$IF ON/OFF

General Form:

$IF { ON / OFF } [device-address, / file-number,] line-number

Where:

device-address = /taa, where taa is the device-address of the peripheral device to be tested.

file-number = #n, where n is either an integer or a numeric variable whose truncated value lies within the range 0-15, inclusive. A file-number identifies the slot in the Device Table to which the address of the I/O device to be tested has been assigned with the SELECT statement. (See Chapter 7, section 7.4.)

Purpose:

The $IF ON/OFF statement is used to determine the ready/busy status of any given device attached to the CPU and branch when ready or busy, depending upon the form of the statement used. Device ready/busy is a signal available to the CPU from every legal device-address. Although the particular meaning of the signal differs with various classes of devices, it is transparent to the programmer when using standard Wang I/O statements. However, when it is necessary to test this signal, $IF ON/OFF allows the user conditional branching depending upon the results of the test.

The $IF ON command tests the device ready/busy signal for the ready condition. If this condition is sensed, the program branches to the line-number specified in the $IF ON statement. If device busy is sensed, program execution continues at the next numbered line. Conversely, the $IF OFF statement tests the ready/busy signal for device busy. If this condition is found, the program branches to the specified line; if not, program execution proceeds in the normal manner.

The following is a list of classes of devices and their ready/busy characteristics:

1. Keyboard Class: Keyboard; 2236MXD Terminal Multiplexer; 2250 Controller, input (even) address; 2252A Controller; 2227 Controller; 2207 Controller. This class indicates device ready if one or more characters are ready to be sent to the CPU.

> **NOTE:**
>
> On the 2200MVP, $IF ON to the terminal keyboard (/001) senses the following status:
>
> READY — if the terminal is attached to the partition (foreground job) and a key has been pressed.
>
> BUSY — if the terminal is detached (background) or if attached (foreground) but no key has been pressed.

2. Printer/Plotter Class: Printers; Plotters; Typewriters and Output Writers; 2254 Controller, main address during data output operations. This class indicates device ready when the controller is ready to receive at least one more byte.

3. Console CRT Class: Console CRT's. The ready/busy characteristics of this class vary with the particular 2200 system used. On the 2200VP, the standard CRT is always ready. If the MXD is used with a 2200VP, the CRT ready condition occurs when the controller is ready to receive at least one more byte. On the MVP, $IF ON/OFF is used to determine if the partition executing the statement is in the foreground or the background (see Chapter 16, section 16.8 and the note on the preceding page).

4. Special Devices Class: 2209A Nine-Track Tape Drive; 2227B and 2228B Telecommunications Controllers. The signal is set to Ready when the special device has completed the previous operation.

5. Model 2250 8-bit Parallel I/O Controller (at odd addresses). This board has no built-in ready/busy signal; instead, a pin on the board's connector is defined by the 2200 system to be the location of this signal. The meaning of the signal at this pin depends upon the external device. It is recommended that, if possible, the device attached to this controller use the ready/busy signal in the same way as devices in category 2.

6. Address /000. This address is always ready on the 2200VP/MVP. This address can be used to test whether the system is a 2200T or a 2200VP/MVP since on the 2200T, device-address /000 will always be busy. Address /000 may also be used as a "dummy" address for undesired output by SELECTing it for PRINT operations.

Addresses can be specified in a $IF ON/OFF statement in three ways. Addresses can be directly supplied within the statement by the hexadecimal address of the device in the form /taa, where t is the device-type and aa is the actual hardware address. An address can also be specified by using a file-number to which an address has already been assigned. If the address to be tested is not specified by either of these two methods, the system will select as the default address the last SELECTed address for TAPE-class operations.

Examples of Valid Syntax:

```
10 SELECT TAPE/02A
20 $IF ON 200

10 SELECT #3/02B
20 $IF ON #3,250

10 $IF ON/028, 450
20 $IF OFF/028,1000
```

$GIO

General Form:

$GIO [comment] [device-address[,] / file-number,] (arg-1 [,arg-2]) [arg-3 [;arg-3...]]

Where:

comment = A character string identifying the particular operation (e.g., WRITE, READ, CHECK READY) performed by the $GIO sequence. The comment is ignored by the system. Only uppercase letters, digits, and spaces are legal comment characters.

device-address = /taa, where taa is the three-hexdigit device-address of a specified I/O device. The first hex digit (t) represents the device-type; it has no significance during a $GIO operation and may be set to zero. The next two hex digits (aa) represent the unit-device-address; they must correspond to the address which is preset on the I/O controller board of the device being accessed.*

file-number = #n, where n is either an integer or a numeric-variable whose truncated value lies within the range 0-15, inclusive. A file-number identifies the slot in the Device Table to which the address of the I/O device to be tested has been assigned with the SELECT statement. (See Chapter 7, section 7.4.)

arg-1 = A customized microcommand sequence which defines the I/O operation as follows:

a) Directly, by a hex literal, alpha literal, or a string of hex digits, with each four-hexdigit code denoting one two-byte microcommand.

b) Indirectly, by an alpha-variable containing the microcommand sequence.

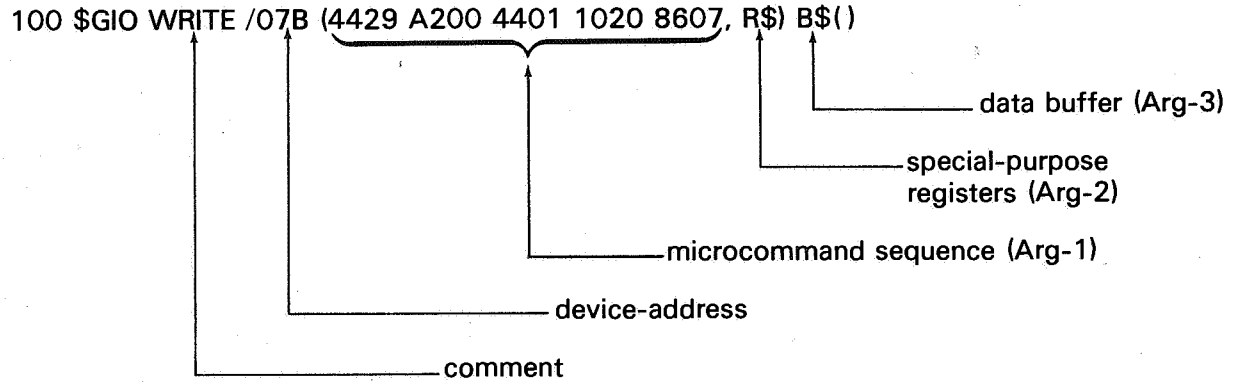
*If neither an indirect nor absolute address is specified, the address currently assigned to TAPE is the default address.

arg-2 = An alpha-variable whose individual bytes (called "registers") are used for storage of special characters and error/status information. The dimensioned length must be at least 10 bytes.

arg-3 = An alpha-variable used as the data buffer for multiple-character input and output operations. (Arg-3 is not required for single-character I/O operations.)

I/O routines are written in a $GIO statement as a series of microcommands represented by a code which is two bytes (four hexadecimal digits) in length. This code instructs the system to perform one or more specific operations such as move a specified character into a designated alpha-variable or read a string of characters from an external device into a buffer memory.

For example, consider the following statement:



The comment and device-address in the illustration are optional components. The Arg-3 data buffer B$( ) is required in this example by the presence of the microcommand A200, which represents a multicharacter output operation. A data buffer is not required, however, when a sequence contains no multicharacter I/O microcommand. The alpha-variable containing the special-purpose registers (R$) is also optional, although without it no error/status information can be passed to the main program. At least one microcommand is required in a $GIO statement.

15.5 OPTIONAL COMMENTS FOR $GIO OPERATIONS

Since the I/O operation represented by a $GIO statement is not readily identifiable from its microcommand sequence, a descriptive comment inserted in the $GIO statement may prove helpful when reviewing or revising a program. The comment has no functional purpose and is ignored by the system when the $GIO statement is executed. The comment may consist of any combination of uppercase letters, digits, and blanks; other characters are illegal in a comment.

15.6 DEVICE-ADDRESSES FOR $GIO STATEMENTS

Three different methods can be used to specify a device-address for a $GIO operation:

1. Direct address specification using a virgule followed by a three-hexdigit device-address. For example:

```
200 $GIO READ /03A (M$,R$) B$( )
```

2. Indirect address specification using a file-number to which the desired device-address has been previously assigned. For example:

```
300 SELECT #2/03A
310 $GIO READ #2, (M$,R$) B$( )
```

3. Omitting an address, thereby implying that the device-address currently SELECTed for TAPE-class operations should be used. For example:

```
400 SELECT TAPE/03A
410 $GIO READ (M$,R$) B$( )
```

15.7 MICROCOMMAND SEQUENCES FOR $GIO OPERATIONS (ARG-1)

Each microcommand in a sequence must be represented by a four-hexdigit code of the form *hhhh*, where *h* is a hexadecimal digit (0 through 9 or A through F). The first two hex digits in a microcommand code usually identify the type of operation to be performed, e.g., single-character output with echo or multicharacter verify. The last two hex digits supply information, e.g., a character to be stored or a register containing a character to be transmitted. There is no limit on the number of microcommands which can be specified in a single $GIO statement.

Types of Microcommands

There are 21 different categories of microcommands available for use in a $GIO statement. The microcommands in these categories are of two basic types: I/O microcommands and control microcommands. Table 15-2 at the end of this chapter summarizes the 21 microcommand categories and their functions.

The majority of $GIO microcommands are I/O microcommands, which are used to send one or more characters to an external device or receive one or more characters from an external device. I/O microcommands are found in 15 of the 21 microcommand categories. Each I/O microcommand represents a unique signal sequence defining a fundamental I/O operation. For example, the microcommand 400D instructs the system to transmit a HEX(0D) character (carriage return) to the specified external device. Specifying a sequence of I/O microcommands is therefore equivalent to programming the signal sequence required to perform the desired I/O operation.

Control microcommands do not directly perform input or output operations, but they do perform a variety of important supplementary programming functions such as:

- Moving, comparing, and testing specified characters. Control microcommands also set the condition code depending upon the result of a test.
- They check error/status information in Arg-2 registers and set the condition code.
- Control microcommands activate and disable the timeout condition. These functions cause the condition code to be set if any expected input strobe or output ready condition is not received within one millisecond to 64 seconds. Timeout has restricted use on the MVP.
- They activate and disable a delay before all output strobes are sent. A "fine delay" can range in duration from 10 microseconds to 1.275 milliseconds, and the length of a "coarse delay" can extend from 50 microseconds to 3.2 seconds.
- They branch to a specified microcommand within the $GIO sequence or terminate $GIO execution on condition code true or false.
- They set up the next specified Arg-3 data buffer.
- They move characters between the Arg-3 data buffer and the Arg-2 register.
- They increment or decrement the specified Arg-2 register or register pair.

The control microcommands provide the capability to program complete I/O routines, including testing, branching, and the use of multiple data buffers, within a single $GIO statement.

Several examples of $GIO usage which demonstrate the types of microcommands are contained in section 15.9.

The Condition Code

Certain operations within a $GIO sequence cause a special flag in memory called the "condition code" to be set. The condition code has two possible values, "true" (or "on") and "false" (or "off"). Initially, the condition code is set to false, and it remains false until a condition occurs which alters its status. In general, the condition code is automatically set to true by a variety of special conditions, including certain error and termination conditions. The specific conditions which set the condition code are documented in the tables at the end of this chapter. In addition to the conditions which automatically cause the condition code to be set, a number of control microcommands are provided to permit the programmer to test for specified conditions and explicitly set the condition code to true depending upon the result of the test.

Once the condition code has been set to true, execution of the $GIO statement is terminated unless the next sequential instruction in the microcommand sequence is one of two special branch instruc-

tions which test the status of the condition code and initiate a branch within the $GIO routine. These two special microcommands have the following forms:

Dhhh (branch to *hhh* if condition code true)
Ehhh (branch to *hhh* if condition code false)

where "hhh" is the three-hexadecimal-digit "address" of a microcommand within the microcommand sequence. The "address" of each microcommand represents its displacement from the first microcommand in the sequence. Thus, hex 000 is the "address" of the first microcommand in a $GIO statement, and hex 001 is the "address" of the second microcommand, etc. For example, a branch instruction such as

D001

causes a branch to the second microcommand in the $GIO statement if the condition code is true. If the condition code is false, no branch is taken, and the next microcommand in the sequence is executed. Conversely, a microcommand such as

E005

generates a branch to the sixth microcommand if the condition code is false, but continues to the next sequential command if it is true. Both branch instructions also automatically reset the condition code to false.

The branch instructions provide, in conjunction with the condition code, a means of responding to special conditions in an I/O operation without terminating $GIO statement execution. In addition, they serve as powerful tools for constructing loops and branches within a $GIO statement, facilitating the implementation of sophisticated, custom-tailored I/O routines in a single statement.

Direct and Indirect Specification of a Microcommand Sequence

The Arg-1 microcommand sequence defining a desired I/O operation can be specified either directly or indirectly in a $GIO statement. Direct specification of the microcommand sequence may occur in three ways. The microcommands can be specified with hex digits, as in the following statement:

```
10 $GIO (A000,R$)A$
```

The microcommands also can be specified in a hex literal:

```
10 $GIO (HEX(A000),R$)A$
```

Finally, they may be specified in an alphanumeric literal. In this case, the ASCII codes of the specified characters are interpreted as the microcommand codes. For example, the statement

```
10 $GIO ("DD",R$)A$
```

is equivalent to

```
10 $GIO (HEX(4444),R$)A$
```

since the ASCII code for the letter *D* is hex 44.

Although all three forms of direct specification are equally legal, the hex literal will execute faster than the hex digits and should be preferred for all but the most simple sequences. Because the alpha literal is an indirect means of representing microcommand codes, the hex literal is recommended for most applications. Note that spaces can be inserted for readability in a string of hex digits, but they are *not* allowed in a hex or alpha literal.

Indirect specification of the microcommand sequence can be made by assigning the microcommand codes to an alpha-variable and specifying the alpha-variable in a $GIO statement. For example, the statements

```
10 A$ = HEX(A000)
20 $GIO (A$,R$)B$
```

are equivalent to

```
20 $GIO (HEX(A000),R$)B$
```

Indirect specification of microcommand codes offers several advantages, including easier modification and debugging of the microcommand sequence and use of the same sequence in several different $GIO statements. Execution is as fast as with a hex literal.

15.8 ERROR/STATUS/GENERAL-PURPOSE REGISTERS (ARG-2)

The alpha-variable specified as the Arg-2 component of a $GIO statement is used as a multipurpose memory area where special characters and error/status information are stored. Each byte of the Arg-2 alpha-variable is called a "register," and the variable itself is commonly referred to as the "register-variable." The dimensioned length of the register-variable must be at least ten bytes because bytes (or "registers") 8, 9, and 10 are used by the system to store special information. The maximum number of registers which can be accessed is 15 (bytes 1-15). If the register-variable contains more than 15 bytes, the additional bytes cannot be used as registers. The Arg-2 component is optional in a $GIO statement. If it is omitted, the system uses 15 bytes in a reserved section of memory and initializes them to all zeroes. In this case, the values of particular registers can be neither read nor modified by the BASIC-2 program.

The registers are used by the BASIC-2 program for a number of purposes. They can store data characters or special termination characters, contain a binary value defining the duration of an implemented delay or timeout condition, or serve as a counter controlling the execution of a loop. The system uses one or more registers under certain conditions to store special characters, error/status information, and a count of the number of characters received into or sent from a buffer during an input routine. Special control microcommands enable the programmer to compare the contents of two registers, to increment or decrement a specified register or register pair by a fixed amount, and to test for specific error bits in the error register.

Although all 15 registers are available for use by the programmer, several registers are used by the system to store information during certain operations and their use should be avoided by the programmer. In particular, register 8 is used to store error/status information (each bit representing a specific error condition), and registers 9 and 10 are used to maintain a count of the characters received during a multicharacter input operation. Registers 5 and 6 are used, respectively, to store the calculated LRC character and ENDI character during certain multicharacter input and output operations. If the application program uses any of these registers (5,6,8,9,10), the user must realize that the system will alter the values of those registers during certain operations. Table 15-8 at the end of this chapter summarizes register usage for each of the 15 registers.

Registers 8, 9, and 10 are automatically initialized to binary zero (HEX(00)) when the $GIO statement begins execution. Subsequently, registers 9 and 10 are reset to zero whenever the data buffer pointer is set to point to a new buffer by using one of the control commands 18hh, 1Ah0, or 1A00. The remaining registers are not initialized automatically and can be assigned an initial value by the BASIC-2 program.

15.9 DATA BUFFERS (ARG-3)

The Arg-3 component of a $GIO statement consists of one or more alpha-variables used as "data buffers." Data buffers are required only for multicharacter input and output operations. They are not required in a $GIO statement which is restricted to single-character input or output.

Size of the Data Buffer

Alphanumeric scalar and array variables can be used as data buffers. Because scalar-variables are limited to a maximum length of 124 bytes, array-variables are more commonly used in multicharacter

I/O operations. The size of the data buffer is equal to the total number of bytes in the defined length of the alpha-variable. The defined length may be the entire dimensioned size of the alpha-variable or some specified portion of its total size. The length of this specified portion can be defined with a STR function.

If an alphanumeric-array (one- or two-dimensional) is used as a data buffer, characters are stored into the array or read from the array sequentially, row by row. The entire array is treated as one contiguous string of characters, starting with the first character of the first element. Element boundaries are ignored.

Multiple Buffers and the Data Buffer "Pointer"

Multiple data buffers must be separated by semicolons (;). The particular data buffer to be used in an I/O operation is indicated by a data buffer "pointer." The pointer is automatically set to point to the first buffer in the Arg-3 sequence when a $GIO statement is executed. The pointer can be set to point to any buffer in the sequence under program control by using one of three special control microcommands. For sequential processing of buffers, the 1A00 control command increments the pointer to the next sequential buffer. A subsequent I/O operation will utilize that buffer. If processing is to be nonsequential, two control microcommands enable the program to move the pointer to a specified buffer in the Arg-3 sequence by specifying the address of the buffer to be used. The address of each buffer in the sequence is simply its displacement from the first buffer in the sequence. Thus, the first buffer has an address of HEX(00), the second has an address of HEX(01), etc. The address of the buffer to be used can be specified immediately in an 18hh microcommand, where *hh* is the address. Alternatively, it can be specified indirectly as the contents of a register with a 1Ah0 command, where the third hex digit specifies the register (1-15) containing the buffer address. (These three instructions, 1A00, 18hh, and 1Ah0, are listed in Table 15-2, "Control Microcommands," found at the end of this chapter.)

The data buffer pointer is moved only by one of the three microcommands described above. The pointer is never moved automatically (i.e., implicitly). Thus, sequential multicharacter I/O commands will continue to use the same data buffer until a new data buffer is designated by explicitly moving the pointer with either the 1A00, 18hh, or 1Ah0 command. Data continues to be sent from or received into the buffer sequentially by each microcommand until a termination condition is sensed or the data pointer is moved to another buffer.

The Character Count

During a multicharacter input or output operation, the system automatically maintains a count of the total number of characters sent from or received into a particular buffer. The count is a two-byte binary number kept in registers 9 and 10. The low-order eight bits of the count are stored in register 10, and the high-order eight bits are stored in register 9. Each time a character is received into or sent from the currently specified buffer, the count is incremented. The count for a particular buffer continues to be incremented by subsequent microcommands which use the same buffer. The count is reset to binary zero only when the data buffer pointer is moved to a new buffer. Thus, the count for each buffer initially starts with a value of zero and is increased cumulatively to reflect the total number of characters transferred or received by all microcommands utilizing that buffer. For example, if a buffer is only partially filled by a multicharacter input command, a subsequent multicharacter input command would store data in the remaining unused portion of the buffer and continue to update the count to reflect the total number of characters received in the buffer. If a multicharacter input command continues to input data after the buffer has been filled, the count continues to be updated to reflect the total number of characters received. The additional characters are not stored, however, but are simply lost.

Terminating Multicharacter I/O Operations

A simple multicharacter output operation always outputs the total number of characters in the output buffer or the defined portion of the buffer. The output operation terminates when the last character has been sent.

A multicharacter input operation, however, can be terminated when one of the following three conditions occurs: the character count equals the input buffer length, an ENDI character is received, or a special termination character is received. Each of these methods is described below.

- **Termination on Count.** Most multiple-character input commands can be terminated when the input buffer is filled. If this termination condition is specified, the system compares the character count with the total number of bytes reserved for the buffer after each character is received. The system terminates the input operation when the character count equals the buffer length (i.e., when the buffer is filled).

- **Termination on ENDI Character.** Certain I/O devices, including the system keyboard, can send a character with a special ninth bit, called the "ENDI bit," in addition to the eight data bits. For example, depressing a Special Function Key on the system keyboard generates a character with an ENDI bit. If this termination condition is specified, the system checks each incoming character for an ENDI bit and terminates the input operation when one is received. The ENDI character is stored in Arg-2 register 6. Note that only the eight data bits are stored; the ENDI bit is stripped off prior to storage. The count does not include the ENDI byte.

- **Termination on Special Character.** The programmer may designate any character as a termination character. The desired character must be stored in register 1 prior to beginning the input operation. As each character is received, it is compared with the character stored in register 1; termination occurs when the specified termination character is received. The special character may be stored with the data characters in the data buffer or discarded.

If the character count is not specified as a termination condition, the number of characters received prior to termination may exceed the available buffer space. In this case, an error bit is set in the error register (register 8), and the excess characters are received but not stored. The count continues to be updated to reflect the total number of characters received, whether stored or not.

It is possible to specify one, two, or all three termination conditions in a single multicharacter input command. If multiple conditions are specified, the input operation terminates when any one of the specified termination conditions is satisfied. The order in which termination conditions are checked by the system is defined below:

First	—	ENDI Character
Second	—	Special Termination Character
Third	—	Character Count

If multiple termination conditions are specified, the system sets a bit in register 8 to indicate which condition caused termination. Subsequently, the program can check register 8 to determine which of the specified termination conditions actually caused the input operation to terminate. No bit is set in register 8 if only one termination condition is specified in the input command.

15.10 SIMPLE EXAMPLES OF $GIO

Output

The following examples illustrate the use of different $GIO routines to output one or more characters to a device. In these examples, the CRT is used as the output device to simplify the illustration of these routines. Normally, one should output to the CRT by using standard BASIC-2 statements such as PRINT or LIST rather than the $GIO statement as in these demonstrations.

This $GIO sequence will display the word *WANG* on the CRT screen using Immediate Mode output microcommands:

```
:10 $GIO /005 (4057 4041 404E 4047)
```

Most simple output devices indicate their readiness to receive a byte by setting their Ready/Busy signal on the I/O bus to Ready. The first byte of the first microcommand (40) waits for this Ready signal (WR) and then outputs the character indicated by the hexcode in the second byte of the microcommand (i.e., 57, which is the character "W"). Data output is accomplished with a single OBS for each character.

Since specifying each character in an individual microcommand is frequently an inefficient method of outputting data, the data is usually output from a variable. This method allows the operator to use one $GIO sequence for all output of this type by simply changing the value of the variable during program execution. Indirect $GIO microcommands are used to change the value of the variable which holds the data. For example, the following program will output *WANG* on the screen, as in the previous example:

```
:10 R$="WANG"
:20 $GIO /005 (4210 4220 4230 4240, R$)
```

The first byte of each microcommand (42) waits for device Ready and then, using an OBS, sends the byte of R$ indicated by the second byte of the microcommand (e.g., 10 sends the first byte, which is the character "W").

Long data strings are output using multicharacter output commands; the data to be output is stored in Arg-3. The following $GIO statement will output the first 100 bytes of A$():

```
:100 $GIO /005 (A000) STR(A$(),1,100)
```

The STR function is used to indicate that not all of A$() is to be output, but only the first 100 bytes.

All the various forms of output demonstrated above can be intermixed within a single $GIO statement.

Input

The following examples illustrate $GIO usage for data input from a single device. The keyboard is used as the input device to provide examples which illustrate typical programming situations. Normally, the keyboard is not accessed via a $GIO statement, but rather through the BASIC-2 statements KEYIN, INPUT, and LINPUT.

The following $GIO sequence receives three characters from the keyboard using single-character input microcommands:

```
:10 $GIO /001 (8701 8702 8703, R$)
```

The first three bytes of R$ will contain the three characters input from the keyboard.

If more than a few characters are to be input, multicharacter input commands should be used. For example, the following sequence inputs ten characters:

```
:10 $GIO /001 (C340, R$) STR (A$,1,10)
```

This microcommand receives one character at a time from the keyboard and stores it in one of the first ten bytes of A$. After receiving and storing each character, the third digit of the microcommand instructs the system to check the buffer to see if it is full. If so, the input sequence terminates according to the last digit in the microcommand (LEND). Tables of these codes and their operation are provided at the end of this chapter.

Other termination conditions are available as well as the one previously mentioned. For example, variable length input, terminated by input of a specific character such as a carriage return, is demonstrated in the following example:

```
:50 STR(R$,1,1)= HEX(0D)
:60 $GIO /001 (C310, R$) A$
```

The first line stores a carriage-return character in the first byte of R$. The digit *1* in the microcommand instructs the system to check byte *1* of R$ after each character is stored and to terminate the input sequence according to LEND if the character received is the same as the character stored in R$. If the character does not match, the sequence is repeated.

The program should check bytes 9 and 10 of R$ to determine how many bytes were actually received. For example:

```
:70 N=VAL(STR(R$,9,2),2)
```

This statement will calculate the number of characters input in the $GIO sequence.

Termination upon reception of a special function character (i.e., by an ENDI bit which is sent) can be achieved in the following manner:

```
:60 $GIO /001 (C320, R$) A$
```

As each character is received, the microcommand checks it before it is stored to determine if it carries a special ninth bit (ENDI). If so, the character is saved in byte 6 of R$ and the sequence is terminated according to LEND. Otherwise, the character is saved in A$ and the sequence continues. Again, the character count is maintained in bytes 9 and 10 of R$.

Selection of combinations of termination conditions and LEND sequences can be accomplished by varying the last two digits of the C3t1-type microcommands according to the desired combination. For instance, C330 terminates the input sequence when either a special character or an ENDI bit is sent.

The next example accepts a string of characters from the keyboard, displaying each character on the CRT as it is keyed:

```
:10 $GIO (010D 7101 8702 7105 4220 1B22 1C12 E001, R$) A$
```

The preceding microcommand sequence works in the following manner. The first microcommand (010D) stores the character whose hex code is the last byte of the microcommand in the register specified by the second digit of the microcommand. In this case, a carriage return (HEX(0D)) is stored in byte 1 of R$. The second microcommand (7101) sends an ABS to the device whose address is equal to the second byte of the microcommand (device type is omitted). In this example, the keyboard is now enabled. The next command (8702) waits for the device to be ready and then receives an IBS (and, hence, one character) from the keyboard, storing it in the register specified by the last digit of the command. The CPU then enables the CRT (7105), waits for CRT ready, and performs an OBS sending the character in register 2 back to the CRT (4220). The next microcommand (1B22) takes the character in register 2, places it in the data buffer (A$), and increments the count maintained in registers 9 and 10. In addition, this command sets the condition code (and terminates) if the data buffer is full. The next microcommand (1C12) compares the character in register 1 to that in register 2 and sets the condition code if the two are the same. The final microcommand (E001) tests the condition code. If it is false, a branch to the second microcommand takes place (001); otherwise, the sequence is ended. Thus, the sequence can be terminated if either the data buffer becomes full or a carriage return is received. The data buffer A$ will contain the characters received (including the carriage return), and the count will again be stored in registers 9 and 10.

Table 15-1
Legend

(Mnemonics used to describe signal sequences for I/O microcommands)

Mnemonic	Operation
ABS	CPU sends an "address bus strobe" with an immediate or indirect address to disable the current address and enable the specified address.
CBS	CPU sends a CBS strobe to the enabled device.
CHECK ENDI	CPU checks for ENDI condition; if ENDI bit has been sent the byte is saved in register 6 (rather than r) and the 20 bit of register 8 is set.
CHECK T_1	CPU checks for ENDI and Special Character termination conditions, and proceeds according to code t.
CHECK T_2	CPU checks for Special Character and full buffer termination conditions, and proceeds according to code t.
CPB	CPU sets its input Ready/Busy signal to Ready.
DATAOUT	CPU sends out next character from $GIO arg-3 buffer, then increments the count in bytes 9 and 10.
ECHO	The received character is echoed (with either OBS or CBS).
IBS	CPU awaits input strobe from enabled device. If the wait is greater than 8ms on the MVP, error 192 results.
IMM	Immediate character is HEX (h_1h_2), specified by the microcommand.
IND	Indirect character is in the register specified by r.
LEND	CPU executes the LRC end sequence specified by l.
OBS	CPU sends an OBS strobe to the enabled device.
SAVE	CPU saves received character in the register specified by r.
SAVE DATA	CPU saves received character in the next location of the arg-3 buffer, then increments the count.
SAVE LRC	CPU saves calculated LRC character in register 5.
SEND LRC	CPU sends calculated LRC character to enabled device.
SET CC	CPU terminates microcommand and sets condition code if specified condition exists.
VERIFY	CPU compares received character; if unequal, the echo-verify error bit (bit 04 in register 8) is set to 1.
WR	CPU awaits Ready signal from enabled device. If the wait is greater than 1ms on the MVP, a break point may result.
W5	CPU waits (5 microseconds) until OBS or CBS is complete.

Table 15-1
Legend (Cont.)

Symbol	Digit in that position represents:
a	portion of address
c	microcode command specification
d	delay specification
h	HEX digit
l	LEND code
r	register
t	Check-T code

Table 15-2
Summary Microcommand Categories

CODE		Operation	Refer To:
$7ch_1h_2$	(c = 1, 3, 6)	Single Address Strobe	Table 15-3
Orh_1h_2		Control — store immediate	Table 15-4
$1h_1h_2h_3$		Control — general	
$75h_1h_2$		Control — delay immediate	
77r0		Control — delay immediate	
$Da_1a_2a_3$		Branch Control	
$Ea_1a_2a_3$		Branch Control	
$4ch_1h_2$		Single Character output	Table 15-5
$5ch_1h_2$		Single Character output with acknowledge	
$6ch_1h_2$		Single Character output with Echo	
$8ch_1h_2$	(c = 0 — 3, 8 — B)	Single Character input with verify	Table 15-6
$8ch_1h_2$	(c = 0 — 3, 8 — B)	Single Character input	
$9ch_1h_2$	(c = 0 — 3, 8 — B)	Single Character input with echo	
AcOl		Multicharacter output	Table 15-7
Bctl	(c = 0, 1, 4, 5)	Multicharacter output with acknowledge	
Bctl	(c = 2,3,6,7)	Multicharacter output with echo	
Bctl	(c = 8, 9, C, D)	Multicharacter output (each character requested)	
BAtl		Multicharacter verify	
Cctl	(c = 2, 3, 6, 7)	Multicharacter input	Table 15-8
Cctl	(c = 0, 1, 4, 5)	Multicharacter input with echo	
Cctl	(c = 8 through F)	Multicharacter input (each character requested)	

Table 15-3
Single Address Strobe

Code	Signal Sequence	Verify Character	Character to be Saved
$71h_1h_2$	ABS/IMM	HEX (h_1h_2)	
73r0	ABS/IND	from register r	
760r	STATUS REQUEST		in register r

NOTE:

Codes in this category can be used repeatedly in a sequence to disable the current device address and enable another.

Table 15-4
Control Microcommands

Code	Operation
$0rh_1h_2$	Store immediate second character, HEX(h_1h_2) in register r.
1000	Set condition code true.
1010	Set condition code if device is ready.
1020	Wait for device ready (with timeout)
1200	Disable previously set delay/timeout condition.
$11r_1r_2$	Move contents of register r_1 to register r_2.
12r1	Set a "coarse" delay before each subsequent OBS or CBS. The length of the delay in units of 50 microseconds is specified by a two-byte binary value stored in registers r and r + 1 (where $1 \leq r \leq 14$). Maximum delay = HEX (FFFF) ~ 3.3 seconds.
12r2	Set a timeout prior to checking each subsequent device ready signal (for output operations) or input strobe (for input operations). The interval in units of one millisecond is specified by a two-byte binary value stored in registers r and r + 1 (where $1 \leq r \leq 14$). Maximum timeout interval = HEX(FFFF) ~ 65.6 seconds. If a timeout interval is exceeded, set condition code, and set error bit (bit 10) in register 8. (The MVP restricts timeout to 15ms for input only.)
$13d_1d_2$	Set a "fine" delay before each subsequent OBS or CBS. The duration of the delay is specified, in units of five microseconds, by the binary value of the second byte of the command (d_1d_2). Minimum delay = 10 microsecond delay, while HEX(03) = 15 microseconds, HEX(04) = 20 microseconds, etc. Maximum delay = HEX(FF) (~ 1.275 milliseconds). (When execution of a $GIO sequence begins, this delay is set to HEX(0A) (50 microseconds).)

Table 15-4
Control Microcommands (Cont.)

Code	Operation
$14r_1r_2$	If contents of register r_1 ≠ contents of register r_2, set compare error bit (bit 08, register 8) to 1.
$15r_1r_2$	If contents of register r_1 ≠ contents of register r_2, set compare error bit (bit 08, register 8). Then, if compare error bit is set, set condition code.
$16a_1a_2$	If complemented status (register 8) code AND a_1a_2 ≠ HEX(00), set condition code(i.e., set cc if any bit specified by the mask a_1a_2 is equal to zero).
$17a_1a_2$	If status (register 8) code AND a_1a_2 ≠ HEX(00), set condition code (i.e., set cc if any bit specified by the mask a_1a_2 is equal to one).
$18a_1a_2$	Set data buffer pointer to specified buffer in Arg-3 sequence. Buffer pointed to is specified by binary value of 2nd byte of command (a_1a_2), which represents displacement from 1st buffer in sequence. Thus HEX(1800) points to 1st buffer, HEX(1801) points to 2nd buffer, HEX(1802) points to 3rd buffer, etc. ERR P47 is signalled if specified buffer does not exist (i.e., not enough buffers). This command also resets the count (Registers 9 and 10) to zero.
19rc	Increment/decrement binary value stored in register r or in pair of registers r and r + 1, depending upon value of c:

c	Operation
0	— increment register r by 1
1	— increment register r by 2
2	— decrement register r by 2
3	— decrement register r by 1
4	— increment register pair by 1
5	— increment register pair by 2
6	— decrement register pair by 2
7	— decrement register pair by 1

	If a register pair is incremented/decremented, it is treated as a 2-byte binary value, with the low-order byte in reg. r + 1.
1ArO	Same as $18a_1a_2$, except displacement is obtained from register r (r ≥ 1).
1A00	Increment data buffer pointer to next buffer to register r. Reset data count to zero.
1Br1	"Write" one byte from data buffer to register r. Increment data count (registers 9,10). Set condition code if buffer empty, without changing count.
1Br2	"Read" one byte from register r into data buffer. Increment data count (registers 9, 10). Set condition code if buffer full, without changing count.
$1Cr_1r_2$	Set condition code if register r_1 = register r_2.

Table 15-4
Control Microcommands (Cont.)

Code	Operation
$1Dr_1r_2$	Set condition code if register pair $r_1r_1 + 1$ = register pair $r_2r_2 + 1$.
$1Er_1r_2$	Set condition code if register r_1 > register r_2
$1Fr_1r_2$	Set condition code if register pair $r_1r_2 + 1$ > register pair $r_2r_2 + 1$.
$75d_1d_2$	Delay immediately. This command waits from 0 to 255 milliseconds, using an 8-bit count specified by d_1d_2, before continuing to the next command. This delay is invoked only once, and is unrelated to other delays.
77r0	Delay immediately. Same as preceding delay, except that count is obtained from register r.
$Da_1a_2a_3$	Branch to microcommand whose "address" is specified by $a_1a_2a_3$ if condition code is *true*. Twelve-bit "address" specified by $a_1a_2a_3$ is displacement from 1st microcommand in Arg-1 sequence. Thus HEX (D000) branches to 1st command in sequence if condition code is true, HEX(D001) branches to 2nd command, etc. If condition code is false, proceed to next microcommand. Reset condition code to false.
$Ea_1a_2a_3$	Branch to microcommand whose "address" is specified by $a_1a_2a_3$ if condition code is *false*. If condition code is true, proceed to next microcommand. Reset condition code to false.

NOTE:

Because 12-bit addresses are used in the branch instructions, there is a limit of 4,096 microcommands in a $GIO sequence (8192 bytes) which may serve as the destination of a branch.

Table 15-5
Single Character Output Microcommands

Code	Signal Sequence	Character To Be Sent	Character To Be Saved
	Single Character Output		
40h_1h_2	**WR, OBS/IMM**	**HEX (h_1h_2)**	
41h_1h_2	OBS/IMM	HEX (h_1h_2)	
42r0	**WR, OBS/IND**	**from register r**	
43r0	OBS/IND	from register r	
44h_1h_2	**WR, CBS/IMM**	**HEX (h_1h_2)**	
45h_1h_2	CBS/IMM	HEX (h_1h_2)	
46r0	**WR, CBS/IND**	**from register r**	
47r0	CBS/IND	from register r	
	Single Character Output with Acknowledge		
50h_1h_2	WR, OBS/IMM, W5, CPB, IBS	HEX (h_1h_2)	
51h_1h_2	OBS/IMM, W5, CPB, IBS	HEX (h_1h_2)	
52r_1r_2	WR, OBS/IND, W5, CPB, IBS, SAVE	from register r_1	into register r_2
53r_1r_2	OBS/IND, W5, CPB, IBS, SAVE	from register r_1	into register r_2
54h_1h_2	WR, OBS/IMM, W5, CPB, IBS	HEX(h_1h_2)	
55h_1h_2	CBS/IMM, W5, CPB, IBS	HEX (h_1h_2)	
56r_1r_2	WR, CBS/IND, W5, CPB, IBS, SAVE	from register r_1	into register r_2
57r_1r_2	CBS/IND, W5, CPB, IBS, SAVE	from register r_1	into register r_2
	Single Character Output with Echo		
60h_1h_2	WR, OBS/IMM, W5, CPB, IBS, VERIFY	HEX (h_1h_2)	
61h_1h_2	OBS/IMM, W5, CPB, IBS, VERIFY	HEX (h_1h_2)	
62r_1r_2	WR, OBS/IND, W5, CPB, IBS, SAVE, VERIFY	from register r_1	into register r_2
63r_1r_2	OBS/IND, W5, CPB, IBS, SAVE, VERIFY	from register r_1	into register r_2
64h_1h_2	WR, CBS/IMM, W5, CPB, IBS, VERIFY	HEX (h_1h_2)	

Table 15-5
Single Character Output Microcommands (Cont.)

Code	Signal Sequence	Character To Be Sent	Character To Be Saved
$65h_1h_2$	CBS/IMM, W5, CPB, IBS, VERIFY	HEX (h_1h_2)	
$66r_1r_2$	WR, CBS/IND, W5, CPB, IBS, SAVE, VERIFY	from register r_1	into register r_2
$67r_1r_2$	CBS/IND, W5, CPB, IBS, SAVE, VERIFY	from register r_1	into register r_2
$68h_1h_2$	WR, OBS/IMM, W5, CPB, IBS, VERIFY, SET CC (if VFY bit set)	HEX (h_1h_2)	
$69h_1h_2$	OBS/IMM, W5, CPB, IBS, VERIFY, SET CC (if VFY bit set)	HEX (h_1h_2)	
$6Ar_1r_2$	WR, OBS/IND, W5, CPB, IBS, SAVE, VERIFY, SET CC (if VFY bit set)	from register r_1	into register r_2
$6Br_1r_2$	OBS/IND, W5, CPB, IBS, SAVE, VERIFY, SET CC (if VFY bit set)	from register r_1	into register r_2
$6Ch_1h_2$	WR, CBS/IMM, W5, CPB, IBS, VERIFY, SET CC (if VFY bit set)	HEX (h_1h_2)	
$6Dh_1h_2$	CBS/IMM, W5, CPB, IBS, VERIFY, SET CC (if VFY bit set)	HEX (h_1h_2)	
$6Er_1r_2$	WR, CBS/IND, W5, CPB, IBS, SAVE, VERIFY, SET CC (if VFY bit set)	from register r_1	into register r_2
$6Fr_1r_2$	CBS/IND, W5, CPB, IBS, SAVE, VERIFY, SET CC (if VFY bit set)	from register r_1	into register r_2

Table 15-6
Single Character Input Microcommands

Code	Signal Sequence	Verify Character	Character To Be Saved
	Single Character Input		
8600	CPB, IBS		
860r	CPB, IBS, SAVE		into register r
862r	CPB, IBS, CHECK ENDI + SAVE		into register r
8700	WR, CPB, IBS		
870r	**WR, CPB, IBS, SAVE**		**into register r**
872r	**WR, CPB, IBS, CHECK ENDI + SAVE**		**into register r**
	Single Character Input with Verify		
$80h_1h_2$	CPB, IBS, VERIFY/IMM	HEX (h_1h_2)	
$81h_1h_2$	WR, CPB, IBS, VERIFY/IMM	HEX (h_1h_2)	
$82r_1r_2$	CPB, IBS, SAVE, VERIFY/IMM	in register r_1	into register r_2
$83r_1r_2$	WR, CPB, IBS, SAVE, VERIFY/IND	in register r_1	into register r_2
$88h_1h_2$	CPB,IBS, VERIFY/IMM, SET CC (if VFY bit set)	HEX(h_1h_2)	
$89h_1h_2$	WR, CPB, IBS, VERIFY/IMM SET CC (if VFY bit set)	HEX (h_1h_2)	
$8Ar_1r_2$	CPB, IBS, SAVE, VERIFY/IND, SET CC (if VFY bit set)	in register r_1	into register r_2
$8Br_1r_2$	WR, CPB, IBS, SAVE, VERIFY/IND, SET CC (if VFY bit set)	in register r_1	into register r_2
	Single Character Input with Echo		
920r	CPB, IBS, SAVE, WR, ECHO/OBS		into register r
930r	CPB, IBS, SAVE, ECHO/OBS		into register r
960r	CPB, IBS, SAVE, WR, ECHO/CBS		into register r
970r	CPB, IBS, SAVE, ECHO/CBS		into register r

Table 15-7
Multicharacter Output Microcommands

(A sequence in parentheses is repeated for each character in the data buffer.)

Code	Signal Sequence
	Multicharacter Output
A00l	**(WR, DATAOUT/OBS), LEND**
A10l	(DATAOUT/OBS), LEND
A20l	25 microsecond version of A00l; no timeout or delay
A40l	(WR, DATAOUT/CBS), LEND
A50l	(DATAOUT/CBS), LEND
A60l	SCAN DATA BUFFER, CALCULATE LRC, LEND
	Multicharacter Output with Acknowledge
B0tl	(WR, DATAOUT/OBS, W5, CPB, IBS, CHECK T), LEND
B1tl	(DATAOUT/OBS, W5, CPB, IBS, CHECK T), LEND
B4tl	(WR, DATAOUT/CBS, W5, CPB, IBS, CHECK T), LEND
B5tl	(DATAOUT/CBS, W5, CPB, IBS, CHECK T), LEND
	Multicharacter Output with Echo
B2tl	(WR, DATAOUT/OBS, W5, CPB, IBS, VERIFY, CHECKT), LEND
B3tl	(DATAOUT/OBS, W5, CPB, IBS, VERIFY, CHECKT), LEND
B6tl	(WR, DATAOUT/CBS, W5, CPB, IBS, VERIFY, CHECKT), LEND
B7tl	(DATAOUT/CBS, W5, CPB, IBS, VERIFY, CHECKT), LEND
	Multicharacter Output with each Character Requested
B8tl	(CPB, IBS, CHECK T, WR, DATAOUT/OBS), LEND
B9tl	(CPB, IBS, CHECK T, DATAOUT/OBS), LEND
BCtl	(CPB, IBS, CHECK T, WR, DATAOUT/CBS), LEND
BDtl	(CPB, IBS, CHECK T, DATAOUT/CBS), LEND
	Multicharacter Verify
BAt0	**(CPB, IBS, VERIFY, CHECK T)**

Valid "Check T" Codes for Table 15-7

Termination Condition / Microcommand	B0tl B4tl B1tl B5tl	B2tl B6tl B3tl B7tl	B8tl BCtl B9tl BDtl	BAt0
None (Loop until buffer is done)	**0**	**0**	**0**	**8**
Terminate output sequence if verify unequal, set cc		1		9
Terminate output sequence if ENDI logic level '1' set cc	2	2	2	A
Terminate output sequence on either condition, set cc		3		B

Valid "Lend" Codes for Table 15-7

LRC* End Sequence / Microcommand	Actl	B0tl thru B7tl	B8tl B9tl BCtl BDtl
None, go to next microcommand	0	0	0
WR, SEND LRC/OBS, SAVE LRC	2	2	
SEND LRC/OBS, SAVE LRC	3	3	
SAVE LRC	**4**	**4**	**4**
WR, SEND LRC/CBS, SAVE LRC	6	6	
SEND LRC/CBS, SAVE LRC	7	7	

*The LRC is the XOR of all bytes transferred to or from the buffer in the present command.

Table 15-8
Multicharacter Input Microcommands[1]

Code	Signal Sequence
	Multicharacter Input
C22I	(CPB, IBS, no timeout or delay, CHECK ENDI, SAVE DATA), LEND
C3tI	**(WR, CPB, IBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND**
C6tI	**(CPB, IBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND**
C7tI	(Delay 50 μs, CPB, IBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND
	Multicharacter Input with Echo
C0tI	(CPB, IBS, WR, ECHO/OBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND
C1tI	(CPB, IBS, ECHO/OBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND
C4tI	(IBS, WR, ECHO/OBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND
C5tI	(CPB, IBS, ECHO/OBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND
	Multicharacter Input with Each Character Requested
C8tI	(WR, OBS,W5, CPB, IBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND
C9tI	(OBS,W5, CPB, IBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND
CAtI*	(CPB, WR, OBS, IBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND
CBtI*	(CPB, OBS, IBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND
CCtI	(WR, CBS,W5, CPB, IBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND
CDtI	(CBS,W5, CPB, IBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND
CEtI*	(CPB, WR, OBS, IBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND
CFtI*	CPB, CBS, IBS, CHECK T_1, SAVE DATA, CHECK T_2), LEND

[1] A sequence in parentheses is repeated until a valid termination condition occurs. In each command, the t-value is the termination code; the I-value the LEND code. See tables.

*The four indicated microcommands may NOT be used on the MVP. An illegal microcommand error will result.

Valid "Check T" Codes For Table 15-8

t	Termination Conditions* (order of checking from left to right)		
	ENDI-level = 1 (when character received)	**Special Character Received (matches char. in reg. 1)**	**Character Count Equals Buffer Length**
0	no action	check, save in buffer, include in LRC and count	no action
1	no action	check, do not save	no action
2	**check, save in reg. 6**	**no action**	**no action**
3	check, save in reg. 6	check, do not save	no action
4	no action	no action	check
5	no action	check, do not save	check
6	check, save in reg. 6	no action	check
7	check, save in reg. 6	check, do not save	check

*Termination conditions do not set condition code.

Valid "Lend" Codes For Table 15-8

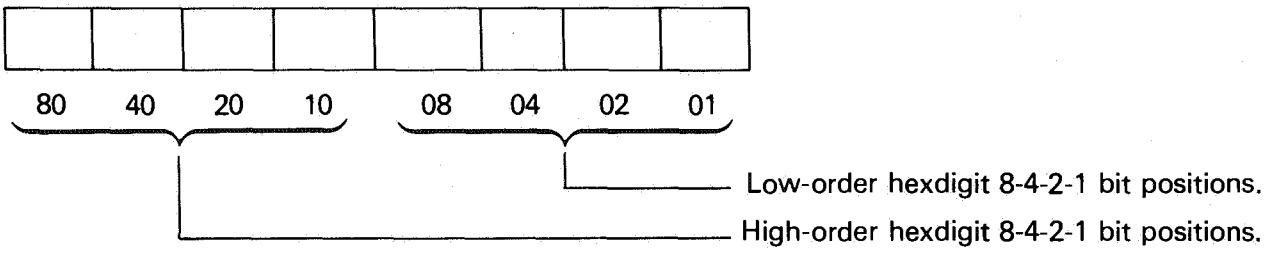
l	LRC End Sequence
0	**None, go to next microcommand**
1	Calculate LRC and save
2	Calculate LRC, save, compare with ENDI character, and set LRC error bit. (Use only if t = 2.)

*The LRC is the XOR of all bytes transferred to or from the buffer in the present command.

Table 15-9
Register Usage

Register (Byte)	Bit Position*	Use
0	all	Dummy location; if written to, data is lost; if read data is always 00. (Read-only register (ROR))
1	all	General-purpose or special termination character.
2, 3, 4	all	General-purpose
5	all	General-purpose or automatic storage of an LRC character.
6	all	General-purpose or automatic storage of an ENDI-level = 1 character.
7	all	General-purpose.
8	01 02 04 08 10 20 40 80	1 = Buffer overflow. 1 = LRC error. 1 = Echo/Verify error. 1 = Compare error. 1 = Timeout. 1 = ENDI-level termination. 1 = Special character termination. 1 = Count termination.
9, 10	all	Automatic storage of the count of transferred characters to or from the currently selected Arg-3 buffer.
11, 12 13, 14, 15	all	General-purpose.

*Bit position labels for status code (register 8) are as follows:



NOTE:

Registers 8, 9, and 10 are zeroed by the system when $GIO begins execution. Registers 9 and 10 are zeroed again whenever an $18a_1a_2$, 1A00, or 1ArO command is executed.

CHAPTER 16 2200MVP OPERATING SYSTEM AND SPECIAL LANGUAGE FEATURES

16.1 THE 2200MVP AND THE 2200VP: A BRIEF COMPARISON

The 2200MVP is an interactive, high-performance, multiuser system programmable in the BASIC-2 language. The 2200MVP operating system is essentially a modified version of the 2200VP operating system which has been enhanced to support multiuser operations. With a few exceptions (noted below), the hardware required to support a 2200MVP operating system is identical to that required by a 2200VP. Therefore, a 2200VP can be field-upgraded to a 2200MVP. In addition, up to 256K of user memory is available on the 2200MVP. Similarly, the BASIC-2 language implemented on the 2200MVP is virtually identical to BASIC-2 on the 2200VP, with the exceptions of several new language features added to support the multiprogramming environment and changes to certain time-sensitive statements such as $GIO. These new and modified BASIC-2 language features are described in the present chapter. (These statements, though generally not functionally useful on the 2200VP, have been added to 2200VP BASIC-2 for program compatibility.)

To summarize, the major differences between the 2200VP and the 2200MVP are as follows:

- The MVP is a multi-user system capable of supporting up to nine terminals concurrently, and up to sixteen jobs. Those jobs which are controlled by a terminal directly are called "foreground" jobs, and those which are assigned to a terminal but executing independently of current terminal control are called "background" jobs. Each terminal may have one or more background jobs.

- User memory on the MVP is divided into "partitions" of any required size. Each partition executes independently from any other one. Some partitions may become "global," permitting the program text and global variables there to be used by several other partitions. Additionally, up to 256K bytes of user memory, in up to four "banks," may be purchased.

- The statements $BREAK, $CLOSE, DEFFN @PART, $INIT, $MSG, $OPEN, $PSTAT, $RELEASE PART, $RELEASE TERMINAL and SELECT @PART, as well as the functions $MSG, #PART, $PSTAT and #TERM have been added to BASIC-2.

16.2 OVERVIEW OF THE 2200MVP

The 2200MVP employs a fixed-partition memory scheme to support multiple users concurrently. In a fixed-partition system, user memory is divided into a number of sections or "partitions," each of which can store a separate program. The number of partitions to be created and the amount of memory to be allocated to each partition are specified by the user in a process called "partition generation." This process also involves specifying certain attributes for each partition and supplying the addresses of peripheral devices attached to the system.

During Master Initialization, the 2200MVP is loaded with the BASIC-2 interpreter in the same manner as the 2200VP. Power-on causes the "MOUNT SYSTEM PLATTER" message to be displayed at terminal #1; the operator at that terminal uses the appropriate SF Key to load the BASIC-2 interpreter and MVP operating system from the system platter.

The special utility program "@GENPART" is now loaded and executed at terminal #1. This program leads the system operator through the necessary steps for partition generation by supplying a series of nontechnical prompts that require the user to provide information pertinent to each partition and shared device. A "system configuration" is created by the "@GENPART" utility; the system configuration defines the number of partitions to be established, the attributes to be associated with each partition, and the peripheral devices attached to the system. Once created, a system configuration can be saved on disk to be recalled later and consequently need be defined only once. A variety of different system configurations can be created for different processing requirements; the operator can then select the appropriate configuration to be loaded each day.

When the user has provided all the requested information or when the desired saved configuration has been selected, @GENPART executes the special BASIC-2 statement $INIT. This statement directs the system to allocate resources in the prescribed manner and create the specified system configuration. The MVP is then configured as a multiuser system with each terminal assigned one or more memory partitions.

Further discussion of both Master Initialization and partition generation can be found in the *2200MVP Introductory Manual.*

Functional Overview

The primary goal of a multiprogramming operating system is to allow several users to share a single computer efficiently. To accomplish this objective, the operating system divides the resources of the computer — memory, peripherals, and CPU time — among the users. Once each user has been allocated a share of the computer's resources, the operating system acts as a monitor, allowing each user to utilize the system in turn while preventing individual users from interfering with each other's computations.

On the Wang 2200MVP, each memory partition behaves much like a single-user 2200VP. From the user's point of view, each partition functions independently from the other partitions in the system. Each user may LOAD and RUN BASIC-2 software, compose a program, or perform Immediate Mode operations. As in a single-user environment, the user has complete control over his or her partition. No user on the system may halt execution in, or change the program text of, any partition controlled by another user.

Each terminal may control several partitions executing independent jobs. At any given time, however, only one of these partitions controls the terminal and can interact with the operator. The partition in control of the terminal is said to be in the "foreground." Other partitions assigned to the terminal may continue to execute in the "background" until operator intervention becomes necessary. If a background job attempts to print to the CRT or obtain input from the keyboard, its execution is suspended until the terminal becomes available to it. The terminal becomes available to the waiting background partition when a $RELEASE TERMINAL instruction is executed in the foreground partition either as part of a program or as an Immediate Mode command entered by the user. A foreground partition thus maintains control of the terminal for as long as desired. Therefore, messages from other partitions cannot appear and disturb the CRT display at undesirable times. A background partition may, however, print to a local printer, even though the partition is not attached to the terminal which controls that printer. The $RELEASE TERMINAL statement and foreground/background processing are discussed further in a subsequent section of this chapter.

Although partitions in general function independently of one another, there are situations in which it is useful for two or more partitions to cooperate. Cooperating partitions may share program text and/or data. The sharing of commonly used programs and data by several partitions eliminates needless duplication and produces more efficient use of available memory. The integrity and independence of a partition which contains shared programs or data are maintained by requiring the partition to explicitly declare itself to be global (sharable) before it can be accessed by other partitions. Correspondingly, a partition wishing to access shared text or data in a global partition must identify the desired global partition. Global partitions are discussed in section 16.9.

The image of multiple partitions functioning as completely independent machines is clouded somewhat by the problem of contention for shared peripheral devices. The situation is familiar to programmers accustomed to working with single-user Wang 2200 systems that share one or more disk drives via disk multiplexers. In such systems, it is sometimes necessary for one CPU to request exclusive control of a disk (i.e., to "hog" the disk) while an update is made. Similarly, on the MVP it is necessary for a partition to exclusively control a printer for the duration of the printing of a report; otherwise, one partition's print lines might become unintelligibly mixed with those of another partition. To solve this problem, the concept of disk hog mode has been extended on the MVP to all shared I/O devices. The $OPEN and $CLOSE statements allow a partition to request exclusive control of any device on the system. These statements are discussed in sections 16.4 and 16.10.

To the programmer who regards the MVP system as a whole, it appears that all partitions are executing simultaneously. Because all partitions share a single CPU, however, only one partition can be executing at any given moment. The operating system creates the illusion of simultaneous execution of several programs by rapidly switching from one to the other in turn. In general, the programmer need not be concerned with the details of how the operating system does its job. However, the following presentation may be helpful in giving the user an overall picture of how a multiprogramming system attempts to maximize system utilization while maintaining good user response time. The programmer who is aware of how the operating system performs its job can enhance system performance by applying a few simple programming techniques.

Partitions in the 2200MVP are serviced by the CPU in a "round-robin" fashion, with some additional priorities given for certain I/O operations. Each partition in turn is given a "timeslice" 30 milliseconds (ms) in duration, during which it has exclusive control of the CPU. The CPU has a 30-ms timer which is set at the beginning of the timeslice; at the completion of each BASIC statement (and at various points in the middle of long statements and I/O operations), the clock is checked to see whether the 30-ms timeslice has been exhausted.

When a partition's timeslice has expired, the operating system saves the status of that partition so that it may be restored later when that partition's turn comes around again. The operating system then loads the status of the next partition in line and begins its 30-ms timeslice. The process of halting execution of a partition at the end of its timeslice is called a "breakpoint." The programmer cannot predict in advance when a breakpoint will occur. This is not a serious problem, however, because except for a few cases involving global variables, the occurrence of breakpoints does not concern the programmer.

Timeslices do not always last exactly 30 ms. Unlike many operating systems, the MVP switches users (breakpoints) whenever it is convenient rather than strictly by the clock. This technique reduces the amount of status information that must be saved, giving the MVP low operating system overhead when compared with most other multiuser systems. More importantly, breakpoints may occur in the middle of BASIC I/O statements. If, for instance, the current partition attempts a disk access and the disk is hogged by another partition, this condition is quickly detected and a breakpoint occurs. I/O breakpoints differ from program breakpoints in that the partition is specifically marked as "waiting for I/O". When the partition's turn comes around again, the system takes only a few microseconds to decide whether processing may proceed or whether the partition is still waiting for the I/O device and may be bypassed. Thus, if a printer goes "busy" while it performs some mechanical function or if a partition that does not currently control the terminal attempts to write to the CRT, the system bypasses that partition almost as effectively as if it were removed entirely from the system until the I/O device becomes available.

The CPU is much faster than any of its peripherals. Therefore, breakpointing during an I/O operation allows the MVP to perform a great deal of work with other partitions while the I/O operation is carried out. For example, when a program uses KEYIN to receive data from the keyboard, the MVP can give several timeslices to other partitions between operator keystrokes. Similarly, several partitions can be serviced during a carriage return on a 2221W printer.

To accomplish the overlap of CPU and I/O processing, I/O devices must be buffered and frequently microprocessors must be used to relieve the CPU of responsibility for controlling the peripheral. The most sophisticated of these "intelligent" peripheral controllers is the 2236MXD terminal controller, which handles many I/O operations to the 2236D terminal which otherwise would require the attention of the CPU. For example, the MVP CPU does not perform INPUT or LINPUT statements; instead, it asks the microprocessor in the 2236MXD to perform such operations. Just as in the case of a busy printer, a partition executing an INPUT or LINPUT statement is marked as "waiting for I/O" and receives no CPU timeslices until the INPUT or LINPUT statement is terminated with a carriage return or by depressing a Special Function Key. The 2236MXD also performs line-editing functions at the terminal CRT to move the cursor and to insert and delete characters. In dealing with the 2236MXD, the MVP operating system must perform some address translation because all partitions refer to the terminal keyboard as address /001, to the CRT as address /005, and to the terminal printer as address /204. The operating system ensures that all output for each partition is sent to the proper terminal and all input is received from the proper keyboard.

The following sections of this chapter describe the important multiprogramming features of the 2200MVP in some detail. These discussions are followed by the general formats of all BASIC-2 statements designed for use on the 2200MVP. The final section of this chapter discusses programming considerations for the MVP and suggests some good programming practices.

16.3 USER MEMORY ALLOCATION

Memory on the 2200MVP is divided into two distinct kinds. The operating system, BASIC-2 interpreter, and other system data are contained in *control memory.* The 2200MVP has approximately 60K bytes of control memory, which is completely separate from memory available for user programs and data. *User memory* is available in increments from 16K bytes to 256K bytes. When referring to the memory size of a 2200MVP, only the user memory is considered.

User memory on the 2200MVP is divided into areas known as "banks", of which a maximum of four are allowed. A bank contains a maximum of 64K bytes of user memory; if a system containing from 16K bytes to 64K bytes of user memory is purchased, user memory is contained in only one bank. In this first bank, memory may be added in 16K-byte increments. A system containing more than 64K bytes of user memory contains more than one bank, depending on the amount of memory purchased. Memory in bank two may be purchased in 32K-byte increments only, while banks three and four may be purchased only as complete 64K-byte banks.

General system overhead on the 2200MVP requires only 3K bytes of user memory in bank #1. In banks #2, #3, and #4, 8K bytes in each bank are unavailable for partitions (regardless of the size of that bank), leaving a maximum of 56K bytes available in each of those banks for partitions. A 256K-byte MVP then provides a total of 229K bytes of user memory available for partitions. In addition to the general system overhead, each partition requires 1K bytes of housekeeping information for program control and buffering, leaving the remaining memory allocated for a partition free for programs and data. Partitions must be at least 1.25K bytes in size; they may be allocated space in 256-byte (0.25K-byte) increments up to a maximum of the full extent of user memory in any one bank (exclusive of housekeeping and unavailable areas).

User memory in one bank is inaccessible to user memory in any other bank; thus, a partition may not extend from one bank to another. Figure 16-1, which follows, illustrates memory bank organization.

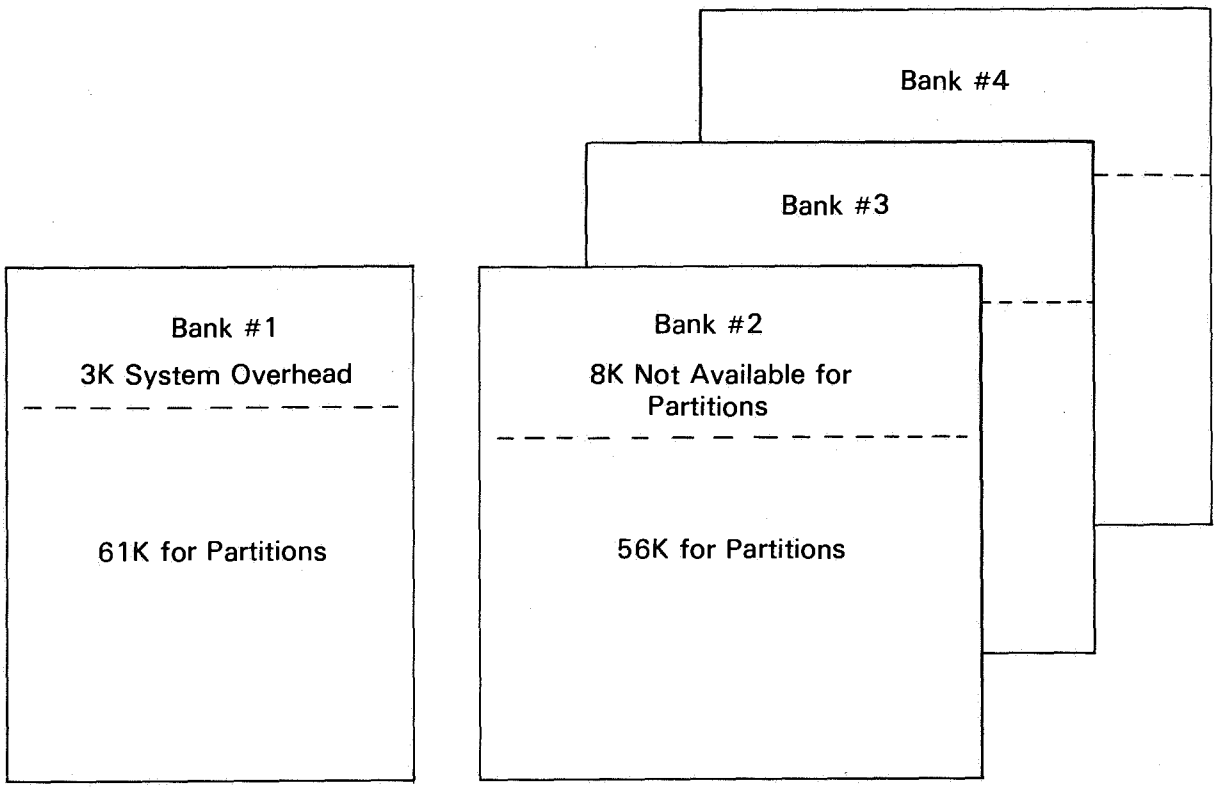


Figure 16-1.
Memory Bank Organization

The MVP permits the user to define one or more "global partitions" within each memory bank. The programs and variables stored in a global partition are accessible to other partitions within that bank. Thus, when the same program is run by two or more users accessing the same memory bank (for example, when several terminals run the same order entry application), only one copy of the program must be stored in that bank.

The availability of global programs can significantly reduce overall memory requirements when several users are running the same program within the same memory bank, because only one copy of the program must be kept in memory. In this case, each user running the global program requires only a small partition within the same bank which contains a subroutine call (branching to the global program) and any variables required for program execution.

A global partition is accessible only by other partitions in the bank where the global partition resides. However, the first available 5K bytes of user memory in bank #1 constitute a special area of memory known as the "universal global" area, which is illustrated in Figure 16-2 below. A global partition which is contained entirely in this area may be accessed by a partition in *any* bank. A universal global partition can be used to store programs and data which are to be shared by all users on the system.

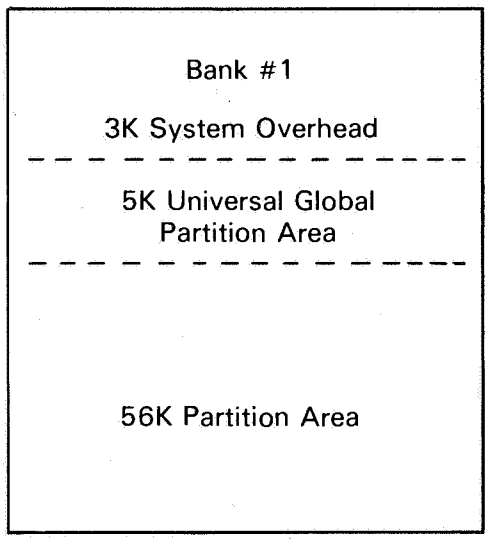


Figure 16-2.
The Universal Global Area

Note that the entire universal global area need not be used for universal global partitions; the only restriction is that a universal global partition reside entirely within that area. For all other purposes, the entire area is treated exactly as all other memory in bank #1.

Consider the multibank system configuration in Figure 16-3 below:

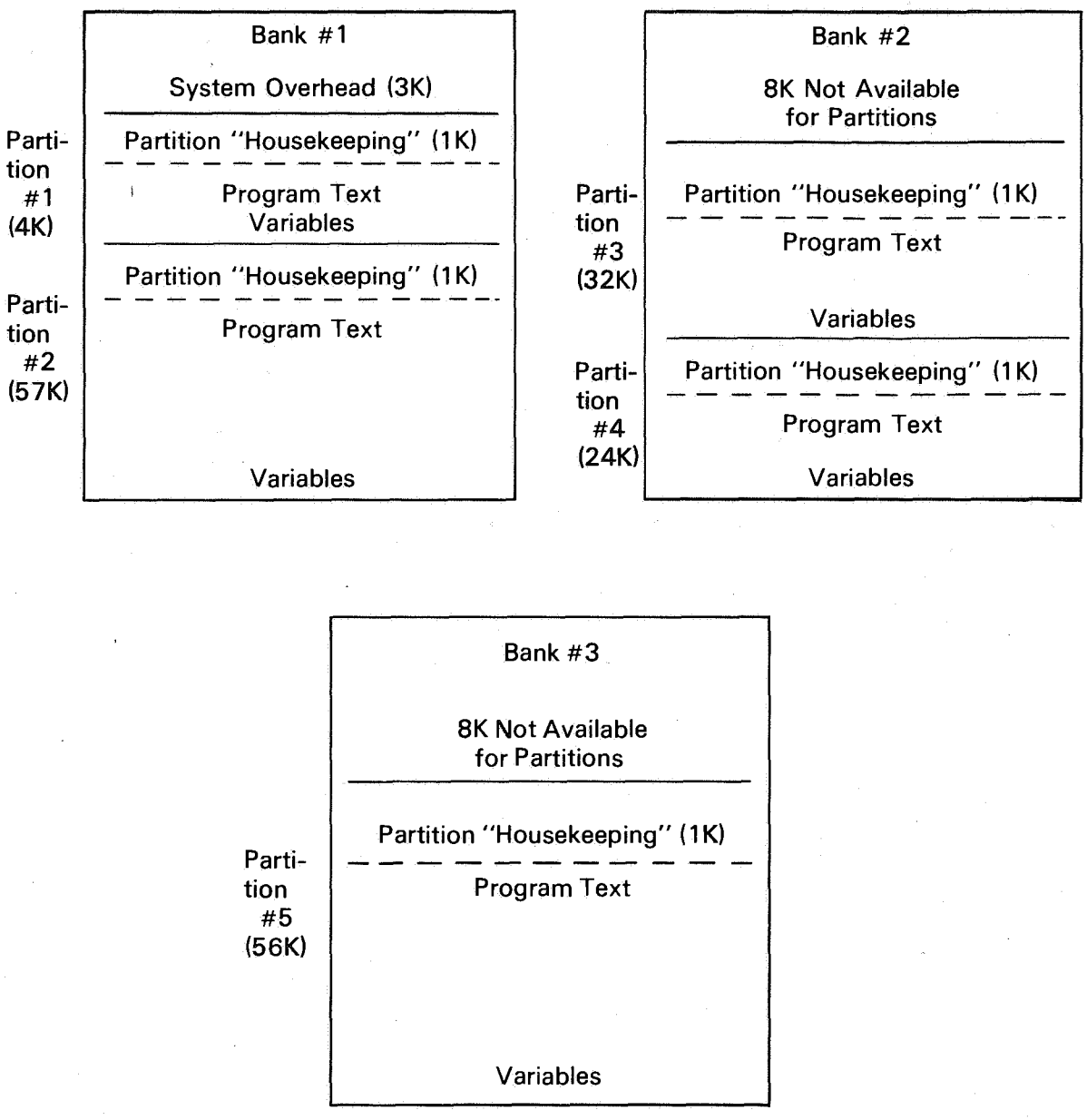


Figure 16-3.
A Multibank System Configuration

If partition #3 were defined as global, it could be accessed only by partition #4 since that is the only other partition in bank #2. However, if partition #1 were defined to be global, it could be accessed by any of the other partitions since partition #1 resides entirely within the universal global area.

The END statement and the SPACE and SPACEK functions apply to the current partition. For example, SPACEK returns the partition size (e.g., 32K for partition #3 above) rather than the total amount of memory in the system. However, before memory has been partitioned, SPACEK returns the total amount of memory available for partitioning in all banks.

16.4 PERIPHERAL ALLOCATION

By default, all peripherals attached to the 2200MVP are available to all users. This situation leads to a conflict if more than one user attempts to use an unsharable device. For example, if two users attempt to print to a line printer simultaneously, the printer will intermix their output. To avoid such a situation, the MVP operating system enables a partition to programmably request exclusive use of a peripheral by specifying the device-address of that peripheral in a $OPEN statement. Once open, the device remains "hogged" by that partition until either a $CLOSE or END statement or a CLEAR, RESET, or LOAD RUN command is executed. Thus, if a disk is "hogged" via a $OPEN statement, only the user who executed that statement may read or write disk files until the device is released by one of the described methods. Note, however, that when an I/O statement references a particular device, the device is automatically hogged for the duration of that statement. Thus, for example, while one partition is listing a program on the printer, other partitions are inhibited from printing until the LIST statement completes execution. In such cases, therefore, it is not necessary to use $OPEN.

At partition generation time, all peripherals attached to the system (other than the terminals and local printers attached to them) must be specified in the Master Device Table. By default, all peripheral devices are available to all partitions. However, a device can be assigned exclusively to a specified partition (until the next system configuration) by entering the number of the partition which is to have control of the device in the Master Device Table. If a partition attempts to use a device which is permanently designated for exclusive use by another partition, ERR P48 is signalled. Console device-addresses (i.e., /005 (CRT), /001 (keyboard), /204 (terminal printers)) are not specified in the Master Device Table. For disk controllers which respond to more than one address, only the primary address must be specified (i.e., /310 but not /B10 or /350). For all other multiaddress controllers, all valid addresses must be listed. For addresses that will differ by the first digit only (device-type), only the normal addresses must be specified. The following device-addresses are reserved:

00: null device (always ready for output)
01-07, 41-47, 81-87, C1-C7: MXD addresses

16.5 AUTOMATIC PROGRAM BOOTSTRAPPING

Normally, after partition generation is performed, the "READY (BASIC-2)" message is displayed on each terminal and the system waits for a command to be entered via the terminal keyboard. The operator can then load and run a program. Alternatively, the MVP operating system provides an automatic means for a program to be loaded and run immediately after partition generation without operator intervention. If a program name is specified for a partition during the partition generation procedure, that program will be automatically loaded and run when partition generation is performed. The programs to be loaded must all reside on the platter currently identified by the default disk address (stored in slot #0 in the Device Table). Unlike the 2200VP, the 2200MVP initially sets the default disk address to the platter address from which BASIC-2 is loaded. It is therefore most convenient for automatically bootstrapped programs to reside on the same disk platter as BASIC-2 (@@) and @GENPART.

Automatic program bootstrapping is particularly useful for setting up background or global partitions and for forcing terminals to execute particular BASIC-2 software.

16.6 DISABLED PROGRAMMING

The MVP provides security capability that allows programming and Immediate Mode operations to be inhibited for any partition(s). A terminal attached to a partition in "Disabled Programming" Mode is kept under BASIC-2 software control, effectively preventing inadvertent or unauthorized use of data files and programs from that terminal. After partition generation or RESET, the operator can only load and execute the program "START". This program, which is user written, may provide a menu of operations that can be performed from that terminal and may require passwords for certain operations.

In Disabled Programming Mode, the terminal is prevented from entering program lines and most Immediate Mode statements. Attempts to perform such operations are illegal and generate ERR A08.

However, the following commands and Immediate Mode statements are allowed:

RESET
CLEAR
RUN
HALT/STEP
CONTINUE
LOAD RUN — a program name cannot be specified ("START" is always implied).

"Disabled programming" for a partition is specified during partition generation at Master Initialization time (see the discussion of partition generation in the *2200MVP Introductory Manual).* Programming remains disabled for a partition until the system is reconfigured.

16.7 BROADCAST MESSAGE

A message can be set up by terminal #1 to be displayed on each terminal's CRT whenever the "READY" message is normally displayed (i.e., after RESET or CLEAR). The user-defined message is displayed on line 0 of the CRT immediately above the "READY" message. The message text may also be examined by a BASIC-2 program.

It is sometimes useful to display a message at each terminal to inform users of some condition concerning the system. The utility program "@GENPART", executed at Master Initialization, allows the operator to specify a broadcast message. The user at terminal #1 alone can alter the system broadcast message at any time by using the $MSG statement. For example:

```
$MSG = "*** SYSTEM WILL GO DOWN AT NOON ***"
```

After RESET, the display would appear as shown below:

```
*** SYSTEM WILL GO DOWN AT NOON ***
READY (BASIC-2) PARTITION 01
:_
```

The message is available to any program in the system via the $MSG function. For example:

```
10 A$ = $MSG
```

sets A$ equal to the broadcast message. The program can then display the message whenever convenient.

16.8 FOREGROUND AND BACKGROUND PROCESSING

A single terminal can be used to run programs in more than one partition. The terminal can be switched from one partition to the next in order to initiate program execution and to perform any necessary user/program interaction. If a program in one partition attempts to communicate with the terminal but the terminal is attached to another partition, program execution is suspended until the terminal becomes attached to this partition.

The partition currently attached to a terminal is referred to as the "foreground" partition, while each partition assigned to a terminal but not currently attached to it is referred to as a "background" partition. When the terminal is switched from one partition to another, the partition currently in the foreground is moved into the background, and a background partition moves into the foreground. With foreground/background processing, a single terminal can run several jobs requiring minimal user interaction in the background and a highly interactive job such as order entry or on-line inquiry in the foreground.

During "partition generation" at Master Initialization time, each partition normally is assigned to a terminal. (A specific exception is a partition which is assigned to terminal #0 and is, therefore, not assigned to any terminal.) Although each partition is assigned to a single terminal, each terminal may have several partitions assigned to it. Initially, the terminal is attached to the lowest numbered partition assigned to it. The terminal remains attached to that partition until a $RELEASE TERMINAL statement is executed in that partition or a $RELEASE PART statement is executed by that terminal. When

$RELEASE TERMINAL is executed, the current foreground partition is placed in the background. The MVP operating system can then attach the terminal to the next partition waiting to communicate with it. Each partition assigned to the terminal has equal priority for using the terminal; the partitions are scanned in a round-robin fashion, assuring that each partition will have access to the terminal as $RELEASE TERMINAL statements are executed.

Optionally, a terminal can be released to a specified partition by using the "TO partition" parameter in the $RELEASE TERMINAL statement. In this case, the terminal is attached to the specified partition even if the partition has not attempted to communicate with the terminal and one or more other partitions are trying to communicate with it. The partition requested by the "TO" parameter *must* be assigned to either the calling terminal or to terminal #0. The $RELEASE TERMINAL TO partition statement is the only way to regain control of a partition assigned to terminal #0 unless RESET is pressed at a terminal with no assigned partitions.

When a $RELEASE PART statement is executed, the current foreground partition is assigned to terminal #0, and the next available partition is assigned to the calling terminal. If no partitions assigned to the terminal are available, the terminal is attached to the lowest numbered partition assigned to terminal #0 (other than the one just released). If the only available partition is released via $RELEASE PART, keying RESET at the terminal reattaches the partition.

Occasionally, an operator needs to request the job status of a background program. The background partition may, if desirable, output data directly to a local printer attached to the terminal. Alternatively, the background program can set status information in global variables. The foreground program can display the status directly or, more likely, call a global subroutine in the background partition to display the status. (See the following description of global text and global variables in section 16.9.)

If all partitions assigned to a particular terminal release the terminal, the user cannot access programs in any partition until at least one of the partitions attempts to output to the terminal. In particular, the terminal keyboard is active only when the terminal is attached to a partition. In order to prevent a lockout situation, the RESET and HALT Keys always remain active. If no partition is attached to the terminal when HALT or RESET is keyed, the lowest numbered partition assigned to the terminal will first be halted or reset and then automatically attached to the terminal. Therefore, it is good practice to set up a small control partition as the lowest numbered partition when no foreground job is to be run from the terminal.

Some background jobs may have no terminal I/O requirements other than to periodically display their current status. To avoid having such jobs "hang" while awaiting availability of the terminal, the $IF ON statement can be used to determine if the terminal currently is attached to the partition. If $IF ON finds that the terminal is attached, the status information is displayed; if not, the program branches to perform normal processing before testing for the terminal again. The use of $IF ON to test for an attached terminal is further discussed in Chapter 15, section 15.2.

Background Terminal Printer Operation

Printers can be physically attached to either a printer controller in the MVP CPU or directly to a 2236D-type terminal. Printers attached to the CPU are referred to as "system printers" because they generally are available to any partition, while printers attached directly to a terminal are called "terminal printers" because they are only available to partitions assigned to the terminal.

Normally, these terminal printers are used only by the current foreground partition. The MVP operating system, however, also supports background printing to a terminal printer (device-address /204). This feature allows a partition in the background to output to the terminal printer while the current foreground partition retains control of the CRT and keyboard. Any partition assigned to the terminal can output to the printer. The $OPEN command should be used to "hog" the printer to prevent more than one background partition from outputting to it simultaneously. Keying RESET flushes any printer output buffered in the 2236MXD controller or terminal only if the foreground partition has $OPENed the printer. Note that $RELEASE TERMINAL releases the terminal CRT and keyboard from the current foreground partition, but does not affect the terminal printer or the partition outputting to it.

Background printing is not recommended to terminals using communication rates less than 1200 baud since keystroke echoing may noticeably be delayed. Release 1.5 or later of the MVP operating system, release 1.4 or later of the MXD firmware, and release 2.0 or later of the 2236D firmware are all required for proper background printing to the terminal.

16.9 GLOBAL PARTITIONS

The 2200MVP provides the capability for one or more partitions to define themselves as "global." Global partitions contain global programs (which can be accessed from other partitions) and global variables (which can be referenced by other partitions). Global partitions thus serve an important function in permitting the sharing of common programs and data among many partitions.

With the exception of a partition contained entirely within the first 5K bytes of bank #1, a partition can be global only with respect to partitions in the same bank. Partitions within one bank may reference programs or variables in global partitions in that bank, but in no other bank. However, programs and variables in any global partition contained entirely in the "universal global" area of bank #1 can be referenced by any partition.

A partition defines itself as "global" by executing a DEFFN @PART statement. DEFFN @PART must specify the name which will be used by other partitions to identify the global partition. For example, the statement

```
10 DEFFN @PART "GLOBAL"
```

defines the current partition as global and assigns it the name GLOBAL.

A partition which attempts to access a global partition must first SELECT the particular global partition to be accessed by executing a SELECT @PART statement. SELECT @PART specifies the name of the global partition to be accessed. For example, the statement

```
10 SELECT @PART "GLOBAL"
```

selects the partition named GLOBAL for use by the current partition for all subsequent global subroutine calls or references to global variables.

Access to a global partition may be restricted only to certain designated terminals. In this case, the numbers of the terminals which are to have access to the global partition must be specified in the DEFFN @PART statement. For example, the statement

```
10 DEFFN @PART "GLOBAL" FOR 1, 2, 3
```

defines the current partition as global and specifies that it will be accessible only to partitions assigned to terminals #1, #2, and #3 (including any background partitions associated with those terminals). Partitions associated with all other terminals on the system would receive an error message if they attempted to access global program text or global variables in GLOBAL.

A nonglobal partition may access more than one global partition by executing a new SELECT @PART statement for each global partition to be accessed. Only one global partition can be selected at any time, however; the execution of each SELECT @PART statement automatically deselects the previously selected global partition.

Global Program Text

The program text in a global partition is called "global program text" because it can be shared by other partitions via global subroutine calls. Such sharable text is often described as "reentrant" because it can be reentered and reexecuted by several partitions concurrently. A partition issues a global subroutine call by executing a GOSUB' statement which references a DEFFN' subroutine in the global partition. Prior to executing the GOSUB', the calling partition must have executed a SELECT @PART statement identifying the global partition in which the global subroutine is to be found.

When a GOSUB' statement is executed, the system first searches the current partition for a corresponding DEFFN' subroutine. If no corresponding DEFFN' is found, the system proceeds to search the last-SELECTed global partition and executes the corresponding DEFFN' subroutine in the global partition. The global text is executed until a RETURN statement is encountered, at which point execution of the calling program is resumed.

Each calling partition must declare its own set of variables for the global program text. Such variables, referred to as "local" variables to distinguish them from a special class of variables called "global" variables, are referenced and modified in each calling partition by the global program text.

Figure 16-4 shows the program flow of two partitions sharing global text in a third partition:

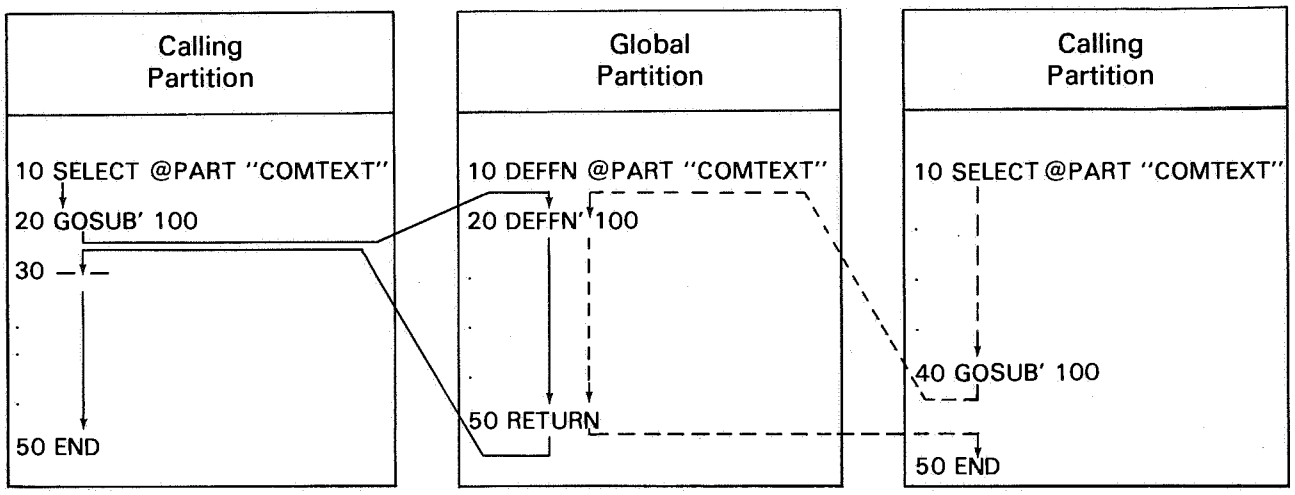


Figure 16-4.
Two Partitions Accessing Global Program Text

While a global subroutine is executing, all line-number, DEFFN', and user-defined function references apply to the global text. Thus, the global text may have the same line-numbers and DEFFN's as the calling program. READ statements in the global text will access DATA in the calling program *until* a RESTORE statement is executed in the global subroutine. After returning from a global subroutine, READ statements in the calling program will continue to reference DATA in the global partition (assuming a RESTORE was executed in the global partition) until a RESTORE is executed in the calling program. The above strategy enables both the calling and the global text to reference their own and each other's DATA statements. Local variables referenced in the global subroutine must have been defined in the original calling program or an error will result when the statement referencing such a variable in the global subroutine is executed.

Local Variables Referenced in a Global Partition

Local variables (normal BASIC variables) referenced in a global partition following the DEFFN @PART statement are *not* entered into the global partition's Variable Table and thus are not physically located within the global partition. Local variables are entered into the Variable Table in a global partition only if:

1. They are declared explicitly in a DIM or COM statement or
2. They are referenced *prior* to DEFFN @PART.

During execution of a global subroutine, all references to local variables refer to the Variable Table of the partition from which the subroutine execution was initiated. When a calling partition issues a global subroutine call, therefore, references to local variables in the global subroutine actually refer to

variables in the original calling partition. Therefore, all local variables used by a global subroutine must be declared in each originating partition which accesses the global text. The global subroutine references and modifies the local variables in each originating partition exactly as if it were loaded and running in that partition.

In the more general case of nested global subroutines, references to local variables in all nested subroutines refer back to the original calling partition exactly as if all global subroutines were loaded and running in the originating partition.

Nesting Global Subroutines

Nesting of global subroutines is accomplished by selecting a global partition within a global subroutine, as illustrated in Figure 16-5 below:

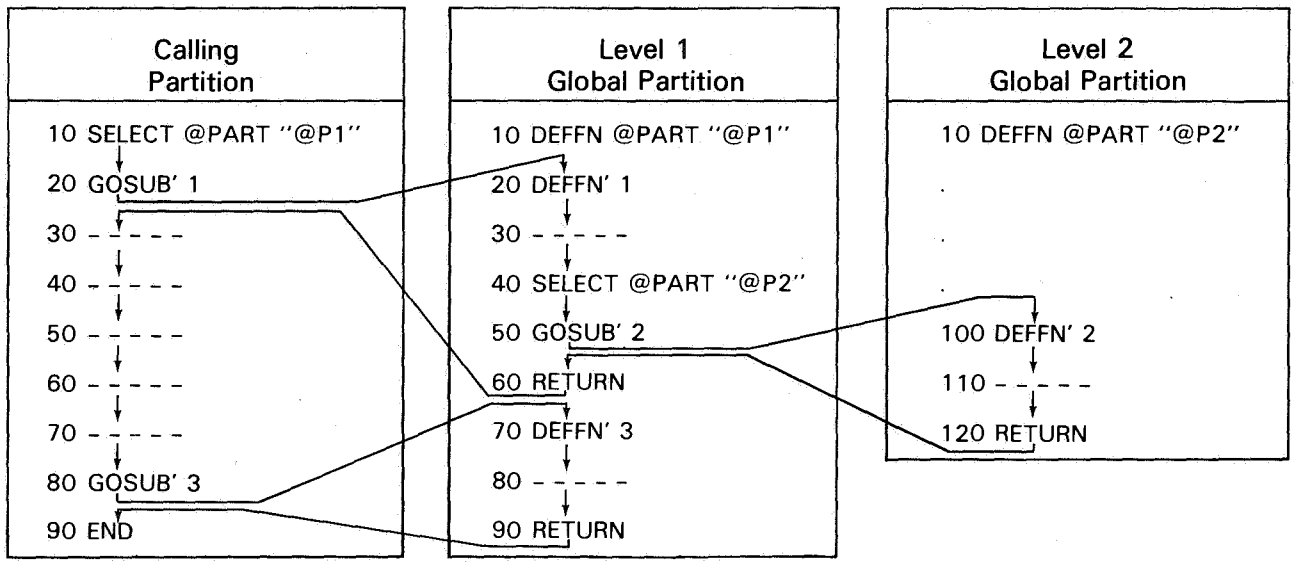


Figure 16-5.
Nesting Global Subroutines

It is important to note that it is not possible to use this nesting technique to access a partition in one memory bank from a partition in another via the universal global area. For example, a partition in bank #2 is not allowed to call a global partition in the universal global area, which in turn calls another global partition in bank #1. Thus, in general, a global partition in the universal global area which is to be accessed by partitions in other memory banks may not reference partitions outside the universal global area.

When a global subroutine is called, the pointer to the currently selected global partition is placed on the subroutine stack along with the subroutine return address. Executing the subroutine RETURN causes the global partition pointer to be restored. Thus, a global subroutine can select another global partition without affecting the pointers of the originating partition. (See the subsection "Further Details on Global Partitions" for a more detailed discussion.)

Some Programming Considerations for Global Program Text

A program which more than one user will run can be loaded into a global partition. Each calling program need only consist of a declaration of the variables used (DIM, COM), a SELECT of the global partition containing the program, and a GOSUB' to the beginning of the shared program. Such an arrangement makes efficient use of memory by eliminating the need to maintain a separate copy of the entire program in each user's partition.

Global subroutines are also useful for controlling access to shared resources. For example, if several

users must update a common disk file, a shared global subroutine could be used to perform all file accesses. Whenever a user wishes to access the file, a call is made to the global subroutine. In this way, one routine coordinates the activities of all users with respect to the shared file.

The DEFFN @PART statement used to define a global partition is an *executable* statement. Until the program in a global partition is run and DEFFN @PART is executed, the partition is not defined to be global. If a calling partition executes a SELECT @PART statement before the corresponding DEFFN @PART statement has been executed in the global partition, an error will result. In general, therefore, a global program should be specified as a "bootstrap" program for the global partition. In this case, the global program is automatically loaded and run when the system is Master Initialized. If the calling programs are loaded and run by operators at the various terminals, a calling program will never issue a global subroutine call before the global program is run.

If, however, the calling programs are also bootstrapped into their respective partitions, there is no guarantee that one or more of the calling programs will not be loaded and run before the global program. To avoid having a calling program terminate with an error in such a case, the ERROR clause can be used following the SELECT @PART statement. For example:

```
10 SELECT @PART "GLOBAL": ERROR $BREAK: GOTO 10
```

This line continues to loop until the global partition is defined. The $BREAK statement is included to reduce the wait time. If the global partition is not defined, the current partition does not continue to loop and check for its full timeslice; instead, it "puts itself to sleep" temporarily, giving up some of its processing time to enable the system to more rapidly load and run other programs, including the global program. When the period defined by $BREAK is ended, the current partition again loops to execute SELECT @PART. It is likely that the global program is now resident, and normal execution can begin.

Note that while many terminals can communicate with a global program, only *one* terminal can modify the global text. Global text, like nonglobal text, can be modified *only* by the terminal to which the partition containing the text is assigned. There is no way for operators at other terminals to modify or list the global text.

NOTE:

An "@" precedes the line-number of a line of global program text whenever it is displayed by the system (e.g., for error messages or when stepping through a global program). For example:

```
@100 X = Y/Z
          ↑ ERR C62
```

A global program should not be cleared or modified while it is being accessed by a calling partition. If such an operation is performed, it will produce an unpredictable error message in the calling partition.

Global Variables

In addition to the standard types of variables used in each partition, called "local" variables, the 2200MVP supports a special class of variables called "global" variables. Global variables which are declared in a global partition can be referenced by programs in other partitions, thus providing a convenient medium for interpartition communication and data sharing.

Global variables are identified by prefixing an at-sign (@) to the variable name. The following are examples of global variable names:

```
@A
@B$
@N(2)
@B2$(5,5)
```

Global variables constitute a completely separate set of variables from standard or local variables. Thus, for example, @A$ and A$ are separate and distinct variables.

Unlike local variables, global variables cannot be declared implicitly or by reference. They must be declared explicitly in a DIM or COM statement. Thus, although a global variable may be referenced in several partitions, it is actually allocated physical storage and entered into the Variable Table *only* in the partition in which it is explicitly declared. A global variable which is declared in a global partition can be referenced in one or more nonglobal partitions. During program execution, when a reference is made to the global variable, it is the single entry for that variable in the global partition's Variable Table which actually is referenced.

Before a nonglobal partition can reference a global variable, however, it must identify the global partition containing the variable by executing a SELECT @PART statement. Following execution of SELECT @PART, all references to global variables will access the designated global partition.

Figure 16-6 shows the Variable Tables for a global partition which declares two global variables (@X and @Y) and two nonglobal partitions which reference those variables.

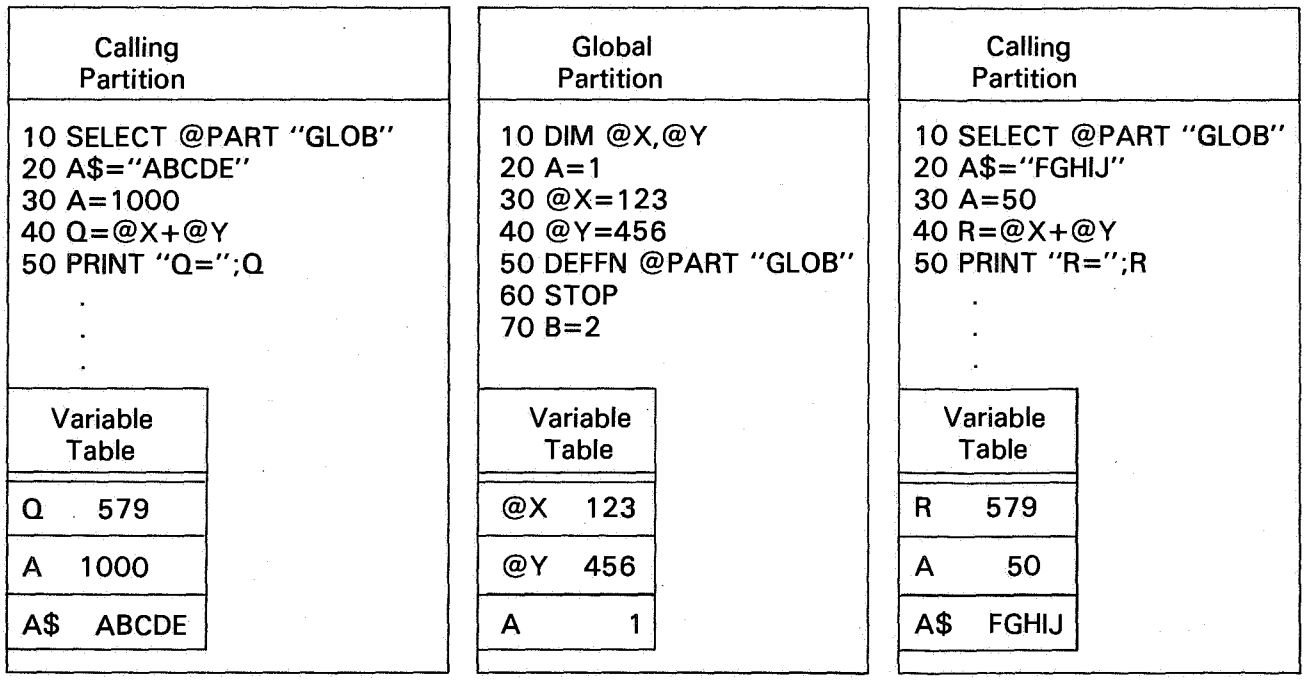


Figure 16-6.
Variable Table Entries in Global and Nonglobal Partitions for Global and Local Variables

Notice that, although the global variables @X and @Y are referenced in all three partitions, they are entered into the Variable Table *only* in the global partition. Notice also that both @X and @Y are explicitly declared in a DIM statement, although they are numeric scalars. Global variables *must* be declared explicitly in the global partition, irrespective of their variable type. Notice also that B does not appear in the Variable Table of the global partition; local variables may not be implicitly defined following a DEFFN @PART statement.

Declaring Global Variables in a Nonglobal Partition

Global variables which are explicitly declared in a nonglobal partition will be entered into that partition's Variable Table. Such variables are not truly "global," however, because they are not accessible to other partitions; they constitute, in effect, a second set of local variables available to each partition.

It is not good programming procedure to use global variable names for local variables, however, because this policy can make documentation of the program very confusing, particularly if the program also references global variables in a global partition. *All* references to global variables following the SELECT @PART statement are directed to the selected global partition, *even* if the variable is declared within the calling partition. Thus, references to global variables following a SELECT @PART statement will produce an error if the variable is not declared in the global partition, even if the variable is declared in the calling partition.

Using Global Variables for Task Control

Sometimes in a multiuser environment it is necessary to allow only one user at a time to perform a specific task (e.g., in a KFAM key file update, other users must be blocked from accessing the key file until the update is complete). A test-and-modify operation is required to determine if an operation is currently in progress for another user and, if not, to set the state to "operation in progress" for this user. This operation can be accomplished by using an IF statement with a global variable which indicates whether or not the operation is in progress for some user. The following example illustrates such a test-and-modify routine:

```
.
.
.
100 $BREAK: IF @T = 0 THEN @T = #TERM : ELSE GOTO 100
110 REM PERFORM OPERATION
.
.
.
980 REM OPERATION COMPLETE
990 @T = 0
```

The global variable @T is a control variable which is set to the terminal number of the terminal for which the operation is currently being performed or to zero if no operation is in progress. Line 100 inhibits the current user from performing the operation until no other users are performing a conflicting operation (i.e., until @T = 0); the $BREAK statement relinquishes the remainder of the current partition's timeslice in order to enable the conflicting task to complete more quickly. Line 990 sets the control variable to 0, allowing another user to perform an operation. The above strategy works only if all programs with conflicting operations contain the line 100 logic. Test-and-modify operations are thus best included as part of the shared program text in the global partition.

The test-and-modify operation must be performed in a single IF statement. If the test (IF @T = 0) were done in one statement and the modify (@T = #TERM) in another, the MVP operating system might switch partitions between the execution of those statements, enabling another user to modify @T before the current partition has the opportunity to do so. In addition, it is necessary that the statement following THEN and ELSE execute to completion before a breakpoint occurs. Otherwise, the modification performed by this statement will be incomplete when the system switches partitions. I/O statements and statements which take an exceptionally long time to execute may be interrupted with a breakpoint before execution is complete. Such statements should not be used in a test-and-modify

operation. Certain statements always execute to completion, with no possibility of a breakpoint during execution. They include:

FOR	ON/GOTO	$RELEASE TERMINAL
GOSUB	ON/GOSUB	ROTATE
GOSUB'	READ	$TRAN
GOTO	REM	$MSG
LET	RESTORE	SELECT @PART
NEXT	RETURN	DEFFN @PART

Further Details on Global Partitions

The preceding discussion of global partitions has concentrated on functional descriptions of global program text and global variables and practical considerations for their use. This subsection provides a more comprehensive and systematic explanation of the relationship between nonglobal or "calling" partitions and global partitions.

It may be helpful to preface the discussion of global subroutines with a brief description of how the system oversees the execution of a DEFFN' subroutine within the current partition. Consider, for example, the following program segment:

```
10 DEFFN'1 (A,B)
20 C = A/B
30 RETURN
40 GOSUB'1 (10,5)
50 PRINT C
60 END
```

This program consists of a DEFFN' subroutine (lines 10-30) and a main program which issues a call to the subroutine, prints the result, and ends (lines 40-60). As written, however, the program will not run. Because the subroutine is placed before the main body of the program, the system will attempt to execute it first, and an ERR C63 (attempt to divide by zero) will be signalled at line 20. (The variable B, having been assigned no value, is initialized to 0.)

It is necessary then to remove the subroutine from the mainstream of execution and ensure that it is executed only when explicitly called from the main program. This objective can be achieved in two ways:

```
 5 GOTO 40
10 DEFFN'1 (A,B)
20 C = A/B
30 RETURN
40 GOSUB'1 (10,5)
50 PRINT C
60 END
```

```
40 GOSUB' 1 (10,5)
50 PRINT C
60 END
70 DEFFN'1 (A,B)
80 C = A/B
90 RETURN
```

In the first instance, a GOTO statement is inserted at the beginning of the program to branch around the subroutine. In the second, the subroutine's lines are renumbered to follow the main body of the program. In each case, the result is the same: the subroutine is placed outside the normal sequence of execution and can be executed only when an appropriate GOSUB' is encountered in the main program. Although the location of the subroutine within the program is flexible, this example illustrates that the subroutine must be placed outside the normal sequence of execution. There would be no disruption to the logic of the program if the subroutine were lifted entirely out of the current partition and placed in a global partition as long as the system could transfer execution from the current partition to the global partition and back again.

Before addressing the issue of how a program in one partition is connected to a subroutine in another partition, however, it may be instructive to consider how a program and a subroutine are connected in the same partition. When a GOSUB' statement is executed, the system must perform two basic tasks:

1. It must "remember" the location of the statement following the GOSUB' statement so that the normal sequence of execution can be resumed at that point upon returning from the subroutine and

2. It must locate and execute the specified DEFFN' subroutine.

The system "remembers" the location of the statement following GOSUB' by placing this information on an internal stack prior to branching to the subroutine. The internal stacks and their uses are described in some detail in Chapter 2, section 2.4. Each partition has two stacks, an Operator Stack (kept in the 1K of "housekeeping" area reserved in each partition) and a Value Stack (kept in the user-memory portion of the partition). Because these two stacks operate in parallel, it will be simpler to limit the discussion to the Value Stack only.

The manner in which the Value Stack is used cannot be understood until one additional concept is introduced, the "text pointer." A program is a section of text consisting of one or more program lines, each of which contains one or more statements. In the normal sequence of execution, the program is executed in line-number sequence, with multiple statements on a line executed from left to right. The system keeps track of where it is currently executing in a program with a special pointer called the *text pointer,* which always points to the next statement to be executed. As each statement is executed, the text pointer is automatically incremented to point to the next statement in sequence. For purposes of discussion, the text pointer consists of a line-number and a statement number. For example, consider the following line of program text:

```
10 A = 100: PRINT A
```

When the statement "A = 100" is executed, the text pointer is automatically incremented to point to the next statement, "PRINT A". Thus, during execution of the statement "A = 100", the text pointer would have the value:

```
text pointer = 10,2
```

indicating that the next statement to be executed is the second statement on line 10.

When a GOSUB' statement is executed, the current value of the text pointer is saved on the Value Stack and the system searches the program for a corresponding DEFFN'. If the DEFFN' is found, its location replaces the current value of the text pointer, and execution is passed to that point. When the subroutine RETURN statement is executed, the system retrieves the old text pointer from the Value Stack, places it in the current text pointer, and resumes execution at that point. This process is illustrated graphically in Figure 16-7 which follows.

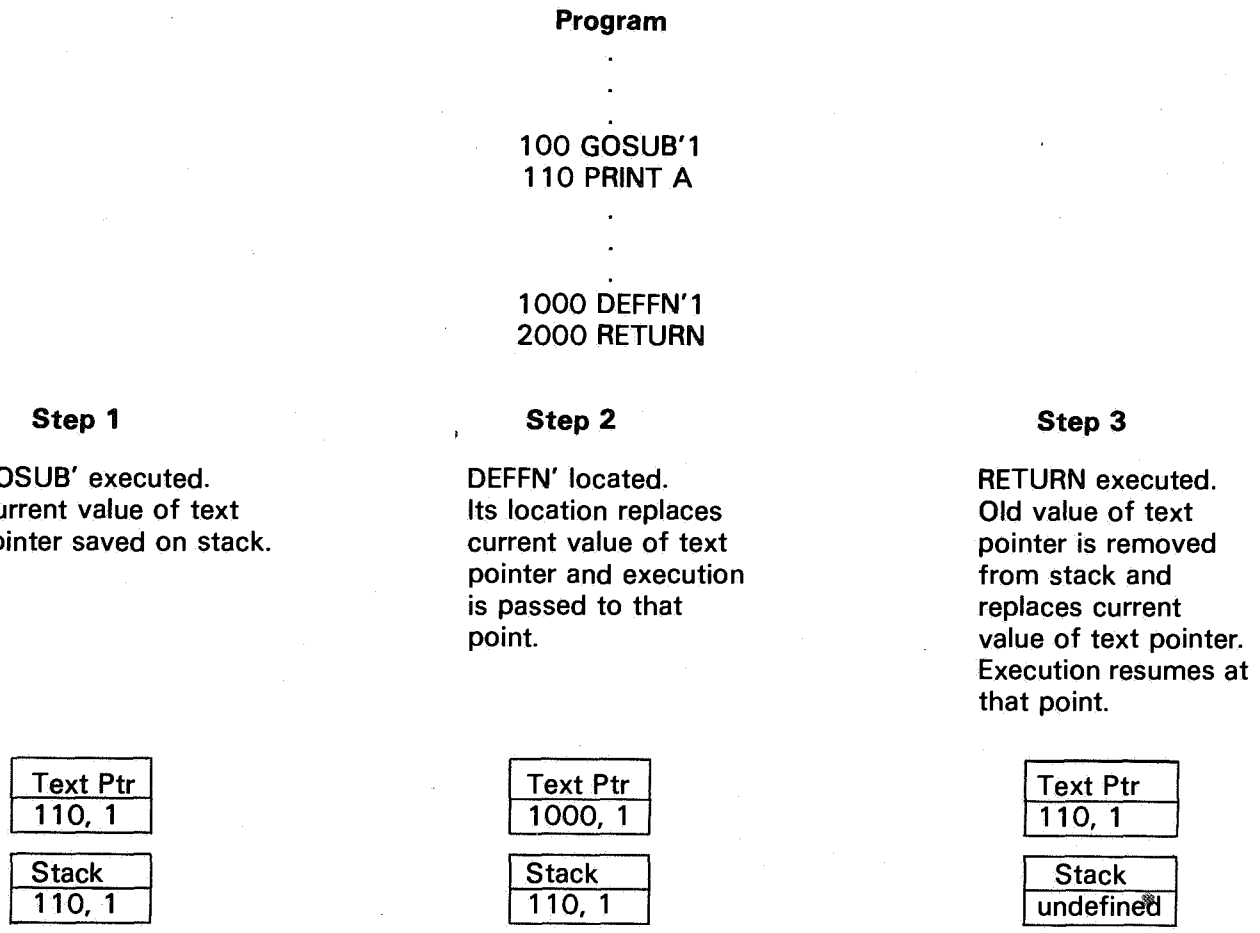


Figure 16-7.
Use of Text Pointer and Stack To Control
Flow of Execution Following a Subroutine Call

When program execution is restricted to a single partition, the text pointer is essentially all the information required to control the flow of execution. In order to execute global subroutines, however, the system requires some additional information to identify which partition contains the currently executing program text. This additional information, together with the text pointer, is stored in a special table referred to as the Pointer Table. A Pointer Table is maintained for each partition. Figure 16-8 below illustrates the contents of the Pointer Table.

Text Pointer	
Text Partition#	
DATA Partition#	
Global Partition#	
Originating Partition#	
Terminal#	

Figure 16-8.
The Pointer Table

Before proceeding with a detailed explanation of how these values are used to control the execution of global subroutines and access to global variables, it will be helpful to introduce two new concepts, "job flow" and "originating partition."

The term "job flow" refers to the sequence of execution followed by a job from beginning to end. The job flow may be restricted to a single partition, or it may extend across several partitions via global subroutine calls. The term "job" is preferred to "program" here because the term "program" is too closely associated with the contents of a single partition. When a global subroutine call is made, the global text is executed exactly as if it were appended to the calling text within the calling partition. The "job" may be regarded, therefore, as the combination of all nonglobal and global program text considered as a logical unit.

The term "originating partition" refers to the partition from which execution of a job originates. Each job has one, and only one, originating partition, and each partition can originate at most one job. For each job in the system, the originating partition of that job maintains the Pointer Table which controls the execution of the job. Even when an originating partition issues a global subroutine call which passes execution to a global partition, execution of the job is controlled by the Pointer Table in the job's originating partition. The global partition is completely "passive" in this situation, serving in effect as nothing more than a physical extension of the originating partition. The job flow between originating and global partitions is diagrammed in Figure 16-9.

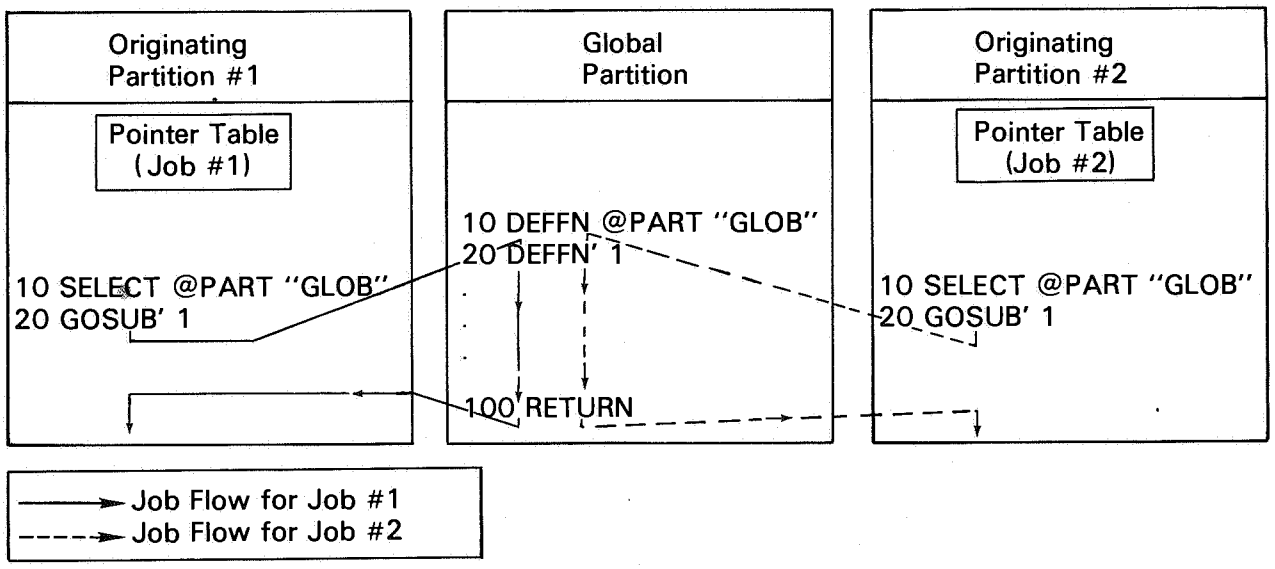


Figure 16-9.
Job Flow Between Originating Partitions and Global Partition

Note that there is no rule to prevent a global partition from originating its own job, which would execute independently of other jobs using the global text in that partition. The RUN command used to resolve the global partition initiates a job flow which either undertakes some useful task or terminates with STOP, $BREAK!, END, or end of program. Note, too, that global partitions can be nested. In this case, no matter how many levels of nesting are performed, the Pointer Table in the originating partition always controls the job's execution.

The manner in which the Pointer Table is used to control the execution of a job can now be explained. Each item in the Pointer Table is used to control the execution of a particular statement or class of statements. During job execution, certain items can be modified by particular statements to reference different partitions. Initially, all items in the table point to the current partition. For example, immediately following Master Initialization, the Pointer Table in partition #2 (assigned to terminal #4) would have the values shown in Figure 16-10.

Text Pointer	0,0
Text Partition#	2
DATA Partition#	2
Global Partition#	2
Originating Partition#	2
Terminal#	4

Figure 16-10.
Pointer Table for Partition #2 Following Master Initialization

The last two items in the table, the Originating Partition# and the Terminal#, are constants which are set at Master Initialization and normally do not change unless the system is reconfigured. The other items can be modified dynamically during job execution. The meanings and uses of all items are described below:

- The Text Pointer — is updated by the system each time a statement is executed to point to the next sequential statement. It is modified by any branching statement (e.g., GOSUB, GOTO, GOSUB') to point to the branched-to statement.

- The Text Partition# — is the number of the partition to which the text pointer applies (i.e., it is the number of the partition containing the currently executing text). It is modified by a GOSUB' statement whenever a branch is made to a DEFFN' in a global partition. In this case, GOSUB' sets the Text Partition# equal to the Global Partition#.

- The DATA Partition# — is the number of the partition containing DATA statements referenced by a READ statement. It can be modified by a RESTORE statement, which always sets it equal to the current Text Partition#.

- The Global Partition# — is the number of the currently selected global partition. It is modified by a SELECT @PART statement. It is the partition searched by GOSUB' for a corresponding DEFFN' if the DEFFN' cannot be found in the text partition. It is also the partition used for all global variable references.

- The Originating Partition# — is the number of the partition in which execution of the job originates and the Pointer Table is stored. The Originating Partition# is a constant for each partition, changed only by reconfiguring the system. It is used for all local variable references, for LOAD operations, and for all system commands issued by the terminal user. Any job's Originating Partition# is returned by the #PART function.

- The Terminal# — is the number of the terminal assigned to the originating partition. Like the Originating Partition#, it is set at configuration time and generally is not modified except by reconfiguring the system. (However, the Terminal# can be altered upon execution of a $RELEASE PART statement.) It is used for all CRT, keyboard, and local printer I/O operations performed during job execution, including CO, CI, PRINT, LIST, INPUT, LINPUT, and KEYIN. For any partition, its Terminal# is returned by the #TERM function.

This information is summarized in Table 16-1 which follows.

Table 16-1.
Functions of Pointer Table Items

Item	Modified By	Used By
Text Pointer	Instruction executions, branching instructions	Program execution control
Text Partition#	GOSUB', RETURN	Text Pointer
DATA Partition#	RESTORE	READ
Global Partition#	SELECT @PART, RETURN	Global variable reference, GOSUB'
Originating Partition#	Fixed	Local variable reference, LOAD, system commands issued by terminal user
Terminal#	Fixed	Terminal/local printer I/O operations during job execution

When a GOSUB' statement is executed, the text partition is first searched for the corresponding DEFFN'. If the DEFFN' is not located, the global partition is searched. If the DEFFN' is found in the global partition, the following sequence of events occurs:

1. The current values of the text pointer, Text Partition#, and Global Partition# are taken from the Pointer Table in the originating partition and saved on the Value Stack in that partition.

2. The Text Partition# is set equal to the Global Partition#.

3. The text pointer is set to the location of the DEFFN' in the global partition, and execution is passed to that point.

All local variable references use the Originating Partition#, which always points to the originating partition. Thus, all references to local variables in the originating partition or in any global partition at any point during job execution refer back to the originating partition of that job.

When a RETURN statement is executed to return from the global subroutine, the following events take place:

1. The old text pointer, Text Partition#, and Global Partition# values are removed from the stack and replaced in the Pointer Table.

2. Execution is passed to the statement pointed to by the text pointer (i.e., the statement following GOSUB' in the calling partition).

Neither SELECT @PART nor GOSUB' modify the DATA Partition#, nor does RETURN affect its value. Following a global subroutine call, therefore, the DATA Partition# continues to point to the calling partition. READ statements in the global text will use DATA statements in the calling partition. The DATA Partition# is modified only by RESTORE, which sets it equal to the current value of the Text Partition#. Following a global subroutine call, the Text Partition# points to the global partition. Thus, execution of a RESTORE statement in the global text resets the DATA Partition# to point to the global partition. Subsequent READ statements in the global text would use DATA statements in the global partition.

When a RETURN is executed in the global subroutine, execution is passed back to the calling partition. However, the DATA Partition# is not altered; if a RESTORE was executed in the global subroutine, the DATA Partition# continues to point to the global partition even when execution is passed back to the calling partition. Subsequent READ statements in the calling partition would use DATA statements in the global partition. In order to reset the DATA Partition# to the Originating Partition#, a RESTORE must be executed in the originating partition.

The functions of the statements which modify the Pointer Table are summarized in Table 16-2 which follows.

Table 16-2.
Statements Which Modify the Pointer Table

Statement	Item Modified	Action
SELECT @PART	Global Partition#	Sets Global Partition# equal to partition number of SELECTed global partition.
GOSUB'	Text Partition# Text Pointer	Sets Text Partition# equal to Global Partition# if DEFFN' is located in global partition. Sets text pointer to location of DEFFN'. Original values of text pointer, Text Partition#, and Global Partition# are saved on stack in originating partition.
RETURN	Text Pointer Text Partition# Global Partition#	Retrieves old values of text pointer, Text Partition#, and Global Partition# from stack and places them back in Pointer Table.
RESTORE	DATA Partition# Text Partition#.	Sets DATA Partition# equal to

The technique of stacking and unstacking the pointers whenever a program calls a global subroutine or returns from executing one makes handling nested global partitions easy. When a global subroutine issues a call to another global partition, the pointers for the calling global partition are saved in the originating partition's stack on top of the pointers previously saved for the originating partition. When a RETURN is made from the nested global partition, the calling global partition's pointers are removed from the stack and placed in the Pointer Table. Execution of the calling global partition then proceeds as usual until a RETURN is executed, at which point the originating partition's pointers are taken from the stack and restored to the Pointer Table. This process is repeated for each level of nested global partitions. An originating partition which calls a global partition is therefore not concerned with any SELECT @PART statements or global subroutine calls executed in the global partition because a subroutine RETURN always restores the Pointer Table to its status prior to the subroutine call.

16.10 SPECIAL 2200MVP BASIC-2 LANGUAGE FEATURES

Certain portions of the BASIC-2 language are specifically useful only on the 2200MVP because it has multiuser capabilities. The following lists explain these particular features.

MVP-Only Functions

The following BASIC-2 functions can be used only on the MVP:

$MSG — returns the current broadcast message specified by terminal #1.

#PART — returns a numeric value equal to the partition number of the originating partition for this job.

$PSTAT — returns the current status of the specified partition. The status information includes a user-defined status message, operating system type (VP or MVP), operating system release number, partition size, terminal number, global name, ERR function value, and I/O device currently in use.

#TERM — returns a numeric value equal to the terminal number of the terminal assigned to the originating partition for this job.

MVP-Only Statements

The following ten BASIC-2 statements are designed principally for use on the MVP:

$BREAK — relinquishes a specified amount of the current partition's execution time for use by other partitions.

$CLOSE — releases one or more hogged devices.

DEFFN @PART — defines the current partition as global.

$INIT — passes configuration parameters to the operating system.

$MSG — defines a broadcast message available to all terminals (can be executed only by terminal #1).

$OPEN — hogs one or more devices.

$PSTAT — sets the user-defined portion of the partition status.

$RELEASE PART — causes ownership of a partition by a terminal to be relinquished.

$RELEASE TERMINAL — detaches the terminal from the current partition.

SELECT @PART — selects a specified global partition for subsequent global subroutine and global variable references.

These statements are described on the following pages.

NOTE:
Although the BASIC-2 statements and functions listed above are designed specifically for use on the 2200MVP, their syntax is supported in BASIC-2 on the 2200VP as well. In general, these statements operate on the 2200VP as if it were a single-partition, single-terminal MVP system. The exception is the $MSG statement, which is ignored on the 2200VP.

$BREAK

General Form:

$BREAK [expression | !]

Where:

0 < = expression < 256, default = 1

Purpose:

The $BREAK statement is used to put the current partition to "sleep" so that other partitions can have more processing time. The $BREAK statement gives up units of processing time (nominally, 30 milliseconds) that would normally have been allocated to this partition. The number of units to be relinquished is specified by the integer portion of the expression. A $BREAK value of 0 indicates that no break is to be performed. The amount of time given up by a partition is affected by the number of partitions awaiting execution.

$BREAK is used primarily when the current partition is waiting for some event to occur before continuing with its processing. For example, the program may need to wait until a global partition is defined:

```
10 SELECT @PART "GLOBAL": ERROR $BREAK: GOTO 10
```

Line 10 loops until the global partition is defined; the $BREAK statement enables the system to spend more time processing other partitions.

$BREAK! puts the partition "to sleep" permanently; the CPU skips processing this partition. $BREAK! can be used to end the execution of a global partition running in the background without the program requiring interaction with the terminal. Thus, subsequent execution of "$RELEASE TERMINAL" will not attach the terminal to the global partition.

If it becomes necessary to attach a terminal to a permanently put-to-sleep partition, "$RELEASE TERMINAL TO" the partition can be executed to attach the terminal. RESET must then be pressed in order to "wake up" the partition.

Examples of Valid Syntax:

```
10 $BREAK
20 $BREAK X
30 $BREAK 5
40 $BREAK !
```

$CLOSE

General Form:

$CLOSE [{ file-number / device-address } [, { file-number / device-address }]...]

Where:

device-address = /taa, where t = device-type and aa = physical device-address.

file-number = #n, where n is an integer or numeric-variable whose value is 0-15.

Purpose:

The $CLOSE statement releases the specified devices which were previously hogged via the $OPEN statement. If no device-addresses are specified, the $CLOSE statement releases all devices currently OPENed for the current partition.

Execution of END (within a program), ending program execution due to no more program text, CLEAR (with no parameters), RESET, or LOAD RUN also CLOSEs all devices currently OPENed for the current partition.

Examples of Valid Syntax:

```
10 $CLOSE
20 $CLOSE /215
30 $CLOSE /215, /02A, /02B
40 $CLOSE #3
```

DEFFN @PART

General Form:

Where:

terminal# = numeric expression

Purpose:

The DEFFN @PART statement defines the current partition as "global," enabling the program text and global variables in the current partition to be shared with other partitions in the same memory bank. A global partition may be shared by partitions in other banks only if it is defined in the universal global area. (See the discussion of the universal global area in section 16.3.) The literal or alpha-variable in the DEFFN @PART statement names the partition; the name can be up to eight characters in length. A calling program SELECTs the global partition by executing a SELECT @PART statement containing the partition name; the calling partition can then access program text in the named global partition by specifying a GOSUB' to the corresponding DEFFN' in the global program. Global variables are referenced by name in the calling program (e.g., @A$).

DEFFN @PART also determines which local variables are to be entered into the global partition's Variable Table during the resolution phase. Only those local variables occurring *before* the DEFFN @PART statement or in DIM or COM statements are entered.

Access to a global partition can be restricted only to partitions associated with specified terminals by declaring the terminal numbers of those terminals allowed to access the partition in the FOR clause of the DEFFN @PART statement. For example, the statement

```
DEFFN @PART "SHARE" FOR 1, 2, 4
```

specifies that the current partition may be accessed only by partitions assigned to terminals 1, 2, and 4 (including any background partitions associated with these terminals). Partitions assigned to other terminals will generate error messages if they attempt to access SHARE. If no FOR clause is included in the DEFFN @PART statement, all partitions are free to access the global information.

Only one DEFFN @PART statement normally occurs within a program; if more than one is found, the name in the last-executed DEFFN @PART is in effect. Error X77 results if the name specified in the DEFFN @PART statement has already been used by another global partition in the same bank or in the universal global area of bank #1. Note that partitions residing in different banks may define themselves with the same global name. A partition that has been defined to be global remains global until a CLEAR (with no parameters), LOAD RUN, or LOAD (overlay) is executed by that partition.

Executing a DEFFN @PART statement with the name equal to all spaces declares that the partition is *not* global. Other partitions cannot SELECT (via SELECT @PART) this partition for global operations.

Examples of Valid Syntax:

```
10 DEFFN @PART "SHARE"
20 DEFFN @PART A$
30 DEFFN @PART "GLOBAL" FOR 1, 2, 4
40 DEFFN @PART B$ FOR X, Y, Z
```

$INIT

General Form:

Program statement (pass configuration parameters to the operating system):

$INIT (alpha-1, alpha-2, alpha-3, alpha-4, alpha-5 [, alpha-6])

Immediate Mode statement (reconfigure system):

$INIT password

Where:

alpha = { literal-string | alpha-variable }

password = system reconfiguration password, which must be a literal-string of one to eight characters in length.

Purpose:

The special-purpose statement $INIT passes the system configuration parameters to the MVP operating system and causes the system to be reinitialized to the specified configuration. Once configured, the system can be reconfigured by executing the $INIT password statement at terminal #1. Control is passed to the system bootstrap; the message

```
MOUNT SYSTEM PLATTER
PRESS RESET
```

is displayed, and the system can be loaded and configured as if it had just been turned on.

In order to protect against inadvertent reconfiguration, $INIT can be executed at terminal #1 only. In addition, reconfiguration is password protected. If the proper password is not included in the Immediate Mode $INIT command, an error is signalled and reconfiguration does not occur. The default password is "SYSTEM"; thus, the operator on terminal #1 would enter:

```
:$INIT "SYSTEM"
```

in order to pass control to the bootstrap. The password can be changed by specifying a new password to the operating system via the "alpha-6" parameter in the $INIT program statement. However, if the system is powered off or an Immediate Mode $INIT is executed, the password reverts back to "SYSTEM". Passwords can range from 1 to 8 characters in length.

The user need not be concerned with the complex form of $INIT unless a customized partition-generator program is required. It is recommended that the Wang-supplied utility "@GENPART" or a modified version of it be used for configuring the system to ensure that the proper configuration parameters are passed to the operating system. If the $INIT parameters are not properly set up, the system may be erroneously configured, produce unpredictable errors, or lock out all the terminals. Following any of these error conditions, it may be necessary to power the CPU off and then back on in order to restore operation.

The configuration parameters are defined as follows:

alpha-1 = size of each partition.

Length of string > = 17 bytes.
Size = binary value indicating number of 256-byte pages of memory allocated for a partition.
Byte 1 = size of partition #1.

Byte 2 = size of partition #2.

.
.
.

Byte n = size of partition #n.
Byte n+1 = HEX(00).

> **NOTE:**
>
> The partition must be set so that the memory used for partitions is completely contiguous (i.e., a bank must be completely filled before partitions in the next bank can be specified). No partition may be split between banks.

alpha-2 = terminal number for each partition.

Length of string > = 16 bytes.
Terminal number= number (in binary) of terminal assigned to a partition.
Byte 1 = terminal number for partition #1.
Byte 2 = terminal number for partition #2.

.
.
.

Byte n = terminal number for partition #n.
Remaining bytes must = HEX(00).

alpha-3 = partition modes.

Length of string > = 16 bytes.
Mode, bit 01 = 1 if and only if programming is not allowed on this partition.
Mode, bit 02 = 1 if and only if a program is to be bootstrapped into this partition.
Byte 1 = mode of partition #1.
Byte 2 = mode of partition #2.

.
.
.

Byte n = mode of partition #n.

alpha-4 = bootstrap program name for each partition.

Length of string > = 128 bytes.
Bootstrap program name =eight-byte alpha string specifying the program to be automatically loaded and run after partition generation.
1st eight bytes = bootstrap name for partition #1.
2nd eight bytes = bootstrap name for partition #2.

.
.
.

Nth eight bytes = bootstrap name for partition #n.

alpha-5 = device table.

Length of string > = 99 bytes.
A device is specified by three bytes.
1st byte, low 4-bits = device-type (disk must be 3 or B).
2nd byte = physical device-address.
3rd byte = number in binary of the partition for which the device is to be OPENed (0 if none).
1st three bytes = device specification for device #1.
2nd three bytes = device specification for device #2.
.
.
.
Nth three bytes = device specification for device #n.
(N + 1) 3 bytes = HEX(000000).

alpha-6 = reconfiguration password.
Length of string > = 8 bytes.
1st eight bytes are the password.

Examples of Valid Syntax:

```
$INIT "SYSTEM"
10 $INIT (S$,T$,M$,N$( ),D$)
20 $INIT (S$,T$,M$,N$( ),D$, P$)
```

$MSG

General Form:

$MSG = alpha-expression

Where:

alpha-expression same as for LET

Purpose:

The $MSG statement can be used by terminal #1 to define a broadcast message to be displayed on each terminal's CRT whenever the "READY" message is normally displayed (i.e., after RESET or CLEAR). The broadcast message is displayed on line 0 of the CRT immediately above the "READY" message. The message is specified by using the $MSG statement. For example:

```
$MSG = "*** SYSTEM WILL GO DOWN AT NOON ***"
```

After RESET, the display would appear as shown below:

```
*** SYSTEM WILL GO DOWN AT NOON ***
READY (BASIC-2) PARTITION 01
:_
```

The message also is available to any program in the system via the $MSG function, which can appear only on the right-hand side of alpha assignment statements. For example, the statement

```
10 A$ = $MSG
```

sets A$ equal to the broadcast message. The program can then display the message whenever convenient.

Examples of Valid Syntax:

```
10 $MSG = "MERRY XMAS"
20 A$ = $MSG
```

$OPEN

General Form:

$OPEN [line-number,] { file-number / device-address } [, { file-number / device-address }] ...

Where:

device-address = /taa, where t = device-type and aa = physical device-address.

file-number = #n, where n is an integer or numeric-variable whose value is 0-15.

Purpose:

A program may request exclusive use of a peripheral for the current partition by specifying the device-address of that peripheral in a $OPEN statement. Once OPEN, the device remains "hogged" by the current partition until one of the following conditions occurs:

- A $CLOSE or END statement is executed (END must be within a program).
- Program execution terminates with the last line of program text.
- A CLEAR (with no parameters), RESET, or LOAD RUN command is executed.

If the requested device has already been OPENed by another partition, a branch is made to the line-number specified in the $OPEN statement. If no line-number is specified, the $OPEN statement will wait until the specified device becomes available for this partition. Error P48 results if the specified device is not in the Master Device Table or if the device has been designated as an exclusive device for another partition. A $IF ON statement always senses device busy when the specified device is OPENed by another partition.

If multiple devices are specified in a $OPEN statement and one of the devices cannot be opened, then *none* of the devices is opened. If more than one device is required by a program, *all* should be OPENed with a single $OPEN statement in order to prevent a device contention deadlock. Such a deadlock could arise when two or more partitions which require control of the same peripherals seize different peripherals in their $OPEN statements. Suppose, for example, that partitions #1 and #2 both require control of a disk and a printer. If partition #1 OPENs the disk and partition #2 OPENs the printer, each partition will "hang", waiting for the device being hogged by the other partition. If both devices are specified in the same $OPEN statement, the first partition to execute this statement gets the devices and the other partition either waits or branches to perform other processing.

Note that the device-type is ignored by the $OPEN statement. It is not possible to hog a single disk platter by specifying /B10 or /310; the entire disk unit must be hogged.

Examples of Valid Syntax:

```
10 $OPEN /215
20 $OPEN 100, /215, /02A, /02B
30 $OPEN #3
```

$PSTAT

General Form (as a statement):

$PSTAT = alpha-expression

Where:

alpha-expression is the same as for LET

General Form (as a function within an alpha expression):

$PSTAT (expression)

Where:

1 < = value of expression < = number of partitions

Purpose:

The $PSTAT function returns an alphanumeric string describing the current status of the partition specified by the value of the expression. The $PSTAT function can be used only within an alpha expression on the right-hand side of assignment statements. The following information is returned by the $PSTAT function:

bytes 1-8 = user-specified status

This area contains the user-specified message set with the $PSTAT statement.

byte 9 = operating system type ("M" if MVP, "V" if VP)

byte 10 = operating system release number

The release number is stored in packed decimal; thus, HEX(12) is release 1.2.

byte 11 = memory bank

A decimal value indicating the memory bank in which the partition resides.

bytes 12-13 = partition size (XX.YY K)

A packed decimal value indicating the partition size. Byte 12 is the integer portion of the partition size (XX) and byte 13 is the fractional portion (YY).

byte 14 = programmability

A "P" is returned if the partition is programmable; if programming has been disabled, a "space" is returned.

byte 15 = terminal number

A packed decimal value indicating which terminal is assigned to the partition. A value of zero is returned if the partition is not assigned to any terminal (i.e., a $RELEASE PART statement was executed, or the partition was originally assigned to the null terminal).

byte 16 = terminal status

"A" if terminal is attached.
"D" if terminal has been detached via $RELEASE TERMINAL and the partition has not requested the terminal.
"W" if the partition is waiting for the terminal to be attached.

bytes 17-24 = global name

If the partition has not declared itself to be global by means of DEFFN @PART, the global name is all spaces.

byte 25 = ERR function value

Numeric portion of the last error encountered in the partition.

byte 26 = Text Partition#

A packed decimal value indicating the number of the partition containing the program text currently being executed. The Text Partition# is the same as #PART except when global text is being executed.

byte 27 = Global Partition#

A packed decimal value indicating the number of the partition selected for global operations via SELECT @PART.

byte 28 = DATA Partition#

A packed decimal value indicating the partition containing the DATA statements that READ currently points at. The DATA Partition# is the same as #PART unless a RESTORE statement has been executed in the global text.

byte 29 = device-address

Address of the device with which the partition currently is communicating or for which it is currently waiting.

If the partition specified in the $PSTAT function does not exist in the current configuration, ERR X77 results.

The first eight bytes of the partition status are user specified using the $PSTAT statement. The first eight bytes of the alpha-expression portion of the $PSTAT statement become the first eight bytes of the partition status, allowing the user to specify a message to be displayed when a $PSTAT function is executed.

Examples of Valid Syntax:

```
10 A$ = $PSTAT (3)
20 B$ = $PSTAT (#PART)
30 C$() = $PSTAT (1) & $PSTAT (2) & $PSTAT (3)
40 $PSTAT = Q$
50 $PSTAT = "OP#12345"
```

$RELEASE PART

General Form:

$RELEASE PART

Purpose:

The $RELEASE PART statement causes a partition to be reassigned from the current controlling terminal to the null terminal (terminal #0). The released partition may then be reassigned to any other terminal in the system which issues a request for it.

The null terminal is not an actual device, but rather a pseudoterminal to which a partition may be assigned. Partitions assigned to this pseudoterminal are available to any requesting terminal. These partitions represent a pool of resources available to all terminals attached to the system on a first-come, first-served basis. Partitions assigned to the null terminal may run programs and are considered to be in the background.

The $RELEASE PART statement allows for more flexible handling of partitions. Normally, each partition can be accessed by only one terminal. That terminal is specified at partition generation time. However, it may be desirable at some time to allow a partition controlled by one terminal to be released so that another terminal may control it. Executing the $RELEASE PART statement permits this to occur.

If the $RELEASE PART statement is executed in a foreground partition, that partition is immediately detached from the terminal and assigned to the null terminal. If there is a single background partition assigned to the terminal, the background partition is attached to the terminal and becomes the foreground partition. In the case of multiple background partitions, the first partition waiting to communicate with the terminal will be attached to it. If no other partitions are assigned to it, the terminal effectively detaches itself from the active MVP system when it releases its partition. *No action may be initiated from such a terminal unless another partition is subsequently attached to it.*

If the $RELEASE PART statement is executed in a background partition, only the partition in which the statement is executed is reassigned; neither the current foreground partition nor any other background partitions are disturbed.

$RELEASE PART does not clear the partition or halt program execution; however, if a program running in a partition assigned to the null terminal attempts to output to the terminal, execution is suspended.

A terminal may request for itself a partition assigned to the null terminal by executing a $RELEASE TERMINAL TO statement to the desired partition. In this case, the current partition is detached (i.e., becomes a background job); the desired partition is reassigned from the null terminal to the requesting terminal and is then attached to that terminal (i.e., becomes the foreground partition). In the case of a terminal which is detached from the active MVP system (due to execution of a $RELEASE PART statement), keying RESET will cause the lowest numbered partition assigned to the null terminal (if such a partition exists) to be reassigned to the requesting terminal and immediately attached to it.

Any partition may be assigned to the null terminal at partition generation time by specifying zero as the terminal assignment parameter of that partition. If a partition is assigned to the null terminal, byte 15 of $PSTAT will be zero.

Examples of Valid Syntax:

```
10 $RELEASE PART
$RELEASE PART
```

$RELEASE TERMINAL

General Form:

$RELEASE TERMINAL [TO { expression / partition-name } [, STOP]]

Where:

partition-name = { literal-string / alpha-variable } (1-8 bytes in length)

Purpose:

The $RELEASE TERMINAL statement detaches the terminal from the current partition. The MVP operating system can then attach the terminal to the next partition waiting to communicate with it. The $RELEASE TERMINAL statement is ignored if the terminal is not attached to the partition at the time the statement is executed.

Optionally, the terminal can be released to a specified partition by using the "TO" parameter in $RELEASE TERMINAL TO. In this case, the terminal is attached to the specified partition even if the partition has not attempted to communicate with it. This directed releasing is necessary when background programs need to be aborted by HALT or RESET. The partition is specified by number (i.e., expression); global partitions, which are named by a DEFFN @PART statement, also may be identified by partition name. Error X77 results if the specified partition is not assigned to the terminal issuing the $RELEASE statement or if the partition has not been released (see the discussion of the $RELEASE PART statement earlier in this section). If more than one partition exists with the specified name, the partitions must reside in separate banks and the terminal is released to the lowest numbered available partition with the specified name.

If the STOP parameter is included in $RELEASE TERMINAL TO, the terminal will be attached to the specified partition and that partition will then be halted. This statement can be used to terminate a background job that continually releases the terminal.

A program can determine whether a terminal currently is attached to its partition by executing a $IF ON statement to the terminal CRT or keyboard. This capability is particularly important for background jobs which use the terminal only to display status information. Normally, the background job would "hang", waiting for the terminal each time it attempted to display its status information. To avoid this problem, $IF ON can be used to test periodically for the availability of the terminal. If the terminal isn't available, the program can resume its normal processing.

$IF ON to address /005 (terminal CRT) senses the following status:

READY if the terminal is attached to the partition executing $IF ON.

BUSY if the terminal is detached.

For example, consider the following routine:

```
10 $IF ON /005, 30
20 - -
30 - -
```

When the $IF ON statement is executed, a branch is made to line 30 if and only if the terminal is currently attached to this partition.

Note that $IF ON also can be used to address /001 (the terminal keyboard) to determine if the terminal is attached and if a character has been entered. See section 16.11, "Programming Considerations," for further details.

Examples of Valid Syntax:

```
$RELEASE TERMINAL
10 $RELEASE TERMINAL
20 $RELEASE TERMINAL TO 3
30 $RELEASE TERMINAL TO "JOBXYZ"
40 $RELEASE TERMINAL TO 5, STOP
```

SELECT @PART

General Form:

SELECT [. . .] @PART partition name [. . .]

Where:

partition name = { alpha-variable / literal string } (1-8 bytes in length)

Purpose:

The SELECT @PART statement specifies a global partition whose text and/or global variables are to be referenced by the partition in which SELECT @PART is executed (the "calling" partition). A global partition in the same memory bank as the calling partition or in the universal global area of bank #1 can be selected. The name of the global partition is specified by the literal or alpha-variable in the SELECT @PART statement; if no partition by that name has been defined via DEFFN @PART or if the global partition is not accessible to this terminal, error X77 results. The program can wait for the specified partition to be defined by using the ERROR statement after the SELECT @PART statement. For example:

```
10 SELECT @PART "SHARE": ERROR $BREAK: GOTO 10
```

Having selected a global partition, global text is accessed by executing a GOSUB' to the corresponding DEFFN' in the global program; global variables are referenced by name (e.g., @A$).

More than one SELECT @PART can occur within a program so that more than one global partition can be accessed. However, only the last-executed SELECT @PART is in effect. When a RETURN from a global subroutine is performed, the global partition that had been selected for the calling program is restored; thus, the global subroutine can select a different global partition without affecting the selection of the calling program.

The global partition selection is cleared by CLEAR (with no parameters), LOAD RUN, and LOAD (overlay) when executed in the calling partition. Following any of these operations, another SELECT @PART must be executed in order to access the global partition.

Executing a SELECT @PART statement with the name equal to all spaces selects the originating partition (i.e., #PART) for global operations. The originating partition need not be defined to be global. SELECT @PART " " is useful when a partition needs to refer to its own global variables or when global text needs to call subroutines in the calling partition.

Examples of Valid Syntax:

```
10 SELECT @PART "SHARE"
20 SELECT PRINT /215, @PART A$, DISK/320
```

16.11 PROGRAMMING CONSIDERATIONS

This section consists of two parts: the first subsection discusses general programming considerations for the 2200MVP, while the second subsection enumerates specific areas of incompatibility between the MVP and other Wang systems. The discussion of programming considerations suggests some simple programming techniques which can contribute to better system performance and explains many of the differences between the 2200MVP and other Wang systems, most notably the 2200VP. The discussion of compatibility focuses on those differences which would prevent a program written for another Wang system from running on the MVP. This discussion is intended primarily for programmers who must modify existing software to run on the MVP.

General Programming Considerations

Although any program may ignore the potential existence of other programs in the system, the use of a few simple techniques which make the most efficient use of CPU time can contribute significantly to overall system performance. These techniques, along with some special features and restrictions, are discussed below.

A. **Considerations for Time-Dependent Programs**

1. The execution time for a given program on the MVP varies from one run to another depending upon the current load of the CPU. Therefore, instrumentation that is critically timed by the CPU may not work properly.

2. The 2236D CRT communication speed is slower than the speed of the 2226CRT and other CRT models. Thus, programs written to update the entire screen may perform slowly when run on the MVP. Program modifications to update only the required portions of the screen are recommended.

3. In general, the INPUT and LINPUT statements are preferred over KEYIN for data entry because these statements are handled by the MXD controller rather than the CPU and, therefore, require no CPU processing between keystrokes. If a program must use KEYIN, use Form 1 (e.g., KEYIN A$) rather than Form 2 (e.g., KEYIN A$, 10, 20) because the partition is automatically put to "sleep" until a key is touched when Form 1 is used. When a program must test a condition repeatedly in a tight polling loop, it should perform the test once and then relinquish the remainder of its timeslice with a $BREAK statement. For the programmer's convenience, this feature has been built into Form 2 of KEYIN (an automatic $BREAK occurs if no character is ready for KEYIN).

4. The use of FOR/NEXT loops or $GIO (75xx) to create time delays (e.g., to maintain a message on the CRT for a specified interval) not only wastes CPU time but results in a delay which varies in duration as a function of CPU loading. In this type of delay, the partition waits for its full timeslice (30 ms). When its turn arrives again, the partition is allowed to waste another 30 ms until the loop is completed. Time-delay loops thus become minimum delay loops on the MVP. The preferred method of implementing time delays requires the use of a SELECT P statement. SELECT P is timed by the 2236D terminal rather than the CPU. With this technique, special characters are sent to the terminal to cause it to delay. Since the terminal is buffered, the program does not wait at the PRINT statement causing the delay. The CPU time which would otherwise be wasted in the delay loop can be used more productively for the delayed partition and others.

B. **I/O Operations**

1. For line printers, plotters, the 2228B Synchronous/Asynchronous Communications Controller, and any other device which must be temporarily allocated to a specified

user exclusively, the $OPEN and $CLOSE statements are provided. Other than making certain that these statements are added, the programmer need not change the body of a program.

2. All Console Input, INPUT, and LINPUT operations utilize the 2236D-type terminals. Therefore, such operations may not be performed with TC boards. Similarly, all Console Output (including echo characters and output from Immediate Mode PRINT and PRINTUSING statements) utilizes the 2236D-type terminal only. Thus, Console Output can never be sent to other output devices such as the line printer. The only exception to this rule is output from TRACE, which may be sent to a printer if the printer is selected for CO.

C. I/O Statement Restrictions

The following table defines the devices with which the MVP operating system permits individual statements to communicate. ERR P48 results if a statement addresses an illegal device.

Table 16-3.
Devices to Which BASIC-2 Statements Communicate

Statement Or Operation	2236D-type Terminal Keyboard	2236D-type Terminal CRT	2236D-type Terminal Printer/ Plotter	Devices Other Than 2236D Terminals
Console Output*		X	X	X
PRINT		X	X	X
PRINTUSING		X	X	X
HEXPRINT		X	X	X
LIST		X	X	X
PLOT		X	X	X
Console Input	X			
INPUT	X			
LINPUT	X			
KEYIN	X			X
$IF ON/OFF	X	X		X
$GIO	X**	X	X	X
SELECT ON (interrupt)				X
Disk Statements				X

D. Default Disk Address

Unlike the 2200VP, whose default disk address is always /310 after power on, the MVP's default disk address is set to the address of the drive from which the system was loaded. For example, loading by pressing SF '00 sets the default address to /310. After partition generation, the default disk address for each partition is set to the default disk address of partition #1 at the time of partition generation.

* Console Output (keystroke echo; error, END, and STOP messages; and LINPUT and INPUT prompts) is *always* directed to the terminal CRT except for TRACE output, which can be sent to another selected device such as a printer.

** Input with timeout is not supported.

E. Special Features of the 2236MXD Controller

1. The 2236 MXD Controller buffers up to 32 keystrokes that have not yet been requested by a program. Such buffering reduces peaks in operator typing speed in data entry applications, but it also allows the operator to anticipate program prompts. Sometimes it is necessary to flush this keystroke buffer (for example, to minimize the effect of an operator repeatedly keying RETURN while a program overlay loads from a diskette). It is easy to flush the keyboard buffer with a KEYIN statement that branches to itself such as:

   ```
   10 KEYIN A$, 10, 10
   ```

2. The 2236 MXD Controller allows a maximum of 480 bytes to be entered into a single line request. This limit restricts the maximum length of a field that may be entered with a single INPUT or LINPUT statement. This restriction also limits the length of a program line that may be entered or edited on the MVP.

3. Character Insert Mode is not supported on the MVP. Characters must be inserted into text lines in Edit Mode, using the Edit Insert Key.

4. Edit Mode on the MVP is signalled by a blinking cursor rather than an asterisk at the left of the character field.

Software Conversion — Compatibility With Earlier Wang 2200 Systems

This section discusses BASIC-2 language compatibility between the 2200MVP and previous Wang 2200 systems. The native language of the 2200MVP is Wang BASIC-2, which is almost 100% compatible with the 2200VP BASIC-2 on the 2200VP. Like the 2200VP, the MVP also supports most earlier Wang BASIC syntax. Users with existing software are therefore encouraged to try it on the MVP. As a first approximation for the required MVP partition size, use the size of the single-user system for which the software is designed. (This figure may be slightly small for programs designed for the 2200T and slightly large for 2200VP programs.) Once the program has been loaded and resolved, the Immediate Mode command PRINT SPACEK-SPACE/1024 will reveal the partition size needed by the program in K bytes.

The remainder of this section deals with the few software incompatibilities between the MVP and other Wang systems in the order of frequency with which they are usually encountered. Even if an existing program runs correctly on the MVP, it will probably need a few $OPEN and $CLOSE statements to hog the printer. Also, some structural changes may be required if the program is not designed for a multiplexed disk environment. While it is desirable to get existing software up and running quickly, program modifications which utilize special MVP features to facilitate cooperation among programs can enhance system performance significantly. Frequently, the memory requirements of 2200T programs can be substantially reduced by recoding them using the more powerful BASIC-2 statements. (Users converting 2200T programs for use on the MVP should refer to Appendix C, "Compatibility With Other 2200 Series CPU's.") The following is a list of four suggestions for improvement of existing code:

1. Some programs use KEYIN to input text atom codes from the BASIC Keyword keys. This technique works properly on the MVP, but the standard 2236D terminal keyboard does not have all the atom keys found on a 2226 console keyboard. The absence of a PRINT Key seems to create the most frequent software compatibility problem.

2. Many programs test for the existence of printers and disks before attempting to access them. Such tests present a problem on the MVP because several lines of buffering separate the terminal printer from the I/O bus. The $OPEN and ERROR $BREAK statements can be used to determine if the device-address was declared in the Master Device Table when the system was configured using the @GENPART program.

3. Some programs use $GIO with timeout to the keyboard to insist that an operator respond in a fixed period of time. $GIO with timeout is not supported at address /001 on the MVP.

4. The MVP does not permit CI or CO to be selected to any device other than the 2236D terminal. The output of a TRACE command, however, may be SELECTed to a printer by using a SELECT CO statement. The width of the Console Output device may not be redefined to be other than 80 bytes for an INPUT or LINPUT operation.

APPENDICES

APPENDIX A: BASIC-2 ERROR CODES

A.1 MISCELLANEOUS ERRORS (NONRECOVERABLE)

ERR A01

Error: MEMORY OVERFLOW

Cause: There is not enough memory free space remaining to enter the program line or accommodate the defined variable. System commands (e.g., SAVE) and some Immediate Mode statements still can be executed. (See Chapter 2, section 2.5 for a more detailed explanation of this error.)

Recovery: Make more space available by entering a CLEAR P, N, or V command to shorten the program or reduce the number of variables defined.

ERR A02

Error: MEMORY OVERFLOW

Cause: There is not enough memory free space remaining to execute the program or Immediate Mode line. Commands (e.g., SAVE) and some Immediate Mode statements still can be executed. (See Chapter 2, section 2.5 for a more detailed explanation of this error.)

Recovery: Make more space available by shortening the program or reducing the amount of variable space used by executing a CLEAR P, N, or V command.

ERR A03

Error: MEMORY OVERFLOW

Cause: There is insufficient free space in memory to execute the LIST DC, MOVE, or COPY statement (approximately 1K bytes of free space are required for MOVE and COPY and 100 bytes for LIST DC).

Recovery: Make more space available by executing a CLEAR P, N, or V command.

ERR A04

Error: STACK OVERFLOW

Cause: A fixed-length system stack (the Operator Stack) has overflowed. A maximum total of 64 levels of nesting for subroutines, FOR/NEXT loops, and expression evaluation is permitted. Often this error occurs because the program repeatedly branches out of subroutines or loops without executing a terminating RETURN or NEXT statement.

Recovery: Correct the program text, possibly by using a RETURN CLEAR statement to clear subroutine or loop information from the stacks.

ERR A05

Error: PROGRAM LINE TOO LONG

Cause: The program line being entered can not be saved in one disk sector because its length exceeds 253 bytes. The line can be executed, but cannot be saved on disk.

Recovery: Shorten the line by breaking it up into two or more smaller lines.

ERR A06

Error: PROGRAM PROTECTED

Cause: A program or program overlay loaded into memory was protected; therefore, no program text in memory can be SAVEd, LISTed, or modified (except by LOAD or CLEAR).

Recovery: Protect Mode must be deactivated with a CLEAR command. (However, executing a CLEAR command also clears all memory.)

ERR A07

Error: ILLEGAL IMMEDIATE MODE STATEMENT

Cause: An attempt was made to execute an illegal statement in Immediate Mode.

Recovery: Delete the illegal statement and reexecute the line.

ERR A08

Error: STATEMENT NOT LEGAL HERE

Cause: The statement cannot be used in this context.

Recovery: Correct the program line.

ERR A09

Error: PROGRAM NOT RESOLVED

Cause: An attempt was made to execute an unresolved program.

Recovery: Resolve the program by running it with RUN.

A.2 SYNTAX ERRORS (NONRECOVERABLE)

ERR S10

Error: MISSING LEFT PARENTHESIS

Cause: A left parenthesis [( ] was expected.

Recovery: Correct the statement text.

ERR S11

Error: MISSING RIGHT PARENTHESIS

Cause: A right parenthesis [ )] was expected.

Recovery: Correct the statement text.

ERR S12

Error: MISSING EQUAL SIGN

Cause: An equal sign (=) was expected.

Recovery: Correct the statement text.

ERR S13

Error: MISSING COMMA

Cause: A comma (,) was expected.

Recovery: Correct the statement text.

ERR S14

Error: MISSING ASTERISK

Cause: An asterisk (*) was expected.

Recovery: Correct the statement text.

ERR S15

Error: MISSING ">" CHARACTER

Cause: The required ">" character is missing from the program statement.

Recovery: Correct the program statement syntax.

ERR S16

Error: MISSING LETTER

Cause: A letter was expected.

Recovery: Correct the statement text.

ERR S17

Error: MISSING HEX DIGIT

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

Recovery: Correct the program text.

ERR S18

Error: MISSING RELATIONAL OPERATOR

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

Recovery: Correct the statement text.

ERR S19

Error: MISSING REQUIRED WORD

Cause: A required BASIC word is missing (e.g., THEN or STEP).

Recovery: Correct the statement text.

ERR S20

Error: EXPECTED END OF STATEMENT

Cause: The end of the statement was expected. The statement syntax is correct up to the point of the error message, but one or more following characters make the statement illegal.

Recovery: Complete the statement text.

ERR S21

Error: MISSING LINE-NUMBER

Cause: A line-number in the program statement is missing.

Recovery: Correct the statement syntax.

ERR S22

Error: ILLEGAL PLOT ARGUMENT

Cause: An argument in the PLOT statement is illegal.

Recovery: Correct the PLOT statement.

ERR S23

Error: INVALID LITERAL STRING

Cause: A literal string was expected. The length of the literal string must be > = 1 and < = 255.

Recovery: Correct the invalid literal string.

ERR S24

Error: ILLEGAL EXPRESSION OR MISSING VARIABLE

Cause: The expression syntax is illegal or a variable is missing.

Recovery: Correct the syntax, or insert the missing variable.

ERR S25

Error: MISSING NUMERIC-SCALAR-VARIABLE

Cause: A numeric-scalar-variable was expected.

Recovery: Correct the statement text.

ERR S26

Error: MISSING ARRAY-VARIABLE

Cause: An array-variable was expected.

Recovery: Correct the statement text.

ERR S27

Error: MISSING NUMERIC-ARRAY

Cause: A numeric-array is required in the specified program statement syntax.

Recovery: Correct the program statement.

ERR S28

Error: MISSING ALPHA-ARRAY

Cause: An alpha-array is required in the specified program statement syntax.

Recovery: Correct the program statement.

ERR S29

Error: MISSING ALPHANUMERIC-VARIABLE

Cause: An alphanumeric-variable was expected.

Recovery: Correct the statement text.

A.3 PROGRAM ERRORS (NONRECOVERABLE)

ERR P32

Error: START > END

Cause: The starting value is greater than the ending value.

Recovery: Correct the statement in error.

ERR P33

Error: LINE-NUMBER CONFLICT

Cause: The RENUMBER command cannot be executed. The renumbered program text must fit between existing (nonrenumbered) program lines.

Recovery: Correct the RENUMBER command.

ERR P34

Error: ILLEGAL VALUE

Cause: The value exceeds the allowed limit.

Recovery: Correct the program or data.

ERR P35

Error: NO PROGRAM IN MEMORY

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

Recovery: Enter the program statements or load a program.

ERR P36

Error: UNDEFINED LINE-NUMBER OR ILLEGAL CONTINUE COMMAND

Cause: A referenced line-number is undefined, or the user is attempting to CONTINUE program execution after one of the following conditions has occurred: A Stack or Memory Overflow error, entry of a new variable or a CLEAR command, modification of the user program text, or depressing the RESET Key.

Recovery: Correct the statement text, or rerun the program with RUN.

ERR P37

Error: UNDEFINED MARKED SUBROUTINE

Cause: There is no DEFFN' statement in the program corresponding to the GOSUB' statement that was to be executed.

Recovery: Correct the program.

ERR P38

Error: UNDEFINED FN FUNCTION

Cause: An undefined FN function was referenced.

Recovery: Correct the program by defining the function or referencing it correctly.

ERR P39

Error: FN'S NESTED TOO DEEP

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

Recovery: Reduce the number of nested functions.

ERR P40

Error: NO CORRESPONDING "FOR" FOR "NEXT" STATEMENT

Cause: There is no companion FOR statement for a NEXT statement, or a branch was made into the middle of a FOR/NEXT loop.

Recovery: Correct the program.

ERR P41

Error: RETURN WITHOUT GOSUB

Cause: A RETURN statement was executed without first executing a GOSUB or GOSUB' statement (e.g., a branch was made into the middle of a subroutine).

Recovery: Correct the program.

ERR P42

Error: ILLEGAL IMAGE

Cause: The image is not legal in this context. For example, the image referenced by PRINTUSING does not contain a format-specification.

Recovery: Correct the program.

ERR P43

Error: ILLEGAL MATRIX OPERAND

Cause: The same array-name appears on both sides of the equation in a MAT multiplication or MAT transposition statement.

Recovery: Correct the statement.

ERR P44

Error: MATRIX NOT SQUARE

Cause: The dimensions of the operand in a MAT inversion or identity are not equal.

Recovery: Correct the array dimensions.

ERR P45

Error: OPERAND DIMENSIONS NOT COMPATIBLE

Cause: The dimensions of the operands in a MAT statement are not compatible; the operation cannot be performed.

Recovery: Correct the dimensions of the arrays.

ERR P46

Error: ILLEGAL MICROCOMMAND

Cause: A microcommand in the specified $GIO sequence is illegal or undefined.

Recovery: Use only legal or defined microcommands.

ERR P47

Error: MISSING BUFFER VARIABLE

Cause: A buffer (Arg-3) in the $GIO statement was omitted for a data input, data output, or data verify microcommand.

Recovery: Define the buffer if it was omitted.

ERR P48

Error: ILLEGAL DEVICE SPECIFICATION

Cause: The #n file-number in a program statement is undefined, or the device-address is illegal. On the MVP, the selected device is not contained in the Master Device Table; the error is signalled when communication is attempted and *not* when the SELECT statement is executed.

NOTE:

P48 is a recoverable error.

Recovery: Define the specified file-number in a SELECT statement, or correct the device-address.

ERR P49

Error: INTERRUPT TABLE FULL

Cause: Interrupts were defined for more than eight devices. The maximum number of devices allowed is eight.

Recovery: Reduce the number of interrupts.

ERR P50

Error: ILLEGAL ARRAY DIMENSIONS OR VARIABLE LENGTH

Cause: An array dimension or alpha-variable length exceeds the legal limits. The limits are as follows:

one-dimensional array: 1 <= dimension < 65536
two-dimensional array: 1 <= dimension < 256
alpha-variable length: 1 <= length < 125

Recovery: Correct the dimension or variable length.

ERR P51

Error: VARIABLE OR VALUE TOO SHORT

Cause: The length of the variable or value is too small for the specified operation.

Recovery: Correct the program.

ERR P52

Error: VARIABLE OR VALUE TOO LONG

Cause: The length of the variable or value is too long for the specified operation.

Recovery: Correct the statement or command.

ERR P53

Error: NONCOMMON VARIABLES ALREADY DEFINED

Cause: A COM statement was preceded by a noncommon variable definition.

Recovery: Correct the program by making all COM statements the first-numbered lines, or clear the noncommon variables with a CLEAR N command.

ERR P54

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) or the variable containing the program name(s) in a multiple-file LOAD command is not a common variable.

Recovery: Redefine the variable to be common.

ERR P55

Error: UNDEFINED VARIABLE (PROGRAM NOT RESOLVED)

Cause: An array which was not defined properly in a DIM or COM statement is referenced in the program, or a variable has been encountered which was not defined because the program was not resolved (e.g., a Special Function Key was used to initiate program execution, but the program was never RUN).

Recovery: Correct the text or RUN the program.

ERR P56

Error: ILLEGAL SUBSCRIPTS

Cause: The variable subscripts exceed the defined array dimensions or the dimensions of the variable, which were defined in a DIM or COM statement, do not agree with the array definition.

Recovery: Change the variable subscripts or the variable definition in a DIM or COM statement.

ERR P57

Error: ILLEGAL STR ARGUMENTS

Cause: The STR function arguments exceed the maximum defined length of the alpha-variable.

Recovery: Correct the STR arguments, or redefine the alpha-variable.

ERR P58

Error: ILLEGAL FIELD/DELIMITER SPECIFICATION

Cause: The field or delimiter specification in a $PACK or $UNPACK statement is illegal.

Recovery: Correct the illegal field or delimiter specification.

ERR P59

Error: ILLEGAL REDIMENSION

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

Recovery: Reserve more space for the array in the initial DIM or COM statement, or redimension the array to fit in the available space.

A.4 COMPUTATIONAL ERRORS (RECOVERABLE)

ERR C60

Error: UNDERFLOW

Cause: The absolute value of the calculated result was less than 1E-99 but greater than zero.

Recovery: Correct the program or the data. Underflow errors can be suppressed by executing SELECT ERROR > 60; a default value of zero will be used.

ERR C61

Error: OVERFLOW

Cause: The absolute value of the calculated result was greater than 9.999999999999E+99.

Recovery: Correct the program or the data. Overflow errors can be suppressed by executing SELECT ERROR > 61; a default value of ± 9.999999999999E+99 will be used.

ERR C62

Error: DIVISION BY ZERO

Cause: Division by a value of zero is a mathematically undefined operation.

Recovery: Correct the program or the data. A division-by-zero error can be suppressed by executing SELECT ERROR > 62; a default value of ±9.999999999999E+99 will be used.

ERR C63

Error: ZERO DIVIDED BY ZERO OR ZERO ↑ ZERO

Cause: A mathematically indeterminate operation (0/0 or 0↑0) was attempted.

Recovery: Correct the program or the data. Errors of this type can be suppressed by executing SELECT ERROR > 63; a default value of 0 will be used.

ERR C64

Error: ZERO RAISED TO NEGATIVE POWER

Cause: Zero raised to a negative power is a mathematically undefined operation.

Recovery: Correct the program or the data. This error can be suppressed by executing SELECT ERROR > 64; a default value of 9.999999999999E+99 will be used.

ERR C65

Error: NEGATIVE NUMBER RAISED TO NONINTEGER POWER

Cause: This is a mathematically undefined operation.

Recovery: Correct the program or the data. This error can be suppressed by executing SELECT ERROR > 65; a default value of the absolute value of the negative number raised to the noninteger power will be used.

ERR C66

Error: SQUARE ROOT OF NEGATIVE VALUE

Cause: This is a mathematically undefined operation.

Recovery: Correct the program or the data. This error can be suppressed by executing SELECT ERROR > 66; a default value of SQR(ABS(X)), where X is the negative value, will be used.

ERR C67

Error: LOG OF ZERO

Cause: This is a mathematically undefined operation.

Recovery: Correct the program or the data. This error can be suppressed by executing SELECT ERROR > 67; a default value of -9.999999999999E±99 will be used.

ERR C68

Error: LOG OF NEGATIVE VALUE

Cause: This is a mathematically undefined operation.

Recovery: Correct the program or the data. This error can be suppressed by executing SELECT ERROR > 68; a default value of the LOG of the absolute value of the number will be used.

ERR C69

Error: ARGUMENT TOO LARGE

Cause: The absolute value of the SIN, COS, or TAN function is >= 1E+10; the system cannot evaluate the function meaningfully. Or, the absolute value of the ARCSIN, ARCCOS, or ARCTAN argument is > 1.0; the value of the function is mathematically undefined.

Recovery: Correct the program or the data. This error can be suppressed by executing SELECT ERROR > 69; a default value of zero will be used.

A.5 EXECUTION ERRORS (RECOVERABLE)

ERR X70

Error: INSUFFICIENT DATA

Cause: There are not enough DATA values to satisfy READ or RESTORE statement requirements.

Recovery: Correct the program to supply additional DATA, or modify the READ or RESTORE statement.

ERR X71

Error: VALUE EXCEEDS FORMAT

Cause: The number of integer digits in the PACK or CONVERT image specification is insufficient to express the value of the number being packed or converted.

Recovery: Change the image specification.

ERR X72

Error: SINGULAR MATRIX

Cause: The operand in a MAT inversion statement is singular and cannot be inverted.

Recovery: Correct the program or the data. Inclusion of a normalized determinant parameter in the MAT INV statement will eliminate this error; however, the determinant must be checked by the application program following the inversion.

ERR X73

Error: ILLEGAL INPUT DATA

Cause: The value entered as requested by an INPUT statement is expressed in an illegal format.

Recovery: Reenter the data in the correct format starting with the erroneous number, or terminate run with RESET and RUN again. Alternatively, LINPUT can be used to enter the data, and the data can be verified within the application program.

ERR X74

Error: WRONG VARIABLE TYPE

Cause: The variable type (alpha or numeric) does not agree with the data type. For example, during a DATALOAD DC operation a numeric value was expected, but an alphanumeric value was read.

Recovery: Correct the program or the data, or verify that the proper file is being accessed.

ERR X75

Error: ILLEGAL NUMBER

Cause: The format of a number is illegal.

Recovery: Correct the number.

ERR X76

Error: BUFFER EXCEEDED

Cause: The buffer variable is too small or too large for the specified operation.

Recovery: Change the size of the buffer variable.

ERR X77

Error: INVALID PARTITION REFERENCE

Cause: The partition referenced by SELECT @PART or $RELEASE TERMINAL is not defined, or the name specified by DEFFN @PART has already been used.

Recovery: Use the proper partition name; wait for the global partition to be defined.

A.6 DISK ERRORS (RECOVERABLE)

ERR D80

Error: FILE NOT OPEN

Cause: The file was not opened.

Recovery: Open the file before attempting to read from it or write to it.

ERR D81

Error: FILE FULL

Cause: The file is full; no more information may be written into the file.

Recovery: Correct the program, or use MOVE to move the file to another platter and reserve additional space for it.

ERR D82

Error: FILE NOT IN CATALOG

Cause: A nonexistent file name was specified, or an attempt was made to load a data file as a program file or a program file as a data file.

Recovery: Make sure the correct file name is being used, the proper disk platter is mounted, and the proper disk drive is being accessed.

ERR D83

Error: FILE ALREADY CATALOGED

Cause: An attempt was made to catalog a file with a name that already exists in the Catalog Index.

Recovery: Use a different name, or catalog the file on a different platter.

ERR D84

Error: FILE NOT SCRATCHED

Cause: An attempt was made to rename or write over a file that has not been scratched.

Recovery: Scratch the file before renaming it.

ERR D85

Error: CATALOG INDEX FULL

Cause: There is no more room in the Catalog Index for a new name.

Recovery: Scratch any unwanted files and compress the catalog using a MOVE statement, or mount a new disk platter and create a new catalog.

ERR D86

Error: CATALOG END ERROR

Cause: The end of the Catalog Area is defined to fall within the Catalog Index, or an attempt has been made to move the end of the Catalog Area to fall within the area already occupied by cataloged files (with MOVE END), or there is no room left in the Catalog Area to store more information.

Recovery: Either correct the SCRATCH DISK or MOVE END statement, increase the size of the Catalog Area with MOVE END, scratch unwanted files and compress the catalog with MOVE, or open a new catalog on a separate platter.

ERR D87

Error: NO END-OF-FILE

Cause: No end-of-file record was recorded in the file by using either a DATASAVE DC END or a DATASAVE DA END statement and, therefore, none could be found by the DSKIP END statement.

Recovery: Correct the file by writing an end-of-file trailer after the last data record.

ERR D88

Error: WRONG RECORD TYPE

Cause: A program record was encountered when a data record was expected, or a data record was encountered when a program record was expected.

Recovery: Correct the program. Be sure the proper platter is mounted and be sure the proper drive is being accessed.

ERR D89

Error: SECTOR ADDRESS BEYOND END-OF-FILE

Cause: The sector address being accessed by the DATALOAD DC or DATASAVE DC operation is beyond the end-of-file. This error can be caused by a bad disk platter.

Recovery: Run the program again. If the error persists, use a different platter or reformat the platter. If the error still exists, contact your Wang Service Representative.

A.7 I/O ERRORS (RECOVERABLE)

ERR 190

Error: DISK HARDWARE ERROR

Cause: The disk did not respond properly to the system at the beginning of a read or write operation; the read or write has not been performed.

Recovery: Key RESET and run the program again. If the error persists, ensure that the disk unit is powered on and that all cables are properly connected. If the error still occurs, contact your Wang Service Representative.

ERR 191

Error: DISK HARDWARE ERROR

Cause: A disk hardware error occurred because the disk is not in file-ready position. If the disk is in LOAD mode or if the power is not turned on, for example, the disk is not in file-ready position and a disk hardware error is generated.

Recovery: Key RESET and run the program again. If the error recurs, check to ensure that the program is addressing the correct disk platter. Be sure the disk is turned on, properly set up for operation, and that all cables are properly connected. Set the disk into LOAD mode and then back into RUN mode by using the RUN/LOAD selection switch. If the error persists, call your Wang Service Representative.

> **NOTE:**
>
> The disk must *never* be left in LOAD mode for an extended period of time when the power is on.

ERR 192

Error: TIMEOUT ERROR

Cause: The device did not respond to the system in the proper amount of time (time-out). In the case of the disk, the read or write operation has not been performed.

Recovery: Key RESET and run the program again. If the error persists, be sure that the disk platter has been formatted. If the error still occurs, contact your Wang Service Representative.

ERR 193

Error: FORMAT ERROR

Cause: A format error was detected during a disk operation. This error indicates that certain sector-control information is invalid. If this error occurs during a read or write operation, the platter may need to be reformatted. If this error occurs during formatting, there may be a flaw on the platter's surface.

Recovery: Format the disk platter again. If the error persists, replace the media. If the error continues, call your Wang Service Representative.

ERR 194

Error: FORMAT KEY ENGAGED

Cause: The disk format key is engaged. The key should be engaged only when formatting a disk.

Recovery: Turn off the format key.

ERR 195

Error: DEVICE ERROR

Cause: A device fault occurred indicating that the disk could not perform the requested operation. This error may result from an attempt to write to a write-protected platter.

Recovery: If writing, make sure the platter is not write-protected. Repeat the operation. If the error persists, power the disk off and then on, and then repeat the operation. If the error still occurs, call your Wang Service Representative.

ERR 196

Error: DATA ERROR

Cause: For read operations, the checksum calculations (CRC or ECC) indicate that the data read is incorrect. The sector read may have been written incorrectly. For disk drives that perform error correction (ECC), the error correction attempt was unsuccessful. For write operations, the LRC calculation indicates that the data sent to the disk was incorrect. The data has not been written.

Recovery: For read errors, rewrite the data. If read errors persist, the disk platter should be reformatted. For write errors, the write operation should be repeated. If write errors persist, ensure that all cable connections are properly made and are tight. If either error persists, contact your Wang Service Representative.

ERR 197

Error: LONGITUDINAL REDUNDANCY CHECK ERROR

Cause: A longitudinal redundancy check error occurred when reading or writing a sector. Usually, this error indicates a transmission error between the disk and the CPU. However, the sector being accessed may have been previously written incorrectly.

Recovery: Run the program again. If the error persists, rewrite the flawed sector. If the error still persists, call your Wang Service Representative.

ERR 198

Error: ILLEGAL SECTOR ADDRESS OR PLATTER NOT MOUNTED

Cause: The disk sector being addressed is not on the disk, or the disk platter is not mounted. (The maximum legal sector address depends upon the disk model used.)

Recovery: Correct the program statement in error, or mount a platter in the specified drive.

ERR 199

Error: READ-AFTER-WRITE ERROR

Cause: The comparison of read-after-write to a disk sector failed, indicating that the information was not written properly. This error usually indicates that the disk platter is defective.

Recovery: Write the information again. If the error persists, try a new platter; if the error still persists, call your Wang Service Representative.

APPENDIX B BASIC-2 RULES OF SYNTAX

B.1 BASIC-2 SYNTAX SPECIFICATION RULES

The following editorial rules are used in this manual to define and illustrate the components of BASIC-2 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. Two forms of the RESTORE statement which are legal appear below:

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

4. Braces, { }, enclosing vertically stacked items indicate that one of the items is required. For example:

   COM { scalar-variable / array-variable }

   means that the COM statement elements can be either scalar-variables (e.g., C2) or array-variables (e.g., D(4,8)).

5. Ellipsis (. . .) indicates that the preceding item may occur only once or many times in succession. For example:

   INPUT variable [, variable,] . . .

6. Except within double quotation marks, BASIC-2 syntax ignores blanks.

7. When one or more items appear in sequence, these items or their replacements must appear in the specified order.

APPENDIX C COMPATIBILITY WITH OTHER 2200 SERIES CPU's

Wang Laboratories, Inc., has introduced a number of different CPU models in the 2200 series, including the 2200B, 2200C, 2200S, 2200T, 2200VP, and 2200MVP. Of these, the 2200VP and 2200MVP are the newest, most refined, and most powerful. The 2200VP and 2200MVP central processors support a completely new and more powerful version of the BASIC language, called "BASIC-2." The BASIC-2 language is fundamentally incompatible with the several versions of BASIC supported by earlier 2200 series CPU's. (These versions of BASIC are all subsets of the BASIC supported by the 2200T; henceforth, this version of BASIC will be referred to as "Wang BASIC" to distinguish it from BASIC-2.) In order, however, to provide downward compatibility with the other 2200 series CPU's, the BASIC-2 interpreter has been programmed to recognize the syntax of *both* the BASIC-2 instruction set *and* the Wang BASIC instruction set. Most programs written for 2200 CPU's other than the 2200VP/MVP can, therefore, be loaded and run on a 2200VP/MVP *without modification.* The remaining programs will require slight modification. Because the routines which interpret Wang BASIC syntax are a resident part of the BASIC-2 interpreter, the programmer can freely intermix BASIC-2 and Wang BASIC instructions within the same program.

The areas of incompatibility between the 2200VP/MVP and other 2200 series CPU's fall into two categories: Wang BASIC language features not supported by the 2200VP/MVP, and language incompatibilities resulting from improved 2200VP/MVP design. A third incompatibility problem which is independent of language may arise in time-dependent programs as a result of the increased processing speed and improved accuracy of the 2200VP/MVP.

C.1 WANG BASIC LANGUAGE FEATURES NOT SUPPORTED IN BASIC-2

There are two types of features not supported in BASIC-2 on the 2200VP and 2200MVP which limit the ability of Wang BASIC programs to be transferred from other 2200 series systems.

1. **Teletype, Tape Cassette, and Manual-Feed Card Reader Statements**

   Teletype, Tape Cassette, and Manual-Feed Mark Sense Card Reader statements are not supported on the 2200VP or 2200MVP. Versions of the LOAD, DATALOAD, and DATALOAD BT statements designed to access these devices will produce syntax and/or execution errors when the program is run on a 2200VP or 2200MVP. (In some situations, a $GIO statement can be used to emulate the nonsupported statement.) Note that there are a number of peripherals not supported on the 2200MVP.

2. **Lowercase Literal Strings**

   Lowercase literal strings are not supported in BASIC-2. Lowercase literals are a special form of literal enclosed in single quotes (e.g., 'ABC'); they are used to specify lowercase letters in Wang BASIC. (On the 2200VP and 2200MVP, lowercase letters are easily entered in normal character strings by placing the keyboard in A/a mode.) Occurrences of lowercase literal strings in a program are flagged as syntax errors by the BASIC-2 interpreter. The problem is easily corrected by changing the single quotes to double quotes.

C.2 INCOMPATIBILITIES RESULTING FROM DESIGN CHANGES

Several incompatibilities between Wang BASIC and BASIC-2 arise as a result of changes in the design of the 2200VP and 2200MVP internal architecture and in the implementation of certain BASIC-2 language features:

1. **ON ERROR GOTO Statement**

   The ON ERROR GOTO statement is supported as part of Wang BASIC, although it is not a part of the BASIC-2 instruction set. However, error codes on the 2200VP/MVP are

different from those used on other 2200 CPU's. Both the 2200VP and the 2200MVP employ a three-byte error code, which consists of a letter prefix and two digits, as opposed to the two-byte code utilized by other 2200 CPU's. For this reason, the variable designated to receive the error code in an ON ERROR GOTO statement should be an alpha-variable at least three bytes in length when this statement is used on the 2200VP or 2200MVP. (On earlier 2200 CPU's, a two-byte variable was sufficient.) If a two-byte variable is used for the error code on a 2200VP or 2200MVP, only the letter prefix and first digit of the error code will be returned. This incompatibility should not present a serious problem in most cases, however, because the error detection and control facilities supported by BASIC-2 (including the SELECT ERROR and ERROR statements and the ERR function) are significantly more powerful and versatile than the ON ERROR GOTO statement. The programmer is therefore encouraged to modify his programs with the improved BASIC-2 instructions and avoid using the ON ERROR GOTO statement whenever possible.

2. **Internal Representation of Array Subscripts (MAT SORT and MAT MOVE STATEMENT)**

The 2200VP and 2200MVP utilize an improved technique for representing the subscripts of one-dimensional arrays, leading to incompatibilities in the interpretation of subscript arrays produced by the MAT SORT statement and used by the MAT MOVE statement. In all 2200 series processors, array subscripts are represented internally as two-byte binary values. In earlier 2200 processors, however, one-dimensional arrays were treated as two-dimensional arrays with a single column and up to 255 rows. For example, the subscript for the fifth element of the one-dimensional array A$( ) would be represented internally as 0501, where the 05 signifies row 5 and the 01 signifies column 1. On the 2200VP and 2200MVP, this notation has been abandoned in favor of one in which the column subscript is implicit and both bytes of the binary value represent the row subscript for one-dimensional arrays. On the 2200VP and 2200MVP, therefore, the fifth element of the one-dimensional array A$( ) is represented as 0005. Using the entire two-byte value to represent the row subscript enables the 2200VP and 2200MVP to support a maximum one-dimensional array subscript of 65535 (the maximum value of a two-byte binary number). Previous models supported a maximum dimension of 255 (the maximum value of a one-byte binary number). In general, this difference is completely transparent to the programmer since it is a purely internal matter which does not determine how array subscripts are written in a BASIC program. There are two statements, however, which produce and interpret arrays containing subscripts in the system's two-byte internal binary format: MAT SORT and MAT MOVE. The subscript array produced by MAT SORT, called the "locator-array," typically is used as input to a MAT MOVE statement to produce an output data array. Ordinarily, there is *no* compatibility problem; when a BASIC program containing these statements is run on a 2200VP or 2200MVP, the locator-array created by MAT SORT will automatically receive subscripts in 2200VP/MVP format, and MAT MOVE will interpret them correctly. When the user has written a custom routine to either interpret the subscripts in a locator-array produced by MAT SORT, generate a locator-array for MAT MOVE, or use a locator-array generated on another 2200 CPU, however, the program must be modified to correctly represent and interpret the subscripts of one-dimensional arrays on a 2200VP or 2200MVP. One method which corrects this incompatibility in a program is the replacement of one-dimensional arrays with two-dimensional arrays having a second dimension of 1.

This procedure involves changing the original DIM statement (e.g., DIM A$(X) becomes DIM A$(X,1)) and modifying all references to the array within the program (e.g., references to A$(X) must be changed to A$(X,1)). In this case, the locator-array produced by MAT SORT will be the same on all systems. An alternate approach is to change references to the generated subscripts in the locator-array. Often this solution involves merely changing a one-byte VAL function to a two-byte VAL function (e.g., VAL(A$(X)) becomes VAL(A$(X),2)). Note that no incompatibility problem arises for two-dimensional arrays.

3. Operation of the PRINTUSING Statement

An anomaly in the operation of the PRINTUSING statement in Wang BASIC has been corrected in BASIC-2. On earlier 2200 processors, a PRINTUSING statement which referenced an image whose length exactly equalled the selected line width generated a *double* line feed after printing the image line. In BASIC-2, this anomaly has been corrected, and the PRINTUSING statement now generates single line feeds only.

4. Use of Colons in a PRINTUSING Image Statement

In Wang BASIC, the colon is not allowed as a legal character in a PRINTUSING Image statement. Rather, the colon is interpreted as a statement separator. Thus, the line

```
50 %+##.##: GOTO 100
```

is interpreted to contain two statements, the Image statement "%+##.##" and the GOTO statement "GOTO 100". In BASIC-2, however, a colon is allowed as a legal character in a PRINTUSING Image statement; it is *not* interpreted as a statement separator in this case. Thus, line 50 above would be interpreted as a *single* statement in which the characters ":GOTO 100" are regarded as a text string embedded in the Image statement. In BASIC-2, therefore, multiple statements are *not* allowed on the same line following an Image statement. Wang BASIC programs containing such multiple-statement lines must be modified (by moving all statements following the Image statement to a new line) before they will run correctly on a 2200VP or 2200MVP.

5. Setting and Checking the End-of-File Flag With an IF END THEN Statement

In Wang BASIC, the end-of-file flag is set while reading a disk file *only* when a software trailer record (written with a DATASAVE DC/DA END statement) is read. The hardware trailer record (written automatically by the system following each DATASAVE DC OPEN, DATASAVE DC END, or MOVE statement) does *not* cause the end-of-file flag to be set. Thus, in order to test for end-of-file with an IF END THEN statement, it was necessary to be certain that a software trailer was written in the file following the last data record. In BASIC-2, *both* a software trailer *and* a hardware trailer cause the end-of-file flag to be set when reading a file. IF END THEN will, therefore, detect the end-of-file even if no software trailer has been written in the file.

6. Reading Past the End-of-File With DATALOAD DC and DATALOAD DA Statements

In Wang BASIC, an attempt to read beyond the end-of-file with DATALOAD DC or DATALOAD DA produces an ERR 62. In BASIC-2, no error is generated when the program attempts to read beyond the end-of-file. Instead, the DATALOAD DC/DA statement continues to read the trailer record, and the variables in its argument list remain *unchanged,* retaining the values of the last valid data record read.

C.3 LANGUAGE-INDEPENDENT INCOMPATIBILITIES

There is a third class of incompatibility problems arising from changes and improvements to the 2200VP/MVP design, but not related to specific language features. Programs which are particularly sensitive to timing considerations or to the accuracy of mathematical computations may display unpredictable behavior when run on a 2200VP/MVP processor. There are two potential problem areas to consider:

1. **Problems in Time-Dependent Programs**

   The significantly improved processing speed of the 2200VP CPU and the variations in speed on the 2200MVP due to multiprogramming processing may cause problems for time-dependent programs. Prompts which remained on the screen long enough to be easily read when a program was run on an earlier 2200 model may flash by at unreadable speed when the same program is run on a 2200VP. This problem can be solved by using pauses or making changes in timing loops. More serious problems may arise in data capture or telecommunications programs which employ the $GIO statement to transmit and receive data. The programmer should carefully examine the discussion of data transfer speeds in Chapter 15, section 15.4 or utilize standard program packages designed for the 2200VP or 2200MVP.

2. **Problems Arising From Improved Math Accuracy**

   Math package functions and the MAT INVerse statement are generally more accurate on the 2200VP and 2200MVP than on other 2200 series CPU's. This increased accuracy may affect a few programs which rely on a specific value being returned by a certain expression.

C.4 COMPARISON OF BASIC-2 AND WANG BASIC INSTRUCTION SETS

The following tables present a complete listing of all instructions in the BASIC-2 and Wang BASIC instruction sets. These tables are provided to enable the programmer to readily compare the features of both languages. The tables assume that the reader has a general knowledge of Wang BASIC; many instructions are accompanied by comments indicating changes or improvements in the BASIC-2 version of the instruction over the corresponding Wang BASIC instruction. Note, however, that *both* instruction sets are *fully supported* by the 2200VP and 2200MVP (with the exceptions of tape cassette, manual-feed card reader, and teletype I/O instructions; see the preceding discussion). In general, the programmer need realize the differences *only* if he is coding programs which will be shared by a 2200VP or 2200MVP processor and another 2200 model. In this case, the use of BASIC-2 instructions must be avoided since these instructions are recognized only by the 2200VP and 2200MVP.

Table C-1.
Comparison of the Wang BASIC and BASIC-2 General Instruction Sets

> **NOTE:**
>
> Although BASIC-2 is the native language of the 2200VP and 2200MVP, *both* the Wang BASIC and BASIC-2 instruction sets are supported by these systems. Thus, a "No" in the BASIC-2 column indicates only that the instruction is not part of the BASIC-2 language; it does *not* imply that the instruction cannot be executed on a 2200VP or 2200MVP.

Instruction	Wang BASIC		BASIC-2	
ADD	Yes	(Statement)	Yes	(Alpha-expression operator)
ALL	No		Yes	
AND,OR,XOR	Yes	(Statements)	Yes	(Alpha-expression operators)
BIN	Yes	(Statement)	Yes	(Function: two-byte conversion supported)
BOOL	Yes	(Statement)	Yes	(Alpha-expression operator)
COM	Yes		Yes	(Variables can be used to specify dimensions.)
COM CLEAR	Yes		Yes	
CONVERT	Yes		Yes	(Extended image)
DAC	No		Yes	
DATA	Yes		Yes	(Hex values are legal.)
DEFFN	Yes		Yes	
DEFFN'	Yes		Yes	
DIM	Yes		Yes	(Variables may be used to specify dimensions, and numeric-scalar-variables are allowed.)
DSC	No		Yes	
END	Yes		Yes	
ERR	No		Yes	
ERROR	No		Yes	
EXP	Yes		Yes	

Instruction	Wang BASIC	BASIC-2	
FIX	No	Yes	
FOR/TO	Yes	Yes	
$FORMAT	No	Yes	
$GIO	Yes	Yes	(New microcommands and faster transfer speeds are provided.)
GOSUB	Yes	Yes	
GOSUB'	Yes	Yes	
GOTO	Yes	Yes	
HEX	Yes	Yes	
HEXPACK	No	Yes	
HEXPRINT	Yes	No	(Replaced by PRINT HEXOF)
HEXUNPACK	No	Yes	
IF END THEN	Yes	Yes	(A statement after THEN is allowed and an ELSE clause is supported.)
$IF ON	Yes	Yes	
$IF OFF	No	Yes	
IF/THEN	Yes	Yes	(Multiple conditions are allowed, a statement after THEN is allowed, and an ELSE clause is supported.)
Image (%)	Yes	Yes	(May be specified as literal in PRINTUSING statement or as value of an alpha-variable; expanded numeric formats)
INIT	Yes	No	(Variable initialization can be performed using ALL function)
INT	Yes	Yes	
INPUT	Yes	Yes	(Hex literal legal for INPUT message)
KEYIN	Yes	Yes	(Line-numbers optional, explicit device-address allowed)
LET	Yes	Yes	(Wider range of alpha expressions allowed)
LEN	Yes	Yes	

Instruction	Wang BASIC	BASIC-2	
LINPUT	No	Yes	
LOG	Yes	Yes	
LGT	No	Yes	
MAT Addition	Yes	Yes	
MAT CON	Yes	Yes	
MAT CONVERT	Yes	No	(Replaced by MAT MOVE variation)
MAT COPY	Yes	Yes	(Alpha-scalar-variables may be specified.)
MAT Equality	Yes	Yes	
MAT IDN	Yes	Yes	
MAT INPUT	Yes	Yes	
MAT INV	Yes	Yes	(Normalized determinant returned; error messages suppressed if determinant specified)
MAT MERGE	Yes	Yes	
MAT MOVE	Yes	Yes	(Automatic alpha-to-numeric and numeric-to-alpha conversions performed; locator-array optional)
MAT Multiplication	Yes	Yes	
MAT PRINT	Yes	Yes	
MAT READ	Yes	Yes	
MAT REDIM	Yes	Yes	
MAT Scalar Mult.	Yes	Yes	
MAT Subtraction	Yes	Yes	
MAT SEARCH	Yes	Yes	(Alpha-scalar-variable or literal can be searched; literal can specify target substring.)
MAT SORT	Yes	Yes	
MAT TRN	Yes	Yes	
MAT ZER	Yes	Yes	

Instruction	Wang BASIC	BASIC-2	
MAX	No	Yes	
MIN	No	Yes	
MOD	No	Yes	
NEXT	Yes	Yes	(Multiple indices allowed)
NUM	Yes	Yes	
ON ERROR GOTO	Yes	No	
ON/GOTO,GOSUB	Yes	Yes	(Line-numbers optional; alpha-variable may be used for expression)
ON/SELECT	No	Yes	
PACK	Yes	Yes	
$PACK	Yes	Yes	(Additional formats)
$PI	Yes	Yes	
POS	Yes	Yes	(Last occurrence supported. Target character can be specified by an alpha-variable. Can search a literal.)
PRINT	Yes	Yes	(Numeric value with absolute value which can be expressed in 14 digits is printed in normal format.)
PRINT AT	No	Yes	
PRINT HEXOF	No	Yes	
PRINTUSING	Yes	Yes	(Expanded image; hex literals legal)
PRINTUSING TO	No	Yes	
PRINT TAB	Yes	Yes	
READ	Yes	Yes	
REM	Yes	Yes	(Special formatting features added for LIST)
RESTORE	Yes	Yes	(Line-number of DATA statement may be specified.)
RETURN	Yes	Yes	
RETURN CLEAR	Yes	Yes	(ALL parameter added)
RND	Yes	Yes	(New algorithm)

Instruction	Wang BASIC	BASIC-2	
ROTATE	Yes	Yes	(Entire string rotation supported; left or right rotation)
ROUND	No	Yes	
SELECT	Yes	Yes	
SELECT ERROR	No	Yes	
SELECT LINE	No	Yes	
SELECT ON,OFF	No	Yes	
SELECT [NO] ROUND	No	Yes	
SGN	Yes	Yes	
SPACE	No	Yes	
SQR	Yes	Yes	(Increased accuracy)
STOP	Yes	Yes	(Hex literals allowed; line-number can be displayed)
STR	Yes	Yes	(Arrays may be specified; starting byte optional)
SUB	No	Yes	
Trig Functions (All)	Yes	Yes	(Increased accuracy)
$TRAN	Yes	Yes	(Literal can be used as table.)
UNPACK	Yes	Yes	
$UNPACK	Yes	Yes	(Additional format)
VAL	Yes	Yes	(Two-byte conversion supported)
VER	No	Yes	

Table C-2.
Comparison of the Wang BASIC and BASIC-2
System Commands

Command	Wang BASIC	BASIC-2	
CLEAR	Yes	Yes	(Line-numbers optional in CLEAR P)
CONTINUE	Yes	Yes	(Immediate Mode statements)
HALT/STEP	Yes	Yes	
LIST S	Yes	Yes	(Programmable; title may be specified.)
LIST D	No	Yes	
LIST DT	No	Yes	
LIST I	No	Yes	
LIST V	No	Yes	
LIST #	No	Yes	
LIST'	No	Yes	
RENUMBER	Yes	Yes	(Expanded flexibility, clearer syntax)
RESET	Yes	Yes	(CI and CO automatically reselected to keyboard and CRT, respectively.)
RUN	Yes	Yes	(Program execution can be started at a specified statement within a line.)
Special Function Keys	Yes	Yes	(INPUT logic used for subroutine entry)
TRACE	Yes	Yes	(Clearer, more readable output)

Table C-3.
Comparison of the Wang BASIC and BASIC-2 I/O Instructions

I/O Instruction	Wang BASIC	BASIC-2	
BACKSPACE (tape drive)	Yes	No	(Device not supported)
COPY	Yes	No	(Replaced by COPY TO)
COPY TO	No	Yes	(Drive-to-drive copy)
DATALOAD (2214 Manual-Feed Card Reader)	Yes	No	(Manual-Feed Card Reader not supported)
DATALOAD (tape cassette drive)	Yes	No	(Device not supported)
DATALOAD (teletype)	Yes	No	(Teletype not supported)
DATALOAD (all other devices)	Yes	Yes	
DATALOAD BA	Yes	Yes	(Return-variable optional)
DATALOAD BT (2214 Manual-Feed Card Reader)	Yes	No	(Device not supported)
DATALOAD BT (tape cassette drive)	Yes	No	(Device not supported)
DATALOAD BT (teletype)	Yes	No	(Device not supported)
DATALOAD BT (all other devices)	Yes	Yes	(Certain devices not supported on 2200MVP; see *2200MVP Introductory Manual.)*
DATALOAD DA	Yes	Yes	(Return-variable optional)
DATALOAD DC	Yes	Yes	
DATALOAD DC OPEN	Yes	Yes	
DATASAVE (tape cassette drive)	Yes	No	(Device not supported)
DATASAVE (teletype)	Yes	No	(Device not supported)
DATASAVE (all other devices)	Yes	Yes	(Certain devices not supported on 2200MVP; see *2200 MVP Introductory Manual.)*
DATASAVE BA	Yes	Yes	(Return-variable optional)

I/O Instruction	Wang BASIC	BASIC-2	
DATASAVE BT (tape cassette drive)	Yes	No	(Device not supported)
DATASAVE BT (teletype)	Yes	No	(Device not supported)
DATASAVE BT (all other devices)	Yes	Yes	(Certain devices not supported on 2200MVP; see *2200 MVP Introductory Manual.)*
DATASAVE DA	Yes	Yes	(Return-variable optional)
DATASAVE DC	Yes	Yes	
DATASAVE DC CLOSE	Yes	Yes	
DATASAVE DC END	Yes	Yes	
DATASAVE DC OPEN	Yes	Yes	(Syntax for renaming scratched file conforms to SAVE DC.)
DATARESAVE (tape cassette drive)	Yes	No	(Device not supported)
DBACKSPACE	Yes	Yes	
DSKIP	Yes	Yes	
FILE NUMBERS	Yes (0-6)	Yes	(0-15)
LIMITS	Yes	Yes	(Status of file may be returned.)
LIST DC	Yes	Yes	("S" parameter allowed; available free space returned for each file.)
LOAD (2214 Manual-Feed Card Reader)	Yes	No	(Device not supported)
LOAD (tape cassette drive)	Yes	No	(Device not supported)
LOAD (teletype)	Yes	No	(Device not supported)
LOAD (all other devices)	Yes	Yes	(Certain devices not supported on 2200MVP; see *2200MVP Introductory Manual.)*
LOAD DA	Yes	Yes	(Return-variable is optional.)

I/O Instruction	Wang BASIC	BASIC-2	
LOAD DC	Yes	Yes	("DC" parameter optional; new "BEG" parameter enables execution to begin at any specified line; multiple program module loading capability.)
LOAD RUN	No	Yes	
MOVE	Yes	No	(Replaced by MOVE TO)
MOVE TO	No	Yes	(Drive-to-drive move; individual files can be moved.)
MOVE END	Yes	Yes	
REWIND (tape cassette drive)	Yes	No	(Device not supported)
SAVE DA	Yes	Yes	(Return-variable is optional.)
SAVE DC	Yes	Yes	(Programmable; "DC" parameter optional; new "!" parameter provides more sophisticated program protection capability; ability to automatically delete REM's and spaces from saved program.)
SCRATCH	Yes	Yes	
SCRATCH DISK	Yes	Yes	
SKIP (tape cassette drive)	Yes	No	(Device not supported)
VERIFY	Yes	Yes	(Location of bad sector can be returned in a variable without a system error message.)

Table C-4.
Special Instructions for the 2200MVP

> **NOTE:**
>
> These instructions operate on the 2200VP as if it were a single-terminal, single-partition MVP, with the exception of $MSG. That instruction returns a value of HEX(00) on the 2200VP. These instructions are illegal on other 2200 CPU's.

Instruction	Purpose
$BREAK statement	Temporarily inhibits processing in a specified partition.
$CLOSE statement	Releases one or more specified I/O devices previously seized by the current partition with a $OPEN statement.
DEFFN @PART statement	Defines a global partition.
$INIT statement	Passes configuration parameters to the MVP operating system and causes the system to be initialized to the specified configuration.
$MSG function	Defines a message at terminal #1 to be displayed at other terminals when READY is displayed.
$OPEN function	Seizes one or more specified I/O devices for exclusive use by the current partition.
#PART function	Returns the partition number of the current partition.
$RELEASE PART statement	Makes a partition available to another terminal.
$RELEASE TERMINAL statement	Detaches the terminal from the current partition, enabling it to become attached to another partition waiting to communicate with it.
SELECT @PART statement	Specifies a global partition whose program text and/or global variables are to be used by the current program.
#TERM function	Returns the terminal number of the terminal currently assigned to this partition.

APPENDIX D SERIES 2200 DEVICE-ADDRESSES

This appendix lists the standard and alternate device-types and device-addresses for various classes of I/O devices on the 2200VP and 2200MVP as well as for several specialized controllers.

Device or Class	Device-Type Usual, (Alternates)	Usual Address	Alternate Addresses
CRT	0	05	-
Keyboard	0	01	-
2236D Terminal Printer	2,(0)	04	-
2236D Terminal Plotter	C,(4)	04	-
Printers	2,(0)	15	16
Plotters	C,(4)	13	14,15,16
Disk Drives	3,(B,D)	10	20,30
Second drive or third floppy drive	3,(B,D)	50	60,70
Nine-Track Tape Transport	0	7B	7D,7F
Card Reader	6,(4)	28-2F*	-
2207A RS-232-C I/O Controller			
Input	0	19	1A,1B
Output	0	1D	1E,1F
2228B Synchronous/Asynchronous Communications Controller			
Input	0	1C	1A-1F
Output	0	9C	9A-9F
2550 8-Bit Parallel I/O Controller			
Input	0	3A	3C,3E
Output	0	3B	3D,3F
2252A Scanning Input Interface Controller (BCD 1-10 Digit Parallel)	0	5A	5B-5F
2254 IEEE-488 Interface Controller			
General	0	4C	-
Keyin	0	4D	-
$IF ON	0	4E	-

*All listed addresses are required by the single device.

APPENDIX E 2200VP AND 2200MVP CPU SPECIFICATIONS

Dynamic Range:

$-10^{+100} < n <= -10^{-99}, 0, 10^{-99} <= n < 10^{+100}$

Accuracy:

13 digits (typical)

Memory Size:

VP: 32 bytes (standard). Expandable to 64K bytes.

MVP: 32 bytes (standard). Expandable to 64K, 128K, 196K or 256K bytes.

Power Requirements:

Voltage:	115 VAC ± 10%, 60Hz ± 1 cps 230 VAC ± 10%, 50Hz ± 1 cps
Power:	230 Watts
Fuses:	3.0A(SB) @ 115V 1.5A(SB) @ 230V

Operating Environment:

50°F to 90°F (10°C to 32°C)
20% to 80% relative humidity, noncondensing (maximum range permitted)
35% to 65% relative humidity (recommended range)

CPU Dimensions:

Height	16.0 in.(40.6 cm)
Width	24.8 in.(63.0 cm)
Depth	10.5 in.(26.7 cm)

Shipping Weight:

47 lbs

APPENDIX F CRT CHARACTER SETS

F.1 24 x 80 CRT CHARACTER SET

CONTROL CODES					
HEX	ACTION	HEX	ACTION	HEX	
00	NULL	06	CURSOR OFF	0A	CURSOR ↓
01	HOME CURSOR	07	AUDIBLE TONE	0C	CURSOR ↑
03	CLEAR SCREEN, HOME CURSOR	08	BACKSPACE	0D	CARRIAGE RETURN
05	CURSOR ON	09	NON-DESTRUC-TIVE SPACE		

CHARACTERS															
HEX	CHAR	HEX	CHAR	HEX	CHAR	HEX	CHAR	HEX	CHAR	HEX	CHAR	HEX	CHAR	HEX	CHAR
10	â	20	SPACE	30	0	40	@	50	P	60	°	70	p	80	NULL
11	ê	21	!	31	1	41	A	51	Q	61	a	71	q	81	◆
12	î	22	"	32	2	42	B	52	R	62	b	72	r	82	▶
13	ô	23	#	33	3	43	C	53	S	63	c	73	s	83	◀
14	û	24	$	34	4	44	D	54	T	64	d	74	t	84	→
15	ä	25	%	35	5	45	E	55	U	65	e	75	u	85	←
16	ë	26	&	36	6	46	F	56	V	66	f	76	v	86	|
17	ï	27	'	37	7	47	G	57	W	67	g	77	w	87	¨
18	ö	28	(	38	8	48	H	58	X	68	h	78	x	88	´
19	ü	29	)	39	9	49	I	59	Y	69	i	79	y	89	`
1A	à	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z	8A	^
1B	è	2B	+	3B	;	4B	K	5B	[	6B	k	7B	ß	8B	■
1C	ù	2C	,	3C	<	4C	L	5C	\	6C	l	7C	£	8C	!!
1D	Ä	2D	—	3D	=	4D	M	5D	]	6D	m	7D	é	8D	↕
1E	Ö	2E	.	3E	>	4E	N	5E	↑	6E	n	7E	ç	8E	β
1F	Ü	2F	/	3F	?	4F	O	5F	←	6F	o	7F	¢	8F	¶

F.2 16 x 64 CRT CHARACTER SET

High Order Hexadecimal Digit of Code

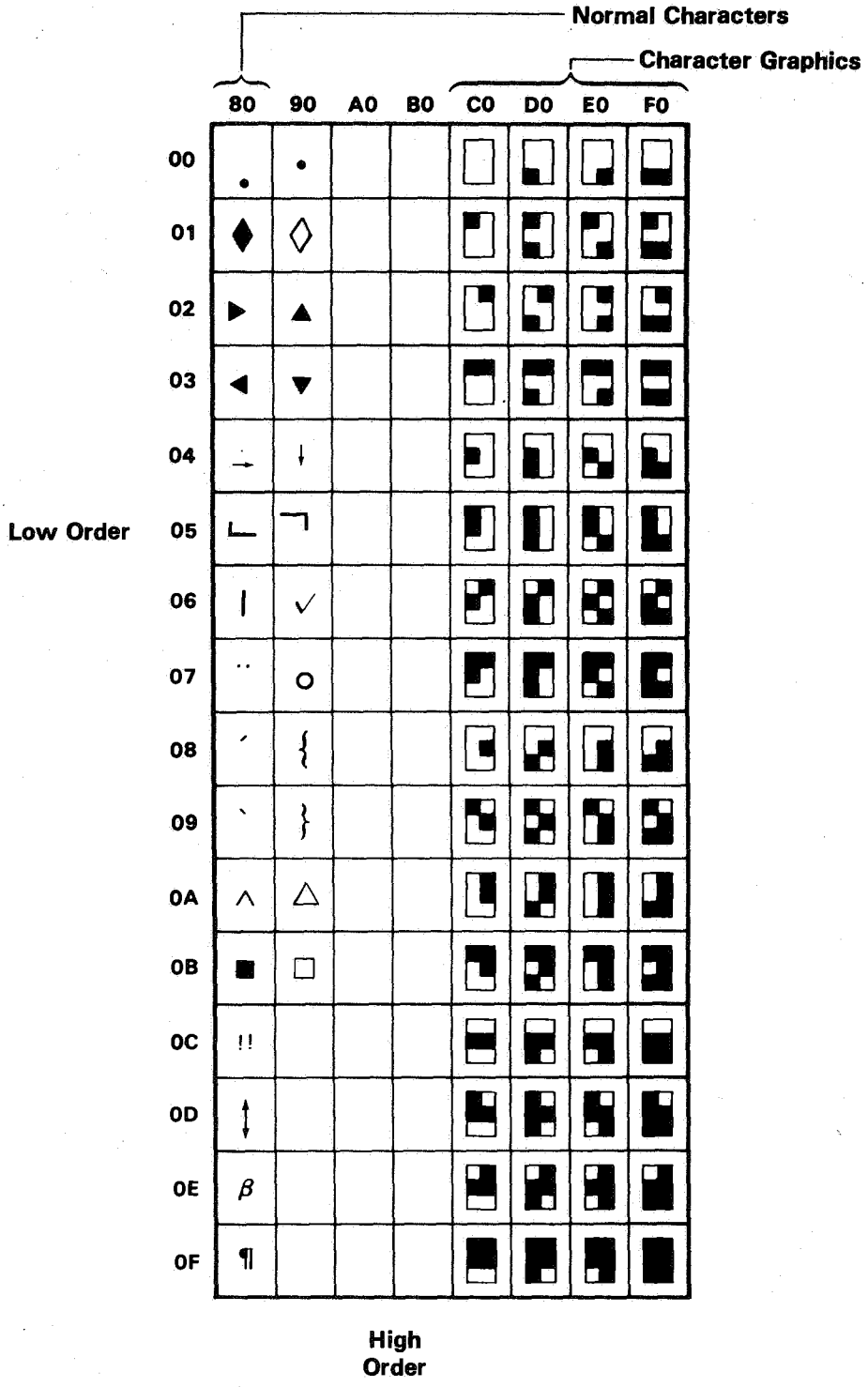
Low Order Hexadecimal Digit Of Code

	0	1	2	3	4	5	6	7
0	NULL		SPACE	Ø	@	P	prime	p
1	HOME (CRT)	X-ON	!	1	A	Q	a	q
2			"	2	B	R	b	r
3	CLEAR SCREEN (CRT)	X-OFF	#	3	C	S	c	s
4			$	4	D	T	d	t
5			&	5	E	U	e	u
6			&	6	F	V	f	v
7	BELL		' (apos)	7	G	W	g	w
8	BACKSPACE (CRT CURSOR ←)		(	8	H	X	h	x
9	HT(TAB) or (CRT CURSOR →)	CLEAR TAB	)	9	I	Y	i	y
A	LINE FEED (CRT CURSOR ↓)	SET TAB	*	:	J	Z	j	z
B	VT (VERTICAL TAB)		+	;	K	[	k	
C	FORM FEED OR REV. INDEX (CRT CURSOR ↑)		,	< or [	L	\	l	
D	CR (CARRIAGE RETURN)		-	=	M	]	m	
E	SO	¢	.	> or]	N	↑ or ^ or !	n	~
F	SI	°	/	?	O	← or _	o	■

NOTE:

Codes 60 to 7F are available only on an upper/lower-case CRT and on a Model 2221W printer.

F.3 STANDARD USA 2236DE UPPER CHARACTER SET



APPENDIX G GLOSSARY OF TERMS AND KEYWORDS

G.1 GLOSSARY OF TERMS

alpha: Short for "alphanumeric."

alphanumeric: One of the two BASIC data types; capable of containing a sequence of characters from the ASCII character set.

argument: An item provided as input to a function or subroutine. The type of argument generally must conform to rules set by the called function or subroutine.

array: A group of variables referred to by the same name in which one or two numeric expressions (subscripts) select particular variables (elements).

array-designator: An item in the BASIC language which represents all elements of an array, taken in subscript or row-by-row order. Denoted by the array name followed by opening and closing parentheses, but with no subscripts inside the parentheses, e.g., E( ) and M9$( ).

ASCII: The abbreviation for the American Standard Code for Information Interchange. The ASCII code pairs numbers to displayable characters and is used to represent alpha data in BASIC-2 where applicable.

binary coded decimal (BCD): Form of numeric representation in which the binary value of each byte represents two decimal digits. This form is used by the PACK statement to store numeric values in alpha-variables. (See the entry on "packed decimal" format.)

binary value: Value expressed in base two notation.

Boolean: A function which logically operates on a value on the basis of its individual bits. The AND, BOOL, OR, and XOR operators perform Boolean functions.

character: A number, letter, or symbol: the elementary unit of information in the BASIC "alpha" data type.

concatenate: Combine two or more BASIC alpha-expressions to form a long string in which the last character from the first expression precedes the first character from the second expression. Also, to combine program lines with EDIT/RECALL. Represented in BASIC by the "&" operator.

constant: An item in the BASIC language whose value is self-defining and not subject to change. In this manual, constants are classified by type (numerics or literal strings).

device-address: The number corresponding to the controller-set hardware address of an I/O device. It is most commonly specified in the form "/taa", where "t" is the device-type and "aa" is the device-address, which corresponds to the switch setting in the controller.

device table: Table containing the addresses of all peripheral devices available to system users and the status of each device. Information for each device is contained in a "slot" of the table. There are 16 slots available, numbered from 0 to 15. In the case of disks, the starting, ending, and current sectors of the file being accessed (if any) are listed in the slot corresponding to that device.

device-type: The character describing to the operating system the characteristics of the addressed device, instructing the system as to the protocol for I/O operations. Commonly specified as the "t" in "/taa", the device-address.

element: An element of an array is one of the individual variables which comprise that array. The word "element" is also used to refer to an item in the BASIC language which may appear in a particular situation.

expression: A general term for an item in the BASIC language constructed from operands, operators, and functions according to proper syntax. For example, in the statement "X=Y+4*#PI", "Y+4*#PI" is an expression.

file: A set of related data on disk; the unit of data by which a BASIC program gains access to data stored on disk.

file-number: A numeric expression preceded by a "#" character that references a slot in the device table.

function: An item in the BASIC language which takes as input one or more values and produces a value which can be used in further operations by the program such as ARCTAN(x), ROUND(x,y), and VAL(x), where x and y are appropriate expressions.

global partition: On the MVP, a partition which through explicit declaration allows its program text to be used by any requesting partition and permits its variables to be updated by this partition.

global text: Program text contained in a global partition available to any requesting partition.

global variable: A variable of the form "@variable-name" which can be modified by a user in a partition other than the one declaring the variable initially.

hexadecimal: Notation in base 16. A shorthand notation for binary data. Allowable digits are 0-9 and A-F.

Immediate Mode: Mode in which statements or commands are executed without line-numbers; contrasts with Program Mode.

interrupt: A subroutine call that occurs because a device ready/busy signal has been set to a desired state (contrast with a GOSUB or GOSUB' statement). Such subroutine calls are termed "interrupts" because they interrupt normal program flow at an unpredictable time. In BASIC-2, interrupts are programmable using the SELECT ON/OFF statement. All interrupts are disabled unless specifically enabled.

keyword: A sequence of letters which are interpreted as a unit by the BASIC interpreter. Examples of keywords are LET, IF, THEN, and ARCTAN.

leading: Instances of a character occurring to the left of all instances of any other character. For example, there are two leading zeros in "003320".

length: The "defined" length of an alpha value is the number of characters which it contains. For variables and array strings, this is defined in the COM, DIM, or MAT REDIM statements. The "actual" or "current" length of an alpha-variable or array string is the length of the string excluding trailing blanks. A string which contains all blanks, however, has an "actual" length of one.

line-number: An integer from 0000 to 9999 inclusive, placed at the beginning of each line of program text to define its place in the sequence of program lines. Also used as a label for program branches and subroutines.

literal-string: An alpha constant; typically a sequence of characters enclosed in quotes or expressed using hexadecimal notation. "hello, ABCDEF" and HEX(6162CF0F) are literals.

locator-array: An array created during a MAT SORT operation containing the subscripts of the arrays being sorted in ascending order. Used by the sort routines to create a sorted, merged file from one or more other files.

matrix: A two-dimensional array for which two numeric expressions are required to uniquely identify an element.

numeric: Floating-point; as opposed to alpha.

operand: A constant, variable, or any expression "operated upon" and taken as input by a statement, operator, or function.

operating system: The control microprogram for driving the BASIC-2 interpreter, contained in control memory. It is not modifiable by the user.

operator: A symbol which operates upon two expressions, indicating that, at execution time, their values should be combined using a particular function. The "+" symbol is an operator signifying the addition function; the example "A+B" illustrates the use of an operator indicating that the operands "A" and "B" are to be combined by using the addition function.

packed decimal: Numeric format created by the PACK statement in which numbers are stored in alpha-variables. Numbers are stored in binary coded decimal form (BCD), two digits per byte. Useful for storing and operating upon numbers with more than 13 digits of precision.

partition: A section of user memory within the MVP which is reserved for an individual user and which no other user may access unless it is declared "global."

Program Mode: Mode in which statements and commands are not executed immediately, but rather in line-number order. Contrasts with Immediate Mode.

receiver: An item which can receive and store a new value. Variables and portions of alpha-variables specified by the STR function are examples of valid receivers.

scalar: A variable which has a single value associated with it, not an array or array-element.

statement: The smallest unit in the BASIC language capable of invoking a complete action.

string: A sequence of characters. String data is contained in a BASIC alpha-variable.

subscript: A numeric expression used to select one of the elements of an array. Depending upon the array, either one or two subscripts may be required to uniquely select an array-element.

substring: A portion of an alpha-variable, defined by relative starting position and length, using a STR expression. For example, STR(A$,4,3) represents the substring consisting of three characters from A$ beginning at the fourth.

trailing: Instances of a character in a string which are located to the right of all instances of any other characters. There are two trailing blanks in "ABC ". The zeros in the number "200", however, are not called trailing zeros because in references to numbers, "trailing" also implies a digit occurring to the right of a decimal point.

value: The value of an expression is the string or number obtained when the listed operands are combined using the specified operators and functions; the value of a variable is the current contents of that variable (the string or number which that variable stands for at the present time). For variables, the current value is always that which has been most recently assigned to it.

variable: A named area of memory whose value may change during the execution of a program. In this manual, variables are classified by type as either alpha or numeric.

variable name: An item in the BASIC language which represents the current value of a particular variable.

vector: A one-dimensional array which requires only one subscript to uniquely select an element.

G.2 BASIC-2 KEYWORDS

The following is a partial glossary of BASIC-2 keywords, including all verbs, functions, and operators.

ABS
function: Returns the absolute value of an expression.

ADD: Adds binary values of two arguments in a logical expression one byte at a time and places result in alpha-receiver.

ADDC: Like ADD, but treats the two arguments as multibyte binary numbers.

ALL
function: Used in logical expressions to generate an argument consisting entirely of the same specified character.

AND:
1. Logically ANDs two arguments one byte at a time.
2. AND of two relations in an IF statement.

ARCCOS
function: Returns the arccosine of an expression.

ARCSIN
function: Returns the arcsine of an expression.

ARCTAN
function: Returns the arctangent of an expression.

ATN
function: Same as ARCTAN.

BIN
function: Converts integer value of an expression to an alphanumeric value which is the binary equivalent of the expression. Inverse of VAL function.

BOOLh: Does one of 16 possible logical (Boolean) operations on alpha values depending upon the value of h.

$BREAK: Relinquishes a specified amount of the current partition's execution time for use by other partitions. The $BREAK statement is executed on the MVP only.

CLEAR: Clears user memory (or, optionally, program text, all variables, or noncommon variables) and returns control to the operating system.

$CLOSE: Relinquishes exclusive control of a device gained by the current user executing a $OPEN statement. The $CLOSE statement can be executed on the MVP only.

COM: Establishes common storage area for variables used by more than one program. Like DIM, it reserves space for arrays and sets length for alpha-variables.

COM CLEAR: Repositions the Common Variable Pointer in memory, causing certain common variables to become noncommon.

Concatenation operation (&): Combines two strings, the second being put directly after the first without intervening characters. The result is treated as a single string.

CONTINUE: Resumes program execution following a HALT/STEP command.

CONVERT:

1. Converts a number represented by ASCII characters in an alphanumeric expression to a numeric value and sets a numeric-variable equal to that value.
2. Converts a numeric value to an ASCII character string representing it and places that string in an alpha-receiver in a specified format.

COPY TO: Copies the entire contents of one specified disk platter to another, including information not stored in the catalog area and scratched files.

COS function: Returns the cosine of an expression.

DAC: Performs decimal addition on a pair of unsigned packed decimal numbers, with carry.

DATA: Provides DATA values which can be used by variables in a READ statement, enabling constants to be stored within a program.

DATALOAD: Loads data from one or more cards in a card reader into memory. Also, loads data from a paper tape into memory.

DATALOAD BA: Loads one sector of unformatted data from disk (blocked data in Direct Addressing Mode). Not normally used if the data is stored within the catalog area on disk.

DATALOAD BT: Loads data from one card in a card reader into memory or loads one set of data from a paper tape into memory.

DATALOAD DA: Reads one or more logical records from the disk, starting at the absolute sector address specified. The data must be in standard 2200 format.

DATALOAD DC: Reads logical data records from a cataloged disk file and sequentially assigns them to the specified variables.

DATALOAD DC OPEN: Opens a data file that has been previously cataloged on disk. May be used to reopen a temporary work file (TEMP file).

DATASAVE: Performs a read lookahead operation on a card reader.

DATASAVE BA: Saves data on disk with no accompanying control bytes (block format).

DATASAVE DA:	Saves data on disk in Absolute Sector Addressing Mode.
DATASAVE DC:	Saves one logical record on disk beginning at the current sector address specified in the device table for the current file-number.
DATASAVE DC CLOSE:	Closes one or all currently open data files.
DATASAVE DC END:	Writes an end-of-file record in a cataloged data file at the current sector address.
DATASAVE DC OPEN:	Reserves space for a data file on disk and stores this information in the catalog area of the disk.
DBACKSPACE:	Backspaces over the specified number of logical records or sectors in a data file. May optionally backspace to the beginning of the file.
DEFFN:	Defines a single-valued, user-written numeric function referenced by FN.
DEFFN':	1. Defines SF Key or program entry point for subroutine with argument-passing capability. 2. Defines literal to be supplied for text entry when SF Key is used.
DEFFN @PART:	Defines the current partition to be global and assigns a global name which may be used by other partitions to call the global partition. The DEFFN @PART statement can be executed on the MVP only.
DIM:	Reserves space for arrays and sets the length for alpha-variables.
DSC:	Performs decimal subtraction with carry on a pair of unsigned packed decimal numbers.
DSKIP:	Skips over the specified number of logical records or sectors in a data file. May optionally skip to the current end-of-file record.
END:	Terminates program prior to physical end and returns amount of free space available in user memory.
ERR:	Returns the number of the most recent error condition. After reference is made, the value of the ERR function returns to zero.
ERROR:	Suppresses the normal error message when a recoverable error occurs and branches to a user-specified routine.
EXP function:	Finds the value of "e" raised to the value of the expression.
FIX:	Returns the integer portion of the expression.
FOR/TO:	Sets up loop control information, providing the initial counter value, final counter value, and step value.
$FORMAT:	Creates a format specification for the field forms of $PACK and $UNPACK.

FN function:	Calls a function previously defined in a DEFFN statement.
FOR:	Initiates a loop ending with a NEXT statement.
$GIO:	Specifies a special I/O routine by a series of microcommands. Allows programmable control of devices not normally supported.
GOSUB:	Transfers control to first program line of an internal subroutine.
GOSUB':	Transfers control to an internal subroutine marked by DEFFN'; unlike GOSUB, can pass arguments.
GOTO:	Transfers control to specified line-number.
HEX literal:	Allows the user to supply ASCII code for characters for which no keyboard characters exist.
HEXPACK:	Converts an ASCII character string representing a string of hex digits into the binary equivalent of those digits.
HEXUNPACK:	Converts the binary value of an alpha-expression to a string of ASCII characters representing the hexadecimal equivalent of that value.
IF END THEN:	Branches to the specified line-number or executes the specified statement if an end-of-file record is read.
IF THEN ELSE:	Tests relation and causes conditional transfer or statement execution based on result of test.
Image(%):	Used with PRINTUSING to format output.
$INIT:	Passes system configuration information to the MVP operating system to allow partition generation to occur.
INPUT:	1. Allows user to supply data during program execution. 2. In conjunction with DEFFN' statement, allows user to enter defined text or branch to marked subroutine by using a SF Key.
INT function:	Returns the largest integer less than or equal to the value of an expression.
KEYIN:	Receives a single character from an input device.
LEN function:	Determines the actual length in bytes of an alpha value.
LET:	Assigns the value of an expression to one or more receivers.
LGT function:	Returns the base ten logarithm of an expression.

LIMITS: Obtains the beginning and ending sector addresses of the current data file and either the current sector address for an Absolute Sector Address file or the number of sectors in a cataloged file. Also determines the status of the specified file.

LINPUT: Allows input and concurrent editing of an alpha-variable directly from the keyboard.

LIST: Lists the current program text on the device currently selected for LIST operations. On the CRT, LIST may be set to output only one screenful of lines at a time by using the *S* parameter. The *D* parameter causes each statement within a program line to be listed separately.

LIST DC: Lists the contents of the catalog area of the specified disk platter on the device currently selected for LIST operations.

LIST DT: Lists the contents of the device table on the device currently selected for LIST operations.

LIST I: Lists all currently defined interrupts on the device currently selected for LIST operations.

LIST T: Lists all occurrences in the current program text of the specified character string on the device currently selected for LIST operations.

LIST V: Lists the line-numbers of all occurrences of the specified variables in the current program text on the device currently selected for LIST operations.

LIST #: Lists the line-numbers of all lines which reference the specified line(s) in GOTO or GOSUB statements.

LIST ': Lists the line-numbers of all lines which reference the specified labeled subroutine in a DEFFN' or a GOSUB' statement.

LOAD command: Loads the specified BASIC program or program segment from the specified platter and appends it to the program currently in memory.

LOAD statement: Loads a BASIC program or program segment into memory and automatically executes it. Optionally deletes only a portion of the existing program and may also begin execution of the new text at any line.

LOAD DA command: Loads BASIC program text within the specified sectors into memory and appends it to the existing text. The specified starting sector must contain a program header record.

LOAD DA statement: Similar to LOAD command, but text is loaded in Absolute Sector Addressing Mode.

LOAD RUN command: Clears memory, loads in the specified catalog program from the specified platter, and immediately executes it. If no program name is specified, the program called "START" is assumed to be specified.

LOG function: Returns the natural logarithm of an expression.

MAT +: Adds two arrays of the same dimensions.

MAT CON: Sets all elements of an array equal to 1. Can also redimension an array.

MAT COPY: Transfers an alpha value to an alpha-receiver one byte at a time.

MAT =: Replaces each element of one array with corresponding element of another array. Redimensions first array to conform to second array.

MAT IDN (MAT identity): Causes specified matrix to assume form of matrix identity.

MAT INPUT: Allows user to supply values for an array from the keyboard during program execution.

MAT INV (MAT inverse): Causes one matrix to be replaced by inverse of another matrix.

MAT MERGE: Performs merge of two or more sorted input files into one sorted output file.

MAT *: Stores product of two arrays in a third array.

MAT PRINT: Prints arrays.

MAT READ: Assigns values contained in DATA statements to array-variables without referencing each member of the array individually.

MAT REDIM: Redimensions an array.

MAT ()* (MAT scalar multiplication): Multiplies each element in an array by an expression. Result is stored in a second array or in the same array.

MAT – (MAT subtraction): Subtracts numeric-arrays of same dimensions.

MAT SEARCH: Searches alpha value for strings of the same length as a second alpha-expression which satisfy one of the following relations:

$$\left\{\begin{array}{l} > \\ >= \\ < \\ <= \\ = \\ <> \end{array}\right\}$$

and places starting positions of substrings satisfying the relationship into numeric-array or alpha receiver.

MAT SORT: Creates a locator-array consisting of the subscripts of the array to be sorted in ascending order.

MAT TRN (MAT transpose): Causes one array to be replaced by transpose of another array.

MAT ZER: Sets all elements of an array to zero. Can redimension an array.

MAX function: Returns maximum value in numeric list.

MIN function: Returns minimum value in numeric list.

MOD function: Returns value of first expression modulo second expression.

MOVE TO: Moves the contents of the catalog area of one specified disk platter to another. No scratched files are moved.

MOVE END: Increases or decreases the size of the catalog area on a disk platter by moving the end record set by the SCRATCH DISK statement.

$MSG:
1. Returns the broadcast message specified by terminal #1.
2. When executed by terminal #1, specifies a system broadcast message of up to 80 bytes. The $MSG statement can be executed on the MVP only.

NEXT: Marks the end of a loop initiated by FOR.

NUM function: Counts number of sequential ASCII characters in an alpha-expression that represent a legal BASIC number.

ON GOTO or ON GOSUB: Computed GOTO or GOSUB statement. Branches to one of a number of lines depending upon the value of the expression following the ON clause.

ON/SELECT: Conditional SELECT statement. Performs the same operations as the SELECT statement based on the value of the specified argument.

$OPEN: Allows the current partition to acquire exclusive use of a device ("hogs" the device) until CLEAR is keyed or a $CLOSE statement is executed. The $OPEN statement can be executed on the MVP only.

OR:
1. Logically ORs two arguments in a logical expression.
2. OR of two relations in an IF statement.

PACK: Packs numeric values into an alphanumeric receiver, reducing storage requirements.

$PACK: Packs numeric and character data into an alpha receiver in a user-specified format.

#PART: Returns a numeric value equal to the number of the current partition. This function can be used only on the MVP.

#PI function: Assigns the value 3.14159265359.

POS function:	Searches an alpha value for a character that satisfies a defined relationship to another alpha value and outputs this character's position.
PRINT:	Sends output to the printer or the CRT (as chosen by SELECT).
PRINT AT:	Sends output to the CRT at the specified cursor position. Optionally erases characters following the specified cursor position.
PRINT HEXOF:	Prints the value of an alphanumeric-variable or literal string in hexadecimal notation.
PRINTUSING:	Sends formatted output to the CRT or printer. The format is determined by the referenced Image (%) statement.
PRINTUSING TO:	Stores formatted output in an alpha-variable.
PRINT TAB:	Outputs data in tabulated format.
READ:	In conjunction with the DATA statement, assigns elements in a DATA list to receivers in a READ list.
$RELEASE PART:	Relinquishes control of the partition to the null terminal, making the partition available to other users. The $RELEASE PART statement can be executed on the MVP only.
$RELEASE TERMINAL:	Places the current partition in the background and attaches the terminal to the next partition waiting to communicate with it. The $RELEASE TERMINAL statement can be executed only on the MVP.
REM:	Denotes comment; remainder of statement is ignored by system.
RENUMBER:	Assigns new line-numbers to the specified range of program lines in memory. The user may also specify the increment between successive line-numbers. RENUMBER automatically readjusts the entire program to accommodate subroutine calls and program branches made to the newly numbered lines.
RESET:	Immediately halts program execution and I/O operations and returns control to the operating system. On the MVP, RESET may also be used to attach a terminal to a partition assigned to the null terminal if the calling terminal does not have any partition assigned to it.
RESTORE:	Allows repetitive use of DATA statement values by READ statements by setting the DATA pointer back to the specified DATA value.
RETURN:	Used in a subroutine to return processing of the program to the statement immediately following the last-executed GOSUB or GOSUB' statement.
RETURN CLEAR:	Used in a subroutine to clear subroutine return address information from memory. Execution continues with the statement following the RETURN CLEAR statement.
RND function:	Produces a pseudorandom number between 0 and 1.

ROTATE [C]: Rotates bits in an alpha-receiver.

ROUND function: Rounds an expression to a specified number of decimal places.

RUN: Initiates program execution.

SAVE: Saves the current program or a portion thereof on the specified disk platter in Catalog Mode.

SAVE DA: Saves the current program or a portion thereof on the specified disk platter beginning at the specified sector address.

SCRATCH: Sets the status of the named disk files to "scratched."

SCRATCH DISK: Reserves space for the catalog area and catalog index on the specified platter.

SELECT:

1. Selects devices for input or output operations and the characteristics of such operations.
2. Specifies whether mathematical operations will be performed by using degrees, radians, or grads.
3. Selects a variable-length pause between output of one line on the CRT and the next. Pauses range from 1/6 second to 1 1/2 seconds.

SELECT ERROR: Suppresses error messages when a computational error (type C) occurs which satisfies the relation specified in the statement.

SELECT LINE: Sets the line length of an output device such as the CRT to expedite output operations.

SELECT ON/OFF: Defines interrupts for a program in the order of precedence.

SELECT @PART: Selects the named partition as the global partition of reference for the calling partition. This statement can be used on the MVP only.

SELECT [NO] ROUND: Causes numeric computational results to be rounded (i.e., truncated).

SGN function: Returns the value "1" if the argument is any positive number, "0" if the argument is zero, and "-1" if the argument is any negative number.

SIN function: Returns the sine of an expression.

SPACE: Returns the number of bytes of free space in memory.

SPACEK: Returns the total user memory size divided by 1,024.

SQR function: Finds the square root of an expression.

STOP:	Interrupts program execution until 1) CONTINUE is keyed or 2) a SF Key corresponding to a subroutine marked by a DEFFN' causes the program to continue at the entry point of the subroutine.
STR function:	Specifies a substring of an alpha-variable or alpha-array string. A portion of an alpha value can be examined, extracted, or changed using this function.
SUB:	Performs binary subtraction on a pair of binary values contained within alpha-variables or literal strings.
SUBC:	Like SUB, but performed with carry.
TAN function:	Returns the tangent of an expression.
#TERM:	Returns a numeric value equal to the number of the terminal executing the statement. This function can only be used on the MVP.
TRACE:	Traces program execution, producing output on the CRT or printer.
$TRAN:	Translates (in place) the characters in an alpha-receiver via an alpha-expression which is used as a translation table or list.
UNPACK:	Unpacks data that was packed by the PACK statement.
$UNPACK:	Unpacks data that was packed by the $PACK statement and places it in alpha-variables.
VAL function:	Converts alphanumeric expression to integer value which is the binary equivalent of the expression. Inverse of BIN function.
VER function:	Verifies that the value of an alpha-variable or literal string conforms to a specified format.
VERIFY:	Performs CRC and LRC checks on each sector in the specified range to determine that the information written in each sector is free from error.
XOR:	1. Logically exclusive ORs two arguments in a logical expression. 2. Exclusive OR of two relations in an IF statement.

APPENDIX H BASIC-2 ERROR CODES (SUMMARY)

H.1 NONRECOVERABLE ERRORS

Miscellaneous Errors

A01	Memory Overflow (Text ↔ Variable Table)
A02	Memory Overflow (Text ↔ Value Stack)
A03	Memory Overflow (LISTDC, MOVE, COPY)
A04	Stack Overflow (Operator Stack)
A05	Program Line Too Long
A06	Program Protected
A07	Illegal Immediate Mode Statement
A08	Statement Not Legal Here
A09	Program Not Resolved

Syntax Errors

S10	Missing Left Parenthesis
S11	Missing Right Parenthesis
S12	Missing Equal Sign
S13	Missing Comma
S14	Missing Asterisk
S15	Missing ">" Character
S16	Missing Letter
S17	Missing Hex Digit
S18	Missing Relational Operator
S19	Missing Required Word
S20	Expected End of Statement
S21	Missing Line-Number
S22	Illegal PLOT Argument
S23	Invalid Literal String
S24	Illegal Expression or Missing Variable
S25	Missing Numeric-Scalar-Variable
S26	Missing Array-Variable
S27	Missing Numeric-Array
S28	Missing Alpha-Array
S29	Missing Alpha-Variable

Program Errors

P32	Start > End
P33	Line-Number Conflict
P34	Illegal Value
P35	No Program in Memory
P36	Undefined Line-Number or CONTINUE Illegal
P37	Undefined Marked Subroutine
P38	Undefined FN Function
P39	FN's Nested Too Deep
P40	No Corresponding FOR for NEXT Statement
P41	RETURN Without GOSUB
P42	Illegal Image
P43	Illegal Matrix Operand
P44	Matrix Not Square

Program Errors

P45	Operand Dimensions Not Compatible
P46	Illegal Microcommand
P47	Missing Buffer Variable
P48	Illegal Device Specification (Recoverable)
P49	Interrupt Table Full
P50	Illegal Array Dimensions or Variable Length
P51	Variable or Value Too Short
P52	Variable or Value Too Long
P53	Noncommon Variables Already Defined
P54	Common Variable Required
P55	Undefined Variable (Program Not Resolved)
P56	Illegal Subscripts
P57	Illegal STR Arguments
P58	Illegal Field/Delimiter Specification
P59	Illegal Redimension

H.2 RECOVERABLE ERRORS

Computational Errors

C60	Underflow
C61	Overflow
C62	Division by Zero
C63	Zero Divided by Zero or Zero ↑ Zero
C64	Zero Raised to Negative Power
C65	Negative Number Raised to Noninteger Power
C66	Square Root of Negative Value
C67	LOG of Zero
C68	LOG of Negative Value
C69	Argument Too Large

Execution Errors

X70	Insufficient Data
X71	Value Exceeds Format
X72	Singular Matrix
X73	Illegal INPUT Data
X74	Wrong Variable Type
X75	Illegal Number
X76	Buffer Exceeded
X77	Invalid Partition Reference

Disk Errors

D80	File Not Open
D81	File Full
D82	File Not in Catalog
D83	File Already Cataloged
D84	File Not Scratched
D85	Index Full
D86	Catalog End Error
D87	No End-of-File
D88	Wrong Record Type
D89	Sector Address Beyond End-of-File

I/O Errors

I90	Disk Hardware Error
I91	Disk Hardware Error
I92	Timeout Error
I93	Format Error
I94	Format Key Engaged
I95	Device Error
I96	Data Error
I97	Longitudinal Redundancy Check Error
I98	Illegal Sector Address or Platter Not Mounted
I99	Read-After-Write Error

APPENDIX I DETAILS OF THE I/O BUS

I.1 INPUT, OUTPUT, AND ADDRESS STROBES

The microcommand tables at the end of chapter 15 list the signal sequences generated by each available $GIO I/O microcommand. Within these signal sequences, reference is made to four types of strobes: IBS input strobes, OBS output strobes, CBS output strobes, and ABS address strobes. For those readers who require a more detailed description of strobes than provided in chapter 15, a brief explanation follows.

A strobe is a short-duration change in the voltage level of a direct-current circuit. For example, in the circuitry of Wang's Model 2250 I/O Interface Controller, a logic "0" is represented by a "high-level" signal between +2.4 and 3.6 volts DC, and a logic "1" is represented by a "low-level" signal between 0 and +0.4 volts DC. As shown in Figure I-1, output strobes from a Wang CPU to an external device via the Model 2250 I/O Interface Controller have a pulse width of five microseconds ±10%. Input strobes from an external device to the Model 2250 I/O Interface Controller must have a pulse width between five and 20 microseconds.

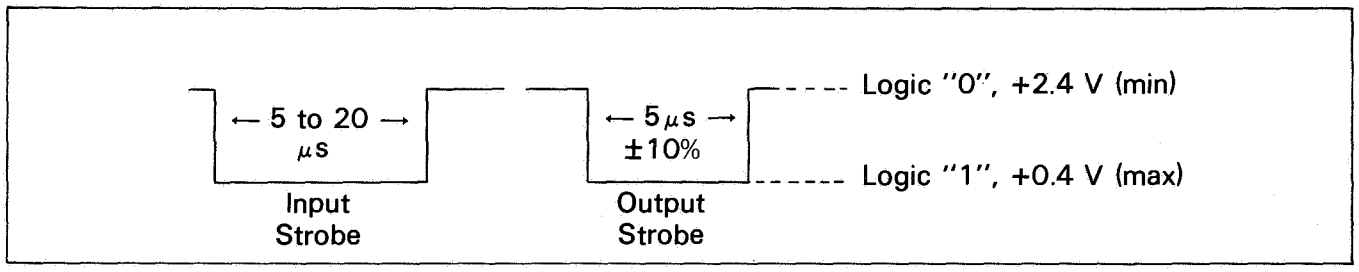


Figure I-1.
Schematic of Input and Output Strobes for the Model 2250
I/O Interface Controller

An input strobe (IBS) received by the Model 2250 I/O Interface Controller on a particular circuit from an external device indicates that data signals (8-bits in parallel or one byte) are available on other circuits, awaiting transfer into the CPU. Similarly, an output strobe sent from the CPU via the Model 2250 I/O Interface Controller on a particular circuit indicates to an external device that data signals (8-bits in parallel) are available on other circuits, awaiting reception by the external device.

For some applications, bytes of output information to be sent to a device fall into two classifications: (1) control data (e.g., instructions) and (2) output data to be stored. For this reason, the Model 2250 I/O Interface Controller provides two different circuits for output strobes. On one circuit, an output strobe is designated as an OBS strobe (Output Data Strobe); on the other circuit, an output strobe is designated as a CBS strobe (Control Output Strobe). Thus, a byte of information on the eight parallel data circuits can be identified by using either an OBS or a CBS strobe to indicate whether it is output data or control data.

If a microcommand table contains a code for a signal sequence which sends information via an OBS strobe, the table usually contains another code for a signal sequence identical except for its use of a CBS strobe to send the information. When selecting microcommand codes to control a device interfaced via a Model 2250 I/O Interface Controller, the programmer must know whether both the CBS and OBS output strobe circuits are connected or whether only the OBS circuit is connected. The ABS strobe, lastly, is an address strobe. It indicates to a controller that the byte to follow is an 8-bit code representing the device-address rather than a data or control character. The 8-bit address code must, in this case, correspond to the last two hex digits of the three-digit device-address preset in the device controller board. ABS strobes are generated at the beginning of a $GIO statement by microcommands of the form *7hhh* and are used to select or deselect a particular I/O device.

NOTE:
The microcommands whose signal sequences output characters with an OBS strobe rather than a CBS strobe are appropriate for most standard Wang peripherals such as CRT's and printers, which are interfaced to a System 2200 CPU via controllers other than the Model 2250 I/O Interface Controller. The connector on the Model 2250 I/O Interface Controller supports both OBS and CBS strobes; therefore, a programmer must know how an interfaced device is connected to such controllers before making decisions regarding the two types of output strobes.

INDEX

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

Z

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

700-4080D

TITLE OF MANUAL **BASIC-2 LANGUAGE REFERENCE MANUAL**

COMMENTS:

Fold

Fold

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

WANG

Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 16 LOWELL, MA

POSTAGE WILL BE PAID BY ADDRESSEE

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS 01851

Attention: Technical Writing Department

Fold

Cut along dotted line.

Printed in U.S.A.
13-1019

International Representatives

Argentina
Bahamas
Bahrain
Bolivia
Botswana
Brazil
Canary Islands
Chile
Colombia
Costa Rica
Cyprus
Denmark
Dominican Republic
Ecuador
Egypt
El Salvador
Finland
Ghana
Greece
Guam
Guatemala
Haiti
Honduras
Iceland
India
Indonesia
Ireland
Israel
Italy
Ivory Coast
Japan
Jordan
Kenya
Korea
Kuwait
Lebanon
Liberia
Malaysia
Malta
Mexico
Morocco
New Guinea
Nicaragua
Nigeria
Norway
Paraguay
Peru
Philippines
Portugal
Qatar
Saudi Arabia
Scotland
Senegal
South Africa
Spain
Sri Lanka
Sudan
Tasmania
Thailand
Turkey
United Arab Emirates
Uruguay
Venezuela
Zimbabwe

United States

Alabama
Birmingham
Mobile

Alaska
Anchorage
Juneau

Arizona
Phoenix
Tucson

California
Anaheim
Burlingame
Culver City
Emeryville
Fountain Valley
Fresno
Los Angeles
Sacramento
San Diego
San Francisco
Santa Clara
Ventura

Colorado
Englewood

Connecticut
New Haven
Stamford
Wethersfield

District of Columbia
Washington

Florida
Coral Gables
Hialeah
Hollywood
Jacksonville
Miami
Orlando
Sarasota
Tampa

Georgia
Atlanta
Savannah

Hawaii
Honolulu
Maui

Idaho
Boise

Illinois
Arlington Heights
Chicago
Morton
Oakbrook
Park Ridge
Rock Island
Rosemont
Springfield

Indiana
Fort Wayne
Indianapolis
South Bend

Iowa
Ankeny

Kansas
Overland Park
Wichita

Kentucky
Louisville

Louisiana
Baton Rouge
Metairie

Maine
Portland

Maryland
Baltimore
Bethesda
Gaithersburg
Rockville

Massachusetts
Boston
Burlington
Chelmsford
Lawrence
Littleton
Lowell
Marlboro
Shrewsbury
Worcester

Michigan
Grand Rapids
Kalamazoo
Lansing
Southfield

Minnesota
Eden Prairie
Minneapolis

Mississippi
Jackson

Missouri
Creve Coeur
St. Louis

Nebraska
Omaha

Nevada
Las Vegas

New Hampshire
Manchester

New Jersey
Bloomfield
Clifton
Edison
Mountainside
Toms River

New Mexico
Albuquerque
Santa Fe

New York
Albany
Jericho
Lake Success
New York City
Rochester
Syosset
Syracuse
Tonawanda

North Carolina
Charlotte
Greensboro
Raleigh

Ohio
Akron
Cincinnati
Cleveland
Independence
Toledo
Worthington

Oklahoma
Oklahoma City
Tulsa

Oregon
Eugene
Portland
Salem

Pennsylvania
Allentown
Erie
Harrisburg
Philadelphia
Pittsburgh
State College
Wayne

Rhode Island
Providence

South Carolina
Charleston
Columbia

Tennessee
Chattanooga
Knoxville
Memphis
Nashville

Texas
Austin
Dallas
El Paso
Houston
San Antonio

Utah
Salt Lake City

Virginia
Newport News
Norfolk
Richmond
Rosslyn
Springfield

Washington
Richland
Seattle
Spokane

Wisconsin
Appleton
Brookfield
Green Bay
Madison
Wauwatosa

International Offices

Australia
Wang Computer Pty., Ltd.
Adelaide, S.A.
Brisbane, Old.
Canberra, A.C.T.
Perth, W.A.
South Melbourne, Vic. 3
Sydney, NSW

Austria
Wang Gesellschaft, m.b.H.
Vienna

Belgium
Wang Europe, S.A.
Brussels
Erpe-Mere

Canada
Wang Canada Ltd.
Burlington, Ontario
Burnaby, B.C.
Calgary, Alberta
Don Mills, Ontario
Edmonton, Alberta
Halifax, Nova Scotia
Hamilton, Ontario
Montreal, Quebec
Ottawa, Ontario
Quebec City, Quebec
Toronto, Ontario
Victoria, B.C.
Winnipeg, Manitoba

China
Wang Industrial Co., Ltd.
Taipei

Wang Laboratories, Ltd.
Taipei

France
Wang France S.A.R.L.
Paris
Bordeaux
Lille
Lyon
Marseilles
Nantes
Nice
Rouen
Strasbourg

Great Britain
Wang (U.K.) Ltd.
Richmond
Birmingham
London
Manchester

Hong Kong
Wang Pacific Ltd.
Hong Kong

Japan
Wang Computer Ltd.
Tokyo

Netherlands
Wang Nederland B.V.
IJsselstein
Groningen

New Zealand
Wang Computer Ltd.
Auckland
Christchurch
Wellington

Panama
Wang de Panama
ICPECI S.A.
Panama City

Puerto Rico
Wang Computaderas, Inc.
Hato Rey

Singapore
Wang Computer (Pte) Ltd.
Singapore

Sweden
Wang Skandinaviska AB
Stockholm
Gothenburg
Malmö

Switzerland
Wang A.G.
Zurich
Basel
Bern
Geneva
Lausanne
St. Gallen

Wang Trading A.G.
Zug

West Germany
Wang Deutschland, GmbH
Frankfurt
Berlin
Cologne
Dusseldorf
Essen
Freiburg
Hamburg
Hannover
Kassel
Mannheim
Munich
Nürnberg
Saarbrücken
Stuttgart

WANG

ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

Printed in U.S.A.
700-4080D
6-81-3M
Price: see current list