# Multiuser BASIC-2 Language Reference Manual

7th Edition – March 1991

700-4080F

WANG

**Wang Laboratories, Inc.**
One Industrial Avenue, Lowell, MA 01851
Tel. (508) 459-5000, Telex 172108

## Disclaimer of Warranties and Limitation of Liabilities

### Software Notice

## FCC and DOC Notices

The Federal Communications Commission (FCC) and the Canadian Department of Communications (DOC) require that information regarding the interference potential of electrical equipment be included in the user documentation for the equipment. They also require that specific information be provided for components that will be connected to the Public Switched Telephone Network (PSTN).

### Electromagnetic Interference Requirements

Wang Laboratories, Inc., manufactures both Class A verified computers and peripherals and Class B certified computers and peripherals. To determine which of the following warnings apply to the equipment, the user should refer to the label affixed to the device.

An example of a Class A verified label is the following:

> This device complies with Part 15 of the FCC Rules. Operation is subject to the following two conditions: (1) This device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

An example of a Class B certification label is the following:

> FCC I.D. B4YPC250-16
> This device complies with Part 15 of the FCC Rules. Operation is subject to the following two conditions: (1) This device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

### FCC Warning

**For Class A** – This equipment(1,2,4) has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case the user will be required to correct the interference at his or her own expense.

**For Class B** – This equipment(1,2,3,4) has been tested and found to comply with the limits for a Class B digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

- Reorient or relocate the receiving antenna.
- Increase the separation between the equipment and the receiver.
- Connect the equipment into an outlet on a circuit different from that to which the receiver is connected.
- Consult the dealer or an experienced radio/television technician for help.

### DOC Warning

**For Class A** – This digital apparatus does not exceed the Class A limits for radio noise emissions from digital apparatus set out in the Radio Interference Regulations of the Canadian Department of Communications.

**For Class B** – This digital apparatus does not exceed the Class B limits for radio noise emissions from digital apparatus set out in the Radio Interference Regulations of the Canadian Department of Communications.

### Connection to the PSTN

#### FCC Part 68 Requirements

If this equipment connects to the PSTN, it complies with Part 68 of the FCC Rules. A label is affixed to this equipment that contains, among other information, the FCC Registration Number (REN) and Ringer Equivalence Number (REN) for this equipment. Upon request, the user must provide this information to the telephone company.

The REN is useful to determine the number of devices that the user may connect to the telephone line and still have all those devices ring when the telephone number is called. In most but not all areas, the sum of the RENs of all devices connected to one line should not exceed five (5). To be certain of the number of devices that the user may connect to the line, as determined by the REN, the user should contact the local telephone company to determine the maximum REN for the calling area.

If the telephone equipment causes harm to the telephone network, the telephone company may discontinue service temporarily. If possible, the user will be notified in advance. However, if advance notice is not practical, the user will be notified as soon as possible. The user will be informed of the right to file a complaint with the FCC.

The telephone company may make changes in its facilities, equipment, operations, or procedures that could affect the proper functioning of the equipment. If such changes are made, the user will be notified in advance to give the user an opportunity to maintain uninterrupted telephone service.

If the user experiences trouble with this telephone equipment, the telephone company may ask that the user disconnect the equipment from the network until the problem has been corrected or until the user is sure that the equipment is not malfunctioning. For service or repairs, the user should contact the point of sales representative or Wang Laboratories, Inc.

This equipment may not be used in conjunction with coin service provided by the telephone company. Connection to party lines is subject to state tariffs.

## DOC Notice for Canadian Users

**Notice** – If this equipment contains a Canadian Department of Communications label, it identifies DOC-certified equipment. This certification means that the equipment meets certain telecommunications network protective, operational, and safety requirements. The DOC does not guarantee that the equipment will operate to the user's satisfaction.

Before installing this equipment, the user should ensure that it is permissible for it to be connected to the facilities of the local telecommunications company. The equipment must also be installed using an acceptable method of connection. In some cases, the company's inside wiring associated with a single line individual service may be extended by means of a certified connector assembly (telephone extension cord). The user should be aware that compliance with the above conditions may not prevent degradation of service in some situations.

Repairs to certified equipment should be made by an authorized Canadian maintenance facility designated by the supplier. Any repairs or alterations made by the user to this equipment, or equipment malfunctions, may give the telecommunications company cause to request that the user disconnect the equipment.

Users should ensure for their own protection that the electrical ground connections of the power utility, telephone lines, and internal metallic water pipe system, if present, are connected together. This precaution may be particularly important in rural areas.

***Caution:** Users should not attempt to make such connections themselves, but should contact the appropriate electrical inspection authority or electrician, as appropriate.*

**Load Number** – The Load Number (LN) assigned to each terminal device denotes the percentage of the total load that can be connected to a telephone loop. The termination on a loop may consist of any combination of devices, subject only to the requirement that the total of the Load Numbers of all devices does not exceed 100. An alphabetical suffix is also specified in the Load Number for the appropriate ringing type (A or B), if applicable. For example, LN=20A designates a Load Number of 20 and an "A" type ringer. The Load Number for this equipment can be found in the installation instructions pertaining to connection to the PSTN.

---

(1) This equipment requires the use of shielded cables.

(2) If the user attaches a Class A verified device to equipment otherwise labeled as Class B, the combined system meets Class A regulations only. Class A specifications provide reasonable protection against radio and television interference in a commercial environment. Operation of Class A equipment in a residential environment may cause interference.

(3) This device complies with Part 15 of the FCC Rules. Operation is subject to the following two conditions: (1) This device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

(4) Any modifications to this equipment not expressly approved by Wang Laboratories, Inc., could void the user's authority to operate this equipment.

# Preface

Chapter 1

# Introduction

Chapter 2

# Editing and Debugging Features

## Figures

# Preface

This manual is designed as a primary resource for using the BASIC-2 language on Wang computer systems. Users unfamiliar with the BASIC language are encouraged to refer to a standard textbook for an introduction to the language.

Throughout this manual, a general format accompanies each description of a command or statement. When more than one specific arrangement is permitted, there are separate numbered formats. Within a format, key words, connectives, and special characters appear in proper sequence. Unless otherwise stated, you can use only the sequence shown.

This manual uses the following conventions to define and illustrate the components of BASIC-2 program statements and commands:

- Uppercase letters (A through Z), digits (0 through 9), and special characters (such as *, /, +) must always be used for program entry exactly as presented in the general format.
- All lowercase words represent information that you must supply.

  *Example:*

  In the following statement, you must supply the line-number.

  ```
  GOTO line-number
  ```

- When braces, { } enclose a vertically stacked list in a portion of a format, you must select one of the options within the braces.

  *Example:*

  ```
         expression
  ON
         alpha-variable
  ```

- Brackets, [ ] indicate that the enclosed information is optional. When brackets contain a vertical list of two or more items, you can use one or none of the items.

  *Example:*

  ```
  LOAD RUN [filename]
  ```

- The presence of an ellipsis (...) within any format indicates that the unit immediately preceding the notation can occur one or more times in succession.
- *Example:*

```
COM com-element [,com-element] ...
```

- When one or more items appear in sequence, these items or their replacements must appear in the specified order.

1

# Introduction

## Overview

BASIC-2 is a high-level programming language designed for interactive programming and ease of use. Many extensions and enhancements are included in BASIC-2 to facilitate the tasks of writing, documenting, and debugging programs and to provide flexible language capabilities for a wide range of applications. BASIC-2 includes features that support system operation.

### Multiuser Operation

BASIC-2 allows several users to share a single computer efficiently. The operating system divides the resources of the computer (i.e., memory, peripherals, and CPU time) among the users. Memory in the system is divided into a number of sections, called "partitions," each of which can hold a separate BASIC-2 program. Once each user has been allocated a share of the resources, the operating system acts as a monitor, allowing each user to use the system in turn while preventing individual users from interfering with each other.

Most of the discussion in this manual is from the perspective of the operation of a single BASIC-2 program in a partition. Refer to Chapter 16 for a detailed discussion of multiuser operation.

### Interpretation Process

BASIC-2 analyzes each line syntactically as it is entered and compacts it into a smaller, more easily interpreted form. By distributing interpretation between entry time and run time, BASIC-2 significantly accelerates program development.

### Editing Features

Editing features and system commands enable direct keyboard control of the development and execution of BASIC-2 programs. Extensive debugging features help to identify and isolate possible areas of program failure. In addition, a number of one-line, immediately-executable BASIC-2 statements provide a quick and convenient means of performing arithmetical calculations.

### Compatibility Features

Software compatibility was the principal goal in the design of BASIC-2. BASIC-2 includes nearly all the capabilities of single-user VP BASIC-2. However, several language features have been added to BASIC-2 in order to support the multiprogramming environment. BASIC-2 also includes debugging capability and a few other language enhancements not found in VP BASIC-2. In addition, BASIC-2 supports earlier Wang BASIC syntax, providing a significant degree of compatibility with systems that use Wang BASIC. For the Wang Professional Computer (PC) Series, Wang offers PC BASIC-2, a language highly compatible with BASIC-2.

## Types of BASIC–2 Instructions

BASIC-2 consists of various language elements, 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. Commands control system operations. Operator and system-defined functions construct numeric or alphanumeric expressions within a statement.

### System Commands

System commands are instructions that allow you to control major system functions directly from the keyboard. Commands enable you to perform functions such as initiating program execution, clearing system memory, listing the program in memory, and renumbering the program in memory. The system executes commands immediately after entry; commands are not stored in memory as part of a program.

### Statements

A statement is a programmable instruction that 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 statement directs the system to perform a specific operation or sequence of operations. In most cases, a statement includes one or more expressions that provide the information to be operated on by the system. An expression can consist of numeric or alphanumeric data, variables containing such data, or a combination of functions or operators and data.

BASIC-2 statements are divided into two groups: executable and nonexecutable. Executable statements direct the system to perform certain tasks during program execution. Nonexecutable statements initiate no system action when encountered during program execution. These statements, such as COM, DATA, and REM, provide the system with information or aid in program documentation.

## BASIC-2 Statements and Program Execution

The system generally processes BASIC-2 statements as a sequence of numbered lines within a program. However, certain BASIC-2 statements function independently of a program context and can be executed individually. BASIC-2 defines the former operating mode as Program mode and the latter as Immediate mode.

### Program Mode

A BASIC-2 program consists of one or more numbered program lines, each consisting of one or more statements. Program lines are not executed immediately upon entry but are stored in memory for execution at a later time.

A line number consisting of one to four digits must preface all program lines. 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:

```
0010 PRINT A+B-5
10 PRINT A+B-5
```

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. However, program lines can be entered in any order. When you enter a new line having a unique line number, the system automatically inserts the line into proper line number sequence in the program residing in memory.

## Immediate Mode

Immediate mode lines, unlike program lines, are not permanently stored in memory for subsequent execution as part of a program. Rather, the system executes these instructions immediately upon entry. When you enter a new Immediate mode line or program line, the system deletes the last entered Immediate mode line from memory.

Although BASIC-2 statements are by definition programmable instructions, you can execute most statements in Immediate mode by entering them without a preceding line number and then pressing RETURN. All system commands, including those that control the loading, running, editing, debugging, and saving of programs, belong to the group of statements that can be executed in Immediate mode.

The PRINT statement, which displays the results of evaluating an expression, can be used in Immediate mode. Using the PRINT statement in Immediate mode makes the system a powerful on-line calculator.

For example, the following statement evaluates the expression and displays the result on the screen:

```
PRINT 150 + 350 + 500
```

Immediate mode lines can be elaborate programs, but they must be self-contained. An Immediate mode line cannot transfer program execution to numbered program text. Therefore, statements that reference program line numbers are not allowed in Immediate mode. Also, the following statements cannot be executed in Immediate mode.

```
DATA      ERROR     INPUT          READ
DEFFN     GOSUB     LINPUT         RETURN
DEFFN'    GOSUB'    ON GOSUB/GOTO
```

***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. However, other Immediate mode statements (in particular, assignment statements) can alter the contents of variables in memory. Variables these statements reference should be carefully selected, since the alteration of one or more variables can affect the operation of the program currently in memory.*

## Multiple-Statement Lines

BASIC-2 permits the specification of more than one statement on a program line or in an Immediate mode line. Individual statements on the same line must be separated by colons. For example, the following three program lines

```
10 A = A+1
20 PRINT A
30 GOTO 100
```

can also be written in a single line as

```
40 A = A+1: PRINT A: GOTO 100
```

Line 40 contains three separate statements that instruct the system to add 1 to the value of A, print the value of A, and branch to Line 100. The statements in Lines 10, 20, and 30 perform the same task.

The use of multiple-statement lines allows program statements to be logically grouped for more readable programs. You can also execute multiple-statement lines in Immediate mode. For example, the following statement displays the natural logarithms of the integers 1 through 10:

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

## Phases of the Language Processor

The system accomplishes the entry and execution of a BASIC-2 program or Immediate mode line in three distinct phases: Entry phase, Resolution phase, and Execution phase. During each phase, the system performs a specific set of actions and checks for certain types of errors.

### Entry Phase

Entry phase occurs as soon as the system enters a program line or Immediate mode line. During entry phase, the system checks the entered line for such local syntax errors as misspelled key words and erroneous punctuation. Only the entered program line is checked, not its surrounding context. If the line contains an error, the screen displays an appropriate error number or message.

The system saves in memory all entered lines, correct or otherwise, and condenses the line into a form that uses significantly less storage space and executes faster than the original form. The need for separate source and object files is thereby eliminated. When you recall a line for inspection or correction, the system automatically regenerates the original source text from its condensed form.

## Resolution Phase

Resolution phase occurs when you execute an Immediate mode RUN or LOAD RUN command or a Program mode LOAD or LOAD RUN statement. During Resolution phase, the system sequentially scans the entire program for overall consistency. The system performs such tasks as checking that statements throughout the program are in the proper sequence, testing the validity of all program references, and ensuring that arrays are properly defined.

If the entire pass is error-free, program execution begins immediately. If the system detects an error, the screen displays an error number or message, program resolution ceases, the program is unresolved, and the system returns to Entry phase. Only after a program undergoes error-free resolution does the system enter Execution phase.

## Execution Phase

During Execution phase, the system executes program lines in line number sequence, unless program execution is transferred out of the normal sequence by a branch statement (e.g., GOTO, GOSUB, FOR...NEXT). Program execution continues until one of the following conditions occurs:

- The system executes a STOP statement, an END statement, or the last statement in the program.
- An error occurs (unless the error termination is suppressed with the SELECT ERROR or ERROR statement; refer to Chapter 8).
- You press HALT or RESET.

2

# Editing and Debugging

## Overview

The BASIC-2 language offers a variety of useful editing and debugging features. The editing capabilities of the system enable the programmer to edit program lines, Immediate mode lines, and data values during and after entry. Additionally, the debugging features of BASIC-2 facilitate the tasks of identifying and isolating bugs in a BASIC-2 program.

## Line Entry

When BASIC-2 is ready to accept program lines and Immediate mode text, the colon (:) prompt is displayed in the first column of the next screen line; the cursor appears directly to the right of this colon. As you enter text, the cursor always appears one character beyond the most recently entered character. Lines being entered can span several CRT lines. If a line is entered longer than a CRT line, the cursor automatically advances to the beginning of the next CRT line.

Pressing RETURN terminates the entry of a line. This action passes control to the system, which then processes the line. If the entered line is a numbered program line, it becomes part of the program in memory. If the entered line is an Immediate mode line, it is executed and saved for possible recall.

### Spaces

Spaces can be included within a program line to enhance readability, but they are not necessary for the system to interpret the line.

### Maximum Line Length

The maximum length of line that can be entered is determined by the amount of memory available for buffering the line. If you attempt to type more characters than this maximum, the system emits a beep and does not accept any additional text until you press RETURN.

### Upper/lowercase Entry

With BASIC-2, program lines can be entered in either uppercase or lowercase characters. Lowercase characters are automatically converted to uppercase when the line is entered. However, lowercase characters within REM statements, Image (%) statements, and literal strings enclosed in quotes are not translated to uppercase. For example, if the following line is entered:

```
100 rem print title: print at(0,10); "Summary of New Features"
```

it is converted to:

```
100 REM print title: PRINT AT(0,10); "Summary of New Features"
```

## Line Editing

A line being entered can be edited by positioning the cursor at the desired character and deleting, inserting, or overtyping characters. When editing a line, the system makes all changes to a copy of the line. No changes are made to the actual line until RETURN is pressed.

If an error is discovered in a line that has already been entered, the line can be recalled. If the last entered line is an Immediate mode line, pressing EDIT then RECALL recalls that line for editing. If a program line is to be edited, the line number of the program line is entered before pressing RECALL. These procedures retrieve the line from memory and displays it on the screen. The line can then be edited and reentered by pressing RETURN.

### Edit Keys and Their Operation

Table 2-1 describes the editing functions performed by various keys in BASIC-2.

**Table 2-1. Editing Functions**

| Key | Function |
|---|---|
| EDIT | The EDIT key on terminals with DE-style keyboards activates and deactivates the function keys for editing use. Pressing EDIT causes the system to enter Edit mode, indicated by a blinking cursor. In Edit mode, the function keys are activated for use in editing. Pressing EDIT again causes the system to leave Edit mode, indicated by a non blinking cursor. When not in Edit mode, the function keys are not available for edit functions. Note, terminals with DW-style keyboards have dedicated editing keys and do not need to use the function keys for editing. |
| East cursor control key → | Moves the cursor one position to the right (or to the beginning of the next screen line when the cursor is on the 80th character of a line), as long as this does not move the cursor out of the current program or Immediate mode line. |
| East five cursor control key (⟶ ) | Moves the cursor five positions to the right, as long as this does not move the cursor out of the current program or Immediate mode line. On a DW-style keyboard, SHIFT with the East key performs this function. |
| West cursor control key (← ) | Moves the cursor one position to the left (or to the end of the previous screen line when the cursor is on the first character of a line), as long as this does not move the cursor out of the current program or Immediate mode line. |
| West five cursor control key (←—) | Moves the cursor five positions to the left, as long as this does not move the cursor out of the current program or Immediate mode line. On a DW-style keyboard, SHIFT and the West key performs this function. |
| North cursor control key ( ↑ ) | Moves the cursor up one screen line, as long as this does not move the cursor out of the current program or Immediate mode line. |
| BEG cursor control key | Moves the cursor to the beginning of the current line. On a DW-style keyboard, pressing SHIFT and the North cursor control key performs this function. |
| South cursor control key (↓) | Moves the cursor down one screen line, as long as this does not move the cursor out of the current program or Immediate mode line. |
| END cursor control key (↓) | Moves the cursor to the end of the current line. On a DW-style keyboard, pressing SHIFT and the South cursor control key performs this function. |
| INSERT | Inserts a blank character at the current cursor position. All text to the right of the cursor moves over one position to the right, with the 80th char acter on a screen line moving to the beginning of the next screen line. |
| DELETE | Deletes the character at the current cursor position. All text to the right of the cursor moves over one position to the left, with the first character on a screen line moving to the end of the previous line. |
| BACKSPACE | Replaces the character to the left of the current cursor position with a blank. The cursor then moves one position to the left. |
| Space bar | Replaces the character at the current cursor position with a blank. The cursor then moves one position to the right. |
| ERASE | Deletes all text from the current cursor position to the end of the line. |
| SHIFT and ERASE | Deletes the entire line of text in which the cursor is positioned. |
| RETURN | Enters a line of text. The colon then appears, prompting you to enter an other line of text. |
| RECALL | Recalls a specific program line or last entered Immediate mode line. |

## Program Development

BASIC-2 facilitates program development through the following methods:

- Replacing lines
- Deleting lines
- Renumbering program lines
- Combining program lines

### Replacing Lines

To replace an existing program line with a new line, enter the new line with a line number identical to that of the original line and press RETURN. This procedure removes the original line from memory, replaces it with the new line, and displays the message "Replaced" on line 25 of the screen.

*Example:*

To replace line 20,

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

Enter a new line with line number 20. This replaces the existing line 20 with the new line.

```
:20 PRINT LOG(A) + LOG(B)
:_
```

### Deleting Lines

To delete a program line from memory, enter only the line number and press RETURN.

*Example:*

The following entry deletes Line 20 from memory.

```
:20
:_
```

If it is necessary to delete entire sections of a program, you can delete a series of program lines by executing an Immediate mode CLEAR P command. For additional information, refer to the discussion of the CLEAR P command in Chapter 10.

## Renumbering Program Lines

To change the line number of a specific program line, Recall the line, move the cursor to the beginning of the line, and edit the line number. Because the edited line has a new line number, it is saved in memory as a new line, or replaces any existing line with the same line number. However, the original line, with the original line number, is not automatically deleted from memory. For example, if Line 20 is recalled and its line number is changed to 30 and saved, both Lines 20 and 30 (identical except for different line numbers) are stored in memory. The original line (Line 20) can be removed only if you delete it or replace it with a new Line 20.

The Immediate mode RENUMBER command renumbers individual lines or an entire program, automatically moving renumbered text from the original location and inserting it in proper sequence. RENUMBER also allows you to specify the increment between successive line numbers. This enables you to leave adequate space between successive lines for inserting of new program text.

RENUMBER also automatically changes all program references to line numbers in GOTO, GOSUB, and IF...THEN statements in accordance with the new numbering scheme. Refer to Chapter 9 for a detailed description of the RENUMBER command.

## Combining Program Lines

To combine two or more program lines in memory into a single program line, perform the following procedure:

1. Recall a program line by entering the line number and pressing EDIT then RECALL.
2. Enter a colon (:) at the end of the program line, followed by the line number of the program line that is to be added to the end of the currently displayed line.
3. Press the EDIT then RECALL keys. This recalls the specified line from memory and adds it to the end of the currently displayed line. This procedure can be repeated to combine multiple lines into a single, complex line.
4. Press RETURN to store the new line in memory. The new line replaces only the first recalled line in memory, because the new line has the line number of the first recalled line. When you save the new line, the system does not delete from memory any of the program lines that were recalled and added to the original line. To delete each of these lines, enter each line number and press RETURN.

*Example:*

The following two lines exist in memory:

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

To add Line 20 to the end of Line 10, recall Line 10 and enter ":20" at the end.

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

To combine Line 20 with Line 10, press RECALL.

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

When you press RETURN, the new extended version of Line 10 replaces the original Line 10 in memory. Line 20, however, is not affected, as the following listing discloses:

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

To delete Line 20, enter the line number and press RETURN.

## Program Debugging Features

The process of locating and identifying errors in a BASIC-2 program requires you to analyze the performance of the program at critical stages in execution. To aid you in this task, the system provides the following debugging tools:

- Descriptive error messages
- Stopping and resuming program execution
- Halting and stepping through a program
- Tracing through a program
- Listing and cross-referencing a program

### Descriptive Error Messages

The system automatically scans program text for errors during program entry, resolution, and execution. When the system encounters an error, it displays the erroneous line and an arrow points to the approximate position of the error. On the next line, an error message is displayed. The error message includes the word "ERROR", the error number, and an explanation of the error condition.

*For example:*

```
DATALOAD DC #1, X,A$
         ↑
ERROR D80:  File Not Open
```

If the system detects an error during text entry, it stores the erroneous line in memory. If the system encounters an error during program resolution or execution, it immediately terminates resolution or execution. The system stops error scanning when it encounters the first error. If a line contains more than one error, the system detects and reports only the first error. Refer to Appendix B for a list of errors and suggested recovery procedures.

## Stopping and Resuming Program Execution

When you are debugging a program, it is helpful to stop program execution at a particular point to examine and modify the values of critical variables. Execution of a STOP statement in a program suspends program execution and displays the word STOP, optionally followed by a user-supplied message and/or the line number of the STOP statement.

Strategic placement of STOP statements enables you to monitor the performance of a program at critical points. Upon execution of a STOP statement, you can examine the current values of variables with an Immediate mode PRINT statement and modify them to observe the effect on subsequent processing. There is no limit to the number of STOP statements a program can contain.

BASIC-2 allows a program stopping point to be set from Immediate mode. STOP, when used in Immediate mode, sets a stop point at the specified program line. Subsequently, when a program is run, execution of that program stops just before the specified line is to be executed, as if the STOP statement were the first statement of that line.

To resume program execution following a STOP statement, issue a CONTINUE command. The CONTINUE command restarts execution at the statement immediately following the STOP statement, provided you do not modify program text. Refer to Chapters 10 and 11 for additional information on the CONTINUE command and STOP statement, respectively.

## Halting and Stepping Through a Program

You can halt program execution after the currently executing statement by pressing HALT. At this time, you can examine the values of critical variables with an Immediate mode PRINT statement and, if necessary, modify the values of variables.

You can also step through program execution one statement at a time by repeatedly pressing HALT. Each time you press this key, the system lists and executes the next statement before halting. In this manner, you can closely observe the action taken by a program following execution of each statement.

When halting and stepping are no longer necessary, execution of a CONTINUE command resumes normal program execution. Refer to Chapter 10 for additional information on the HALT and the CONTINUE commands.

## Tracing Through a Program

Trace mode allows you to monitor certain critical operations. The TRACE statement turns on Trace mode; the TRACE OFF statement turns off 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. You can obtain a more comprehensive picture of what is happening in the program by stepping through program execution while in Trace mode.

To produce a printed copy of the Trace output, select the printer for Console Output operations prior to executing TRACE. You can use the SELECT P statement to slow the rate at which output is displayed on the screen. Refer to Chapter 7 for information on selecting devices for Console Output and selecting pauses; refer to Chapter 10 for information on the TRACE statement.

## Listing and Cross-Referencing a Program

The LIST command has a variety of forms that provide listing and cross-referencing capabilities. These forms include the following:

- LIST produces a listing of all or selected portions of program text in memory.
- LIST D produces a formatted listing of all or selected portions of program text. A formatted listing prints each statement of a multiple-statement line on a separate line.
- LIST T produces a cross-reference listing of all program lines containing a specified text string.
- LIST # produces a cross-reference listing of line numbers the program references.
- LIST V produces a cross-reference listing of variables the program references.
- LIST DIM and LIST COM lists the currently defined variables and their values.
- LIST ' produces a cross-reference listing of marked subroutines (DEFFN') defined and referenced in the program.
- LIST DT lists the contents of the device table. (Refer toChapter 7.)

The LIST command is discussed in detail in Chapter 10.

3

# Screen Operations

## Overview

In addition to normal character output to the screen, BASIC-2 provides control of cursor movement, character display attributes, box graphics, and alternate character sets (including character graphics). A BASIC-2 print function, the PRINT BOX statement, allows easy implementation of the box graphics feature.

All other features are programmed by outputting a series of one or more control codes to the screen. Although the HEX function is used for most examples in this chapter, cursor control codes can also be stored and transmitted to the screen from alpha variables.

The HEX function is a special kind of literal string used to describe one or more characters in terms of their hexadecimal representation. The HEX codes are composed of a pair of hexadecimal digits (the integers 0 through 9 and the letters A through F). There is no limit to the number of characters that may be described in a single HEX function. Since a HEX literal must describe complete characters, a HEX literal must consist of an even number of hexadecimal digits.

Any character may be represented by a HEX literal. However, HEX literals usually are used to describe characters not found on the keyboard or codes that perform control functions.

## Screen Control Codes

The codes HEX(00) - HEX(0F) are reserved by the screen for controlling such features as cursor movements, display attributes, and alarms. The codes HEX(0E) and HEX(0F) are used for controlling the character attributes, while the code HEX(02) introduces the start of a multibyte sequence. The various uses of these three codes are detailed in the following sections. The code HEX(00) represents a null action. All remaining codes control cursor appearance and movement.

### Cursor Control Codes

The cursor can be positioned at any specified row and column on the screen with the PRINT AT function. In addition, cursor control codes can be sent to the screen by using the PRINT HEX function. For example, either PRINT AT (0,0) or PRINT HEX(01) moves the cursor to the top left corner of the screen. Table 3-1 lists the available cursor controls.

**Table 3-1. Cursor Controls**

| Function | Description |
|---|---|
| HEX(01) | Moves cursor to the home position (top left of the screen) |
| HEX(03) | Clears the screen and moves the cursor to home position |
| HEX(05) | Cursor on |
| HEX(06) | Cursor off |
| HEX(08) | Cursor left 1 position (nondestructive backspace) |
| HEX(09) | Cursor right 1 position (nondestructive space) |
| HEX(0A) | Cursor down 1 line (line feed) |
| HEX(0C) | Cursor up 1 line (reverse line feed) |
| HEX(0D) | Cursor to the beginning of current line (carriage return). |

HEX codes can be combined in a single statement to perform several functions. Each function is executed as it occurs in the sequence. For example, the statement HEX(030A0909) will clear the screen and home the cursor (03), insert a line feed (0A), and indent two spaces to the right (0909). PRINT and PRINTUSING statements automatically issue a carriage return and a line feed if they are not terminated with a comma or a semicolon.

```
EXAMPLE 1 - EXAMPLES OF CONTROL CODES

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS

WANG LABORATORIES, INC.
                        ONE INDUSTRIAL AVENUE
                                                LOWELL, MASSACHUSETTS
:
```

**Figure 3-1. Examples of Control Codes**

## Character Display Attributes

To highlight information on the screen, BASIC-2 provides display attributes that can be selected for any character displayed on the screen. The following display attributes are available:

- Bright – Characters are displayed in high intensity.
- Blink – Characters blink.
- Reverse Video – Characters are dark while the character background display is light (dark on light).
- Underline – Characters are displayed with an underscore.

## HEX Codes Used to Invoke Display Attributes

When BASIC-2 starts, the screen displays characters in normal intensity, non-blinking, normal video (light on dark), and non-underlined (normal intensity). The power-on default meaning of HEX(0E) is bright, non-blinking, normal video, and non-underlined.

The display attribute to be used is selected by sending a command of the following form to the screen:

```
      HEX(02 04 xx yy 0E)
                or
      HEX(02 04 xx yy 0F)
```

```
where:

      02 04 =  The control code sequence that indicates to the
               screen that special character display attributes are
               to be selected.

      xx yy =  The HEX codes specifying the display attributes to be
               selected, where:

               xx = 00 for normal intensity, no blink
               02 for bright, no blink
               04 for normal intensity, blinking
               0B for bright, blinking

               yy = 00 for normal video, no underline
               02 for reverse video
               04 for underline
               0B for reverse video, underline

      OE or OF = A terminator character that causes the display at
                 tributes selected by xx yy to be turned on or off;
                 HEX(0E) turns the selected attributes on, HEX(0F)
                 turns them off.
```

There are two ways to code the attribute "blinking." However, on most terminals, blinking normal-intensity and blinking high-intensity characters both appear as blinking normal-intensity characters.

## Turning on Character Display Attributes

To highlight portions of the display area, you must execute the appropriate HEX(0204...) sequence *before* the character or string of characters that require an attribute is output. A sequence ending in 0E, e.g., HEX(020400020E), selects and immediately activates (turn on) an attribute. However, a sequence ending in 0F, e.g., HEX(020402040F), selects an attribute but does not turn it on. Execute the following program to see the possible display attributes, i.e., bright, blinking, underline, and reverse video. Each HEX statement is located before output to be highlighted, and each HEX sequence ends with an 0E. (Refer to Figure 3-2.)

```
5 PRINT HEX(03)
10 PRINT "EXAMPLE 2 - THE DISPLAY ATTRIBUTES"
20 PRINT
30 PRINT HEX(020402000E); "THE STAR IS BRIGHT."
40 PRINT
50 PRINT HEX(020400040E); "PLEASE UNDERLINE YOUR NAME."
60 PRINT70 PRINT HEX(020400020E); "DO YOU LIKE REVERSE
VIDEO?"80 PRINT 90 PRINT HEX(020404000E); "THE EMERGENCY LIGHT
IS BLINKING."
100 PRINT HEX(0F)
```

```
EXAMPLE 2 - THE DISPLAY ATTRIBUTES.

THE STAR IS BRIGHT.

PLEASE UNDERLINE YOUR NAME.

DO YOU LIKE REVERSE VIDEO?

THE EMERGENCY LIGHT IS BLINKING.

:
```

**Figure 3-2. The Display Attributes**

If the appropriate code is used, any combination of one or more attributes is possible. The following HEX sequences and their respective screen displays are examples of possible combinations of attributes.

- PRINT HEX(020402020E) – Sequence for a bright, reverse video display.
- PRINT HEX(02040B0B0E) – Sequence for a bright, blinking, underlined, reverse video display.

By placing the HEX(0204...) sequence in the appropriate position, Figure 3-2 could be modified to highlight only the key words that describe an attribute. Also, instead of using the PRINT statement to insert blank lines between each displayed sentence, use the control code for line feed, HEX(0A). In Figure 3-3, notice the changed location of the HEX sequence and the difference between the two screen displays.

```
5 PRINT HEX(03)
10 PRINT "EXAMPLE 3 - HIGHLIGHTING KEY WORDS"
20 PRINT HEX(0A)
30 PRINT "THE STAR IS "; HEX(020402000E); "BRIGHT."
40 PRINT HEX(0F0A)
50 PRINT "PLEASE "; HEX(020400040E); "UNDERLINE"; HEX(0F);
" YOUR NAME."
60 PRINT HEX(0A)
70 PRINT "DO YOU LIKE THE "; HEX(020400020E); "REVERSE VIDEO?"
80 PRINT HEX(0F0A)
90 PRINT "THE EMERGENCY LIGHT IS "; HEX(020404000E); "BLINK-
ING."
100 PRINT HEX(0F)
```

```
EXAMPLE 3 - HIGHLIGHTING KEY WORDS

THE STAR IS BRIGHT.

PLEASE UNDERLINE YOUR NAME.

DO YOU LIKE THE REVERSE VIDEO?

THE EMERGENCY LIGHT IS BLINKING.

 :
```

**Figure 3-3. Highlighting Key Words**

## Turning off Character Display Attributes

Once turned on, the selected attribute remains in effect until it is turned off. Since there are several ways to turn an attribute off, first consider the following example.

```
5  PRINT HEX(03)
10 PRINT "EXAMPLE 4 - THE USE OF HEX(0F)"; HEX(0A)
20 PRINT HEX(020402000E); "WE HAVE SELECTED THE BRIGHT
   ATTRIBUTE."
30 PRINT "THE LIGHT IS VERY BRIGHT."
40 PRINT "THE ATTRIBUTE REMAINS IN EFFECT UNTIL IT IS TURNED
   OFF."
50 PRINT "ALL THESE LINES ARE BRIGHT."; HEX(0A)
60 PRINT "HEX(0F)"; HEX(0F); " IS USED TO TURN OFF AN
   ATTRIBUTE."
```

The HEX sequence in Statement 20 selects and activates the attribute "bright intensity" (normal video, no blink, no underline). Notice how the attribute remains in effect for as many lines as desired. (Refer to Figure 3-4.) Each of the four sentences (Statements 20-50) appear on the screen in bright intensity. In this example, the HEX(0F) in Statement 60 is used to turn off the selected attribute and restore normal intensity. An isolated HEX(0F) will *always* turn off a selected attribute and restore normal intensity.

```
EXAMPLE 4 - THE USE OF HEX (OF)

WE HAVE SELECTED THE BRIGHT ATTRIBUTE.
THE LIGHT IS VERY BRIGHT.
THE ATTRIBUTE REMAINS IN EFFECT UNTIL IT IS TURNED OFF.

HEX (OF) IS USED TO TURN OFF AN ATTRIBUTE.

:
```

**Figure 3-4. Using HEX(0F)**

The second way to turn off a selected attribute is to select another attribute. As demonstrated in the next example, each new HEX(0204...) sequence turns off the previous attribute.

```
5  PRINT HEX(03)
10 PRINT "EXAMPLE 5 - SELECTING ANOTHER ATTRIBUTE"; HEX(0A)
20 PRINT HEX(020402000E); "THIS LINE IS BRIGHT."; HEX(0A)
30 PRINT HEX(020400040E); "OUR SECOND LINE IS UNDERLINED."
40 PRINT "THIS LINE IS ALSO UNDERLINED."; HEX(0A)
50 PRINT HEX(020400020E); "NOW WE HAVE SELECTED REVERSE
   VIDEO."; HEX(0A)
60 PRINT HEX(0F); "NORMAL INTENSITY RESTORED."
```

The HEX sequence in Statement 20 selects and activates the attribute "bright intensity." Therefore, the sentence "This line is bright." appears on the screen in bright intensity. However, the new HEX sequence in Statement 30 selects and activates the attribute "underline," thus turning off the bright intensity attribute. Both Statements 30 and 40 are underlined when displayed on the screen. Similarly, the HEX sequence in Statement 50 selects and activates the attribute "reverse video," thus turning off the underline attribute. Lastly, the HEX(0F) in Statement 60 turns off the attribute and restores normal intensity. (Refer to Figure 3-5.)

```
EXAMPLE 5- SELECTING ANOTHER ATTRIBUTE

THIS LINE IS BRIGHT.

OUR SECOND LINE IS UNDERLINED.
THIS LINE IS ALSO UNDERLINED.

NOW WE HAVE SELECTED REVERSE VIDEO.

NORMAL INTENSITY RESTORED.

:
```

**Figure 3-5. Selecting Another Attribute**

## Using Isolated HEX (0E)

An isolated HEX(0E) may be used to activate the *last* attribute selected by a HEX(0204...) sequence. However, when an attribute is turned on in this manner, the attribute will remain in effect for a maximum of *one* text line. Therefore, either an automatic carriage return, a programmed carriage return issued with a HEX(0D), or a HEX(0F) turns the attribute off. Execute the following program:

```
5  PRINT HEX(03)
10 PRINT "EXAMPLE 6 - TESTING ISOLATED HEX(0E)"
20 PRINT
30 PRINT HEX(020400020E); "SELECTING REVERSE VIDEO"; HEX(0F)
40 PRINT
50 PRINT "HOW MUCH OF THIS LINE "; HEX(0E); "APPEARS IN RE
         VERSE VIDEO?"
60 PRINT "NOTICE THAT NORMAL INTENSITY HAS BEEN RESTORED.
         WHY?"
70 PRINT
80 PRINT HEX(0E); "REVERSE VIDEO HAS BEEN REACTIVATED.";
   HEX(0D0A);"WHAT HAPPENED WHEN WE PROGRAMMED A
   CARRIAGE RETURN?"
```

Statement 30 selects and activates reverse video and then immediately turns the attribute off after one line. The HEX(0F) statement turns the attribute off and restores normal intensity. The beginning of Statement 50 appears on the screen in normal intensity until the isolated HEX(0E) reactivates the reverse video attribute for the remainder of the line. Since the attribute was activated by a HEX(0E), the attribute is turned off by the implied carriage return produced by not ending the statement with a comma or semicolon. Therefore, Statement 60 appears in normal intensity. The attribute is reactivated with the HEX(0E) in Statement 80. In this case, the programmed carriage return, HEX(0D), turns off the reverse video attribute and again restores normal intensity. HEX(0A) in Statement 80 issues a line feed so that the second statement of Line 80 does not strike over the first statement of Line 80. HEX(0A) itself does not deactivate the current attribute. In any of these cases, the attribute also could have been turned off by a HEX(0F). (Refer to Figure 3-6.)

```
EXAMPLE 6 - TESTING ISOLATED HEX (OE)

SELECTING REVERSE VIDEO

HOW MUCH OF THIS LINE APPEARS IN REVERSE VIDEO?
NOTICE THAT NORMAL INTENSITY HAS BEEN RESTORED. WHY?

REVERSE VIDEO HAS BEEN REACTIVATED.
WHAT HAPPENED WHEN WE PROGRAMMED A CARRIAGE RETURN?

:
```

**Figure 3-6. Testing Isolated HEX(0E)**

The isolated HEX(0E) can be extremely helpful when highlighting portions of one or more lines that require the same attribute. Consider the following example:

```
5  PRINT HEX(03)
10 PRINT "EXAMPLE 7 - USE OF ISOLATED HEX(0E)"; HEX(0A)
20 PRINT HEX(020400040E); "THIS ENTIRE SENTENCE IS UNDER
   LINED."; HEX(0F0A)
30 PRINT "ONLY THE WORD "; HEX(0E); "ATTRIBUTE"; HEX(0F); " IS
   UNDERLINED."; HEX(0A)
40 PRINT "PART OF THIS LINE "; HEX(0E); "IS UNDERLINED."
```

Statement 20 selects and activates the underline attribute for the first line of output. The beginning of Statement 30 appears in normal intensity without underline, but the HEX(0E) reactivates the last attribute selected (in this case, underline). After just one word, the attribute is again turned off and the remainder of the sentence appears in normal intensity. The HEX(0E) in Statement 40 then reactivates the underline attribute for the last part of the sentence. Since a HEX(0E) was used to reactivate the attribute, the underline attribute will be turned off by the automatic carriage return. (Refer to Figure 3-7.)

```
EXAMPLE 7 - USE OF ISOLATED HEX(OE)

THIS ENTIRE SENTENCE IS UNDERLINED.

ONLY THE WORD ATTRIBUTE IS UNDERLINED.

PART OF THIS LINE IS UNDERLINED.

:
```

**Figure 3-7. Using Isolated HEX(0E)**

## Special Uses of Alternate Display Attributes

The following list explains special uses of alternate display attributes.

1. LIST D

   The CPU sends out a HEX(0E) at the beginning of each REM% statement in the program. Thus, comment statements appear in the most recently selected alternate display attribute.

2. 100 PRINT "PROMPT";: LINPUT HEX(0E), A$: PRINT A$

   The field to be entered appears in the most recently selected alternate display attribute. When entry is terminated with a carriage return, the alternate attribute is cancelled, so the PRINT statement prints A$ in normal intensity.

3. 150 PRINT HEX(0E); "PROMPT"; HEX(0F);

   160 LINPUT A$

   This time, only the prompt appears in the most recently selected alternate attribute.

## Summary of Display Attribute Rules

The following list contains the general rules discussed in the previous sections for governing the use of display attributes:

1. HEX(02 04 xx yy 0E) selects and activates a display attribute. Attributes activated in this manner are turned off only by HEX(0F) or by another HEX(0204...) sequence. The attribute is *not* turned off by carriage return, HEX(0D). Therefore it is possible to highlight a portion of either one or several lines.
2. HEX(02 04 xx yy 0F) selects, but does *not* activate, a display attribute. Normal intensity is activated instead.
3. An isolated HEX(0E) activates the attribute selected by the last HEX(0204...) sequence for a maximum of one text line. The attribute remains in effect until the occurrence of either an automatic carriage return, a programmed HEX(0D), or a HEX(0F).
4. Rule 1 takes precedence over Rule 3. If an attribute is selected and activated by Rule 1, a subsequent HEX(0E) will *not* cause the attribute to be turned off by the next carriage return.

5. An isolated HEX(0F) always turns off the alternate attribute and restores normal intensity.
6. Screen clear, HEX(03) clears the screen to black but otherwise has no effect on the meaning of HEX(0E) or the attribute currently in effect. Likewise, scrolling the screen scrolls in a black line, but otherwise has no effect on attributes.
7. Reverse video spaces are white, not black. Zoned format PRINT statements, i.e., PRINT, PRINT TAB, and the third parameter of PRINT AT, use spaces to clear the screen. These statements will leave white areas on the screen when reverse video is activated.
8. The RESET key causes normal-intensity characters to be selected and HEX(0E) to be defined as high-intensity characters.
9. The system considers all codes HEX(00) - HEX(0F) to occupy no space on the output medium. Thus, attribute selection sequences do not cause the system to issue automatic carriage returns or throw off the column count used by TAB and zoned format PRINT statements.
10. Control codes HEX(00) - HEX(0F) do not have attributes. It is not possible to change the attribute of a character by passing the cursor through it with a PRINT AT statement.

## Selection of Character Sets

BASIC-2 offers two character sets: the normal character set and the alternate character set. (Refer to Appendix A.) The following sequence is used for selecting either character set.

```
               HEX (02 02 xx 0F/0E)

where:

      02 02 =  The control code sequence that indicates to the
               screen that a character set will be selected.

         xx =  A HEX code specifying the character set to be
               selected.

              If xx = 00 The normal character set is selected.  The
              codes HEX(90) to HEX(FF) are underline
              versions of characters from HEX(10) to HEX(7F).


              If xx = 02 The alternate character set is selected.
              The codes HEX(80) to HEX(FF) represent the
              graphic characters and symbols.

      0F/0E =  A terminator character that signals the end of the
              character selection sequence.
```

In the character set selection, the following items should be noted:

1. With the exception of the HEX(80) code, the characters represented by the codes HEX(10) to HEX(8F) are identical in both the normal and the alternate character set.
2. In the alternate character set, the codes HEX(9C) to HEX(BF) are presently undefined and reserved for future expansion. Any use of these codes involves the risk of being incompatible with future use of the screen.

The 64 graphic characters, HEX(C0) to HEX(FF), are represented by all the combinations of sixths of a character space, where the character space is divided as shown in Figure 3-8. When displayed, graphic characters are extrapolated to fill the entire character position. For this reason, adjacent areas of two graphic characters will touch; thus, continuous lines (bars) of light or dark areas can be displayed on the screen. When combined with display attributes, character graphics are useful for the construction of bar graphs, histograms, and other special displays.

**Figure 3-8. Division of a Character Space**

## Summary of Character Set Selection

The rules concerning the use of character set selection can be summarized as follows:

1. HEX(02 02 00 0F) selects the normal character set. The meaning of codes HEX(90) to HEX(FF) are defined to be the normal characters HEX(10) to HEX(7F) with underline.
2. HEX(02 02 02 0F) selects the alternate character set. The codes HEX(80) to HEX(FF) represent the graphic characters and other special symbols.
3. Power on and RESET select the default character set.
4. Carriage return does *not* affect character set selection. The sequences given in Rules 1 to 3 are the only methods for changing character sets.
5. As with attributes, the character set selection sequences affect the interpretation of characters at the time they are received by the screen. Therefore, underlined and graphic characters may be used in different areas of the same display. Once on the screen, a character is modified only by explicitly striking over it with another character or by screen clear.
6. All display attributes can be used with both the normal and the alternate character set.

Box graphics can also be used for highlighting entry fields as shown in the following example:

```
10 PRINT "PROMPT"; BOX(1, 17);:LINPUT A$
```

## Box Graphics

By using the PRINT BOX (height, width) statement, one can display continuous horizontal or vertical lines, enabling forms to be drawn or information to be separated by lines or boxes. The horizontal line unit is a line segment the width of a character space but positioned from the middle of one character space to the middle of the next character space. Horizontal lines are displayed between rows of characters.

The vertical line unit has the height of a character space. Vertical lines are drawn through the middle of a character space; the line coexists with the character at that location.

> ***Note:*** *Since the height and width of a character space are not the same unit measurement, boxes are not drawn proportionally. However, because of these measurements, you can easily box fields of characters.*

Figures 3-9 and 3-10 illustrate the placement of box graphic lines. Figure 3-9, which shows the smallest possible box, was produced by the statement PRINT BOX(1,1); "AB". It illustrates the placement of horizontal and vertical box graphic lines relative to the character position. Figure 3-10, which was produced by the statement PRINT BOX(1,1); HEX(0202020F); HEX(E1CC), demonstrates where box graphic lines appear relative to character set graphic blobs.

You can consider the screen as both a box graphics display and a character display that just happen to be displayed on the same screen. While in Character mode, only the characters and their attributes are modified while box graphics remain intact. For example, within a boxed area used to highlight a prompt, the prompt may be rewritten a number of times without altering or erasing the box itself. The one exception to this rule is screen clear, HEX(03) which clears both characters and box graphics. During a box graphics sequence, characters and their attributes are undisturbed.

Because the Character and Box Graphic modes are independent, you can easily update portions of either display. The third argument of PRINT AT is useful for clearing portions of the display. Though slower than screen clear, the statement PRINT AT (0,0) is useful for clearing the characters from the screen without disturbing the box graphics.

**Figure 3-9. Box Graphic Line Placement Relative to Character Position**

Figure 3-10. Box Graphic Line Placement Relative to Graphic Character Set

Refer to Chapter 10 for more information on the PRINT BOX statement.

# 4

# Numeric Operations

## Overview

BASIC-2 distinguishes 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 discussed in Chapter 6.

## Numeric Values

The system stores floating-point values in a format containing 13 digits, a sign, and a signed 2-digit integer exponent. Numeric values can be stored in memory in the form of a constant or as the value of a numeric-variable. Numeric values can be values you enter or values the system produces as the result of evaluating a numeric expression. In any case, the legal range of numeric values that the system can handle is as follows:

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

If more than 13 digits are entered for a numeric value, the systemlsignals an error and the number is rejected. However, the system ignores leading zeros before the decimal point. The numeric values you enter from the keyboard can be specified in either fixed-point or exponential format.

### Fixed-Point Format

A numeric value entered in fixed-point format can contain a maximum of 13 digits, a sign, and a decimal point. The system ignores embedded spaces in the value. The sign of the value must precede the digits; unsigned values are regarded as positive numbers. If a decimal point is not entered, it is assumed to be at the right of the last digit. The following examples illustrate legal numeric entries in fixed-point format:

```
12.003
+1234567890.123
-.00725
```

### Exponential Format

A numeric value entered in exponential format can contain a maximum of 13 digits, a sign, a decimal point, and a signed 1- or 2-digit exponent that is denoted by the letter E. In exponential format, the value is equal to the entered number times 10 to the power of the exponent. The number must conform to the rules for fixed-point values. The exponent consists of the letter E followed by an integer value. The integer value can contain a maximum of two digits. The sign of the exponent must precede the exponential digits; unsigned exponents are regarded as positive. The following examples illustrate legal numeric entries in exponential format.

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

## Numeric Constants

A constant is a legal numeric value that appears in a BASIC statement. Constants must conform to the format rules for legal numeric entries detailed in the section entitled "Numeric Values". Numeric constants 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 can be used wherever numeric expressions are permitted. The value of a constant is not altered during program execution. In the following statements, the boxed items are examples of numeric constants:

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

## Numeric Variables

Numeric variables 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 statements.

You must identify each variable with a unique variable name. In BASIC-2, numeric variable names consist of an an uppercase letter optionally followed by a digit. The following are examples of legal variable names:

```
A, X0, Y9
```

A numeric-scalar variable stores a single value.

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 store and process multiple numeric values. Numeric-array-variable names consist of a single uppercase letter, optionally followed by a digit, with parentheses enclosing the subscripts.

BASIC-2 defines an array as the array name immediately followed by parentheses. The following variables represent an entire array:

```
A(), B0(), Z9()
```

A particular element in an array is identified by specifying its subscript(s) in parentheses following the array name. The following examples represent an element of a numeric-array:

```
T(12,6), C1(55)
```

A scalar-variable and an array-variable can have the same name since they are independent variables, but a 1-dimensional array-variable and a 2-dimensional array-variable cannot have the same name in the same program.

Before referencing or assigning data to variables, you must define and reserve space in memory for certain variables. You do not need to explicitly define numeric-scalar-variables; they are automatically defined when first referenced in a program. However, you must explicitly define array-variables with a DIM or COM statement. Any reference to an undefined variable results in an error.

*Example:*

The following statement defines two numeric-arrays: N() is a 1-dimensional array of 10 elements; Q1() is a 2-dimensional array of 5 rows and 5 columns (25 elements).

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

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 defining array-variables.

- The numbering of array elements starts at 1 rather than 0. Thus, a subscript of 0, as in A(0), is not permitted.
- An array can be defined to have one or two dimensions.
- The dimension of an array with one dimension can not exceed 65535; dimensions of arrays with 2 dimensions cannot exceed 255.

## Numeric Expressions

A numeric expression can consist of a single variable or constant or a series of variables and constants separated by arithmetic operators and numeric functions. Numeric expressions can be evaluated in a variety of BASIC statements. For example, the assignment (LET) statement evaluates numeric expressions and assigns the result to a variable; the PRINT statement evaluates numeric expressions and displays the result on the specified device. The following examples illustrate numeric expressions.

```
A = B
PRINT 4*A+B
N = N+SQR(A+B*2)
ON 3*J-1 GOTO 100, 200, 300
```

## Arithmetic Operators

The following arithmetic symbols or operators perform mathematical operations in BASIC-2:

| | |
|---|---|
| + | 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") |

A BASIC-2 expression cannot contain two consecutive arithmetic operators. For example, the following statement violates BASIC-2 syntax and is illegal:

```
20 PRINT A * -J
```

Parentheses must be used to separate the operators. Line 20 can be written correctly with the following statement:

```
20 PRINT A * (-J)
```

## Order of Evaluation

Numeric 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:

1. Exponentiation (↑) is performed from left to right.
2. Multiplication and division (*, /) are performed from left to right.
3. Addition, subtraction, and negation (+,-) are performed from left to right.

## Altering the Standard Order of Evaluation

Parentheses can alter the standard order in which a numeric expression is evaluated. 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, which are identical except for parentheses:

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

In constructing expressions, you can nest parentheses 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 is always evaluated first.

The use of parentheses can be particularly critical when raising a negative number to an even power.

*Example:*

The following statement yields the result -9 because the exponentiation is performed prior to the negation:

```
30 PRINT -3 ↑ 2
```

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 of 9.

## Round/Truncate Option

The results of all arithmetic operations (+, -, *, /, ↑ ), as well as the square root (SQR) and modulo (MOD) functions, are rounded to 13 significant digits. Alternatively, results can be truncated to 13 significant digits by executing a SELECT NO ROUND statement.

Once a SELECT NO ROUND statement is executed, all results are truncated to 13 digits. Rounding can be restored subsequently with a SELECT ROUND statement. Rounding is selected automatically when the system is master initialized.

## Computational Errors

Computational errors can be produced in the course of performing arithmetic operations or evaluating numeric functions. Usually, 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 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.

## System-defined Numeric Functions

The system provides a variety of built-in trigonometric and mathematical functions. These functions are summarized in Table 4-1; those that require a detailed explanation are discussed in the following pages.

**Table 4-1. System-Defined Numeric Functions**

| Function | Meaning | Examples |
|---|---|---|
| INT(x) | Returns the greatest integer value of the expression. | INT(13.5) = 13<br>INT(-5.2) = -6 |
| FIX(x) | Returns the integer portion of the expression. | FIX(13.5) = 13<br>FIX(-5.2) = -5 |
| ABS(x) | Returns the absolute value of the expression. | ABS(7 ↑ 2) = 49<br>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<br>SGN(-.123) = -1<br>SGN(0) = 0 |
| MOD(x,y) | Returns the remainder of x/y. | MOD(8,3) = 2<br>MOD(4.2,4) = .2 |

(continued)

Table 4–1. System-Defined Numeric Functions (continued)

| Function | Meaning | Examples |
|---|---|---|
| ROUND(x,n) | Returns 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<br>ROUND(5.06,1) = 5.1<br>ROUND(5.5,0) = 6<br>ROUND(-3.8,0) = -4<br>ROUND(-3.2,0) = -3<br>ROUND(1256,-2) = 1300<br>ROUND(1256,-1) = 1260 |
| RND(x) | Produces a random number between 0 and 1. | RND(1) = .8392246561935 |
| SQR(x) | Returns the square root of the expression. | SQR(18+7) = 5<br>SQR(25) = 5 |
| MAX x,y,...,z) | Returns the maximum value among the specified expressions or numeric array(s). | MAX(1,3,2) = 3 |
| MIN (x,y,...,z) | Returns the minimum value among the specified expressions or numeric array(s). | MIN(-6,-3,-1) = -6 |
| LGT(x) | Returns the common logarithm (log base 10) of the expression. | LGT(1000) = 3 |
| LOG(x) | Returns the natural logarithm (log base e) of the expression. | LOG(3052) = 8.023552392404 |
| EXP(x) | Returns the value of e raised to the power of the expression (i.e., the natural antilog of the expression). | EXP(.34*(5-6)) = .7117703227626<br>EXP(1) = 2.718281828459 |
| #PI | Assigns the value 3.14159265359. | 4*#PI = 12.56637061436 |
| SIN(x) | Returns the sine of the expression. | SIN(#PI/3) = .8660254037847 |
| COS(x) | Returns the cosine of the expression. | COS (.693 ↑ 2) = .8868799122685 |
| TAN(x) | Returns the tangent of the expression. | TAN(10) = .6483608274591 |
| ARCSIN(x) | Returns the arcsine of the expression. | ARCSIN(.003) = 3.00000450E-03 |
| ARCCOS(x) | Returns the arccosine of the expression. | ARCCOS(.587) = .9434480794406 |
| ARCTAN(x) or ATN(x) | Returns the arctangent of the expression. | ARCTAN(3.3) = 1.276561761684 |

## INT and FIX Functions

The INT function is the greatest integer function, often depicted in algebra with square brackets (e.g., [5.6]). 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 that is less than the value.

*Examples:*

```
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 the same as the INT function, 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, allowing the FIX function to be expressed in terms of the following functions:

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

*Examples:*

```
FIX(3.8)   =     3
FIX(6.1)   =     6
FIX(-2.6)  =    -2
FIX(-2.1)  =    -2
FIX(30)    =    30
FIX(-30)   =   -30
```

## MAX and 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 largest value of the expression or numeric-array-element in the list of arguments; the MIN function returns the smallest value.

*Examples:*

This example assumes that the following values exist for a numeric-array and a numeric-variable.

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

The following values are returned for the given expression:

```
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, a function of two expressions, returns the remainder when the first expression is divided by the second. The MOD function simulates modulo arithmetic if the second expression has a positive integer value. 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 0, MOD returns the first expression. The MOD function is equivalent to the following expression:

```
x - y * INT(x/y)
```

*Examples:*

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

## RND (Random Number) Function

The RND function generates uniformly distributed random numbers with values between 0 and 1. RND can 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 it is necessary to reuse the same series of random numbers (e.g., when debugging a program), you can use the RND function with a zero argument to reset the list to the first random number.

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, and so on. As a result, each time RND is executed with a nonzero argument, it produces a new random number. The value of the RND argument has, apart from the fact that it is nonzero, no relation to the random number produced.

*Example:*

The following routine prints out the first 100 random numbers in the random number list each time the program is run. If Line 10 is deleted, the program produces a different set of random numbers each time it is run.

```
10 X = 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, and so on. The use of RND with a zero argument resets the list pointer to the first number in the list.

On a multiuser system, eachpartition has a list of random numbers. Resetting the list to the beginning by using RND(0) only affects the partition that executes the RND(0).

## ROUND Function

The ROUND function rounds 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. The second argument is the rounding factor.

*Example:*

In the following statement, the value to be rounded is 5.374, and the rounding factor is 2:

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

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 0.

A positive rounding factor rounds the value to a specified decimal place (i.e., a specified position to the right of the decimal point). A rounding factor of 1 rounds the value to the nearest tenth; a rounding factor of 2 rounds the value to the nearest hundredth, and so on.

*Examples:*

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

A rounding factor of 0 rounds the value to the nearest integer.

*Examples:*

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

A negative rounding factor rounds the value 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 1 (factor + 1) specifies the integer position for rounding. In effect, a rounding factor of -1 rounds the value to the nearest ten; a rounding factor of -2 rounds the value to the nearest hundred, and so on.

*Examples:*

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

The ROUND function is equivalent to the following expressions, where X is the value to be rounded and N is the rounding factor:

```
SGN(x)*INT(ABS(x)*10 ↑ (FIX(n))+.5)/10 ↑ (FIX(n))
```

## 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 0, 0 if the argument equals 0, or +1 if the argument is greater than 0.

*Examples:*

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

## Trigonometric Functions

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 degrees = 400 grads). Trigonometric functions are evaluated in radians, unless the system is explicitly instructed to use degrees or grads. If degrees or grads are required, they must be specified with the following SELECT statements prior to performing trigonometric calculations:

SELECT D – Use degrees in all subsequent trigonometric calculations.

SELECT G – Use grads in all subsequent trigonometric calculations.

SELECT R – Use radians in all subsequent trigonometric calculations.

Radian measure is automatically selected upon Master Initializing the system or when a CLEAR command is issued.

## 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. With the exceptions of the #ID, ERR, SPACE, and SPACEK functions, the remaining special-purpose numeric functions operate on alphanumeric arguments and are described in detail in Chapter 5. The ERR function is discussed with the error control features in Chapter 9.

### SPACE Function

The SPACE function returns the amount of memory not currently occupied by program text or data, minus the amount of memory occupied by the value stack. The value returned represents the actual amount of free space in memory at any point during execution.

The Value Stack initially occupies zero bytes but expands during program execution. To determine how much free space is actually available, check the value of 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.

### SPACEK Function

Before memory has been partitioned, the SPACEK function returns the total amount of available user memory divided by 1,024. For example, a system with 56K of user memory returns SPACEK = 56. After a system has been partitioned, SPACEK returns the size of the partition that executes the SPACEK function.

### SPACE S and SK

SPACE S determines the amount of memory that is not currently occupied by any partition. This is Ramdisk Memory. SPACE SK returns the total amount of memory including all allocated and Ramdisk memory. This is the total memory on the CPU board.

### #ID Function

The #ID function returns the value of the CPU identification number (a number from 1 to 65535). With the #ID function, a program can distinguish one system from another. This capability is useful in licensing software to specific installations.

**Table 4-2. Special-Purpose Numeric Functions**

| Function | Meaning | Examples |
|---|---|---|
| BIN | Converts an integer value to a binary number. | A$ = BIN(65) |
| ERR | Returns the error code of the last error condition. | X = ERR |
| LEN | Determines the length of a character string. | X = LEN(A$) |
| NUM | Determines whether or not a character string is a legal representation of a BASIC number. | X = NUM(A$). |
| POS | Returns the position of the first (or last) character in a character string that meets a specified condition. | X = POS(A$="$") |
| VAL | Computes the decimal equivalent of a binary value. | X = VAL(A$) |
| VER | Verifies that a character string conforms to a specified format. | Y = VER(B$,"###") |
| SPACE | Determines the amount of free space available in memory. | Z = SPACE |
| SPACE K | Returns the total user memory size or partition size divided by 1,024. | Z1 = SPACE K |
| SPACE S* | Determines the amount of memory that is not currently occupied by any partition (Ramdisk Memory). | Z2 = SPACE S |
| SPACE SK* | Returns the total amount of memory including all allocated and Ramdisk memory. | Z3 = SPACE SK |
| #ID | Returns the CPU identification number. | PRINT #ID |

Note: * CS/386 ONLY

5

# Alphanumeric Strings

## Alphanumeric Character Strings

In addition to its capability for manipulating and operating upon numeric values, BASIC-2 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 can consist of any combination of keyboard characters, including the letters A to Z, the numbers 0 to 9, and special symbols such as the plus sign ( + ), minus sign ( - ), and dollar sign ( $ ). 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: as the values of alphanumeric string variables or as literal strings.

## Alphanumeric String Variables

Alphanumeric character strings are stored and processed in a type of variable called the alphanumeric string variable or simply the alpha-variable. The programmer must identify each variable with a unique variable name. Alpha-variables consist of an uppercase letter optionally followed by a digit and are distinguished from numeric-variables by the presence of a dollar sign ( $ ). The following are examples of legal alphanumeric-variable names:

```
N$, S0$, A9$
```

A numeric-variable and an alpha-variable are different variables that are distinguished by a dollar sign. 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 it cannot take part in standard numeric arithmetic operations. Only numeric data can be used in arithmetic operations. (Refer to Chapter 4 for a discussion of numeric operations.)

Alpha-variables are of two types: scalar and array. An alpha-scalar- variable can store a single character string. An alpha-array-variable consists of one or more array elements, each of which can store a character string. 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 is explained in the section entitled "Using the Alpha-Array as a Scalar-Variable".)

Alpha-array-variables can be either 1-dimensional or 2-dimensional arrays. A 1-dimensional array resembles a list because it has a single column of elements. The dimension of a 1-dimensional array must not exceed 65535. A 2-dimensional array resembles a table because it has both rows and columns of elements. Dimensions of 2-dimensional arrays must not exceed 255.

BASIC-2 regards scalar-variables and array-variables as different types of variables. However, array-variables containing different numbers of dimensions are different but related kinds of arrays. This allows the same name to be used in a program for an alpha-scalar-variable and an alpha-array-variable, but the same name cannot be used for a 1-dimensional alpha-array and a 2-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.

## Alphanumeric-variable Length

An alphanumeric-variable identifies a unique location in memory reserved for the storage of alphanumeric data. During program resolution, the system scans the program for all variable references and then reserves space for each variable. You use a DIM or COM statement to specify the amount of space reserved for each variable. The maximum length of an alpha-scalar-variable or of an element in an alpha-array is 124 characters (bytes), while the minimum length is one byte in each case. If you do not explicitly dimension an alpha-scalar-variable in a DIM or COM statement, the system automatically reserves 16 characters (bytes) for the variable. Similarly, if you do not specify an element length when dimensioning an alphanumeric-array, the system automatically reserves 16 characters (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 does not occupy the entire defined length. The end of the value of an alpha-variable is usually 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. Trailing blanks generally are not considered part of the value of an alpha-variable.

*Example:*

In the following program, the trailing blanks of A$ are not printed:

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

The character string stored in an alpha-variable is called the current value of the alpha-variable. 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.

*Example:*

In the following program, the LEN function does not consider trailing blanks to be part of the value of the alpha-variable.

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

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

## STR Function

The STR (string) function examines or operates on a specific portion of the value of an alpha-variable. 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.

*Example:*

The following examples assume that A$ = "ABCDEFG":

| **Statements** | **Results** |
|---|---|
| STR(A$,1,4) | Defines a substring beginning with the first byte of A$, four bytes in length. A$ = "ABCD" |
| STR(A$,5,3) | Defines a substring beginning with the fifth byte of A$, three bytes in length. A$ = "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$ = "A12DEFG" |
| STR(A$) | Defines a substring beginning with the first byte of A$ and ending with the last byte of A$, including any trailing spaces. A$ = "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
```

## 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.

*Example:*

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

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

Literal strings also can be assigned to alphanumeric-variables. A literal string can 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.

*Example:*

In the following program, the value is truncated to five characters because the defined length of A$ is five bytes:

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

The minimum length of a literal string is one; the null string ("") is not allowed. An alphanumeric literal string can contain any character, with the exceptions of a carriage return (RETURN), the quotation mark ("), and the characters represented by codes HEX(FB) to HEX(FF), which are reserved for system use.

## Hexadecimal Literal Strings

Hexadecimal literal strings are a form of literal strings consisting of one or more hexadecimal codes specified in a HEX function. (Refer to the discussion of the HEX function in the section entitled "General Forms of the Alphanumeric and Special-Purpose Functions and Operators".) Hexadecimal codes are composed of a pair of hexadecimal digits (0 to 9 or A to F). Each pair of hexadecimal digits specifies the value of a byte in the string. Hex codes are particularly useful for representing special characters not found on the system keyboard and system control codes.

Hex literal strings are legal wherever alphanumeric literal strings are allowed. In particular, the programmer can assign hex literal strings to alpha-variables in an assignment statement.

*Example:*

In the following program, the characters 123 are printed, since they are represented by the hex codes 31, 32, and 33.

```
:60 A$ = HEX(313233)
:70 PRINT A$
:RUN123
```

## Concatenation of Strings

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

*Example:*

```
:10 A$ = "BRIAN"
:20 B$ = "JAMES."
:30 C$ = A$ & B$
:40 PRINT C$
:RUN
BRIANJAMES.
```

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

*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.

*Example:*

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

The concatenation operator can be used only on the right side of an alphanumeric assignment statement; it is not legal in any other statement. Although more than one concatenation operator can be used in the same assignment statement, the concatenation operator cannot appear in combination with other alphanumeric operators in the same statement. For example, the following program statements are legal:

```
:10 A$ = B$ & C$ & D$
:10 A$ = B$ AND C$ AND D$
```

The following statement, however, produces an error:

```
:10 A$ = B$ & C$ AND D$
```

## Using 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, cannot be referenced as a scalar-variable in certain statements that 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$( ), B4$( ).

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 n x m array is

```
(1,1), (1,2), ..., (1,m),
(2,1), (2,2), ..., (2,m),
  .
  .
  .,
(n,1), (n,2), ..., (n,m)
```

For example, the following statement dimensions A$ to be a 2 x 3 array with each element four bytes in length.

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

*Example:*

Assume 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, where the character "b" denotes a space:

```
              1            2            3

1          ABCD         EFGH         IJKL
2          MNOb         Pbbb         Qbbb
```

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 next-to-last element are not considered part of the value of the array.

```
: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$(2,2) do not precede any nonblank character and are no longer counted as part of the value of A$( ).

## Alphanumeric Expressions

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

```
alpha-variable [,alpha-variable] ... = alpha-expression
```

A variable on the left side of the equal sign is called a receiver-variable because it receives a value. For example, in the following statement A$ is the receiver:

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

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 as follows:

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

The concatenation operator (&) cannot be combined with other operators in the same expression. An alpha expression is processed from left to right, one term at a time. Parentheses *cannot* 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.

*Example:*

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

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

Any number of alpha operators and operands can be combined in a single expression, with the exception of the concatenation operator. Table 5-1 summarizes the available alphanumeric operators. Table 5-2 summarizes the available alphanumeric operands; these can be alpha-variables, literal strings, or alpha-functions.

## General Forms of the Alphanumeric and Special-purpose Functions and Operators

General forms of the alpha functions and operators are discussed in alphabetical order on the following pages. The special-purpose numeric functions BIN, LEN, NUM, POS, VAL, and VER are discussed in this chapter because they operate on alphanumeric arguments and are used primarily in the analysis and manipulation of alphanumeric string data. However, the special alpha operators ADD, DAC, DSC, and SUB, which are used for binary and packed decimal operations, are discussed in Chapter 6.

## ALL Function

*Format:*

```
          hh
  ALL     alpha-variable
          literal-string

  where:

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

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 can be specified with a pair of hex digits 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. (Refer to the discussion of alpha expressions and the alpha assignment statement in the section entitled "Alphanumeric Expressions".) For example, the following statement sets each character in A$( ) equal to binary 0:

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

ALL is also useful in logical operations for changing certain bits in every character of an alpha-variable or array.

*Example:*

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

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

*Examples of valid syntax:*

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

## AND, OR, XOR Operators

*Format:*

```
AND
OR
XOR
```

The logical operators AND, OR, and XOR (exclusive or) perform the specified logical operations on the value of an alpha-variable. These operators can be used only in the alpha-expression portion of an alphanumeric assignment statement. (Refer to 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.

*Example:*

The following statement logically ANDs the value of B$ with A$; the result is stored in A$:

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

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 is operated on.
- The entire contents of the receiver-variable, including trailing spaces, are operated on.
- The entire contents of the operand, including any trailing spaces, are used. (Trailing spaces usually 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 following statement operates only on the third and fourth bytes of A$:

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

The logical operators AND, OR, and XOR also can be used in the IF...THEN statement to separate multiple conditions. Refer to Chapter 10.

*Examples of valid syntax:*

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

## BIN Function

*Format:*

```
BIN( expression [,length] )

where:

      length=  numeric-variable or the digit 1, 2, 3, or 4

      If length= 1, 0 <= value-of-expression < 256
      If length= 2, 0 <= value-of-expression < 65,536
      If length= 3, 0 <= value-of-expression < 16,777,216
      If length= 4, 0 <= value-of-expression < 4,294,967,296
```

BIN is an alphanumeric function that uses a numeric argument, but returns an alphanumeric value; it is the inverse of the VAL function. The BIN (binary) function converts the integer value of the expression to a binary value. The number of bytes in the binary value is specified by the digit; if no digit is included, the length is assumed to be one byte. A numeric-variable can now be used to specify the length of the binary result of the BIN function. The BIN function can only be used in the alpha-expression portion of an alphanumeric assignment statement. BIN is especially useful for code conversion and conversion of numbers from internal decimal format to binary.

*Example:*

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

```
$ = BIN(65)
```

*Examples of valid syntax:*

```
B$ = BIN(X,L)
A$ = BIN(X)
TR(A$,I,2) = BIN(X,2)
C$ = BIN(X*Y/Z,4)
```

## BOOL Operator

*Format:*

```
    BOOL h

  where:

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

BOOL is a generalized logical operator that performs a specified operation on the value of the receiver-variable. The operation to be performed is specified by the hexadecimal digit following BOOL (refer to Table 5-3). BOOL can be used only in the alpha-expression portion of an alpha assignment statement. (Refer to the discussion of alpha expressions and the alpha assignment statement in the section entitled "Alphanumeric Expressions".) 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 following statement logically not-ANDs the value of B$ with the value of A$ and assigns the result to A$:

```
A$ = BOOL7 B$
```

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. (Trailing spaces usually 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 following statement operates only on the third and fourth bytes of A$:

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

In every case, the logical operation to be performed is identified by the hex digit following BOOL. The hex digit used to identify each operation is a kind of mnemonic that 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 5-3.

**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:*

```
A$ = BOOL1 B$
B$ = C$ BOOL7 D$
STR(A$,3,2) = BOOL9 ALL(HEX(FF))
```

## Date

*Format (as a statement):*

```
                alpha-variable                          alpha-variable
 DATE =                              PASSWORD
                literal-string                          literal-string
```

*Format (as a function):*

```
 DATE
```

The DATE statement sets or changes the system date. The new date is specified by the alpha-variable or literal-string following the equal sign. The date is specified as a 6 character ASCII value of the form YYMMDD (year, month, day).

The alpha-variable or literal-string following PASSWORD specifies the system password. If the password is correct (i.e., matches the password set in the @GENPART utility), the date is updated. If the password is incorrect an error results and the date is not updated.

*Examples of valid syntax:*

```
DATE = "820601" PASSWORD "SYSTEM"
DATE = D$ PASSWORD P$
```

The DATE function is an alphanumeric function that returns a 6 character ASCII string containing the current date in the form YYMMDD. DATE is used as an operand within alphanumeric expressions.

*Examples of valid syntax:*

```
T$ = DATE & TIME
A$ = DATE AND HEX(FFFF00)
```

## HEX Literal

*Format:*

```
     HEX(hh [hh] ...)

  where:

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

The HEX literal string permits the use of any 8-bit (1-byte) character code in a BASIC-2 program. A HEX literal can 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 results.

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

```
PRINT HEX(03)
```

Any character can be represented by a pair of hex digits. Refer to Appendix A for a complete chart of control codes; refer to the appropriate peripheral manual for codes pertaining to other devices.

*Examples of valid syntax:*

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

## LEN Function

*Format:*

```
LEN (alpha-variable)
```

The LEN (length) function determines the number of characters in the value of an alphanumeric-variable. LEN is a special-purpose numeric function that uses an alphanumeric argument but returns a numeric value as a result. LEN can be used wherever numeric functions are legal. (Refer to the discussion of numeric functions in Chapter 4.)

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, including leading and embedded spaces. In the special case of a variable that contains all blanks, the length is 1.

*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 an STR function returns the length parameter specified in the STR function, irrespective of the contents of the variable.

*Example:*

```
:10 DIM A$16
:20 A$ = "AB   "
:30 PRINT LEN(STR(A$,,5))
:40 PRINT LEN(STR(A$))
:RUN
 5
 16
```

*Examples of valid syntax:*

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

## NUM Function

*Format:*

```
     NUM (alpha-variable)
```

The NUM function determines the number of sequential ASCII characters in the specified alphanumeric-variable that represents a legal BASIC-2 number. A BASIC-2 number consists of numeric characters in a standard format. A numeric character is defined to be one of the following characters.

- Blanks
- Digits 0 to 9
- Decimal point (.)
- Leading plus or minus sign (+, -)
- Letter E

The counting of numeric characters begins with the first character of the specified variable or STR function. Leading and trailing spaces are included in the count. The counting of numeric characters ends when a nonnumeric character occurs, when the sequence of numeric characters fails to conform to standard BASIC-2 number format, or when all characters in the specified variable have been scanned. 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. (The BASIC-2 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 that utilizes an alphanumeric argument, but returns a numeric result. The NUM function can be used wherever numeric functions are legal. Refer to the discussion of numeric functions in Chapter 4.

**Examples** / **Comments**

```
:10 A$ = "+24.37#JK"
:20 X = NUM(A$)
:30 PRINT "X ="; X
:RUN
X = 6
```

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

```
:10 INPUT A$
:20 IF NUM(A$)=16 THEN 50
:30 PRINT "REENTER": GOTO 10
:50 CONVERT A$ TO X
:60 PRINT "X ="; X
:RUN
? 123A5
REENTER
? 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 (ASCII characters). If the value represents a legal number, NUM(A$) = 16, the number or characters in the alpha variable.

## POS Function

*Format:*

```
                                         <
                alpha-variable           <=      hh
POS  (  [-]                               =     alpha-variable       )
               literal-string            >=     literal-string
                                          >
                                         <>
where:

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

The POS (position) function returns the position of the first character in an alpha-variable or literal string that satisfies a specified relation to a given value. The value against which comparison is made follows the relational operator and can 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 that 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 equals zero.

The capability 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 that is clearer and more self-explanatory.

*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.

*Example:*

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

POS is a special-purpose numeric function that uses an alphanumeric argument, but returns a numeric value as a result. The POS function can be used wherever numeric functions are legal. Refer to the discussion of numeric functions in Chapter 4.

*Examples of valid syntax:*

```
X = POS(A$ = "$")
PRINT POS(STR(A$,I,J) = HEX(0D))
IF POS(A$( ) = "T") = 0 THEN 100
B$ = STR(A$,POS(A$ = "+"))
Y = POS(-B$ < C$)
X = POS(A$ = 0A)
T = POS("123456789" = T$)
J = POS(HEX(0102040810204080) > A$)
L = POS(-"0:A a" = STR(B$,N,1))
```

## STR Function

*Format:*

```
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 < maximum no. of characters in alpha-variable.
```

The STR (string) function 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 following statement sets B$ equal to the 3rd, 4th, 5th, and 6th characters of A$:

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

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

*Example:*

```
:10 A$ = "ABCDE"
:20 PRINT STR (A$,,2)
:RUN
AB
```

If the n parameter is omitted, the remainder of the alpha-variable is used, including trailing spaces.

*Example:*

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

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

*Example:*

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

If the STR function is used on the left 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.

*Example:*

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

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

*Examples of valid syntax:*

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

## Time

*Format (as a statement):*

```
                alpha-variable                      alpha-variable
TIME =                             PASSWORD
                literal-string                      literal-string
```

*Format (as a function):*

```
TIME
```

The TIME statement sets or changes the system time. The new time is specified by the alpha-variable or literal-string following the equal sign. The time is specified as a 6 character ASCII value of the form HHMMSS (hour, minute, second). The hour value is based on a 24 hour clock.

The alpha-variable or literal-string following PASSWORD specifies the system password. If the password is correct (i.e., matches the password set in the @GENPART utility), the time is updated. If the password is incorrect an error results and the time is not updated.

The system time can only be set if a system clock exists in the system. If a clock is not present and the time is set, an error is returned.

*Examples of valid syntax:*

```
TIME = "170001" PASSWORD "SYSTEM"
TIME = T$ PASSWORD P$
```

The TIME function is an alphanumeric function that returns an 8 character ASCII string containing the current time in the form HHMMSSCC (hour, minute, second, centisecond). If there is no system clock, the TIME function returns the value 99999999 (i.e., invalid). The TIME function is used as an operand in alpha expressions.

*Example of valid syntax:*

```
A$ = TIME & " " & DATE
```

## VAL Function

*Format:*

```
                alpha-variable
    VAL(                                 [,length] )

                literal-string
 where:

   length = numeric-variable or the digit 1, 2, 3, or 4
```

VAL is a numeric function that uses an alphanumeric argument, but returns a numeric value; it is the inverse of the BIN function. The VAL (value) function converts the binary value in the alpha-variable or literal-string to a numeric value. The number of bytes in the binary value to be converted is specified by the digit; if no digit is included, the length is assumed to be one byte. A numeric-variable can now be used to specify the length of the binary value in the VAL function. The VAL function can be used wherever numeric functions are legal.

VAL is particularly useful in code conversion and table lookup operations since the converted number can be used as a subscript to retrieve the corresponding code from an array. Additionally, VAL can be used with the RESTORE statement to retrieve codes or data from DATA statements.

*Examples:*

```
:PRINT VAL(HEX(20))
 32
:A$=HEX(1234)
:PRINT VAL(A$,2)
 4660
```

*Examples of valid syntax:*

```
X = VAL(A$,L)
X=VAL(A$)
  PRINT VAL("A")
Y=VAL(B$,2)
  RESTORE VAL(STR(A$,I,1))+1
$=A$(VAL(C$)+1)
  IF VAL(X$,2) > 1024 THEN 100
```

## VER Function

*Format:*

```
                  alpha-variable
     VER(                                     , format-specification )
                  literal-string
  where:
                                                alpha-variable

  format-specification =                        literal-string
```

The VER (verify) function verifies 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 that 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 (refer to Table 5-4). 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.

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 that uses an alphanumeric argument but returns a numeric result. The VER function can be used wherever numeric functions are legal. Refer to the discussion of numeric functions in Chapter 4.

**Table 5-2. Format-Character Definitions**

| Format Character | Character Set |
|---|---|
| A | Alphabetic only (A to Z or a to z) |
| # | Numeric only (0 to 9) |
| N | Alphabetic or numeric (A to Z, a to z, or 0 to 9) |
| H | Hexadecimal (0 to 9 or A to F) |
| P | Packed decimal |
| + | Sign (plus (+), minus (-), or blank) |
| X | Any character |
| Other | Only the specified character |

*Examples:*

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

# 6 Decimal Arithmetic Operators

## Overview

This chapter contains general forms for alpha operators that 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. These include the following instructions:

| | |
|---|---|
| ADD | 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 | Performs binary subtraction on a pair of binary values with or without carry propagation between bytes. |

The binary operators can be particularly useful when manipulating binary counters. The packed decimal operators can be useful for extended-precision decimal addition and subtraction since operations1can be performed on packed decimal numbers of almost unlimited length. Packed decimal numbers can be packed two digits for each byte in an alpha variable, and the only effective limit to the size of an alpha variable is the amount of available memory.

## Decimal to Binary Conversion and Two's Complement Notation

In order to perform binary operations on numbers with BASIC-2, you must first convert decimal values to binary values. You can convert a nonnegative integer to binary form by using the following procedure (these examples assume that X is a numeric value):

*Example:* Conversion to a 1-Byte Binary Value (0 <= X < 256)

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

*Example:* Conversion to a 2–Byte Binary Value (0 <= X < 65536)

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

*Example:* Conversion to Binary When the Decimal Value Exceeds Four Bytes

```
10 DIM A$7
    .
    .
    .
90 REM % NUMERIC TO BINARY CONVERSION
100 FOR I = LEN(STR(A$)) -1 TO 1 STEP -4
110 Q = INT(X/4294967296)
120 STR(A$,I,4) = BIN(X-4294967296*Q,4)
130 X = Q
140 NEXT I
150 IF I > 1 THEN STR(A$,,1) = BIN(X,3)
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 assures you that the obtained result is signed correctly. The following example demonstrates how to obtain a two's complement representation of a binary number.

*Example:*

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

To differentiate between positive and negative binary values, you should ensure that the high-order bit of the leftmost byte is reserved1to indicate the sign of the number. The leftmost bit of each value should be either 0 for positive values or 1 for negative values. Results can be left in this form until they are converted back from binary format to decimal format.

To convert these values, you should first check the high-order (leftmost) bit to determine the sign of the value. If the value of the bit is 1, 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.

*Example:*

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(HEX(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 BINARY500 X = INT(X)
510 IF X >= 0 THEN 540
20 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 or BCD) arithmetic operators, you must first convert the value to the packed format. You can convert a nonnegative integer to packed decimal form by using the PACK statement.

*Example:*

```
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, you can determine the ten's complement of the BCD value in the following manner:

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

You differentiate between positive and negative BCD values in the same manner as for binary values. You 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. You can leave the results in this form until you need to convert back to decimal format.

To convert to decimal format, you 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.

*Example:*

The following program illustrates routines for signed and unsigned conversion of decimal to BCD and of 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(HEX(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
.
.
.
```

## 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 discussed in alphabetical order on the following pages. Other operators that take alphanumeric arguments are described in Chapter 5.

## ADD Operator

*Format:*

```
ADD [C]
```

The ADD operator performs binary addition on a pair of binary values. The binary value of the operand is added to the binary value of the receiver-variable, and the sum is assigned to the receiver-variable. You can use ADD only in the alpha expression portion of an alphanumeric assignment statement. (Refer to the discussion of alpha expressions and the alpha assignment statement in Chapter 5.)

Addition is performed on a character-by-character basis from right to left, starting with the low-order (rightmost) character of each field. ADD treats each byte of the operand and receiver-variable as a separate value, 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. The addition continues until ADD operates on the entire value in the receiver-variable.

*Example:*

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

ADD operates on the entire value in the receiver-variable, including trailing spaces. Similarly, if the operand is a variable, ADD uses the entire value of the variable, including trailing spaces. (Trailing spaces usually 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, the following statement operates only on the third and fourth bytes of A$:

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

If the C parameter follows ADD, the value of the operand is treated as a single binary number and added to the binary value of the receiver-variable with automatic carry propagation between characters.

*Example:*

```
:10 DIM A$2
:20 A$ = HEX(0123)
:30 A$ = ADDC HEX(00FF)
:40 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 characters), 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 that cannot be stored in the receiver-variable are truncated.

It is also possible to perform multiple ADDs on one statement. For example, the following statement adds Y$ to X$ three times:

```
10 X$ = ADDC Y$ ADDC Y$ ADDC Y$
```

*Examples of valid syntax:*

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

## DAC Operator

*Format:*

```
DAC
```

The DAC (decimal add with carry) operator 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. You can use DAC only in the alpha-expression portion of an alphanumeric-assignment statement. (Refer to Chapter 5.)

The DAC operator does *not* check the values of the operands for valid packed decimal format. (Any characters other than hex digits 0 to 9 are illegal in a packed decimal number.) Since decimal addition with nonpacked decimal values produces undefined results, use the VER function to verify that the values of the operands are expressed in valid packed decimal format. Decimal addition, like binary addition, is performed on a character-by-character basis from right to left, starting with the low-order (rightmost) characters of the receiver-variable and operand. Each value is treated as a single decimal number, with automatic carry propagation between characters.

DAC operates on the entire value of the receiver-variable, including trailing spaces. Similarly, if the operand is a variable, DAC uses the entire value of the variable, including trailing spaces. (Trailing spaces usually are not considered to be part of the value of an alphanumeric variable.)

*Example:*

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

*Examples of valid syntax:*

```
A$ = DAC HEX(0001)
B$ = A$ DAC F$
```

## DSC Operator

*Format:*

```
DSC
```

The DSC (decimal subtract with carry) operator 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 difference is assigned to the receiver-variable. The programmer can use DSC only in the alpha expression portion of an alphanumeric assignment statement. (Refer to the discussion of alpha expressions and the alpha assignment statement in Chapter 5.)

The DSC operator does *not* check the values of the operands for valid packed decimal format. (Any characters other than hex digits 0 to 9 are illegal in a packed decimal number.) Since decimal subtraction with nonpacked decimal values produces undefined results, the programmer should use the VER function to verify that the values of the operands are expressed in valid packed decimal format.

Decimal subtraction, like binary subtraction (SUB), is performed on a character-by-character basis from right to left, starting with the low-order (rightmost) characters of the receiver-variable and the operand. Each value is treated as a single decimal number, with automatic carry propagation between characters.

DSC operates on the entire value of the receiver-variable, including trailing spaces. Similarly, if the operand is a variable, DSC uses the entire value of the variable, including trailing spaces. (Trailing spaces usually are not considered to be part of the value of an alphanumeric variable.)

*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 "RESULT = "; HEXOF(C$)
:RUN
 RESULT = 0025
```

The answer, 25, occupies the low-order character of the receiver-variable, while the leading zeros indicate that the answer is positive. Suppose, however, that Line 130 is changed to the following statement:

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

If the program is run again, the result is the ten's complement notation for -25.

```
:RUN
 RESULT = 9975
```

*Examples of valid syntax:*

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

## SUB Operator

*Format:*

```
SUB [C]
```

The SUB operator performs 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. You can use SUB only in the alpha-expression portion of an alphanumeric-assignment statement. (Refer to the discussion of alpha-numeric and the alpha-numeric statement in Chapter 5.)

Subtraction is performed on a character-by-character basis from right to left, starting with the low-order (rightmost) character of each field. SUB treats each byte of the operand and receiver-variable as a separate value, with no carry propagation between characters. The last (rightmost) byte of the value of the operand is subtracted from the last (rightmost) byte of the receiver-variable. The next-to-last byte of the operand is then subtracted from the next-to-last byte of the receiver. The subtraction continues until SUB operates on the entire value in the receiver-variable.

*Example:*

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

The entire value in the receiver-variable, including trailing spaces, is operated upon. Similarly, if the operand is a variable, SUB uses the entire value of the variable, including trailing spaces. (Trailing spaces usually 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, the following statement operates only on the third and fourth bytes of A$:

```
STR(A$,3,2) = SUB B$
```

If the C parameter follows SUB, the value of the operand is treated as a single binary number and subtracted from the binary value of the receiver-variable with automatic carry propagation between characters.

*Example:*

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

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

1. **The subtraction 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 that 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 "RESULT = "; HEXOF(C$)
:RUN
 RESULT = 04
```

**In this case, the answer is hex 4 (binary 0100), and the leading hex digit, 0, indicates that it is positive. Changing Line 40 results in the two's complement form of -4.**

```
:40 C$ = B$ SUBC A$
:RUN
 RESULT = FC
```

*Examples of valid syntax:*

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

7

# The Select Statement

## Overview

The SELECT statement is used to direct I/O to specified devices and to select various modes of operation. SELECT is used to select device-addresses for I/O operations, to specify various options for mathematical operations, and to select certain parameters for output devices.

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

- 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 contents of the Device Table 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 form of the LIST command used to list the contents of the Device Table.
- The LIST SELECT command, a form of the LIST command used to display the options currently SELECTED, including SELECT T (date/time) and SELECT H (platter hogging).

- The SELECT function, an alphanumeric function that returns device addresses set in the Device Table.

## Math Mode Selection

This section discusses the SELECT D, SELECT G, SELECT R, SELECT NO ROUND, SELECT ROUND, and SELECT ERROR statements.

### Specifying Degrees, Radians, or Grads

The trigonometric functions can be calculated in one of three modes: degree, radian, or grads (360 degrees = 400 grads). Trigonometric functions are evaluated in radians unless the system is explicitly instructed to use degrees or grads. If degrees or grads are required, they must be specified with the following SELECT statements prior to performing trigonometric calculations:

SELECT D – Use degrees in all subsequent trigonometric calculation.

SELECT G – Use grads in all subsequent trigonometric calculation.

SELECT R – Use radian measure.

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

### Selecting Rounding or Truncation

Results of all arithmetic operations (+, -, *, /, ↑ ) and the square root (SQR) and modulo (MOD) functions are rounded to 13 significant digits. Results can also be truncated to 13 digits by executing a SELECT NO ROUND statement.

Once a SELECT NO ROUND statement is executed, all results are truncated to 13 digits. Rounding can be restored subsequently with a SELECT ROUND statement. Rounding is selected automatically when the system is Master Initialized.

### 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, the system displays an error message and terminates program execution. However, if a statement of the form

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

is executed, the system does not display an error message. Instead, program execution continues with default values 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 an error message and program termination.

The SELECT ERROR statement is further discussed in Chapter 9.

### Default Math Modes

Upon Master Initialization or after a CLEAR command is executed, the following modes of operation are automatically selected:

- Radian measure for all trigonometric functions.
- Rounding to 13 significant digits for all math operations.
- 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.)

## Output Parameter Specification

The following three special output parameters can be specified in a SELECT statement to control the operation of one or more output devices, such as printers and the screen:

- 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 in Program mode or Immediate mode. A pause is used to slow down the rate at which output lines are displayed on the screen, enabling you to read the output easily. The SELECT P statement can also be used during a TRACE operation, allowing you to read debugging information more easily.

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 = approximately 1/6 second
SELECT P6 - Pause = 1 second
SELECT P0 - Pause = null   (SELECT P0 and SELECT P are
                equivalent statements.)
```

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:

- Executing a SELECT P or SELECT P0 statement
- Master Initialization of the system
- Pressing the RESET key
- Executing 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 screen or printer for Console Output (CO) and LIST operations. The LINE parameter has *no* effect on output produced by a PRINT or PRINTUSING statement executed in Program mode. In addition, the SELECT LINE statement can be used to restrict cursor movement to a specified range of lines during INPUT operations.

Upon Master Initialization, the LINE parameter defaults to 24. The LINE parameter is also reset to the default value following the execution of a CLEAR, LOAD RUN, or RESET command.

## Selecting the Line Width

The maximum width of an output line for PRINT and LIST operations can be specified with the optional width select-parameter. Normally, the system 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.

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.

Most printers automatically generate carriage returns when their maximum line widths are exceeded, irrespective of the selected width. In general, the printer automatically continues printing an overlong output line on the next printer line. Since some printers may not follow this procedure, widths that 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. 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.

You can have different line widths selected for CO, PRINT, and LIST operations concurrently. When this situation occurs, a combination of operations in different output classes may produce unpredictable results, particularly on the screen.

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

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

causes the system to continue displaying PRINT output on the same screen 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 0 is selected. This feature is useful for moving the cursor on the screen without altering the column count. (Note, however, that the TAB and AT functions *cannot* be used when a line width of zero is selected; refer to the discussion of the PRINT statement in Chapter 11.)

Upon Master Initialization, the line width for all output classes (CO, PRINT, and LIST) is automatically set 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 values. CLEAR selects PRINT and LIST to be set to the current CO address and INPUT to be set to the current CI address (Address /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 (refer to Table 7.1).

## I/O Device Selection

A principal function of the SELECT statement is to assign the device-addresses of specified 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 the following 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 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 $GIO statements. |
| 8. *DISK* | Input and output for all disk operations. |

## 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 identifies the device-type. The next two hex digits constitute the unit device-address. A device-address should always be preceded by a slash ( / ) when specified. For example, the device-address of the Primary screen is /005; this address is broken down by the system as follows:

```
          0               05

    device-type    unit device-address
```

The system uses the device-type 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. Devices are explained in greater detail in the section entitled "Device Types".

BASIC-2 requires that the addresses of all devices to be used (other than the keyboard (/001), screen (/005), the terminal printer (/004), and null output (/000)) be declared in the Master Device Table when the system is configured with the @GENPART utility. Attempting to access an undefined device results in an error. Refer to Chapter 16 for more information on the Master Device Table.

## 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 to access. There are three ways to select a specific device-address for a particular I/O operation.

- *Default Addresses (Primary I/O Devices)* – The system automatically provides default device-addresses for each class of I/O operations (refer to Table 7-1). 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 system uses the default address for operations in that class.
- *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. 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 section entitled "Explicit Device Table Modification".
- *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

  ```
  SAVE T/D20, "PROG1"
  ```

  saves program PROG1 on the disk image at address /D20, 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. Upon Master Initialization, the default device-addresses are automatically selected for I/O operations. The primary I/O devices and their associated default device-addresses appear in Table 7-1.

**Table 7-1. Default Addresses for Primary I/O Devices**

| I/O Operation Class | Primary I/O Device | Default Address |
|---|---|---|
| Console Input (CI) | Keyboard | /001 |
| INPUT | Keyboard | /001 |
| Console Output (CO) | Screen | /005 |
| PRINT | Screen | /005 |
| LIST | Screen | /005 |
| PLOT | Screen | /005 |
| TAPE | Null Output | /000 |
| DISK | Disk Drive | /D10 |

If the system contains no additional I/O device beyond the primary devices (e.g., if there is only one disk drive), you never have to select device-addresses or specify them explicitly in a DISK or TAPE statement. The system automatically uses the default addresses and accesses the appropriate device for each I/O class. For example, disk operations use the DISK default address, /D10, and access the Primary Disk. If additional devices such as a second disk drive or a plotter are added in any I/O class, device-address selection or (where possible) direct specification is required to identify which device is to be accessed.

When the system enters BASIC-2, 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 operation.

## 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.* A separate Device Table is maintained for each partition in the system. For example, the statement

```
    100 PRINT "ABCD"
```

causes 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.

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 format of the information stored in the Device Table is illustrated in Figure 7-1. When you execute a LIST DT command, a screen with a format similar to the one shown in Figure 7-1 appears.

```
CI      /001                                CO      /005       WIDTH 80
INPUT   /001                                PRINT   /215       WIDTH 132
PLOT    /413                                LIST    /005       WIDTH 80
TAPE    /000
#0      /D11                                #8      /000
#1      /B20  at 530 in 500 to 732          #9      /000
#2      /D12  at 1234 in 1234 to 1945       #10     /000
#3      /320  at 12222 in 11946 to 12223    #11     /000
#4      /000                                #12     /000
#5      /000                                #13     /000
#6      /000                                #14     /000
#7      /000                                #15     /000
MDT:    /004      /215-120    /310      /320-03x

PDT:    /015  @PMO10VO ON      /016 @PMO120VO OFF
```

**Figure 7-1. Sample Device Table Screen**

Upon Master Initialization, default addresses are placed in each I/O slot in the Device Table. Additionally, default line widths are placed in the CO, PRINT, and LIST slots. The line width of the Primary screen, address /005, is used as the default width. The remainder of the Device Table is set to zeros.

Operations in each I/O class refer to the corresponding slot for that class in the Device Table. 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.*

## 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 of the system
- Executing a CLEAR and LOAD RUN command
- Pressing the RESET key

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

## Explicit Device Table Modification

Device Table entries for all I/O classes can be modified explicitly by executing a SELECT statement that specifies the I/O class or classes whose entries are to be modified and the new values for those entries. However, 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.

> ***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.*

### The Console Input Select-Parameter

The Console Input (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. Characters entered during a Console Input operation are automatically echoed to the currently selected Console Output device. BASIC-2 only supports address /001 for CI operations.

### The INPUT Select-Parameter

The INPUT select-parameter specifies the device-address to be used to enter data for INPUT, LINPUT, and KEYIN statements, as in the following 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 entered from the device selected for INPUT operations (Address /01C).

## The Console Output Select-Parameter

BASIC-2 uses the device-address /005 for Console Output operations. 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 operations
- System-generated error messages
- STOP and END messages
- Output from TRACE operations to be redirected to another device (e.g., a printer) through the CO select parameter

  For example, the statement

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

  selects the printer with address /215 for TRACE operations by placing address /215 in the CO slot in the Device Table. The maximum line width is set at 120 characters. Output from console operations, other than TRACE, cannot be redirected with BASIC-2.

## 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 can specify the maximum line width to be used for that device, as in the following example:

```
10 SELECT PRINT /215(100)
20 PRINT"X = "; X,"NAME = "; N$
30 PRINTUSING 40,V
40 %TOTAL VALUE RECEIVED:$#,###.##
```

The SELECT PRINT statement in Line 10 directs all printed output to a printer with device-address /215 by placing address /215 in the PRINT slot in the Device Table. The line width is specified as 100 characters. The screen can be reselected for programmed PRINT output with the following statement:

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

This statement reselects the screen (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 /C013 for PLOT output. The "C" device-type used in place of the standard "0" device-type specifies the use of special control procedures that utilize binary plot vectors. Binary plot vectors can substantially increase the overall speed of a plotting operation. 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.

## The TAPE Select-Parameter

The TAPE select-parameter specifies the device-address to be used by any TAPE-class instruction that does not specify a device-address or file-number in the instruction itself. The TAPE-class instructions are $GIO and $IF ON. For example, the statement

```
100 SELECT TAPE/005
```

selects the screen for $GIO operations.

The 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. The use of file-numbers to reference device-addresses for DISK and TAPE instructions is described in the section entitled "The File-Number Select-Parameter".

## The DISK Select-Parameter

The DISK select-parameter specifies the device-address to be used in any DISK instruction that does not directly specify a device-address or file-number. For example, the statement

```
SELECT DISK /D20
```

selects the disk drive with address /D20 as the Primary Disk. The disk drive is then used automatically by any DISK instruction that does not specify a device-address or file-number. The above SELECT statement places address /D20 in the DISK slot in the Device Table. Because the DISK slot is also identified as the #0 slot in the Device Table, SELECT DISK and SELECT #0 are equivalent.

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/D50
```

lists the Catalog Index of the disk whose device-address is /D50, *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."

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 (refer to Figure 7-1). The #0 slot, which is the DISK I/O slot, contains the default address for the Primary Disk 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 includes the following information:

- 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

However, 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/D20
```

places the device-address /D20 in the #3 slot in the Device Table. Subsequently, a DISK instruction that references file-number #3 uses address /D20 for its I/O operation. For example, the statement

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

opens the data file named "FILE" on the disk image whose address is /D20.

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 Line 10:

```
10 SELECT #2/D10, #3/D30
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 preceding example illustrates, you can specify multiple select-parameters in one SELECT statement. You can specify any combination of select-parameters 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/D50
```

performs all of the following operations:

- Selects degree measure for trigonometric functions
- Selects truncation rather than rounding of numeric results
- Selects device-address /215 for Console Output operations
- Assigns device-address /D50 to the #5 slot in the Device Table

## Implicit Device Table Modification

The SELECT statement is used to explicitly modify one or more slots in the Device Table. However, there are certain system operations that 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 LOAD RUN command is executed. Because each of these operations causes the Device Table to be modified in a different way, you should be familiar with the specific effects of each operation.

### Master Initialization

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

- 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.
- Sets the line widths in the CO, PRINT, and LIST slots to the line width of the primary screen (Address 005).
- Sets the remainder of the #0 slot and all items in the #1 - #15 slots to zeros.

### RESET

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

- The CO entry is set to the system default address for Console Output (/005). The line width is set to 80.
- The CI entry is set to the system default address for Console Input (/001).
- LINE and width are set to 24 and 80, respectively.

No other Device Table slots are affected by keying RESET.

## CLEAR and LOAD RUN

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:

- 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.
- The INPUT device-address entry is set to the current CI device-address.
- The #0 slot is zeroed out except for the DISK device-address, which is not altered.
- 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.

## 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, you 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. |

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 A | = | 2236MXE Asynchronous Communications |
| Type C | = | Vectored PLOT operations** |

* This device-type provides faster plotting than device-type 4.

** 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.

## Conditional Selection of Select-Parameters

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 you enter or a value the program computes 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 the section entitled "General Forms of the Select Statement".

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/D10; #1/B10; #1/D50
```

the numeric-variable I contains the select-value, while #1/D10, #1/B10, and #1/D50 are the select-lists. In this case, if I=1, then SELECT #1/D10 is executed; if I=2, then SELECT #1/B10 is executed; and if I=3, then SELECT #1/D50 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. In a standard SELECT statement, 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
```

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. You can also use an alphanumeric expression as a select-value; for a discussion of alpha select-values, refer to the discussion of ON/SELECT in the section entitled "General Forms of the Select Statement".

## General Forms of the Select Statements

This section contains the general forms of the SELECT, ON/SELECT, and LIST DT statements. The SELECT function is also described.

The LIST DT statement follows the general rules of the SELECT statement. For a detailed description of list control, refer to Chapter 10.

The LIST SELECT statement is also included in Chapter 10.

## LIST DT

*Format:*

```
LIST [title] DT

where:

     title = alpha-variable or literal-string
```

LIST DT lists the contents of the Device Table. The current device selections for the various I/O classes and file numbers are shown. The line width for character output devices is also included.

For data files that are open, the device type in the disk-address of the associated file number indicates whether the file was opened with the T, F, or R platter specification. Device type D indicates T, 3 indicates F, and B indicates R. For example, if a data file has been opened as follows:

```
SELECT #1/320: DATALOAD DC OPEN R #1, "PLAYERS"
```

then, LIST DT displays the selection as: #1 /B20. LIST DT also displays the current sector location within the open data file. For example, for the file above the display might be:

```
#1    /B20  at 530 in 500 to 732
```

This shows that the current location is sector 530 in the data file that begins at sector 500 and ends as sector 732.

LIST DT also lists the device entries in the Master Device Table (MDT). Each entry is displayed in the following format:

```
/taa[-ppx] where:  t = device type (3 if disk, otherwise 0)
                  aa = device address
                  pp = number of the partition using the device
                   x = X if device open exclusively for parti
                         tion pp,or O if open for partition pp
```

The Printer Device Table (PDT) is also listed by LIST DT. An entry for each available printer supported by the Generalized Printer Driver is displayed. Each entry includes the printer address, name of the printer table, and whether the printer table is enabled (ON) or disabled (OFF).

The title parameter and control of the list output is the same as for the other LIST statements. Refer to the LIST command in Chapter 10.

*Example:*

```
:LIST DT

CI     /001                                CO     /005   width 80
INPUT  /001                                PRINT  /215   width 132
PLOT   /413                                LIST   /005   width 80
TAPE  /000

#0     /D11                                 #2     /D10
#5    /B20  at 530 in 500 to 732            #6     /D1F
#7    /D12  at 1234 in 1234 to 1945         #10    /D31
#15   /320  at 12222 in 11946 to 12223      #128   /D5F

MDT:  /004    /215-120   /310       /320-03X

PDT:  /015   @PM010V0 ON    /016 @PM0120V0 OFF
```

## ON SELECT

*Format:*

```
    ON  select-value  SELECT  select-list [;[select-list]]...

  where:

                            numeric-expression
    select-value    =
                            alpha-variable

    select-list     =       select-parameter [,select-parameter] ...
```

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/D10; #0/B10; #0/D20
```

selects #0 to /D10 when X=1, to /B10 when X=2, and to /D20 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 screen (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 (i.e., the current device specification for PRINT remains in effect).

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/D10; DISK/D20; DISK/D30
```

would select DISK to address /D20.

*Examples of valid syntax:*

```
ON I SELECT PRINT/005; PRINT/215
ON A$ SELECT R;D;G
ON (I+1)/2 SELECT #0/D10; #0/D20; #0/D30
ON X SELECT CO/005,PRINT/005,#0/D10;PRINT/215, LIST/215,
#0/B10
```

## SELECT Statement

*Format:*

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

  where:
                                   D
                                   R
                                   G
                                  ERROR [ > error-code][NO] ROUNDP [digit]
                                  LINE numeric-expression
                                  CI device-address
                                  INPUT device-address
                                  CO device-address [(width)]
                                  PRINT device-address [(width)]
    select-parameter =
                                  LIST device-address [(width)]
                                  PLOT device-address
                                  TAPE device-address
                                  DISK device-address
                                  file-number device-address
                                  TC port-number
                                  TERMINAL port-number
                                  DRIVER device-address [OFF]
                                   T ON/OFF
                                   H ON/OFF
                                   NEW *
                                   OLD *

    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 hex digits representing the
                      device type and address
          width    =  an expression 0 < 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
```

Note: * CS/386 ONLY

The SELECT statement is used for the following purposes:

- To select the desired math modes for arithmetic operations; including type of measure for trigonometric functions, rounding or truncation of numeric results, and desired system response to specific math errors. (Refer to the section entitled "Math Mode Selection".)
- To select output parameters for communicating with output devices. (Refer to the section entitled "Output Parameter Specification".)
- To select device-addresses for accessing specified devices with input/output statements and commands. (Refer to Section 7.4.)
- To select a 2236MXE port for telecommunications. Refer to the Asynchronous Communications User Guide for Model 2236MXE Terminal Processor and Option-W Terminal Processor (700-8098) for a discussion of SELECT TC, SELECT TERMINAL, and telecommunications using the 2236MXE.
- To control printer drivers. Refer to the *BASIC-2 Utilities Reference Manual* for a discussion the use of SELECT DRIVER for controlling printer drivers.
- To Select the program saving format mode. The option OLD (Default) signifies format compatible with the current 2200 & VLSI CPUs. The NEW option is *only* compatible with the CS/386 CPU. It should be noted that the CS/386 takes less processing time to resolve the NEW file format. Because the new format takes more space to save a program line, old formats being resaved in the new format may fail with an A05 Error. To overcome this sitution, the program line must be broken into additional line numbers.
- To record DATE and TIME on all programs when using a SAVE or RESAVE command.
- To initiate disk platter hogging on a DS data storage unit with PROMs Revision 3.0 or greater. It allows you to hog only a platter, and not the complete storage unit, using the $OPEN command.

## SELECT Function

*Format:*

```
     SELECT select-parameter

  where:

     select-parameter = CI, INPUT, CO, PRINT, LIST, PLOT, TAPE,
                        DISK, or file#

               file# = #n, where n is an integer or numeric variable
                        with value between 0 and 15 inclusive.
```

The SELECT function is an alphanumeric function that returns the current device selection for the specified I/O class or file number. The value returned is 3 ASCII characters, the first being the device type and the next two being the device address. The SELECT function is used in alpha-expressions in alpha assignment statements (refer to Chapter 5).

*Example:*

```
SELECT #5/D10
A$ = SELECT #5
```

Sets A$ to "D10".

*Examples of valid syntax:*

```
A$=SELECT PRINT
D$()=SELECT #0 & SELECT #1 & SELECT #2
```

## LIST SELECT

*Format:*

```
LIST SELECT
```

The LIST SELECT Command generates a listing of the selected parameters. This allows the user the ability to view the options they have selected. The select statement command will produce a complete discripition of all the options available.

*Example:*

```
10 LIST SELECT
```

Result Shows

```
SELECT   R, ERROR>60, ROUND, P, LINE 24, NEW
SELECT   CI/001, INPUT/001, PLOT/413, TAPE/000
SELECT   PRINT/204(80), LIST/005(80), CO/005(80)
SELECT   DISK/D10
```

*Example of valid syntax:*

```
LIST SELECT
```

8

# Programmable Interrupts

## Overview

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. It subsequently returns control to the interruption point in the main program after the interrupt subroutine processing is finished. Because different devices can be assigned different priorities in the interrupt scheme, this type of interrupt is often called a priority interrupt.

Programmable interrupts are useful for real-time instrument control, as well as in applications where the activity of more than one program needs to be coordinated. However, most users will not use interrupt processing. The system controls all the low level device handling for common system devices such as printers, disk, and terminals.

## Interrupt Programming

The SELECT statement controls interrupts. The following list summarizes interrupt control features, as well as the statements that implement each feature.

- Defining interrupts and priorities
  SELECT ON interrupt-condition GOSUB line-number
  SELECT OFF interrupt-condition GOSUB line-number
- Enabling and disabling individual interrupts
  SELECT ON interrupt-condition
  SELECT OFF interrupt-condition
- Inhibiting and reactivating all currently enabled interrupts
  SELECT ON
  SELECT OFF
- Clearing all currently defined interrupt information for redefinition
  SELECT ON CLEAR

## Interrupt Processing

An I/O device signals an interrupt condition when the device ready/busy condition is ready. A BASIC-2 program can set an interrupt condition for another partition by executing a $ALERT statement. While normal program processing continues, the defined interrupts are constantly polled for a ready signal. When an interrupt condition occurs, the system saves the current BASIC-2 program location and transfers program control to the specified interrupt processing subroutine upon completion of current statement execution. The interrupt subroutine then performs the necessary processing. At the completion of subroutine processing, a RETURN statement sends the program control back to the statement immediately following the last statement executed before the interrupt occurred, and main processing resumes at that point.

*Example:*

```
100 SELECT ON ALERT GOSUB 5000
```

Executing line 100 activates the checking for ALERT interrupts during the execution of the remainder of the program. Control is transferred to the subroutine at line 5000 when the appropriate $ALERT statement is executed by another program. When subroutine 5000 completes, control is returned to the interrupted program.

A program is interrupted between the execution of BASIC-2 statements. The execution of a BASIC-2 statement cannot be interrupted. For example, if the system is awaiting operator entry in response to an INPUT request or is performing a disk operation, all interrupt activity is suspended until the statement completes execution. For this reason, a programmer must be careful when designing a program to handle a device that requires immediate responses. (For example, programmers could use KEYIN loops instead of INPUT if interrupts need to be fielded during keyboard entry.)

The interrupt subroutine itself cannot be interrupted; all interrupt processing is suspended until the interrupt subroutine executes a RETURN or RETURN CLEAR to the main program. (The interrupt routine can, however, call subroutines.) SELECT ON statements executed in an interrupt subroutine do not take effect until the interrupt subroutine executes a RETURN or RETURN CLEAR. In general, all interrupt control SELECT statements can be legally executed within an interrupt subroutine. Since interrupts cannot take effect until the interrupt subroutine executes a RETURN statement, interrupts made active are not effectively active until the RETURN occurs.

If more than one interrupt is enabled, the system checks each interrupt-condition according to the priority. Interrupt priority is established by the order in which interrupts are defined in the program. As a result, the device with the interrupt first defined has the highest priority. Executing a SELECT OFF statement and then reenabling the interrupt with another SELECT ON statement does not change the priority of the device.

The programmer changes interrupt priority by clearing all currently defined information from the system with a SELECT ON CLEAR statement and then redefining interrupts in the desired priority.

## Listing Interrupt Status

Interrupt processing for up to eight different devices is maintained in an internal system table called the Interrupt Table. The LIST I command lists the current contents of the Interrupt Table. Examination of the contents of the Interrupt Table is useful for program debugging. The general form of the LIST I command and the format of its output are discussed in the section entitled "General Forms of the Interrupt Control Statements".

## Definition and Enabling of Interrupts

The programmer can combine the ON/OFF parameters and all associated interrupt parameters with other select-parameters in a single SELECT statement, although such an intermixing of select-parameters is likely to make the program confusing. More significantly, a programmer can specify the ON/OFF parameters 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, determines which one of a number of specified select-parameters is selected.

*Example:*

The following example illustrates the use of ON SELECT to conditionally enable an interrupt for a user-specified device.

```
40 SELECT OFF/017 GOSUB 3000, OFF/018 GOSUB 4000, OFF/019
   GOSUB 5000
50 INPUT "ENTER INTERRUPT DEVICE (1, 2, OR 3)", N
60 ON N SELECT ON/117, ON/018, ON/019
```

Interrupts for all devices are initially defined at line 40; at this point all interrupts are disabled. Line 50 prompts the user to identify which device will be used by specifying a number from 1 to 3. This number represents the select-value and is assigned to the variable N. At line 60, the value of N determines which interrupt is enabled. If N=1, device /017 is enabled. If N=2, device /018 is enabled. If N=3, device /019 is enabled. If N has a value less than 1 or greater than 3, then no device is enabled.

The ON SELECT statement with interrupt parameters is useful in situations where an operator or the program itself must have the capability to enable or disable specified interrupts. The general ON SELECT statement is described in detail in Chapter 7.

## General Forms of the Interrupt Control Statements

The general forms of the $ALERT, SELECT ON/OFF, and SELECT ON CLEAR statements and the LIST I command are discussed in alphabetical order on the following pages.

## $ALERT

*Format:*

```
    $ALERT partition

  where:

    partition = numeric expression
```

The $ALERT statement generates an interrupt to the specified partition. In order for the interrupt to have any effect, the alerted partition must execute a SELECT ON ALERT GOSUB statement. The SELECT ON ALERT GOSUB statement defines that alert interrupts are to be fielded and indicates a subroutine to execute when an alert interrupt occurs.

When an alert interrupt is acknowledged, the programmer knows that at least one $ALERT statement has been executed by some partition since the last occurrence of a $ALERT interrupt or a LOAD, CLEAR, or RUN command. The programmer does not know which partition executed the $ALERT, or whether or not several $ALERTs have been executed since the last $ALERT interrupt was acknowledged.

The alert interrupt is intended to be an indication to the specified partition that some other communication area, such as a global variable or shared disk file, should be polled. Using $ALERT consumes much less CPU or disk I/O time than repeatedly checking disk files or global variables for the occurrence of an interrupt change.

Alert interrupts are defined and fielded according to the same rules as other programmable interrupts. Refer to the discussion of the SELECT ON/OFF statement later in this section.

*Example:*

```
500 $ALERT 5: REM alert partition 5
```

If partition 5 has enabled alert interrupts, the alert interrupt is fielded as soon as partition 5 comes to the end of processing a BASIC-2 statement that is not in the interrupt handling subroutine.

*Examples of valid syntax:*

```
$ALERT 5
$ALERT T(N)
```

## LIST I

*Format:*

```
     LIST [title] I

  where:

     title = alpha-variable or literal-string
```

The LIST I command lists the current contents of the Interrupt Table. LIST I follows the general rules of the LIST statement regarding the title and control of the list output (refer to Chapter 10).

The system stores the following information in the Interrupt Table for each interrupt.

- The condition of the general interrupt inhibit/reactivate
- The currently executing subroutine if the system is currently processing an interrupt
- Status information indicating whether the interrupt is currently active (ON) or inactive (OFF)
- The device address of the device that initiates the interrupt, or the word ALERT if an alert interrupt is defined
- A subroutine line number that specifies the starting point of the interrupt processing routine

The general format of the listing of the Interrupt Table is as follows.

| **Item in Listing** | **Meaning** |
|---|---|
| ON/OFF | Condition of general interrupt inhibit/reactivate |
| GOSUB (or blank) | GOSUB if currently in an interrupt subroutine |
| ON/OFF aa GOSUB xxxx<br>ON/OFF aa GOSUB xxxx<br>ON/OFF ALERT GOSUB xxxx<br>.<br>.<br>. | Parameters of each defined interrupt, including status (ON or OFF), device address (aa) or ALERT, and interrupt subroutine line number (xxxx). |

*Example:*

Consider the following program segment:

```
50 SELECT ON/017 GOSUB 150, OFF/018 GOSUB 250, ON ALERT GOSUB
   300
60 SELECT OFF
```

If a LIST I command is executed immediately after line 60 is executed, the system produces the following listing:

```
 :LIST I
  OFF
  ON   17 GOSUB 0150
  OFF  18 GOSUB 0250
  ON   ALERT GOSUB 0300
```

*Examples of valid syntax;*

```
LIST I
LIST "Interrupt Table" I
```

## SELECT ON/OFF

*Format:*

```
                  ON
    SELECT                    [interrupt-condition [GOSUB line-number]]
                  OFF

  where:
                               /device-address
    interrupt-condition =       file#
                               ALERT

    file# = #n

  where:

  n is an integer or numeric-variable with a value between 0 and 15.
```

The SELECT ON/OFF statement is a form of the general SELECT statement containing special parameters that define, redefine, enable, and disable interrupts. Interrupts cannot be defined in the Immediate mode.

A programmer can include the ON and OFF select-parameters and associated interrupt parameters with any other select-parameters in a SELECT statement. Additionally, they can be specified in an ON SELECT statement to permit conditional definition and enabling of interrupts. Refer to Chapter 7 for a discussion of the general SELECT statement and the ON SELECT statement.

### Defining Interrupts and Priorities

A programmer can define a maximum of eight interrupts. The following statements define interrupts.

SELECT ON  interrupt-condition GOSUB line-number

SELECT OFF interrupt-condition GOSUB line-numer

The interrupt-condition parameter can specify an interrupt for a particular device (device-address or file# specifying a device address) or for another partition (ALERT). The GOSUB parameter specifies the location of the interrupt processing subroutine.

Interrupt priority is established by the order in which the interrupts are defined in a program (i.e., the first interrupt defined has the highest priority, the second interrupt defined has the next highest priority, and so on).

*Example:*

Line 10 defines three interrupts for addresses /017, /018, and /019.

```
10 SELECT OFF/017 GOSUB 100, ON/018 GOSUB 200, OFF/019 GOSUB
300
```

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. Line 10 also defines the subroutine branch addresses associated with each interrupt (i.e., 100, 200, and 300). For example, if the device at address /018 receives a ready signal, program execution continues at line 200 following completion of the current statement.

The ON and OFF parameters define the initial status of an interrupt (either enabled or disabled).

*Example:*

The following statement defines two interrupts.

```
10 SELECT OFF/017 GOSUB 100, ON/018 GOSUB 200
```

The interrupt defined for device /017 is inactive. A ready signal at address /017 does not initiate an interrupt, although an interrupt is defined for that address. The second interrupt, defined for address /018, is active and a ready condition at address /018 initiates an interrupt.

Once an interrupt is defined, a second SELECT statement can be executed later in the program to change the interrupt subroutine address.

*Example:*

Line 10 defines the subroutine address for device /018 as 200. The programmer can change this subroutine address by executing another SELECT statement as in line 50.

```
10 SELECT OFF/017 GOSUB 100, ON/018 GOSUB 200
.
.
.
50 SELECT ON/018 GOSUB 400
```

An interrupt initiated at address /018 now causes a branch to line 400. Changing the subroutine address in this way does *not* alter the originally defined interrupt priority.

## Enabling and Disabling Individual Interrupts

A SELECT ON or SELECT OFF statement with only an interrupt-condition specified selectively enables or disables a previously defined interrupt.

*Example:*

Consider the following program segment.

```
10 SELECT OFF/017 GOSUB 200, ON ALERT GOSUB 500
.
.
.
50 SELECT ON/017
.
.
.
80 SELECT OFF ALERT
```

The interrupt defined for address /017 in line 10 initially is disabled; it is enabled at line 50 of the program. Similarly, the alert interrupt initially enabled in line 10, is disabled in line 80.

Individual interrupts are referenced by their associated interrupt-condition. Once enabled with a SELECT ON statement, an interrupt occurs whenever its associated device becomes ready. When a defined interrupt is enabled or reenabled at any point in a program, it automatically assumes its originally defined priority.

## Inhibiting and Reactivating all Currently Enabled Interrupts

A SELECT ON statement (with no interrupt-condition specified) reactivates all currently enabled interrupts. A SELECT OFF statement (with no interrupt-condition specified) inhibits all currently enabled interrupts.

The general form of interrupt inhibit is executed independently of individual interrupt inhibits (i.e., SELECT OFF interrupt-condition). In effect, when the general interrupt inhibit is removed by SELECT ON, each individual interrupt assumes its previously defined status (ON or OFF). Disabled interrupts are *not* enabled by SELECT ON.

*Examples of valid syntax:*

```
SELECT ON/017 GOSUB 100, ON/018 GOSUB 150, OFF/019 GOSUB 200
SELECT OFF/017, OFF/018, ON ALERT
SELECT ONSELECT ON ALERT GOSUB 5000
SELECT OFF
```

## SELECT ON CLEAR

*Format:*

```
SELECT ON CLEAR
```

The SELECT ON CLEAR statement clears information on all currently defined interrupts from the Interrupt Table and turns off all interrupt processing. The system does not process interrupts again until the execution of a SELECT ON/OFF GOSUB statement.

*Example:*

Following execution of line 85, the system clears all interrupts from the Interrupt Table. A programmer can now define a new set of interrupts with new priorities.

```
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
```

A programmer can include the ON CLEAR select-parameter with any other select-parameters in a SELECT statement. Refer to Chapter 7 for a discussion of the general SELECT statement.

*Examples of valid syntax:*

```
SELECT ON CLEAR
```

9

# Error Control Features

## Overview

BASIC-2 provides an extensive set of error detection features designed to automatically detect and report a wide range of error conditions. The system automatically scans program text for errors during program entry, resolution, and execution.

When the system encounters an error, it displays the erroneous line with an arrow pointing to the approximate position of the error. The error number and a descriptive error message are displayed on the next line. For example:

```
:DATALOAD DC #1, X
ERROR D80:  File Not Open
```

If the system discovers an error during text entry, it stores the erroneous line in memory. If the system encounters an error during program resolution or execution, it immediately terminates resolution or execution. The system stops error scanning when it encounters the first error. For example, if a line contains more than one error, the system detects and reports only the first error. Refer to Appendix B for a list of errors and recovery procedures.

Error codes are numbers preceded by a letter, which indicates the class of the error. Error classes are shown below.

| Letter Prefix | Error Class |
|---|---|
| A | Miscellaneous Errors |
| S | Syntax Errors |
| P | Program Errors |
| C | Computational Errors |
| X | Execution Errors |
| D | Disk Errors |
| I | I/O Errors |

## Error Recoverability

Miscellaneous Errors, Syntax Errors, and Program Errors cause execution of the program to terminate. These types of errors generally indicate incorrect syntax or program logic errors, and they must be corrected before the program can be run. Computational Errors, Execution Errors, Disk Errors, and I/O Errors typically occur during program execution and are called recoverable errors. P48 is also a recoverable error. You can respond to recoverable errors that occur during program execution without aborting the program or disrupting the display with an error message. The following instructions help to intercept and respond to errors:

| | |
|---|---|
| SELECT ERROR | Specifies which computational errors are processed by the program and which are handled with a system response. |
| ERR | Returns the code of the most recent error condition. |
| ERR$ | Returns the descriptive error message for the specified error code. |
| ERROR | Initiates special error processing when an error is detected in a BASIC-2 statement during execution phase. |

The general forms of the ERR and ERR$ functions and the ERROR and SELECT ERROR statements are discussed in alphabetical order on the following pages.

## ERR Function

*Format:*

```
ERR
```

The numeric function ERR returns the code of the most recent error condition. The ERR function does not discriminate between recoverable and non-recoverable errors; any error condition sets the value of ERR. Since ERR is assigned a new value whenever an error occurs, it always returns the code of the most recent error.

Whenever a program references the ERR function (e.g., X = ERR), it is automatically reset to zero. The ERR function is also reset to zero upon the execution of a RUN or CLEAR command.

You can use the ERR function with the SELECT ERROR statement to check for the occurrence of computational errors at the conclusion of a numeric processing routine.

*Example:*

The following program segment sets variable X equal to the error code returned by ERR. Line 250 concludes the numeric processing routine by examining the value of X.

```
100 SELECT ERROR > 69
         .
         .
         .
(numeric processing)

250 X = ERR: IF X <> 0 THEN PRINT "ERROR = ";X
260 END
```

If X does not equal zero (X <> 0), an error occurred. The system then displays an error code to indicate a problem. If X equals zero, then an error did not occur and the program ends.

You can also use the ERR function with the ERROR statement to identify a specific error.

*Example:*

In the following line, the ERROR statement checks for an error following execution of the LOAD statement.

```
10 LOAD "games": ERROR X = ERR: GOSUB'90
```

If an error occurs, X is set equal to the error code, and the program branches to DEFFN'90 to process the error. If an error does not occur, execution continues at the next program line.

*Examples of valid syntax:*

```
X = ERR: PRINT X
LOAD A$: ERROR X = ERR: GOSUB 250
X = ERR: IF X <> 0 THEN 75
```

## ERR$ Function

*Format:*

```
      ERR$( error-code )

  where:

      error-code = expression, 0 <= value <= 99
```

ERR$ is an alphanumeric function that returns the descriptive error message corresponding to the specified error code. For example,

```
:DIM A$80
:A$ = ERR$(62)
:PRINT A$
 Division by Zero
```

A$ is set to the descriptive error message for Error 62. The function takes the numeric portion of the error code as its argument. ERR$ can only be used in the alpha-expression portion of an alphanumeric assignment statement (refer to Chapter 5). ERR$, in conjunction with the ERR function, is useful for displaying error messages in user error recovery routines.

*Examples of valid syntax:*

```
A$=ERR$(ERR)
STR(E$,I)="ERROR: " & ERR$(E)
```

## ERROR

*Format:*

```
                         statement [:statement] ...
        ERROR
                          do-group
```

The ERROR statement provides a means of responding to execution errors under program control. By using the ERROR statement, a programmer can recover from any recoverable error that occurs during program execution. ERROR cannot be used in Immediate mode.

If a recoverable error occurs in a BASIC-2 statement that is immediately followed by an ERROR statement, the normal error response (i.e., program termination and error message) is suppressed and program execution continues at the statement immediately following the word ERROR. If ERROR is followed by a do-group, the statements within the do-group are the error recovery sequence. If ERROR is followed by a statement other than DO, the entire rest of the line is considered to be the error recovery sequence. The error recovery sequence is only executed if the previous statement causes a recoverable error. If an error does not occur, execution continues after the do-group or at the next program line if ERROR is not followed by a do-group. (Refer to Chapter 10 for more information on do-groups.)

*Example:*

```
100 DATALOAD DC OPEN T#1, F$: ERROR E=ERR: GOSUB'50(E)
110 GOSUB 600
```

If an error occurs during the execution of the DATALOAD OPEN statement, execution continues with the statement following ERROR. Subroutine '50 is the error handling routine. Otherwise, execution continues at the next program line (i.e., line 110).

The ERROR statement cannot intercept nonrecoverable errors, nor can it intercept computational errors whose normal system response is suppressed with a SELECT ERROR statement. For example, if the statement SELECT ERROR > 65 has been executed, ERROR cannot intercept computational errors 60 to 65. (See SELECT ERROR.)

Whenever an error occurs, the ERR function is set to the error code corresponding to that error. ERR can be checked following an ERROR statement to identify the error that has occurred.

*Examples of valid syntax:*

```
INPUT "Coordinates",X,Y: ERROR PRINT "Illegal value,
re-enter":
READ X: ERROR DO: PRINT "EOF": ENDDO: GOSUB 500
```

## SELECT ERROR

*Format:*

```
    SELECT ERROR        [ > error code]

  where:

    error code  =  any computational error code (60 to 69)
```

The SELECT ERROR statement suppresses the normal system response to specified computational errors; it has no effect on errors outside the range 60 to 69. Computational errors are those produced by the math package while performing an arithmetic operation or evaluating a function. Usually the system responds to a computational error (other than underflow) by terminating program execution and displaying an error message. Underflow usually does not terminate program execution.

The SELECT ERROR statement produces the following results:

- A SELECT ERROR statement followed by no error code (e.g., SELECT ERROR) terminates program execution with an error message for all computational errors (including underflow).
- A SELECT ERROR statement followed by an error code (e.g., SELECT ERROR > 62), suppresses the system error message for all computational errors whose error codes are *less than or equal to* the specified error code. Program execution then continues with the values shown in Table 9-1. The system terminates program execution and displays an error message for computational errors whose error codes are *greater than* the specified error code.

*Example:*

The following statement suppresses normal error response for errors 60 to 65, while normal system error processing remains in effect for errors 66 to 69.

```
    SELECT ERROR > 65
```

Typically, the ERR function is used at the conclusion of a numeric processing routine to determine whether or not an error occurred during processing. If an error condition is indicated by ERR, you can design a special error recovery routine in the program. (Refer to the discussion of the ERR function in this section.) An ERROR statement cannot handle computational errors suppressed with a SELECT ERROR statement.

Upon Master Initialization or when a CLEAR or LOAD RUN command is executed, normal system error response is selected for all computational errors *except underflow*. Master initializing the system or executing a CLEAR or LOAD RUN command has the same effect as executing a SELECT ERROR > 60 statement.

The ERROR select-parameter and associated error code parameters can appear with any other select-parameters in a SELECT statement. Refer to Chapter 7 for a discussion of the general SELECT statement.

**Table 9-1. SELECT ERROR Return Values**

| Error Code | Error Condition | Value Returned |
|---|---|---|
| 60 | Underflow | 0 |
| 61 | Overflow | ±9.999999999999E+99 |
| 62 | Division by Zero | ±9.999999999999E+99 |
| 63 | Zero / or ↑ Zero | 0 |
| 64 | Zero Raised to Negative Power | +9.999999999999E+99 |
| 65 | Negative Number Raised to Noninteger Power | ABS(X) ↑ Y |
| 66 | Square Root of Negative Value | SQR(ABS(X)) |
| 67 | LOG of Zero | -9.999999999999E+99 |
| 68 | LOG of Negative Value | LOG(ABS(X)) |
| 69 | Argument Too Large | 0 |

*Examples of valid syntax:*

```
SELECT ERROR
SELECT ERROR > 69
SELECT ERROR > 63
```

10

# System Commands

## Overview

System commands enable you to control system operations directly from the keyboard. You can enter and execute a command whenever the colon prompt appears. Special commands enable you to retrieve, examine, modify, document, and save a program. Table 10-1 summarizes the available system commands. Most of these commands are discussed in this chapter. Chapter references are shown for commands described elsewhere. Also, most BASIC-2 statements can be used in Immediate mode as commands; these statements are discussed in the corresponding chapters.

**Table 10-1. BASIC-2 System Commands**

| Command | Function | Chapter |
|---|---|---|
| CLEAR | Clears from memory all or a specified portion of program text and/or variables. | 10 |
| CONTINUE | Continues program execution after a program is halted by a STOP statement or HALT command. | 10 |
| Function key | Can be defined to perform various functions, such as beginning program execution or accessing subroutines. | 10 |
| HALT | Temporarily halts program execution; normal execution can be resumed by pressing CONTINUE, or you can step through the program one statement at a time by repeatedly executing a HALT command. | 10 |
| LIST | Lists a program on the screen or a printer. | 10 |

(continued)

**Table 10–1. BASIC-2 System Commands (continued)**

| Command | Function | Chapter |
|---|---|---|
| LIST COM | Lists the common variables currently defined. | 10 |
| LIST DIM | Lists the noncommon variables currently defined. | 10 |
| LIST DC | Lists the contents of the specified disk platter. | 12 |
| LIST DT | Lists the contents of the Device Table. | 7 |
| LIST I | Lists the contents of the Interrupt Table. | 8 |
| LIST SELECT | Lists the options currently selected. | 10 |
| LIST T | Provides a cross-reference listing of one, some, or all program lines containing a specified character string. | 10 |
| LIST V | Provides a cross-reference listing of some or all variables defined in the program. | 10 |
| LIST # | Provides a cross-reference listing of some or all line numbers referenced in the program. | 10 |
| LIST ' | Provides a cross-reference listing of one or all marked subroutines referenced in the program. | 10 |
| LOAD | Loads a user program from disk into memory. | 12 |
| LOAD RUN | Loads a program from disk into memory and automatically runs the program. | 12 |
| RENAME | Changes the name of a disk file. | 12 |
| RENUMBER | Renumbers all or some of the program with the specified starting line number and increment. | 10 |
| RESAVE | Resaves a program on disk. | 12 |
| RESET | Terminates program execution, clears the screen, resets all I/O devices, and returns control to the keyboard. | 10 |
| RUN | Begins execution of the program in memory. | 10 |
| SAVE | Saves a program on disk. | 12 |
| STOP | Specifies the line number at which the program is to stop. | 10 |
| TRACE | Enables you to trace through program execution. | 10 |

## General Forms of the System Commands

General forms of the BASIC-2 system commands are presented in alphabetical order on the following pages.

## CLEAR

*Format:*

```
              P [starting-line-number] [, [ending-line-number]]
CLEAR         V
              N
```

The CLEAR command removes program text and variables from memory. The CLEAR command without any parameters performs the following operations:

- Removes all program text and variable areas from memory
- Closes all devices currently open
- Turns off Pause and Trace modes
- Selects the current Console Output (CO) device for PRINT, PRINTUSING, and LIST operations
- Selects the current Console Input (CI) device for KEYIN, INPUT, and LINPUT operations
- Clears the #0 slot in the Device Table except for the device-address, which is not altered
- Clears all items in the #1 to #15 slots in the Device Table
- Clears the screen, displays the READY message, and returns control to the keyboard

  CLEAR P removes program text from memory without disturbing any variables. CLEAR P does not alter the Device Table. The CLEAR P command has several forms.

- CLEAR P with no line numbers (e.g., CLEAR P) deletes all user program text from memory.
- CLEAR P followed by only a starting-line-number (e.g., CLEAR P 10) deletes from memory all lines from the specified line to the highest numbered line.
- CLEAR P followed by a comma and an ending-line-number (e.g., CLEAR P,500) deletes from memory all lines from the lowest numbered line up to and including the specified line.
- CLEAR P followed by a starting-line-number, a comma, and an ending-line-number (e.g., CLEAR P 40, 90) deletes from memory the two specified lines and all intervening lines.

  CLEAR V removes all variables from memory but does not clear the program text area or subroutine stacks. CLEAR V also does not alter the Device Table.

  CLEAR N removes all noncommon variables from memory. The names and values of common variables are not changed and the program text area and subroutine stacks are not affected. CLEAR N does not alter the Device Table.

*Examples of valid syntax:*

```
CLEAR
CLEAR P
CLEAR P 10
CLEAR P 200,
CLEAR P,500
CLEAR P 10,20
CLEAR V
CLEAR N
```

## CONTINUE

*Format:*

```
CONTINUE
```

The CONTINUE command resumes program execution after execution is halted by a HALT command or a STOP statement. Program execution continues at the program statement immediately following the last executed statement. If multiple statements with CONTINUE appear in an Immediate mode line, CONTINUE must be the *last* command on the line.

The CONTINUE command can be entered by typing each letter individually or by pressing the CONTINUE key.

Program execution cannot be continued after any of the conditions in the following list:

- Occurrence of an error
- Modification of program variables by the execution of a CLEAR V or CLEAR N command or by defining a new variable in Immediate mode
- Modification of program text by entry of a new program line by execution of a CLEAR, CLEAR P, or RENUMBER command, or by the modification of an existing line.
- Execution of a RESET command

*Example of valid syntax:*

```
CONTINUE
```

## Function Keys

You can define function keys to perform functions such as initiating program execution or accessing subroutines. To perform a specific task, the Function key must be defined with a DEFFN' statement. Refer to the discussion of the DEFFN' statement in Chapter 11.

When a function key is pressed 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, execution of the subroutine is initiated automatically. If the DEFFN' statement defines a character string for text entry, the defined string is displayed on the CRT. If there is no corresponding DEFFN' statement for the pressed Function key, the terminal alarm beeps and no action is taken.

Although a Function key that accesses a DEFFN' subroutine initiates program execution, it does not cause program resolution. Since an unresolved program typically does not execute successfully, every program should be resolved with a RUN command prior to executing it with a function key.

Sixteen (physical) Function keys are available on the keyboard. In conjunction with the SHIFT key, these keys provide access from the keyboard to 32 program subroutines or text entry definitions. (Refer to the appropriate terminal manual for a more complete description of available keys.)

The KEYIN statement can be used within a program to distinguish between Function and other keystrokes. Refer to Chapter 11 for a discussion of the KEYIN statement.

## HALT

The HALT key has two functions. When pressed once, program execution or listing stops. Pressed repeatedly, HALT steps through program execution statement by statement.

When HALT interrupts program listing, the system terminates the listing operation after the currently listing line. You cannot continue listing from the point of interruption.

When HALT interrupts program execution, the system stops execution after completing the currently executing statement. You can resume program execution at the point of interruption by executing a CONTINUE command.

Once the HALT command or a STOP statement has halted program execution, you can again use HALT to display and execute the next program statement and again halt program execution. Continued use of HALT repeats this procedure. When stepping through execution of a multiple statement line, HALT displays only those statements in the line that have not been executed. A colon indicates the position of each executed statement in a multiple statement line. When stepping through program execution is no longer necessary, the CONTINUE command resumes program execution.

If a program has been resolved with a RUN command, you can use a GOTO statement in Immediate mode to begin stepping execution at a particular line number. Refer to the discussion of the GOTO statement in Chapter 11.

You can also use HALT to step through a program in Trace mode and examine the values of variables. Refer to the discussion of the TRACE command later in this chapter.

You cannot use HALT to step through program execution after any of the conditions in the following list:

- Occurrence of an error
- Modification of program variables by the execution of a CLEAR V or CLEAR N command or by defining a new variable in Immediate mode
- Modification of program text by the execution of a CLEAR, CLEAR P, or RENUMBER command by the entry of a new program line or by the modification of an existing line
- Execution of a RESET command

*Example:*

The following example assumes that you ran the program, perhaps with a STOP statement prior to Line 100:

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

You can use HALT to step through the program starting at Line 120. TRACE is turned on so that variables receiving new values are displayed. The instructions you must follow to step through the program in Trace mode appear in the left column. The BASIC-2 statements that must be input and the displays that appear on the screen when HALT is pressed are presented in the right column.

| Programmer Action | CRT Display Following Action |
|---|---|
| Turn Trace mode on. | :TRACE |
| Start stepping at Line 120. | :GOTO 120 |
| Press HALT | :<br>120 X = 1.2: Y = 5*Z+X: GOTO 30<br>120 X <- 1.2 |
| Press HALT | :<br>120: Y = 5*Z+X: GOTO 30<br>120: Y <- 21.6 |
| Press HALT | :<br>120:: GOTO 30<br>120:: TRANSFER TO 30<br>: |

## LIST Command

The LIST command enables you to list and cross-reference a program and then examine the contents of certain system tables used by the program. Because each form of the LIST command performs a different function, each is treated as a separate command with its own format. The following list describes common features of the LIST commands:

- Upon Master Initialization, the screen (Address /005) is initially selected for LIST operations. The SELECT LIST statement can select other devices for listing operations. Refer to the discussion of the SELECT statement in Chapter 7. Execution of a CLEAR or LOAD RUN command automatically reselects the current Console Output device for listing operations.
- If LIST output is directed to the screen, program listing stops when the screen is full. To continue listing, press the RETURN key.
- Pressing HALT during program listing stops the listing after the current line. The listing cannot be continued from that point.
- The optional title parameter provides a convenient means of identifying a program listing. If a programmer specifies a literal string as the title in a LIST command, the system issues a top-of-form command to the currently selected output device which prints the highlighted title, a blank line, and the program listing.
- The LIST command is programmable (i.e., you can use the LIST command as a statement within a BASIC program).

## LIST

*Format:*

```
    LIST [title] [D] [starting-line-number] [, [ending-line-number]]

  where:

    title = alpha-variable or literal-string
```

The LIST command displays program text in memory in line number sequence. You can combine the parameters of the LIST command to produce the following results.

- LIST with no line numbers (e.g., LIST) lists the entire program.
- LIST followed by only a starting-line-number (e.g., LIST 360) lists the specified line.
- LIST followed by a starting-line-number and a comma (e.g., LIST 10,) lists all lines from the specified line to the highest numbered line.
- LIST followed by a comma and an ending-line-number (e.g., LIST ,20) lists all lines from the lowest numbered line up to and including the specified line.
- LIST followed by a starting-line-number, a comma, and an ending-line-number (e.g., LIST 60, 150) lists the specified lines and all intervening lines.

If the D parameter is included in the LIST command, the system displays multiple-statement lines in decompressed form (i.e., one statement on each CRT or printer line). Each line number is output as a 4-digit number and, if necessary, padded with leading zeros (e.g., 0010). Leading zeros permit uniform alignment of listed lines. Additionally, LIST D marks all program lines that are explicitly referenced by a line number or program label. A minus sign ( - ) is displayed before the line number of any referenced line.

You can use the LIST D command with the REM statement to list program titles and subtitles on a separate line in highlighted print. Refer to Chapter 11 for a discussion of the REM statement.

The END, GOTO, LOAD, and RETURN instructions also cause the system to skip a line when listing programs under LIST D.

**The following examples illustrate the various ways in which a programmer can use the LIST command to list all or portions of the program (these examples assume that the following text resides in memory).**

```
10 REM PROGRAM CONVERTS INCHES TO CENTIMETERS
20 REM % Perform Conversion
30 INPUT "Number of inch(es) to be converted", I
40 C = I * 2.54: PRINT HEX(09)
50 REM % Display Results: PRINT I;"inch(es) =";C;"centime
   ter(s)"
60 END
```

*Example:* **Listing a single line**

```
:LIST 30
 0030 INPUT "Number of inch(es) to be converted", I
```

*Example:* **Listing from Line 50 to the highest numbered line**

```
:LIST 50,
 50 REM % Display Results: PRINT I;"inch(es) =";C;"centime
  ter(s)"
 60 END
```

*Example:* **Listing from the lowest numbered line to Line 30**

```
 :LIST,30
10 REM PROGRAM CONVERTS INCHES TO CENTIMETERS
20 REM % Perform Conversion
30 INPUT "Number of inch(es) to be converted", I
```

*Example:* **Listing Lines 20 through 40, inclusive**

```
 :LIST 20,40
20 REM % Perform Conversion
30 INPUT "Number of inch(es) to be converted", I
40 C = I * 2.54: PRINT HEX(09)
```

*Example:* **Decompressed listing of Line 40**

```
:LIST D 40
 0040 C = I * 2.54
:PRINT HEX(09)
```

*Example:* **Decompressed listing of the entire program**

```
:LIST D
 0010 REM PROGRAM CONVERTS INCHES TO CENTIMETERS
 0020 REM %
```

**Perform Conversion**

```
0030 INPUT "Number of inch(es) to be converted", I
0040 C = I * 2.54
:    PRINT HEX(09)
0050 REM %
```

**Display Results**

```
: PRINT I;"inch(es) =";C;"centimeter(s)"
0060 END
```

*Examples of valid syntax:*

```
LIST D
LIST D 100, 500
LIST 90,
LIST D ,200
LIST 10
LIST HEX(03) D
LIST "TITLE" D
LIST T$ D 500,999
```

## LIST COM/DIM

*Format:*

```
                    COM
    LIST [title]
                    DIM
                    COM/DIM
  where:

    title = alpha-variable or literal-string
```

LIST COM/DIM statement lists the currently defined variables and their current values. LIST COM lists the defined common variables. LIST DIM lists the defined noncommon variables. The dimensions of arrays and the length of alpha variables are shown as would appear in a DIM or COM statement.

Values of alpha variables are displayed in both ASCII and hex notation. Nonprintable characters (i.e., hex(00)-hex(0F)) are displayed as periods (.) in the ASCII field. If the value is long, only the first 16 characters are displayed. Alpha array values are displayed as a single string starting at the first element.

For numeric arrays, as many elements that fit on a single line are displayed. The elements in row 1 are displayed, then row 2, etc.

*Example:*

```
:LIST DIM
A           123.45
B1          0
B2(5)       -1 2 0 0 0
B$6         "AB..CD" 41 42 0D 0A 43 44
M$(256)1     "Wang Laboratories" 57 61 6E 67 20 4C 61 62 6F 72
              61 74 6F
N(3,4)      .874539284 .777430912 .314985239222 -.0002438216
            .10138327
```

*Examples of valid syntax:*

```
LIST COM
LIST DIM
LIST COM/DIM
LIST "title" DIM
```

## LIST SELECT

*Format:*

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

  where:
                              D
                              R
                              G
                             ERROR [ > error-code][NO] ROUNDP [digit]
                             LINE numeric-expression
                             CI device-address
                             INPUT device-address
                             CO device-address [(width)]
                             PRINT device-address [(width)]
    select-parameter =
                             LIST device-address [(width)]
                             PLOT device-address
                             TAPE device-address
                             DISK device-address
                             file-number device-address
                             TC port-number
                             TERMINAL port-number
                             DRIVER device-address [OFF]
                              T ON/OFF
                              H ON/OFF
                              NEW *
                              OLD *

    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 hex digits representing the
                       device type and address
            width   =  an expression 0 < 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
```

Note: * CS/386 ONLY

The LIST SELECT statement displays the options currently SELECTED. (Need a "write–up" from Mike Riley.)

## LIST T

*Format:*

```
                         literal-string    ,literal-string
   LIST [title] T                                              ...
                         alpha-variable    ,alpha-variable
 where:

   title = alpha-variable or literal-string
```

The LIST T command generates a cross-reference listing of all program text lines that contain a specified string of characters. The programmer can specify each string to be sought as either the value of an alpha-variable or as a literal-string. When performing the search, the system ignores space characters in the string being sought and in the program text. Including more than one alpha-variable and/or literal-string in the LIST T argument list enables the system to search for more than one character string.

The following examples illustrate the various ways in which a programmer can use the LIST T command to produce a cross-reference listing of character strings (these examples assume that the following program text resides in memory):

```
10 REM PRINTER SELECT IS ON LINE 20
20 SELECT PRINT/215
30 A$ = "CHARLES DICKENS"40 PRINT AT (3,10); A$
50 REM LIST T IGNORES S PA C ES
```

*Example:* Cross-reference listing of a specified alpha-variable

```
:LIST T A$
"CHARLES DICKENS"
-    0030

:B$ = "SPA CE S" :LIST T B$
"SPA CE S"
-    0050
```

*Example:* Cross-reference listing of a specified literal-string

```
:LIST T "SELECT"
"SELECT"
-    0010 0020

:LIST T "/215"
"/215"
-    0020
```

*Example:* Cross-reference listing of multiple literal-strings

```
:LIST T "A", "C", "H", "Z"
"A"
  -     0030 0040 0050
"C"
  -     0010 0020 0030 0050
"H"
  -     0030
"Z"
```

*Examples of valid syntax:*

```
LIST T "SELECT"
LIST T A$
LIST T "A", "B", "C"
LIST "title" T "PRINT"
```

## LIST V

*Format:*

```
    LIST [title] V [variable-name] [,[variable-name]]

  where:

                      letter [digit] (for numeric-scalars)
                      letter [digit]$ (for alpha-scalars)
  variable-name =
                      letter [digit]((for numeric-arrays)
                      letter [digit]$((for alpha-arrays)

          title =      alpha-variable or literal-string
```

The LIST V command generates a cross-reference listing of all references to the specified variables within the current program.

> ***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 only that variable 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.

*Example* (LISTing Variable References)

Assume the following program lines are in memory:

```
10 DIM A(3),B$80,C$(2,3),D9$(81),N40
20 A1=5:X,X1,Y=0.3
30 A(1) = A1 *Y/2
40 B$ = D9$(1)
:LISTV
A( -  0010  0030
A1 -  0020  0030
B$ -  0010  0040
C$(   -  0010
D9$(  -  0010  0040
N( -  0010
X     -  0020
X1 -  0020
Y     -  0020  0030
```

*Examples of valid syntax:*

```
LIST V
LIST "Variables" V
LIST "Numeric scalars" A,
LIST "A variables" A,A9
```

## LIST #

*Format:*

```
     LIST [title] # [starting-line-number] [, [ending-line-number]]

  where:

     title = alpha-variable or literal-string
```

The LIST # command generates a cross-reference listing of all references to the specified line numbers within the current program. You can combine the parameters of the LIST # command to produce the following results:

- LIST # with no line numbers (e.g., LIST #) cross-references the entire program.
- LIST # followed by only a starting-line-number (e.g., LIST # 360) cross-references the specified line.
- LIST # followed by a starting-line-number and a comma (e.g.,
- LIST # 10,) cross-references all lines from the specified line to the highest numbered line.
- LIST # followed by a comma and an ending-line-number (e.g.,
- LIST # ,20) cross-references all lines from the lowest numbered line up to and including the specified line.
- LIST # followed by a starting-line-number, a comma, and an ending-line-number (e.g., LIST # 60, 150) cross-references the specified lines and all intervening lines.

*Example:*

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
```

## LISTing all line number references

```
:LIST #

0040   -   0040  0050
0050   -   0040
0090   -   0020
00100  -   0010  0020
00200  -   0030
00300  -   0040
00500  -   0110
```

*Examples of valid syntax:*

```
LIST #
LIST "title" #
LIST # 10
LIST # 20,LIST # ,999
LIST # 100,200
```

## LIST'

*Format:*

```
    LIST [title] ' [integer]

  where:

    title = alpha-variable or literal-string

    integer = value from 0 to 255
```

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 question marks (????) instead of a line-number.

*Examples:*

Assume the following program lines are in memory:

```
10 DEFFN' 15 "TEXT"
20 GOSUB'0: GOSUB'1
30 DEFFN'0
40 GOSUB'1
50 DEFFN'1
60 GOSUB'2
```

LISTing all DEFFN' subroutine definitions and references

```
:LIST'
0030 DEFFN' 0
     - 0020  0030
0050 DEFFN' 1-
     0020  0040  0050
???? DEFFN' 2
     - 0060
0010 DEFFN' 15
     - 0010
```

LISTing a specified DEFFN' subroutine

```
:LIST '01
0050 DEFFN' 1
    -0030  0040  0050
```

## RENUMBER

*Format:*

```
    RENUMBER [starting-line-number] [- ending-line-number]

       [TO new-starting-line-number] [STEP increment]

  where:

       increment  =  positive integer
```

The RENUMBER command renumbers a program in memory. This statement performs the following renumbering operations:

- RENUMBER with no line numbers (e.g., RENUMBER) renumbers all program lines.
- RENUMBER followed by a starting-line-number and no ending-line-number (e.g., RENUMBER 200,) renumbers all lines from the specified line to the highest numbered line.
- RENUMBER followed by only an ending-line-number (e.g., RENUMBER-20), renumbers all lines from the lowest numbered line up to and including the specified line number.
- RENUMBER followed by a starting-line-number and an ending-line-number (e.g., RENUMBER 80, 400) renumbers the specified lines and all intervening lines.

If you do not specify a new starting-line-number, the system begins renumbering at the STEP value. If no STEP value is specified, the system assumes an increment of 10.

After renumbering, the system automatically places renumbered program lines into the proper position in the program. All references to line numbers within the renumbered program (e.g., GOTO, GOSUB, and PRINTUSING statements) are also automatically modified in accordance with the new numbering scheme.

You can use the RENUMBER command to insert a section of program text between two program lines. However, if the renumbered program section does not completely fit between the two specified lines, the system displays an error message and terminates execution of the command.

The following example illustrates one way in which you can use the RENUMBER command. This example assumes that the following program text resides in memory.

```
: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 "PAYROLL"
```

The following RENUMBER command could be executed:

```
:RENUMBER 800-820 TO 50
```

If you issue a LIST command, the renumbered program now appears as follows:

```
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 "PAYROLL"
```

Lines 800 to 820 have been renumbered to Lines 50 to 70 and moved to the appropriate location in the program. Additionally, the GOSUB reference in Line 500 has been changed from Line 800 to Line 50.

*Examples of valid syntax:*

```
RENUMBER
RENUMBER STEP 20
RENUMBER TO 500 STEP 5RENUMBER 100 TO 1000
RENUMBER 100 STEP 20RENUMBER -1000 TO 10
RENUMBER 100-200 TO 500
RENUMBER 100-200 TO 1000 STEP 5
```

## RESET

When you press the RESET key, the system performs the following operations:

- Terminates program execution or listing
- Clears the stacks of all FOR...NEXT loops and subroutine information
- Turns off Trace and Pause modes
- Clears the screen, displays the READY message, and returns control to the keyboard

RESET does not clear program text from memory or disturb the current values of variables.

Generally, you should use RESET only as a last alternative to rectify a processing problem; program execution cannot be resumed following a RESET command. However, if you use the HALT command to halt program execution, a subsequent CONTINUE command resumes program execution.

## RUN

*Format:*

```
RUN [line-number [, statement-number]]
```

The RUN command resolves and initiates execution of a user program. Program resolution involves the following operations:

- The system sequentially scans the program for syntax errors, verifies all references to variables, line numbers, and line labels (e.g., GOTO, GOSUB, or ON/GOTO/GOSUB statements), and ensures that all referenced array-variables are defined in a DIM or COM statement.
- The system reserves space for all variables not already defined. Numeric variables initially are set to zero, while alphanumeric variables initially are set to all blanks.
- The system initializes the data pointer used by the READ statement to the first data value in the first DATA statement.

If program resolution is completed with no errors, the RUN command immediately begins program execution. If an error is detected during program resolution, the system signals an error and does not execute the program.

If no line-number is specified (e.g., RUN), the RUN command first clears all noncommon variables from memory (common variables are not disturbed), performs program resolution, and then begins program execution from the lowest numbered line.

If a line-number is specified (e.g., RUN 80), the RUN command resolves the program and begins execution at the specified program line. However, noncommon variables are not cleared from memory, and all variables retain current values. As a result, 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 cannot be restarted in the middle of a FOR/NEXT loop or a subroutine.

Program execution can be started at a particular statement within a multi-statement 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.

*Examples of valid syntax:*

```
RUN
RUN 30
RUN 100,3
```

## STOP

*Format:*

```
STOP [line#]
```

STOP, when used in Immediate mode, sets a program stop point at the specified program line. Subsequently, when a program is run, that program's execution will stop just before the specified line is to be executed, as if the STOP statement were the first statement of that line. (Refer to the STOP statement in Chapter 11 for a description of using STOP statements within a program.) When the program stops, the word STOP followed by the line number is displayed. Only one stop point can be set; entering a new Immediate mode STOP replaces the previous stop point setting. To clear a stop point, enter Immediate mode STOP without a line number, or RESET LOAD RUN, or enter a CLEAR command. Immediate mode STOP is especially useful when debugging programs since STOP need not be edited into the program itself.

To continue a program after the program has stopped, do one of the following:

- Press HALT to step through the program execution one statement at a time. Program stepping begins at the statement immediately following the stop point.
- Enter a CONTINUE command to resume program execution at the statement following the stop point.

*Examples of valid syntax:*

```
STOP 100
STOP
```

## TRACE

*Format:*

```
TRACE [OFF]
```

The TRACE statement produces a trace of important operations in program execution as an aid in program debugging. Trace mode is turned on when a TRACE statement is executed within a a program or in Immediate mode; trace mode is turned off by executing a TRACE OFF statement, by pressing RESET, or by executing a CLEAR command. While in trace mode, output is produced from the following program operations:

- Assignment (LET) statements and FOR/NEXT loops
- Program branches (e.g., GOTO, GOSUB, and RETURN)

Whenever a variable receives a new value in an assignment statement or during the execution of a FOR/NEXT loop, the variable name and its new value are output. Whenever the program branches, TRACE outputs the words TRANSFER TO followed by the line number branched to. Immediate mode statements are not traced.

Trace output is sent to the currently selected Console Output (CO) device (see SELECT). The output of each statement traced is preceded by the line number of that statement and colons representing statement separators for multi-statement lines. Trace output adheres to the following conventions:

- The equal sign (=) in an assignment statement is replaced in trace output by an arrow (<-). The arrow is more appropriate than an equal sign since the contents of the variable are not necessarily identical to the value displayed in the trace output.

  *Example:*

  ```
  :TRACE
  :10 X = 456
  :20 A,B,C = 22.11
  :RUN

  10 X <- 456
  20 A <- B <- C <-  22.11
  ```

- Alpha values are displayed in both ASCII and hexadecimal notation. Only the first 16 characters of an alpha value are displayed. Also, the hex output of the value is truncated at the end of the output line.

*Example:*

```
:10 DIM L$26: L$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
:RUN

10: L$ <- "ABCDEFGHIJKLMNOP" 41 42 43 44 45 46 47 48 49 4A
```

- Nonprintable characters (HEX(00) to HEX(0F)) are displayed as decimal-point characters in the ASCII values. This prevents screen control characters from upsetting the display.

*Example:*

```
:10 A$ = HEX(41 03 0A 42)
:RUN

10 A$ <- "A..B " 41 03 0A 42 20
```

- The termination of a FOR/NEXT loop is indicated with the words NEXT END.

*Example:*

```
:10 DIM A(3
:20 FOR I = 1 TO 3
:30 A(I) = I
:40 NEXT I
:RUN

20 I <- 1
30 A( <- 1
40 I <- 2  TRANSFER TO 30
30 A( <- 2
40 I <- 3  TRANSFER TO 30
30 A( <- 3
40 I <- NEXT END
```

- Whenever a branch is made within a program, trace outputs the words TRANSFER TO followed by the line number branched to.

*Example:*

```
:10 GOSUB 500
:20 GOSUB '45: GOTO 600
.
.
.
:500 REM SUBROUTINE: RETURN
:550 DEFFN'45: REM MARKED SUBROUTINE: RETURN
:600 REM
:RUN

10 TRANSFER TO 500
500: TRANSFER TO 2020 TRANSFER TO 550
550:: TRANSFER TO 20
20: TRANSFER TO 600
```

*Examples of valid syntax:*

```
TRACE
TRACE OFF
```

# 11

# General BASIC-2 Statements

## Overview

BASIC-2 general-purpose statements perform such fundamental program operations as assignment, loop control, conditional branching, subroutine definition and access, keyboard data entry, data conversion, and generation of printed output. Table 11-1 lists the general-purpose statements that are available in BASIC-2.

**Table 11-1. General-Purpose BASIC-2 Statements**

| Command | Function |
|---|---|
| COM | Defines one or more common variables within a BASIC-2 program. |
| COM CLEAR | Redefines designated common variables as noncommon or noncommon variables as common. |
| CONVER | Converts a numeric value to an ASCII character string representation of the number, and vice versa. |
| DATA | Provides data values used by a corresponding READ statement. |
| DEFFN | Allows you to define a function of one variable for use in a program. |
| 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 pressing a function key. Arguments can be passed to the subroutine. Defines a character string to be supplied when a function key is used to keyboard text entry. |

(continued)

**Table 11–1. General-Purpose BASIC-2 Statements (continued)**

| Command | Function |
|---|---|
| DIM | Defines one or more noncommon variables. |
| DO | Allows a group of statements to be conditionally executed by IF, ELSE, or ERROR. |
| END | Terminates execution of a BASIC program; returns the amount of free space in memory when executed. |
| FOR | Defines the starting boundary of a loop and determines how many times the loop is executed. Used with the NEXT statement. |
| $FORMAT | Defines a format specification for the $PACK and $UNPACK operations. |
| 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. |
| HEXPACK | Converts an ASCII character string of hexadecimal digits into the binary equivalent of those digits. |
| HEXUNPACK | Converts a binary value in the equivalent ASCII character string of hexadecimal digits. |
| IF | Tests one or more conditions and executes the statement immediately following THEN if the condition is true; otherwise, execution continues at the next statement. |
| Image (%) | Defines a format for printed output generated by the PRINTUSING or PRINTUSINGTO statements. |
| INPUT | Accepts entered data during program execution. (Primarily for user entry from the keyboard; displays an optional prompt to request data.) |
| KEYIN | Inputs a single character from a keyboard-like device. |
| LET | 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 that satisfy a given relation to a specified value. |
| NEXT | Delimits the end of a loop initiated with a FOR...TO statement. |
| ON | Used with the GOTO or GOSUB statement to create a computed branch or call based on the value of a specified expression. |
| PACK | Converts numeric values to Wang packed decimal format. |
| $PACK | Perform packing of data in a variety of user-specifiable formats. |
| PRINT | Evaluates and prints the value of a numeric expression, an alpha-variable, or a character string. Values are printed in a system-defined format. |
| PRINT AT | Moves the cursor to a designated position on the screen. |
| PRINT BOX | Draws or erases boxes and lines of specified dimensions. |

(continued)

**Table 11-1. General-Purpose BASIC-2 Statements (continued)**

| Command | Function |
|---|---|
| PRINT HEXOF | Prints the hexadecimal equivalent of an alphanumeric value. |
| PRINT TAB | Tabs to a specified column in a print line on a CRT or printer. |
| PRINTUSING | Prints values and character strings in a format an image specification defines. |
| PRINTUSING TO | Stores formatted print output in a specified alphanumeric-variable for future processing. |
| 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 | Clears subroutine return information from the internal stacks without causing a branch back to the main program. |
| ROTATE | Rotates the bits of a character or a string of characters. |
| STOP | Terminates program execution and displays an optional message and/or line number of the STOP statement. |
| $TRAN | Uses table-lookup procedures to provide high-speed character conversion. |
| UNPACK | Converts values in Wang packed decimal format to numeric. |
| $UNPACK | Unpacks data from a variety of user-specifiable formats. |

## General Forms of the General-purpose Statements

The general forms of all statements described in Table 11-1 appear in alphabetical order on the following pages. Each general form conforms to a standard notation, which indicates various options, requirements, and features of these general forms. The rules for this notation are discussed in the Preface. You should become familiar with this notation before consulting the general forms of the BASIC-2 statements.

## COM

*Format:*

```
    COM com-element [,com-element] ...


  where:
                    numeric-scalar-variable
                    numeric-array-name (dim1[,dim2])
  com-element  =
                    alpha-array-name (dim1[,dim2]) [length]
                    alpha-scalar-variable [length]

         dim  =     dimension (numeric-scalar-variable or positive
                     integer) such that:

                     For 1-dimensional arrays:  1 ≤ dim1 ≤ 65535
                     For 2-dimensional arrays:  1 ≤ dim1, dim2 ≤ 255

       length =     positive integer or numeric-scalar-variable such
                    that:  1 ≤ length ≤ 124
```

The COM statement, like the DIM statement, defines variables within a BASIC program. Unlike variables defined by a DIM statement, variables defined by a COM statement are maintained when a new program is loaded into memory or the current program is run. This allows you to use common variables to pass common data between successive program module overlays.

When a program is run, the system does not disturb currently defined common variables and their contents. However, the system does clear all noncommon variables from memory. (Noncommon variables include all variables not explicitly defined in a COM statement.) The system clears only common variables from memory following the execution of a CLEAR, CLEAR V, or LOAD RUN command.

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 system processes the COM statement during program resolution, which occurs immediately prior to program execution. You initiate program resolution by executing either a RUN command or a LOAD statement under program control. During program resolution, the system scans the entire program for variable references and reserves space for all variables that have not been previously defined. During program execution, the system ignores a COM statement.

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 is defined prior to a common, the system signals an error and halts program resolution. If a common variable is encountered that has already been defined in the current program or a previous program, its dimensions must be identical to the previously defined dimensions; otherwise, the system signals an error. 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). 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 can, 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 can be defined at the beginning of any subsequent program.

### 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. As a result, 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.

*Example:*

The first program module in a system determines the array dimensions.

```
10 REM THIS IS MODULE 1
20 COM X,Y
30 INPUT "DIMENSIONS",X,Y
40 LOAD "MODULE2"
```

The second module uses the array dimensions for the array definition.

```
10 REM THIS IS MODULE 2
20 COM N(X,Y)
.
.
.
```

*Examples of valid syntax:*

```
COM A(10), B(3,3), C
COM D$(20), E$(2,3)100, F1$64
COM N(R,C), A$(X,Y)L
```

## COM CLEAR

*Format:*

```
                        scalar-variable
    COM CLEAR
                         array
```

The COM CLEAR statement defines some or all previously defined common variables as noncommon or defines some or all previously defined noncommon variables as common. The COM CLEAR statement 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 or change the values of the variables.

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) clears 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 changed to noncommon variables; all common variables that are defined in the program prior to the specified variable or are 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 an undefined variable is specified in a COM CLEAR statement, the system signals an error.

COM CLEAR is an executable statement and is performed in sequence during normal program execution rather than once at program resolution (as in the case of the COM statement).

| Expression | Result |
|---|---|
| 10 COM CLEAR | Redefines all previously defined common variables as noncommon. |
| 110 COM A, B, X, Y(10)<br>.<br>.<br>. | |
| 200 COM CLEAR X | Redefines X and Y( ) as noncommon; A and B remain common. |
| 10 COM X(10,10)<br>20 A = B + C<br>30 COM CLEAR B | Redefines A as a common variable; B and C remain noncommon; X( ) remains common. |

## CONVERT

*Format 1: (alpha to numeric)*

```
CONVERT alpha-variable TO numeric-variable
```

*Format 2: (numeric to alpha)*

```
CONVERT numeric-expression TO alpha-variable, (image)
```

```
where:

                 +                                         +
     image =         [$][#[,]] ... [.][#] ... [↑↑↑↑]       -
                 -                                         ++
                                                           --

               alpha-variable containing image

     length of image < 255
```

The CONVERT statement either converts an ASCII character string to a numeric value or converts a numeric value to an alphanumeric character string. Therefore, BASIC-2 provides two formats of the CONVERT statement.

*Format 1: Alpha-To-Numeric Conversion*

Format 1 of the CONVERT statement converts the character string in the specified alpha-variable to a numeric value.

*Example:*

```
:10 A$ = "1234"
:20 CONVERT A$ TO X
:30 PRINT "X = "; X
:RUN
X = 1234
```

If the ASCII character string in the specified alpha-variable is not a legitimate BASIC representation of a number, the system signals an error. An error-handling routine can anticipate and allow recovery from this error. (Refer to the discussion of the ERROR statement in Chapter 9.) You can use the STR function to convert a portion of an alpha-variable to a numeric value.

*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 useful for reading numeric data from a peripheral device in a format not directly compatible with numeric input or for validating 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.

*Format 2: Numeric-to-Alpha Conversion*

Format 2 of the CONVERT statement converts the numeric value of the specified expression to an ASCII character string according to the specified image. The alphanumeric character string is then stored in the specified alpha-variable.

The image specifies precisely how to convert the numeric value. Each character in the image corresponds to a character in the resultant character string. The image consists of digit-selector characters (#) to signify digits and (optional) plus signs (+, ++), minus signs (-, –), decimal points (.), and up arrows (↑) 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., #.##)*

The CONVERT statement formats numeric values according to the following rules:

- If the image starts or ends with a plus sign (+), CONVERT edits the real sign of the value (+ or -) into the character string at the specified position (beginning or end).
- If the image starts or ends with a minus sign (-), CONVERT edits a blank for positive values or a minus sign for negative values into the character string at the specified position (beginning or end).
- If no sign is specified in the image, CONVERT edits no sign into the character string (i.e., the absolute value of the specified expression is output).
- If the image ends with two plus signs (++), CONVERT edits two spaces for nonnegative values or the characters CR for negative values into the end of the character string.
- If the image ends with two minus signs (–), CONVERT edits two spaces for nonnegative values or the characters DB for negative values into the end of the character string.
- If the image contains a dollar sign ($), comma (,), or decimal point (.), CONVERT edits the specified character into the character string at the corresponding position.

- If the image is fixed-point (i.e., no up arrows (↑) specified), CONVERT edits the numeric value into the character string as a fixed-point number. The fractional portion is automatically truncated or extended with trailing zeros, and the integer portion is padded with leading zeros to fit the image-specification. If the integer portion exceeds the image-specification, the system signals an error.
- If the image is exponential (i.e., four up arrows (↑↑↑↑) specified), CONVERT edits the value into the character string in exponential format. The value is scaled to fit the specified image (leading zeros 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 up arrows (i.e., ↑↑↑↑). The specification of any other number of up arrows produces an error upon entry.*

*Example: Numeric-To-Alpha Conversions*

This example assumes that the following values are provided for X, B$, and C$.

```
X = 12.345          B$ = "#,###+"          C$ = "$##.##++"
```

| Statements | Results |
|---|---|
| 10 CONVERT X TO A$, (###) | A$ = "012" |
| 20 CONVERT X TO A$, (+##.##) | A$ = "+12.34" |
| 30 CONVERT X TO A$, (-#.#↑↑↑↑ | A$ = " 1.2E+01" |
| 40 CONVERT 100 * X TO A$, (B$) | A$ = "1,234+" |
| 50 CONVERT -X TO A$, (C$) | A$ = "$12.34CR" |

*Example: Asterisk Filling*

```
:10 CONVERT 1.23 TO A$, (+####.##)
:20 PRINT "RESULT OF CONVERT: "; A$
:30 $TRAN (STR(A$,, MIN(POS(A$ > "0"), POS(A$ = "."')), "*0") R
:40 PRINT "  ASTERISK FILLED: "; A$
:RUN
RESULT OF CONVERT:  +0001.23
  ASTERISK FILLED:  +***1.23
```

*Example: Removing Leading Zeros*

```
:10 CONVERT 1.23 TO A$, (+####.##)
:20 PRINT "           Result of Convert: "; A$
:30 STR(A$,2) = STR(A$,POS(STR(A$,2) <>"0") + 1)
:40 PRINT " With Leading Zeros Removed: "; A$
:RUN
      Result of Convert:  +0001.23
With Leading Zeros Removed:  +1.23
```

***Note:** In some cases, numeric data converted to alphanumeric format with Format 2 of the CONVERT statement can cause an error when converted back to numeric format with Format 1. Specifically, the following characters are invalid in numeric format:*

- Dollar sign
- Trailing plus or minus sign
- Debit (DR) and credit (CR) signs
- Comma

*Examples of valid syntax (Alpha-to-Numeric Conversion):*

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

*Examples of valid syntax (Numeric-to-Alpha Conversion):*

```
CONVERT 45.6 to A$, (####)
CONVERT -88.735 TO D$, ($##.##++")
```

## DATA

*Format:*

```
                number              ,number
    DATA                                              ...
              literal-string        ,literal-string
```

The DATA statement supplies the values to be used by the variables in a READ statement. In effect, the DATA and READ statements provide a means of storing tables of constants in a program.

Each time a READ statement is executed in a program, READ obtains the next sequential value in the DATA statement value list and assigns the value to a variable in the READ statement variable list. You must specify the values in a DATA statement in the order in which they are to be used, and must separate individual values by commas. The DATA statement cannot be used in Immediate mode.

*Example:*

```
:10 FOR I = 1 TO 3
:20 READ W
:30 PRINT W, W*2
:40 NEXT I
:50 DATA 5, 8.26, -687
:RUN
 5      10
 8.26   16.52
-687    -1374
```

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. If the DATA statement contains an insufficient amount of data, the system signals an error.

The RESTORE statement resets the current DATA statement pointer so that a READ statement can reuse DATA statement values. (Refer to the discussion of the RESTORE statement later in this section.)

*Examples of valid syntax:*

```
DATA 4, 3, 5, 6
DATA 6.56E+45, -644.543
DATA "BOSTON,MASS", "SMITH", 12.2
DATA HEX(0A), "ABC"
```

## DEFFN

*Format:*

```
    DEFFN a (v) = expression

  where:

      a  =  a letter or digit that identifies the function
      v  =  a numeric-scalar-variable
```

The DEFFN (define function) statement enables you to define a function of one variable within a program. The function can 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 the argument specified in the FN function is automatically passed to the DEFFN statement for evaluation; the result is returned to the FN function. A user-defined function can be used in a program wherever system-defined numeric functions are legal. (Refer to Chapter 4 for information on the available system-defined numeric functions.)

A DEFFN statement requires the following three parameters:

- The function name
- A numeric-scalar-variable that serves as a dummy variable
- An expression that defines the function

The function name identifies the defined function when it is referenced with an FN function. The variable specified as a dummy variable is 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 following statement, A is the user-defined function name and X is the dummy variable.

```
    10 DEFFN A(X) = X*4-X
```

An FN function that references a user-defined function requires the following parameters:

- The function name
- Argument whose value is passed to the defined function for evaluation

The arguments specified in an FN function can 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 following statement, A is the function name that identifies the defined function being referenced, and C*2 is the argument whose value replaces the dummy variable in the DEFFN expression:

```
50 V = FNA(C*2)
```

*Example:*

The following brief routine illustrates the use of a user-defined function:

```
:10 DEFFN A(X) = X*4-X
:20 C = 3
:30 PRINT FNA(C*2)
:40 END:RUN
18
```

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 in Line 10. Every occurrence of the dummy variable X in defined function A is replaced with this argument value. Therefore, DEFFN A(6) = 6*4-6 = 18. The result, 18, is then returned to the referencing FN function where it can be assigned to a variable, printed, or used as an argument in a larger expression.

A defined function cannot refer to itself, but it can refer to other defined functions.

*Example:*

```
10 DEFFN 1(A) = A*4-A
20 DEFFN 2(A) = A + FN1(A)
```

Two functions cannot refer to each other (producing, in effect, an endless loop).

*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 can be used in Immediate mode.

*Examples of valid syntax:*

```
DEFFN A(C) = (3*A) - 8*C
DEFFN 1(A3) = (EXP(A3) - EXP(-A3))/2
DEFFN A(C) = FNB(C) * FNC(C)
```

## *DEFFN'* (for Keyboard Text Entry)

*Format:*

```
    DEFFN' integer literal-string [; literal-string] ...

 where:

      0 ≤ integer ≤ 255
```

The DEFFN' statement has the following two purposes:

- To define a character string to be supplied when a function key is used for keyboard text entry.
- To identify the beginning of a subroutine that can be called by pressing a function key or executing a GOSUB' statement.

The DEFFN' statement must be the first statement on the line (i.e., it must immediately follow the line number). The DEFFN' statement cannot be used in Immediate mode.

The integer in the DEFFN' statement represents one of the keyboard function keys. When a function key is pressed, the literal strings defined in the DEFFN' statement are displayed and become part of the text line currently being entered.

All keyboards used with BASIC-2 have at least 16 (physical) function keys, which when used with SHIFT, provide access to 32 text entry definitions or subroutines ('0 to '31). Other keys also produce special function codes; refer to Appendix A for a complete list of key codes for BASIC-2.

*Example:*

Line 100 defines function key '12 as the character string "HEX(".

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

Pressing function key '12 after :200 PRINT is entered 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 300 defines function key '05 as the character string "BOSTON, MASS." followed by a carriage return (HEX (0D)).

```
300 DEFFN' 05 "BOSTON, MASS."; HEX(0D)
```

In this case, the carriage return code causes the line to be executed as soon as function key '05 is pressed; there is no need to press the RETURN key.

The keyboard can be customized by defining the function keys as characters that do not appear on the keyboard. Keyboard customization is achieved 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. 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:*

```
DEFFN'1 "PRINT X,Y,Z";HEX(0D)
DEFFN'31 HEX(5C)
DEFFN'2 "Lowell, MA 01810"
```

## *DEFFN'* (Subroutine Entry Point)

*Format:*

```
      DEFFN' integer [(variable [,variable] ...)]

   where:

        0 ≤ integer ≤ 255
```

The DEFFN' statement, followed by an integer and an optional variable list enclosed in parentheses, indicates the beginning of a subroutine. The DEFFN' subroutine can be entered from the program by using a GOSUB' statement (refer to the discussion of GOSUB' later in this section) or from the keyboard by pressing the appropriate function key. When a function key is pressed or a GOSUB' statement is executed, the system scans the BASIC program for a DEFFN' statement with an integer corresponding to the number of the function key or the integer in the GOSUB' statement. Execution of the program then begins at the corresponding DEFFN' statement. (For example, if function key '02 is pressed, 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 is made by pressing a 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 is waiting for keyboard entry and function key '02 is pressed, program execution commences at Line 100, and upon execution of the RETURN statement, the system terminates execution and returns control to the keyboard.

The DEFFN' statement can 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 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 (refer to 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), the system signals an error.

*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 function key, the arguments to be passed can be entered prior to pressing the 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 in the following way:

```
:1.2, 3.24, JOHN (press 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 entry of 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 question mark (?), and waits for entry of the remaining arguments. You must first reenter a new value in place of the illegal value, then enter any succeeding values, and finally press RETURN. (This procedure is identical to the error response procedure for INPUT; refer to the discussion of the INPUT statement later in this section.)

*Example:*

```
(Press 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.

*Example:*

```
100 DEFFN'4
110 INPUT "RATE", R
120 C = 100*R-50130 PRINT "COST=" ;C
140 RETURN
```

Pressing function 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.

When program execution is initiated with a function key, program resolution is not performed. For this reason, the program should be run initially with a RUN command to ensure that a proper resolution phase is carried out. If the program is not resolved, an error is generated.

The DEFFN' statement can be used with function keys to provide a number of entry points to begin execution of a program. However, because 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:

- The subroutine terminates with a RETURN statement, which deletes return information for that subroutine from the stacks.
- The subroutine return information is removed from the stacks by executing a RETURN CLEAR statement.
- RESET is pressed prior to pressing the function key. (This clears the stacks.)

Failure to meet at least one of these conditions eventually causes a Not Enough Memory error.

*Examples of valid syntax:*

```
DEFFN'1
DEFFN'100
DEFFN'2(27,"help")
DEFFN'50(F,F$,ERR)
```

## DIM

*Format:*

```
    DIM dim-element [,dim-element] ...

  where:

                        numeric-scalar-variable
                        numeric-array-name  (dim1[,dim2])
    dim-element =

                       alpha-array-name (dim1[,dim2]) [length]
                        alpha-scalar-variable [length]

            dim =  dimension (numeric-scalar-variable or positive
                    integer) such that:

                    For 1-dimensional arrays:   1 ≤ dim1 ≤ 65535
                    For 2-dimensional arrays:   1 ≤ dim1, dim2 ≤ 255

         length =   positive integer or numeric-scalar-variable such
                     that:  1 ≤ length ≤ 124
```

The DIM statement defines program variables and reserves space in memory for each specified variable. Variables defined by the DIM statement are called noncommon variables. Noncommon variables are automatically cleared from memory during a program overlay. A single DIM statement can reserve space for more than one variable by separating the successive dim-elements with commas.

You need not explicitly define scalar-variables. Scalar-variables are automatically defined when first referenced in a program. For purposes of documentation, you can specify a numeric-scalar in a DIM statement, but you cannot change the fixed length of eight bytes. However, you can specify the length of an alphanumeric-scalar-variable. Alpha scalars are automatically assigned a length of 16 characters (bytes) if not otherwise specified in a DIM or COM statement.

You must define all array variables in DIM or COM statements before referencing the array in a program.

The system processes the DIM statement during program resolution. Before program execution begins, the system scans the entire program in memory for variable occurrences. When a DIM statement is encountered, the system reserves space for each specified variable, if the variable is not already defined. If an undefined scalar-variable is encountered in the program, the system automatically reserves space for the variable. If a reference to an array occurs outside a DIM or COM statement, an error results if the array is not defined or if the dimensions and lengths do not agree with those specified in the previous definition. If the variable is defined, an error results unless the dimensions and length are identical to those specified in the previous definition.

| Examples | Comments |
|---|---|
| 10 DIM N(45) | Reserves space for a 1-dimensional numeric array with 45 elements. |
| 20 DIM A(8,10) | Reserves space for a 2-dimensional numeric array with 8 rows and 10 columns. |
| 30 DIM K(35), L(3), M(8,7) | Reserves space for two 1-dimensional arrays and one 2-dimensional array. |
| 40 DIM A$32 | Reserves space for an alpha scalar variable with a length equal to 32. |

The dimensions of arrays and lengths of alphanumeric variables or array elements are specified by positive integers; zero is not allowed. However, you can also use numeric-scalar-variables to specify the dimensions and the lengths of array elements and alpha-variables.

The numeric-scalar-variables must have legal values for dimensions and lengths at the time the DIM statement is processed. Since all noncommon variables are cleared from memory during program overlay or following a RUN command without a line number, scalar variables used to specify dimensions and lengths of variables in DIM statements should be defined as common variables. Refer to the discussion of the COM statement earlier in this chapter.

The use of common variables permits array sizes to be dynamically altered at run time if the first program module in the system defines the array sizes for succeeding modules.

*Example:*

The first program module determines the array dimensions.

```
10 REM THIS IS MODULE 1
20 COM X,Y
30 INPUT "DIMENSIONS",X,Y
40 LOAD T/320, "MODULE2"
```

The second module uses the array dimensions for the array definition.

```
10 REM THIS IS MODULE 2
20 DIM N(X,Y)
    .
    .
    .
```

*Examples of valid syntax:*

```
DIM N(5,10), A$(2,3)64
DIM X, A$, B$17
DIM M$(X,Y)L, Q(R,C)
```

## DO group

*Format:*

```
DO
[statement] ...
ENDDO
```

A do-group is a set of statements treated as a unit for conditional execution after an IF, ELSE, or ERROR statement. The do-group is delimited by the statement DO at the beginning and the statement ENDDO at the end. The statements in a do-group can NOT be nested. Improperly paired DO and ENDDO statements are detected at resolution time.

At execution time, if an ENDDO statement is encountered without having executed the DO statement, the ENDDO is ignored.

*Examples:*

```
10 X=Y/Z: ERROR DO: X=9999: GOSUB 100: ENDDO: PRINT X
100 IF P$="PRINTER" THEN DO
110      SELECT PRINT /215
120      PRINT A$,P125      GOSUB 500
130    ENDDO140 ELSE DO150      SELECT PRINT /005160
PRINT A$,P
170      GOSUB 600
180    ENDDO
```

*Examples of valid syntax:*

```
IF F=0 THEN DO: X=Y: Y=Z: Z=X: ENDDO
DATALOAD DC OPEN#1,F$: ERROR DO
ENDDO
```

## END

*Format:*

```
END
```

The END statement is an optional program statement indicating the end of a program. It need not be the last executable statement in a program, which allows you to use more than one END statement in a program.

When the END statement is encountered in a program, the system terminates program execution and displays the amount of free space currently available in user memory (i.e., returns the amount of memory not currently occupied by program text or data). An END statement executed under program control flushes the stacks.

After a program is entered, you can execute an END statement in Immediate mode to obtain the amount of free space currently available in user memory. 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 an END statement is executed is determined in two ways.

- When a program is entered or loaded from a disk image following a CLEAR command, the free space displayed by an Immediate mode END statement reflects only the space occupied by the program text since variable space is not yet allocated.
- After the program executes once, the free space displayed after 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 the SPACE function. Unlike END, SPACE returns a free space value that includes the space occupied by the stack.*

## FOR

*Format:*

```
    FOR counter-variable = initial-value TO exit-value [STEP value]

  where:

    counter-variable  =  numeric-scalar-variable
              value  =  numeric-expression
```

The FOR statement constructs a loop, together with its companion, the NEXT statement. FOR marks the beginning of the loop; NEXT marks the end of the loop. The statements between FOR and NEXT are executed repeatedly until the exit-value of the loop is reached.

FOR defines the loop parameters and stores them in the 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. You can choose to specify the STEP value, which defines the amount by which the counter is incremented or decremented each time through the loop. The STEP value can be positive or negative. If no STEP value is specified, the system uses a default value of +1.

Although the FOR 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 NEXT statement controls the subsequent loop operations.

Each time through the loop, the NEXT statement adds the STEP value to the value of the counter and compares the result 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 or not it exceeds the exit-value. If the STEP value is negative, the counter is tested to determine whether or not it is less than the exit-value. If the tested condition is not met, the NEXT statement reexecutes the loop. If the tested condition is met, the system terminates the loop and resumes program execution with the statement immediately following NEXT.

The relationships between the initial-value, the exit-value, and the STEP value in a FOR statement are significant.

*Example:*

In the following statement, the negative STEP value instructs the NEXT statement to test for a counter value less than the exit-value.

```
50 FOR I = 1 TO 5 STEP -1
```

In this case the loop begins with a counter value less than the exit-value. Following the first execution of the loop, the termination condition is satisfied, and the loop prematurely ends after a single execution.

The following conditions exist regarding the use of FOR/NEXT loops:

- It is illegal to branch into the middle of a FOR/NEXT loop from elsewhere in a program. Since the loop parameters are defined by FOR, the execution of a NEXT statement for which no corresponding FOR statement is executed results in an error.
- It is legal to branch out of a FOR/NEXT loop before the loop is terminated; however, a problem arises if this operation is repeated often within a program. The loop parameters defined in a FOR statement are stored in a stack in memory. When a loop terminates in the normal manner, this information is cleared from the stack. If the loop is not terminated, the information remains in the stack. As a result, repeatedly branching out of an unterminated loop causes the loop parameter information to accumulate in the stack, eventually producing a Not Enough Memory error.
- In addition to normal loop termination, there are two other ways to clear loop information from the stack. 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 are cleared from the stack, execution of a NEXT statement in that loop is illegal.
- 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 stack.
- There is no practical limit to the number of loops that can be nested in a program.
- A loop with a STEP value of zero is executed only once.

*Example: A STEP value of 1 is used if not specified.*

```
10 FOR X = 1 TO 16
20 PRINT X, SQR(X)
30 NEXT X
```

*Example: 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
```

*Example: 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
```

*Example: RETURN properly terminates loops contained within subrou tines.*

```
10 X = 3: Y = 100
20 GOSUB 100
30 END100 S = 0110 FOR B1 = X TO Y
120 S = S+B1130 IF S > 1000 THEN 150
140 NEXT B1
150 RETURN
```

*Example: 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 10060 NEXT I
70 PRINT S
80 GO TO 10
100 I = X: NEXT I: GOTO 10
```

*Example: Valid syntax.*

```
FOR I = 1 TO 10
FOR I = B TO E
FOR K1 = (B+2)/C TO D-E STEP J
FOR L = 10 TO 1 STEP -1
```

## $FORMAT

*Format:*

```
   $FORMAT alpha-variable = field-specification [,field-spec] ...

  where:

                              SKIPxxx       (skip field)
                              Fxxx          (ASCII free format)
                              Ixxx[.dd]      (ASCII integer format)
  field-specification =        Dxxx[.dd]      (IBM display format)
                              Uxxx[.dd]      (IBM USACII-8 format)
                              P+xxx.[.dd]   (IBM packed decimal format)
                              Pxxx.[.dd]    (unsigned packed decimal)
                               Axxx          (alphanumeric format)

                       xxx =   field width, such that: 0 < xxx < 256

                        dd =   implied decimal position, such that:
                                0 ≤= dd < 16
```

The $FORMAT statement is a form of the assignment statement that 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:*

```
$FORMAT F$ = I3, P8.2, A10: $PACK (F = F$)B$( ) FROM X, Y, A$

$FORMAT F1$ = F13, SKIP5, A9: $UNPACK (F = F1$)B$( ) TO N,
P$(1)
```

## GOSUB

*Format:*

```
GOSUB  line-number
```

The GOSUB statement transfers program execution to the first program line of a subroutine. The designated program line can contain any BASIC-2 statement, including a REM statement. The GOSUB statement is illegal in Immediate mode.

The subroutine usually 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 that follow RETURN on the same program line are ignored. A subroutine can be executed within a larger subroutine. This technique is called "nesting" of subroutines. There is no practical limit to the number of subroutines that can be nested in a program.

The system stores subroutine return information in a stack in memory. When a RETURN statement is executed, the system clears return information for the corresponding subroutine from the stack. If a RETURN is not executed, the information is not cleared. Repeated entry to subroutines without executing RETURN statements causes the return information to accumulate in the stack and a Not Enough Memory error eventually occurs. Subroutine return information can be cleared without causing a return to the main program by executing a RETURN CLEAR statement. (Refer to the discussion of the RETURN CLEAR statement later in this section.)

*Example: Single Subroutine*

```
10 GOSUB 30
20 PRINT X: STOP
30 REM THIS IS A SUBROUTINE
40  INPUT A, B
50  X = A*2+B-2
60   RETURN: REM END OF SUBROUTINE
```

*Example: 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
140  R = R + COS(X)
150  RETURN
200 REM THIS IS A NESTED SUBROUTINE
210  INPUT "VALUE",X
220  RETURN: REM END OF NESTED SUBROUTINE
```

## GOSUB'

*Format:*

```
GOSUB' integer [(subroutine-argument [,subroutine-argument] ...)]

where:

0 ≤ integer ≤ 255

                                  literal-string
subroutine-argument   =           alpha-variable
                                  numeric-expression
```

The GOSUB' statement transfers 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. 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 (refer to 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 that can be nested in a program. The use of GOSUB' is not permitted in Immediate mode, but the special function keys can access corresponding subroutines from the keyboard. (Refer to the discussion of the DEFFN' statement found earlier in this section.)

Return information for a marked subroutine is stored in a stack in memory and is cleared from the stack only when a RETURN statement for that subroutine is executed. Repeatedly accessing marked subroutines that do not end with a RETURN statement causes return information to accumulate in the stack, eventually causing an overflow error.

*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. Alphanumeric values must be assigned to alphanumeric-variables and numeric values to numeric-variables.

*Example:*

```
:10 DIM C(2)
: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
```

*Examples of valid syntax:*

```
GOSUB'0
GOSUB'255
GOSUB'50(X,Y)
GOSUB'51(ERR, "Illegal entry")
GOSUB'52(A$,B$(),N+M,3.2)
```

## GOTO

*Format:*

```
GOTO  line-number
```

The GOTO statement transfers program execution to a designated location in the program. When a GOTO statement is executed, program execution is transferred to the first statement in the program line having the specified line number. The GOTO statement does not perform a subroutine call, 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 statement.

The GOTO statement can be used in Immediate mode to step through program execution from a particular line number. However, the program must be 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 you execute a HALT command or a CONTINUE command.

*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
```

*Examples of valid syntax:*

```
GOTO 8000
GOTO 430
```

## HEXPACK

*Format:*

```
HEXPACK alpha-variable-1 FROM alpha-variable-2
```

The HEXPACK statement converts an ASCII character string of hexadecimal digits into the binary equivalent of those hex digits and stores the value into the receiving alpha-variable. Hexadecimal digits entered from the keyboard are always entered as ASCII characters. Therefore, a hexadecimal value must be converted from ASCII code to its true binary equivalent before performing binary operations such as addition or subtraction.

*Example:*

In the following program, the HEXPACK statement converts the binary value of the ASCII character A (= 01000001) into the binary value of the hexadecimal digit A (= 1010). HEXPACK then stores this value in the receiving alpha-variable B$.

```
: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$ = A0
```

Alpha-variable-1 receives the converted binary value. Since each pair of characters in the value of alpha-variable-2 is converted to a 1-byte binary value, 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, the system signals an error and halts program execution. If alpha-variable-1 is longer than the converted binary value, HEXPACK left-justifies the binary value and does not modify the remaining bytes of alpha-variable-1.

*Example:*

```
: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
```

In this example, Line 30 is equivalent to the following statement:

```
30 P$ = HEX(12C9)
```

Alpha-variable-2 contains the ASCII character string of hexadecimal digits. HEXPACK converts each pair of ASCII characters to one byte of the corresponding binary value. As a result, alpha-variable-2 is usually twice as long as alpha-variable-1. Only certain ASCII characters constitute legal representations of hexadecimal digits. These include the characters 0 to 9 and A to F as well as the colon (:), semicolon (;), less than sign (<), equal sign (=), greater than sign (>), and question mark (?). Refer to Table 11-2 for the binary values of these characters.

If the value of alpha-variable-2 contains embedded spaces or any characters other than the ASCII characters listed in Table 11-2, the system displays an error message and terminates program execution. Since trailing spaces within the alpha-variable are not regarded as part of its value, the system does not generate an error.

**Table 11-2. 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 |

The availability of the special characters ":" to "?" (HEX(3A) - HEX(3F)) to represent hexadecimal digits A to F (1010-1111) enables the HEXPACK statement to recognize any ASCII code with a high-order 3-digit (HEX(30) - HEX (3F)) as a legitimate representation of a hexadecimal digit. Therefore, you can transform any code into an acceptable representation of a hexadecimal digit and perform operations such as packing the low-order hexadecimal digits (low-order four bits) from a string of hexadecimal digits.

*Example:*

```
:10 DIM P$2, V$4
:20 V$ = HEX(01020C09)
:30 V$ = OR ALL(30)
:40 HEXPACK P$ FROM V$
:50 PRINT HEXOF(P$)
:RUN
12C9
```

*Example:*

```
: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$: REM RIGHT-JUS
    TIFY UNPACKED HEX
:50 HEXPACK P$ FROM U1$: REM PACK HEX
:60 PRINT "HEXOF(P$) = "; HEXOF(P$)
:RUN
UNPACKED HEX? 123
HEXOF(P$) = 0000000000000123
```

*Examples of valid syntax:*

```
HEXPACK A$ FROM B$
HEXPACK STR(A$,1,3) FROM STR(B$,7)
HEXPACK A$( ) FROM B$( )
```

## HEXUNPACK

*Format:*

```
HEXUNPACK alpha-variable-1 TO alpha-variable-2
```

The HEXUNPACK statement converts the binary value of alpha-variable-1 to a string of ASCII hexadecimal characters that represent that value and stores the resulting characters in alpha-variable-2.

The HEXUNPACK statement functions similarly to the PRINT HEXOF statement. However, the HEXUNPACK statement stores the resulting character string in an alpha-variable; the PRINT HEXOF statement prints the resulting character string to a screen or printer.

The value of alpha-variable-1 is processed left to right in groups of four bits (half-byte). Each group of four bits is converted to an ASCII character that is the hexadecimal representation of the 4-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 group of 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 that results from unpacking the binary value in alpha-variable-1, the system signals an error and halts program execution. If alpha-variable-2 is longer than the converted character string, HEXUNPACK left-justifies the character string within the variable and does not modify the remaining bytes.

*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:*

```
HEXUNPACK A$ TO B$
HEXUNPACK STR(A$,5) TO STR(B$,1,4)
HEXUNPACK A$( ) TO B$( )
```

## IF...THEN

*Format:*

```
                        statement                  statement
IF condition THEN      line-number      :ELSE
                        do-group                   do-group
```

```
where:

  condition =      one or more relations sepa  rated by the logi
                   cal operators AND, OR, or XOR

  relation  =      operand operator operand

                    literal-string
    operand =       alpha-variable
                    numeric-expression

                     <
                     <=
    operator =        =
                     >=
                     >
                     <>

  THEN group =      do-group, line-number, or statement following
                     THEN

  ELSE group =      do-group or statement following ELSE

   statement =      any BASIC-2 statement, except DATA, DEFFN,
                    DEFFN', and Image (%)
```

The IF...THEN statement tests a specified condition. The programmer specifies the condition to be tested as a relation between two operands. The IF...THEN statement can be used to test the following relations. (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 |

The pair of operands being compared can be numeric or alphanumeric; however, a numeric operand cannot be compared with an alphanumeric operand. 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 binary values. (For example, the value of "A" is HEX(41) and the value of "B" is HEX(42). This makes the relation "A"<"B" true, while the relation "A">"B" is false. The character-by-character comparison continues until unequal characters are found (if there are any). The first pair of unequal characters determines the relationship between the values. If no unequal characters are found, the two values are equal.

If two alphanumeric values of different length are compared, the shorter value is extended with spaces (HEX(20)) for the comparison; making the relation "YES" = "YES " true.

*Example:*

If "ABCD" is compared to "ABC", the shorter string is padded with a space character, yielding "ABC ". The first pair of unequal characters if the letter "D" and the space. Therefore, the relation "ABCD" > "ABC" is true since the value for "D", HEX(44), is greater than the value for space, HEX(20).

An IF condition can consist of one or more relations. Relations are separated in an IF statement by the logical operators AND, OR, and XOR. Relations are processed from left to right. The logical operators act as follows:

- If AND separates two relations, the pair of relations is true if and only if both simple relations are true.

  Example: IF A>B AND B<C THEN 100
- If OR separates two relations, the pair of relations is true if and only if at least one simple relation is true.

  Example: IF A$=B$ OR A$=C$ THEN PRINT "ERROR"
- 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.

*Example: IF A<1 XOR B>0 THEN 100*

The IF...THEN statement executes the THEN group if and only if the condition tested is true. If the condition is false, execution continues with the statement following the THEN group.

An ELSE clause can follow the IF...THEN statement. When an ELSE clause is present, only one group of statements is executed. If the tested condition is true, the THEN group is executed. If the tested condition is false the ELSE group is executed. In either case, execution continues at the statement immediately following the ELSE group. ELSE can be put on the same line as the IF...THEN statement or on the following program line.

*Example:*

The following program line prints "OK" if A>B or "ERROR" if A<=B.

```
IF A>B THEN PRINT "OK": ELSE PRINT "ERROR
```

*Example:*

```
10 INPUT A,B,C
20 IF A=B THEN DO
30    C=A-B: PRINT A,B,C: A=A+1
40    ENDDO
50 ELSE DO
60    C=999: A=A+3: B=B+1
70    ENDDO
80 STOP
```

If the value entered for A is equal to B, the statements on line 30 are executed. Control then transfers to line 80. If the value entered for A is not equal to B, the statements on line 60 are executed. Execution then continues at line 80.

*Examples of valid syntax:*

```
IF A<B THEN 100
IF A$<=B$ THEN C$=A$
IF X(1)=Y*Y+Z*Z THEN GOSUB 1000
IF STR(A$,X,Y)=B$(1) THEN GOSUB '50: ELSE GOSUB '51
IF A=B AND C=D THEN DO: A=A+1: B=B+2: C=C+2: D=D+4: ENDDO
IF E<>0 OR ERR<>0 THEN PRINT "ERROR": ELSE GOTO 100
IF B$=" " XOR C$=" " THEN 10
```

## Image (%)

*Format:*

```
    % [character-string] [format-specification] ...

  where:

    character-string = a string of alphanumeric characters that does
                       not contain a "#" character

                                                                +
                           +                                    -
  format-specification=    [$] [#[,]]...[.][#]... [↑↑↑↑]
                           -                                    ++
                                                                --
```

The Image (%) statement, in conjunction with a PRINTUSING statement, provides a format-specification for formatted output. The percent sign (%), which appears as the first character in the line, identifies the Image statement. 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, refer to the discussion of the PRINTUSING statement later in this section.

An Image statement can have any printable characters inserted before and after the format-specifications. Each format-specification contains at least one digit-selector character (#) and any of the following symbols: dollar sign ($), plus signs (+, ++), minus signs (-, –), and decimal point (.). Commas (,) can 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 can 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

*Format:*

```
    INPUT [literal-string [,] ] variable [, variable] ...
```

The INPUT statement allows you to supply data during the execution of a program. Data values you enter are assigned sequentially to the variables specified in the INPUT variable list. A message can be included in the INPUT statement prompting you to enter the required data.

*Example:*

Either of the following statements enable you to enter a pair of numeric values and assign them to the variables A and B:

```
        :40 INPUT A,B
        :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 you 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 (refer to the discussion of the SELECT statement in Chapter 7).

Values you enter are assigned sequentially to the variables in the INPUT statement variable list. If more than one value is entered on a line, a comma must separate each value. Also, each value can be entered on a separate line. Several lines can be used to enter the required INPUT data. The number of lines of input is restricted by the SELECT line statement. If there is a system-detected error in the data, the values must be reentered, beginning with the erroneous value. The values that precede the error are accepted.

You can 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:*

```
:10 INPUT X
:RUN
?12.2 (RETURN)
```

*Example:*

```
:20 INPUT "X,Y",X,Y
:RUN
X,Y? 1.1, 2.3 (RETURN)
```

*Example:*

```
:20 INPUT "MORE INFORMATION" A$
:30 IF A$ = "NO" THEN 50
:40 INPUT "ADDRESS",B$
:50 END
:RUN
MORE INFORMATION? YES (RETURN)
ADDRESS? "BOSTON, MASS" (RETURN)
```

### Use of the Function Keys in INPUT Mode

Function keys can be used in conjunction with the INPUT statement. If a function key has been defined for text entry (refer to the discussion of the DEFFN' statement earlier in this chapter) and the system is awaiting input, pressing the function key causes the character string associated with that key to be entered.

*Example:*

```
:10 DEFFN'01 "HOBBITS"
:20 INPUT A$
:RUN
?
Depressing function key '01 will cause "HOBBITS" to be en-
tered.
? HOBBITS
```

If the function key is defined to call a DEFFN' subroutine (refer to the discussion of the DEFFN' statement earlier in this chapter) and the system is awaiting input, pressing 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 pressing the function key are lost and must be reentered. Subroutine return information is stored in the stack and cleared when a RETURN is executed for that subroutine. Because of this, repetitive subroutine entries with function keys should not be made unless the subroutine RETURN is always executed.

*Example:*

The following program enters and stores a series of numbers. When function key '02 is pressed, 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?11AMOUNT?    (Press function key '02)
TOTAL = 23
AMOUNT?
```

*Examples of valid syntax:*

```
INPUT X
INPUT "Value" X
INPUT "Row and Column", R,C
```

## KEYIN

*Format 1:*

```
            device-address,
  KEYIN                                alpha-variable [,,line-number]
            file#,
```

*Format 2:*

```
           device-address,
  KEYIN                               alpha-variable, line-number, line-num
                                        ber
           file#,
```

The KEYIN statement receives a single character from an input device, usually the keyboard. KEYIN provides a convenient mechanism 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# (#n). If neither a device-address nor a file-number is specified, the device currently selected for INPUT operations is used by KEYIN.

Format 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. 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 function keys. If a standard character is received, execution continues at the next statement. If a function key code is received, execution is transferred to the line specified by the line-number.

Format 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 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 keyboards have visible underline and accent operation. On these keyboards, the accent or underline code is sent preceded by a HEX(FF) code flagging the next code as a special character. The program can display the accent or underline and move the cursor back under the accent by printing a backspace code, HEX(08). The next character received is 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 underline key, a HEX(FF) code precedes the backspace code, HEX(08), when both codes are sent.

### Foreign Character Codes

Foreign characters whose codes are HEX(80) and above are sent with the Function bit set to distinguish them from text atoms, which are codes that represent BASIC-2 keywords. Foreign character codes cause KEYIN to branch to the line-number specified for the function keys.

*Examples of valid syntax:*

*Format 1*

```
KEYIN A$
KEYIN #3, A$,,100
KEYIN /002, A$
```

*Format 2*

```
KEYIN A$, 100, 200
KEYIN #3, A$, 100, 200
```

## LET

*Format:*

```
[LET] numeric-variable [,numeric-variable] ... = numeric-expres
                                                  sion

                          or

[LET] alpha-variable [,alpha-variable] ... = alpha-expression
```

The LET (assignment) statement evaluates the expression on the right side of the equal sign and assigns the result to the variable or variables specified on the left side of the equal sign. Refer to Chapter 4 for a discussion of numeric-expressions and to Chapter 5 for a discussion of alpha-expressions.

If multiple variables appear to the left of the equal sign, a comma must separate each variable. If a numeric value is assigned to an alphanumeric-variable or if an alphanumeric value is assigned to a numeric-variable, an error results. The word LET is optional in an assignment statement.

*Examples of valid syntax:*

```
LET J = 3
X,Y,Z = P+15/2 + SIN(P-2.5)
A$(3) = B$
C$,D$(2) = "ABCDE"
LET A$ = HEX(0000)STR(B$,1,3) = C$
A$, B$, STR(C$(1),3) = STR(E$,N,M)A$() = A$ & B$
C$ = A$ AND B$ OR D$
```

## LINPUT

*Format:*

```
LINPUT [literal-string [,]] [?] [-] alpha-variable
```

The LINPUT (LINe-inPUT) statement allows you 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 cannot exceed 480 characters.

When a LINPUT statement is executed, the following events take place:

- The optional message (literal-string) in the LINPUT statement is displayed, followed by a space character.
- The entire current contents (including trailing spaces) of the specified alpha-variable are recalled and displayed on the screen beginning immediately after the current cursor position.
- If a minus sign (-) is specified immediately before the alpha-variable, each character of the alpha-variable is underlined in the display.
- The cursor is positioned at the first character of the recalled value.
- If a question mark (?) is included in the LINPUT statement, entry starts in Text Entry mode; otherwise, entry starts in Edit mode.

You can 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. 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 pressing RETURN.

*Example:*

```
:10 DIM A$5
:20 A$ = "ABC"
:30 LINPUT "ENTER VALUE"A$
:RUN
ENTER VALUE ABC
```

During a LINPUT operation, cursor movement is restricted to the region of the screen occupied by the value of the receiving alpha-variable. Cursor movement beyond the limits of the alpha-variable is inhibited. (If you attempt to enter data beyond the end of the LINPUT field, the 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 255 bytes of data originally stored in the alpha-variable can be recovered before RETURN is pressed. Recovery of originally stored data can 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 pressing line ERASE and then RECALL. This procedure must be performed before pressing RETURN. Once RETURN is pressed, 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 pressed, the underline is removed from each character in the LINPUT field. The underline feature is useful for generating pseudospace characters that can identify fields to be filled.

If a question mark (?) is included in the LINPUT statement, the LINPUT operation begins in Text Entry mode rather than Edit mode. In Text Entry mode the function keys are active for accessing text definitions or marked subroutines (refer to DEFFN' in this chapter). In Edit mode the function keys are available for edit operations. Text Entry mode is indicated by a steady cursor; in Edit mode the cursor blinks. LINPUT ? is useful when function keys are used to control field operations because the Edit key need not be pressed prior to pressing the function key.

A function key can be pressed in response to a LINPUT request. If the function key is defined for text entry, its associated text string is displayed and inserted into the alpha-variable. If the 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. (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 when the subroutine RETURN statement 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 typing 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 you enter it. 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$
:RUNMESSAGE
:20 LINPUT "MESSAGE", -A$:RUN
MESSAGE
(Blinking cursor at field beginning)
```

*Examples of valid syntax:*

```
LINPUT A$
LINPUT "Filename" - F$
LINPUT ? C$()
```

## MAT COPY

*Format:*

```
MAT COPY [-] source-alpha-variable TO [-] output-alpha-variable
```

The MAT COPY statement constructs 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 an STR function) to the output-variable (or a portion of the output-variable specified with an STR function).

MAT COPY treats each variable 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, MAT COPY fills the remaining bytes with spaces. The source- and output-variables can be the same variable.

You can combine the parameters of the MAT COPY statement to produce the following results:

- If no minus signs precede the variables, MAT COPY transfers the data in the order specified in the source-variable and left-justifies the data in the output-variable.
- If a minus sign precedes the source-variable, MAT COPY transfers data in reverse order and left-justifies the data in the output-variable.
- If a minus sign precedes the output-variable, MAT COPY transfers the data in reverse order and right-justifies the data in the output-variable.
- If a minus sign precedes the source-variable and the output-variable, MAT COPY transfers the data in the order specified in the source-variable and right-justifies the data in the output-variable.

The following examples assume the following dimensions for the input-variable A$() and the output-variable B$():

```
DIM A$(1,5)1, B$(1,7)1
```

These examples also assume the following value for A$().

```
A$( )=  A    B    C    D    E
```

*Example: Duplication of Alpha-Variable, Left-Justified*

```
MAT COPY A$( ) TO B$( )
B$( )=   A    N    C    D    E    space    space
```

*Example: Reversal of Alpha-Variable, Right-Justified*

```
MAT COPY A$( ) TO -B$( )
B$( )=   space   space   E   D   C   B   A
```

*Example: Reversal of Alpha-Variable, Left-Justified*

```
MAT COPY -A$( ) TO B$( )
B$( )=   E   D   C   B   A   space   space
```

*Example: Duplication of Alpha-Variable, Right-Justified*

```
MAT COPY -A$( ) TO -B$( )
B$( )=   space   space   A   B   C   D   E
```

*Examples of valid syntax:*

```
MAT COPY A$( ) TO A$( )
MAT COPY -W$ TO B$( )
MAT COPY STR(A$( ),10,50) TO -W$
MAT COPY -STR(W$,5,10) TO -STR(A$,20)
```

## MAT MOVE

*Format (Simple):*

```
   MAT MOVE move-array [,locator-array] [,n] TO receiver-array

 where:

                          alpha-array [(x[,y])]
   move-array    =
                          numeric-array


                          alpha-array
  locator-array =
                          alpha-array-element

             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.

                        alpha-array-element [(x[,y])]
 receiver-array =        numeric-array-element
                        alpha-array [(x[,y])]
                         numeric-array

          (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 charac
                         ters in the field.  If y is not specified,
                         its value is assumed to equal the number of
                         remaining characters in the element.
```

**In its simple form, the MAT MOVE statement transfers data element by element from one array to another. Optionally, a complex form of the MAT MOVE statement converts numeric data from Wang internal numeric format to alphanumeric data in sort format and restores 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.**

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.

*Example:*

Assume that two arrays A$( ) and B$( ) are defined.

```
        10 DIM A$(3), B$(3)
```

The following two statements are equilavent:

```
        20 MAT MOVE A$( ) TO B$( )
        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. Subscripts in the locator-array are binary values that are two bytes in length. If the move-array is a 2-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 1-dimensional array, the subscript is interpreted as a 2-byte binary value (from 1 to 65535) identifying an element. Arrays with more than two dimensions are treated like 1-dimensional arrays; the locator-array element is not a subscript but is the element number relative to the beginning of the array.

You can 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.

*Example:*

The following statement specifies that a 5-character substring starting at the tenth character position of each element of A$( ) is to be moved to a 5-character field starting at the fourth character position in each element of B$( ).

```
        20 MAT MOVE A$( )(10,5) TO B$( )(4,5)
```

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:

- The end of the move-array is reached for direct data transfer
- The end of the array of subscripts (locator-array) is reached for indirect data transfer
- A binary 0000 is found in the array of subscripts
- The number of elements specified by the counter-variable (the numeric-scalar-variable preceding TO) is moved
- The receiver-array is 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 alpha- or numeric-array of up to 255 dimensions |
| Locator-Array – | An alpha-array with elements of length 2 |
| Receiver-Array – | Any alpha- or numeric-array of up to 255 dimensions |

*Example:*

Suppose the following array of data D$( ) and an associated array of subscripts S$( ) exist in memory. The problem is to move the data from D$( ) to E$( ) in the order specified by S$( ).

$$
\texttt{D\$( )} = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & B & A & C & D \\ 2 & C & B & E & A \\ 3 & A & F & A & E \end{array}
$$

$$
\texttt{S\$( )} = \begin{array}{cccc} 0102 & 0303 & 0301 & 0204 \\ 0202 & 0101 & 0103 & 0201 \\ 0104 & 0203 & 0304 & 0302 \end{array}
$$

Execution of the statements

```
100 DIM E$(3,4)1
110 MAT MOVE D$( ), S$( ) TO E$( )
```

produces an array E$( ) with the following values:

$$
\texttt{E\$( )} = \begin{array}{cccc} A & A & A & A \\ B & B & C & C \\ D & E & E & F \end{array}
$$

*Examples of valid syntax:*

```
MAT MOVE A$( ) TO B$(1,1)
MAT MOVE A$( )(5,3), L$( ) TO B$( )
MAT MOVE A( ), L$(3,1), X TO B(Y)
MAT MOVE A( ), L$( ) TO B( )
MAT MOVE A$( ), L$(X) TO B$(E)(3,4)
```

## MAT SEARCH

*Format:*

```
                                         <
                 search-variable,       <=     alpha-variable
   MAT SEARCH                            =
                 search-literal,        >=     literal-string
                                         >
                                         <>

      TO pointer-variable [STEP s]

  where:

     search-variable = an alpha-variable containing the text to be
                       searched

      search-literal = a literal string representing the text to be
                       searched

    pointer-variable = an alpha-variable containing pointers to oc
                     currences of the desired character string in
                     the search-variable or search-literal

                   s = a numeric expression such that: 1 ≤ s < 65536
```

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 that satisfy the given relationship are placed into the pointer-variable in the order in which they are found. Each position is stored as a 2-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 an STR function is used) or search-literal. If any space remains 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.

Trailing spaces are not usually considered to be part of alpha values. If trailing spaces or values that end in HEX(20) are checked, the STR function specifies the exact number of characters to be checked.

*Example:*

The following statement searches for substrings in A$( ) that are five characters in length and equal to the first five characters of Z$.

```
100 MAT SEARCH A$( ), = STR(Z$, 1, 5) TO B$( )
```

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, a literal string saves assigning the data to another array and produces code that is more self-explanatory than if an array is used.

*Example:*

The following example accepts a 3-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.

```
10 DIM A$3, C$2
20 LINPUT "DEVICE ADDRESS", A$
30 MAT SEARCH "D10D20D30D50D60D70", = STR(A$,,3) TO C$ STEP 3
40 ON 1 + VAL(C$,2)/3 SELECT #N/D10; #N/D20; #N/D30;#N/D50;
   #N/D60; #N/D70 : ELSE GOTO 50
     .
     .
     .
```

Usually, 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 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.

*Example:*

Assume that the array G$( ) is defined with the following values (all occurrences of the value A must be found in G$( )):

|  |  | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
|  | 1 | B | A | C | D |
| G$( )= | 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:

```
                    1      2      3      4      5      6

                1   0002   0008   0009   000B   0000   FFFF
       P$( )=

                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$( ), counting from the first character (element 1,1). Element boundaries are ignored in this process. The value HEX(0000) is inserted in the element following the last valid pointer stored in P$( ). The remaining bytes of P$( ) are filled with HEX(FFFF) because P$( ) is initialized to this value in Line 100 prior to beginning the MAT SEARCH operation.

*Examples of valid syntax:*

```
DIM A$100, B$5, S$(10)3, W$(50)2, W1$50
MAT SEARCH S$( ), = "XX" TO W1$
MAT SEARCH A$, = HEX(0102) TO W1$
MAT SEARCH STR(S$( ),5,20), < B$ TO W1$ STEP 5
MAT SEARCH A$, <> STR(S$( ),18) TO STR(W$( ), 30, 20)
MAT SEARCH "LDASTAADC", = STR(Z$,1,3) TO B$( ) STEP N
MAT SEARCH HEX(0020413749), = B$ TO C$
```

## NEXT

*Format:*

```
   NEXT counter-variable [,counter-variable] ...

 where:

   counter-variable = numeric-scalar-variable
```

The NEXT statement constructs a loop, together with the FOR statement. The NEXT statement marks the end of the loop and must be the last statement in the loop. The counter-variable(s) specified in the NEXT statement must correspond to the counter-variable(s) defined in one or more preceding FOR statements. The NEXT statement controls such loop operations as incrementing the counter, comparing it with the exit value, and reexecuting the loop.

Each time through the loop, the NEXT statement adds the STEP value to the current value of the counter and compares the result with the exit-value specified in the corresponding FOR statement. The type of comparison made is determined by the sign of the STEP value. (If no STEP value is explicitly specified in the FOR statement, the system uses a default value of +1.) If the STEP value is positive, the counter is tested to determine whether or not it exceeds the exit value. If the STEP value is negative, the counter is tested to determine whether or not it is less than the exit value. If the tested condition is not met, the NEXT statement reexecutes the loop. If the tested condition is met, the system terminates the loop and resumes program execution with the statement immediately following NEXT or with the first statement of the next outer loop.

You can end nested loops with a single NEXT statement. For example, the following statements are equivalent:

```
NEXT I,J,K
NEXT I: NEXT J: NEXT K
```

In Immediate mode, a NEXT statement and its corresponding FOR statement must both appear in the same line.

*Examples of valid syntax:*

```
FOR M = 2 TO N-1 STEP 30: J(M) = X(M)*2NEXT M
FOR X = 8 TO 16 STEP 4
FOR A = 2 TO 6 STEP 2
LET B(A,X) = B(X,A)
Nested Loops
NEXT A
NEXT X
```

## ON GOTO/GOSUB

*Format:*

```
     expression        GOSUB
ON                                [, [line-number]] ... line-number
     alpha-variable    GOTO

                  statement
     :ELSE
                  do-group
```

The ON statement is a computed or conditional GOTO or GOSUB statement. 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
```

transfers 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. The ELSE clause contains either a single statement or a group of statements enclosed in DO and ENDDDO. Refer to DO in this chapter for a discussion of do-groups.

For example, the following statement branches to Line 10, 20, or 40 if I = 1, 2, or 4, respectively. For all other values of I, the ELSE clause causes a branch to the subroutine beginning at Line 1000.

```
100 ON I GOTO 10,20,,40: ELSE GOSUB 1000
```

The branch value can 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 following statement causes a transfer to Line 200:

```
50 ON A$ GOTO 100, 200, 300
```

*Examples of valid syntax:*

```
ON I GOSUB 10, 15, 100, 900 :ELSE GOSUB'20
ON 3*J-1 GOSUB 100, 200, 300, 400
ON X GOTO,,200,300
ON VAL(C$) GOSUB 10,20,30: ELSE DO: PRINT "Illegal": GOSUB
  '50: ENDDO
```

## PACK

*Format:*

```
                                          numeric-array
   PACK (image) alpha-variable FROM
                                          numeric-expression

 where:

              +  [#] ... [.][#] ... [↑↑↑↑]
   image =    -

             alpha-variable containing image

   length of image < 255
```

The PACK statement packs numeric values into an alphanumeric variable or array. Using the PACK statement reduces the storage requirements for large amounts of numeric data where only a few significant digits are required.

Numeric values are formatted into packed decimal form according to the specified image and then stored sequentially in the alpha-variable. Array-variables are filled sequentially row by row from the beginning of the first array-element until all numeric data is stored. If the receiving alpha-variable is not large enough to store all the numeric values to be packed, an error results.

The image consists of digit-selector characters (#) to signify digits and, optionally, plus signs (+), minus signs (-), decimal points (.), and up arrows (↑) 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., #.###↑↑↑↑)*

The PACK statement packs numeric values according to the following rules:

- The PACK statement packs two digits per byte and stores one digit for each digit-selector character (#) in the image.
- PACK left-justifies the packed numeric value in the alpha-variable. If specified, plus or minus signs (+ or -) occupy the high-order half-byte, followed by the number in packed decimal format. If specified, the exponent occupies the two low-order half-bytes.

- A single hex digit represents 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
- If no sign is specified, PACK stores the absolute value of the number and assumes the sign of the exponent to be plus (+).
- PACK does not store the decimal point. When unpacking the data, you must specify the decimal point position in the image. Refer to the discussion of the UNPACK statement in this section.
- If the image is expressed in fixed-point format, PACK edits the numeric value as a fixed-point number, truncating or extending with zeros any fraction and inserting leading zeros for nonsignificant integer digits according to the image specification. If the number of integer digits exceeds the format-specification, an error results.
- If the image is expressed in exponential format, PACK edits the numeric value as an exponential number. The value is scaled as specified by the image; there are no leading zeros.

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 one and one-half bytes, but two bytes are required. In such cases, the value of the unused half-byte (low-order half-byte) is not altered by the PACK operation.

*Examples of storage requirements:*

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

*Examples of valid syntax:*

```
PACK (###) A$ FROM X
PACK (####) STR(A$,4,2) FROM N(1)
PACK (##.##) A1$( ) FROM X,Y,N( ),M( )
F$ = "+##.##": PACK (F$) B$ FROM N2
C$(2) = "-#.###↑↑↑↑": PACK (C$(2)) A$( ) FROM N( )
```

## $PACK

*Format:*

```
                   F     alpha-variable
 $PACK     (             =                          )     alpha-variable
                   D     literal-string

              array                         ,array
   FROM      numeric-expression             ,numeric-expression...
             literal-string                 ,literal-string
              variable                      ,variable
```

The $PACK statement stores 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. If the buffer is too small to hold all the values specified, an error results. The buffer is the alpha-variable immediately preceding the word FROM.

*Example:*

In the following statement, the variable B$( ) before the word FROM represents the buffer that receives data. The variables X, A$, Y( ), B$( ) supply the data.

```
$PACK B$( ) FROM X, A$, Y( ), B$( )
```

The following three forms of $PACK are available:

- Delimiter Form (specified by the D= parameter)
- Field Form (specified by the F= parameter)
- Internal Form (assumed if neither the D= nor the F= parameter is specified)

### 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 are stored.

The first two bytes of the alpha-variable or literal string following the D= parameter serve as the delimiter specification.

*Example:*

In the following statement, A$ is the delimiter specification variable.

```
$PACK (D = A$) B$( ) FROM X( )
```

In the following statement, HEX(002C) is the delimiter specification expressed as a hex literal.

```
$PACK (D = HEX(002C)) A$( ) FROM N( )
```

The first byte of the delimiter specification must be HEX(00), HEX(01), HEX(02), or HEX(03); however, $PACK ignores this byte and uses it 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:*

```
  Alphanumeric Values

      C    C  ...  C
where:

      C  =  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. In effect, 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.

```
  Numeric Values
```

*Format 1:*

```
       s   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:*

The following 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 data 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 are stored.

The alpha-variable or literal string following the F= parameter contains the field specifications for the buffer. Each field specification consists of two bytes. The first byte specifies the type of field; the second byte specifies the field width (i.e., the number of characters in that field).

*Example:*

The following two examples illustrate that the field specifications for a buffer can either be contained within an alphanumeric-variable or expressed as a hexadecimal literal string:

```
$PACK (F = F$) B$( ) FROM X( )
$PACK (F = HEX(1008)) B$( ) FROM X( )
```

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 to 6 identifies a unique numeric format. Refer to Table 11-3. 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 results if a numeric value is packed into an alphanumeric field or if an alphanumeric value is packed into a numeric field.

**Table 11-3. 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 |
| 7d0y | packed decimal with binary overflow format |
| 8d0y | signed binary format |
| 9d0y | unsigned binary format |
| A0xx | alphanumeric field |
| A1xx | compressed alphanumeric format |

where:

| | | |
|---|---|---|
| xx | = | field width in binary (xx > 0) |
| y | = | field width in binary (0 < y < =4) |
| d | = | implied decimal position in binary |

You must supply a separate field specification for every variable or array in the variable list. All elements in an array use the field specification specified for that array.

*Example:*

The following statement requires three field specifications:

```
$PACK (F = F$) B$( ) FROM A$, B( ), C$

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$
```

You can also mnemonically define a field specification in a $FORMAT statement. $FORMAT permits the use of simple mnemonic codes rather than hex codes to specify field formats. Refer to the discussion of the $FORMAT statement in this section.

*Example:*

The field specifications defined for F$ above 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:*

```
Alphanumeric Fields (A0xx)

    C      C       ...         C

  where:

    C  =  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.

```
  Numeric Fields
```

*ASCII free format (10xx)*

```
    s      d      d      ...      d   .  d     ...      d

    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 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, the following rule applies: If the value is shorter than the length of the field, leading zeros are inserted; if the value is too long, an error results.

*ASCII integer format (2dxx)*

```
      s  d  d   ...  d

  where:

      s  =  sign (ASCII + or -), required
      d  =  ASCII digit
```

*IBM display format (3dxx)*

```
      Fd   Fd   ...  Fd  sd

  where:

      s  =  sign (C = +, D = -)
      d  =  digit (0-9)
```

*IBM USASCII-8 format (4dxx)*

```
      5d    5d       ...      5d    sd

  where:

      s  =  sign (A = +, B = -)
      d  =  digit (0-9)
```

*IBM packed decimal format (5dxx)*

```
      dd    dd       ...      dd    ds

  where:

      s  =  sign (C = +, D = -)
      d  =  digit (0-9)
```

*Unsigned packed decimal format (6dxx)*

```
      dd   dd       ...      dd   dd

  where:

      d  =  digit (0-9)
```

The above formats are shown in hexadecimal notation.

Decimal arithmetic can be performed on unsigned packed decimal numbers. (Refer to the discussion of the DAC and DSC operators in Chapter 7.)

*Example:*

The following example assumes that the buffer B$ has three fields (one alpha and two numeric), each five characters long. $PACK packs the values of A$, X, and Y 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$, X, Y
:50 PRINT "B$ = "; B$
:RUN

B$ = "ABC  -0012 1.23"
```

*Example:*

The following examples assume that X = 12.345.

| Statements | Results |
|---|---|
| :10 F$ = HEX(100A)<br>:20 $PACK (F = F$) B$ FROM X | B$ = "12.345" |
| :10 F$ = HEX(240A)<br>:20 $PACK (F = F$) B$ FROM X | B$ = "+000123450"<br>Note that there is an implied decimal point four digits from the right end of the field. |
| :10 F$ = HEX(3305)<br>:20 $PACK (F = F$) B$ FROM X | B$ = HEX(F1F2F3F4C5) |
| :10 F$ = HEX(4206)<br>:20 $PACK (F = F$) B$ FROM X | B$ = HEX(5050515253A4) |
| :10 F$ = HEX(5506)<br>:20 $PACK (F = F$) B$ FROM X | B$ = HEX(00001234500C) |
| :10 F$ = HEX(2304)<br>:20 $PACK (F = F$) B$ FROM X | Results in an error because the receiving field is too small to hold the value. |

*Packed Decimal with Binary Overflow Format (7d0y)*

The Packed Decimal with Binary Overflow Format is used to pack numeric values. The number is stored in packed decimal format (same as type 5dxx) if it will fit in the specified field. The maximum field length allowed is 4. If the number is too large to be stored in packed decimal, it is converted to binary and stored in a binary format. If the binary number is still too large for the field, an error (ERROR X71) results. The last hexdigit of the packed value identifies the value as being either packed decimal or binary. If the last hexdigit is hex(C-F), the value is packed decimal. If the value is hex(0-B), the value is binary.

When a number is stored in binary format, it is first converted to a binary value the same length as the field. The number is too large to be packed if the upper hexdigit is greater than 5. The number is then shifted left by one hexdigit (4-bits). The low 3-bits of what was the upper hexdigit now become the upper 3-bits of the low hexdigit of the value. The lowest bit of the last hexdigit stores the sign of the value: zero for nonnegative and one for negative values.

*Example:*

```
10 X = 1234567
20 $PACK (F=HEX(7004)) D$ FROM X
30 $PACK (F=HEX(7003)) B$ FROM X

Results in D$ = HEX(12 34 56 7C) and
B$ = HEX(2D 68 72)
```

*Signed Binary Format (8d0y)*

The Signed Binary Format is used to pack numeric values. The maximum field length allowed is 4. The value is converted to a binary value the same length as the field. If the binary number is too large for the field, an error (ERROR X71) results. Negative values are stored in 2's complement. Thus, the highest bit in the field can be used to determine the sign of the value: the bit is zero for nonnegative values and one for negative values.

*Example:*

```
10 X = 1234567
20 $PACK (F=HEX(8004)) P$ FROM X
30 $PACK (F=HEX(8004)) N$ FROM -X

Results in P$ = HEX(00 2D 68 72) and
N$ = HEX(FF D2 97 8E)
```

*Unsigned Binary Format (9d0y)*

The Unsigned Binary Format is used to pack numeric values. The maximum field length allowed is 4. The sign of the number is ignored. The absolute value is converted to a binary value the same length as the field. If the binary number is too large for the field, an error (ERROR X71) results.

*Example:*

```
10 X = 1234567
20 $PACK (F=HEX(8004)) P$ FROM X
30 $PACK (F=HEX(8004)) N$ FROM -X

Results in P$ = HEX(00 2D 68 72) and
N$ = HEX(00 2D 68 72)
```

*Compressed Alphanumeric Format (A1xx)*

The Compressed Alphanumeric Format provides a means to more compactly store characters with ASCII values hex(20) through hex(5F). These include the uppercase characters, digits, space, and certain symbols. Other characters in the string to be packed will cause an error (ERROR X71). The characters in the string to be packed are converted to 6-bit values. Specifically, characters hex(20) through hex(5F) are converted to the 6-bit values hex(00) through hex(3F). Then, the 6-bit characters are stored left-justified in the pack field. Thus, each four characters in the string to be packed is stored as three bytes in the pack field.

If the compressed value is shorter than the length of the field, the value is is left-justified in the field and the remainder of the field is filled with 0 bits (effectively, the original value is padded with space characters on the right). If the compressed value is too long, it is truncated to fit within the field.

*Example:*

```
10 S$="ABCDEFGH"
20 $PACK (F=HEX(A103)) P$ FROM S$

Results in P$ = HEX(86 28 E4)
```

## The Internal Form of the $PACK Statement

The internal form of $PACK stores data in the standard Wang 2200 disk record format. 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.

```
80  01  SOV  value  SOV  value  SOV  value  EOB

  16  16

control bytes
```

The 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.

```
                                binary count (number of bytes in
                                 value)

                                0 if numeric, 1 if alphanumeric
```

The EOB (End-Of-Block, HEX(FD)) character indicates the end of valid data in the record.

*Data format:*

```
Alphanumeric Values

    C    C ...   C

where:

    C  =  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.**

```
Numeric Values
```

*Numeric values are stored in Wang Internal Numeric Format.*

```
    s  e  e  d  d  d  d  d  d  d  d  d  d  d  d  d (8 bytes)
        L   H

where:

      s =   sign:
           0 if mantissa +, exponent +
           1 if mantissa -, exponent +8 if mantissa +, exponent -
           9 if mantissa -, exponent -
    e e =  exponent (2 digits)L H
      d =  mantissa digit (always 13)
```

**Numeric values are normalized (i.e., leading zeros are eliminated). All digits must be BCD.**

*Example:*

**The following routine packs the values of A$ and X into a buffer B$ in internal format:**

```
:10 DIM B$20, A$5
:20 A$ = "ABC" :X = 123.45
:30 $PACK B$ FROM A$, X
```

**The resulting buffer appears as follows:**

```
B$ = 80 01 85 41 42 43 20 20 08 02 01 23 45 00 00 00 00 FD 20
20
           SOV   value of A$ SOV      value of X      EOB
```

*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$
```

## PRINT

*Format:*

```
                                  ,
   PRINT [print-element]              [print-element]   ...
                                  ;
```

```
 where:

                          alpha-variable
                          literal-string
                          numeric-expression
   print-element =        AT function
                          BOX function
                          HEXOF function
                          TAB function
```

The PRINT statement prints the values of the specified print–element(s) on a designated output device in a system–defined format. The PRINT statement can contain alphanumeric and numeric print–elements, as well as the PRINT functions AT, BOX, HEXOF, and TAB. These functions are described under separate headings later in this chapter.

In Program mode, PRINT outputs to the output device currently selected for PRINT operations. In Immediate mode, PRINT outputs to the currently selected Console Output device. The use of the Console Output device for Immediate mode PRINT output is a debugging feature that enables you to halt program execution and examine the results of Immediate mode PRINT statements on the screen while programmed PRINT output is selected to a printer. (Refer to Chapter 8 for a discussion of selecting PRINT and CO devices.)

### Alphanumeric Print–Elements

An alphanumeric print–element can be a literal string, a HEX literal, or an alpha–variable. A literal string is printed exactly as it appears within the quotation marks, including trailing spaces. The quotation marks are not printed.

*Example:*

```
:10 PRINT "ABCD"
:RUN
ABCD
```

HEX literals can specify special characters not found on the keyboard, execute system control codes, or initiate special character graphics and attributes. 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 an STR( ) function.

*Example:*

```
:10 A$ = "ABC"
:20 PRINT A$; "DEF"
:30 PRINT STR(A$,1,5); "DEF"
:RUN
ABCDEFABC  DEF
```

In Line 30, two trailing spaces from A$ are printed since they are included within the 5–character range defined by the STR( ) function.

## Numeric Print–Elements

Numeric print–elements can be numeric–expressions, numeric–variables, or constants. If a numeric–expression is specified, the PRINT statement evaluates the expression and prints the result. (The PRINT statement never alters the value of a variable.) Depending upon its magnitude, a numeric value is printed in either standard numeric format or scientific notation format. A numeric value is always printed with a leading sign or space and a trailing space. A leading minus sign is printed for negative values, and a leading space is printed for nonnegative values.

A numeric value is printed in *standard format* (i.e., leading minus sign or blank, one to 13 digits plus decimal point, and a trailing space) if it satisfies either of the following conditions:

- The absolute value is in the range 0.1 ≤ value < 10+13
- The absolute value is less than 0.1, but can be expressed in 13 or fewer digits

Leading and trailing zeros are ignored, and the decimal point is not printed if the value is an integer. Numeric values printed in standard 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, 13 digits, a decimal point, and a trailing blank).

*Example:*

```
:10 A = .005: B = 9999999999999: C = -55
:20 PRINT A
:30 PRINT B:40 PRINT C
:RUN.005
9999999999999
-55
```

A numeric value is printed in *scientific notation format* if the absolute value does not correspond to either of the conditions for printing in standard 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 2–digit exponent, and a trailing space. Trailing zeros 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 plus sign (+) rather than a space).

*Example:*

```
:10 A = -2.56E18: B = 1E13
:20 PRINT A
:30 PRINT B
:40 PRINT 275634*112913534
:50 PRINT 5/75
:RUN
-2.56000000E+18
+1.00000000E+13
+3.11228090E+13
+6.66666666E-02
```

***Note:*** *A maximum of nine significant digits is 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 with a PRINTUSING statement.*

The two factors that 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, you cannot control the format of printed output generated with the PRINT statement. If greater format control is desired, you can explicitly define the output format of values with the PRINTUSING statement.

### Print–Element Separators

A comma or a semicolon must separate multiple print–elements in a PRINT statement. Both the comma and the semicolon can also suppress the carriage return. The comma can also function as a format control character.

The semicolon can separate successive print–elements. The semicolon does not insert additional spaces 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 are always printed with a leading minus sign or space and a trailing space.

*Example:*

```
:10 A = 1234: B = 5678
:20 PRINT "ABC";"DEF"
:30 PRINT A;B
:40 PRINT A;"ABC";B;"DEF"
:RUN
ABCDEF
1234  5678
1234 ABC 5678 DEF
```

In Line 20, the pair of alphanumeric values are printed without any intervening spaces, while the numeric values in Lines 30 and 40 are each accompanied by leading and trailing spaces.

A semicolon following the last print–element in a PRINT statement suppresses the carriage return usually generated by the system at the end of the PRINT line. For example:

```
:10 PRINT "ABC";
:20 PRINT "DEF":RUN
ABCDEF
```

In general, a value that 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. However, the character strings defined in HEX literals and HEXOF functions are split if they overlap from one line to the next.

### Using the Comma as an Element Separator and Format Control Character

The comma can separate successive print–elements. The comma also functions as a control character in a PRINT statement. A comma between two print–elements signals to the system to print the values 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 24 x 80 screen, for example, is divided into five zones (0 to 15, 16 to 31, 32 to 47, 48 to 63, and 64 to 79).

A comma preceding a print–element signals that the value is printed starting at the beginning of the next zone. If the 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. 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.

*Example:*

```
:10 PRINT 12*2,"ABC"
:RUN
24              ABC
```

The PRINT statement usually issues a carriage return after the last print–element is output. You can use a comma 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.

*Example:*

```
:10 N = 50: P = 100
:20 PRINT "N=";N,
:30 PRINT "P=";P
:RUN
N= 50           P= 100
```

The trailing comma at the end of Line 20 suppresses the carriage return that is usually issued at that point. As a result, the output from Line 30 appears in the next zone on the same 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.

*Example:*

```
:10 FOR I = 1 TO 8
:20 PRINT I↑2,
:30 NEXT I
:RUN
1 4  9 16  25
36 9   64
```

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 place print–elements more than one zone apart. Each comma shifts the succeeding print–element one zone to the right in the output line. Spaces are printed to position the 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.

A PRINT statement with no print–elements or element separators issues a carriage return and advances the cursor or paper one line.

## Reducing the Rate of PRINT Output

The length of time during which PRINT output is displayed on the screen can be extended by invoking a pause with a SELECT P statement. SELECT P permits you to select a pause ranging from 1/6 second to 1.5 seconds, in increments of 1/6 second (refer to Chapter 7).

### Using 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 often convenient to terminate an output line before it is completely filled. The system provides the following means for issuing carriage returns:

- A carriage return is automatically issued when the last print–element in a PRINT statement is 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 usually should perform a line feed to advance to the next output line. Some output devices, such as printers and output writers, automatically perform a line feed whenever they receive a carriage return signalling line termination. You must explicitly instruct other devices, such as the screen, to perform a line feed after each carriage return by using a line feed code (HEX(0A)).

The device type in the output device address instructs the system whether or not a line feed is issued following each carriage return for a particular device. The device–type is the first hexadecimal digit in a 3–digit device address. In the device address /005, for example, the device type is 0; in the device–address /215, the device–type is 2. In general, output device addresses have one of three device–types: 0, 2, and 7. Each device–type has a particular significance.

Device Type 0 —Instructs the system to automatically issue a line feed code (HEX (0A)) following each carriage return. Type 0 is gener ally used with the screen.

Device Type 2 —Instructs the system to issue a null character (HEX(00)) instead of a line feed code following each carriage return. Type 2 is generally used with printers and output writers.

Device Type 7 —Instructs the system to issue no extra characters following each carriage return. (For most applications, device type 7 is equivalent to device type 2.)

To avoid possible printing problems, the device type must match the output device. For example, using device type 2 with the screen (e.g., SELECT PRINT /205) can create printing problems. Since device type 2 does not automatically output a line feed and the screen does not generate its own, the cursor is not advanced to the next screen line following a carriage return. A subsequent PRINT statement, therefore, prints its output on the same screen line, overwriting the previous output line. Similarly, using device type 0 with a printer (e.g., SELECT PRINT /015) produces double–spaced output. In addition to the line feed supplied by the printer, device type 0 automatically issues a line feed code after each carriage return.

## Line Width

You define the maximum width of an output line for each output device in a SELECT statement. The line width is enclosed in parentheses immediately following the output device address. For example, the following statement selects the line printer for PRINT operations and sets a maximum line width of 132 columns.

```
10 SELECT PRINT/215(132)
```

As a PRINT line is output, the system maintains a column count. When the count exceeds 132, a carriage return is automatically issued, which terminates the line. However, if the last print–element in a PRINT statement is not followed by trailing punctuation, or if a carriage return code (HEX(0D)) is encountered in a HEX literal, the line terminates before the entire 132 characters are output.

The line width is selected independently for each class of output operations. The same output device can receive a different line width for different operations.

*Example:*

```
50 SELECT PRINT/215(132), LIST/215(80)
```

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 equals 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 case, 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. (The occurrence of an automatic line feed when the carriage width of a device is exceeded is device–dependent. Most printers issue an automatic line feed; however, the screen wraps around on the same line.) When you use a line width of zero to manipulate the screen cursor, it permits you to move the cursor without updating the column count and to explicitly control when and where a carriage return is issued. (Because no column count is maintained, the TAB and AT functions cannot be used with a line width of zero.)

## PRINT AT

*Format:*

```
    AT (row, column [,[erase-count]])

  where:

    row, column , erase-count  =  numeric-expressions
```

The AT function, which is legal only within a PRINT statement, positions the cursor at a specified row and column of the screen and erases a specified number of characters from that point onward. The position of the cursor at the conclusion of the PRINT AT operation defines the starting position for subsequently displayed characters. You specify the cursor location with the row and column parameters. Rows are numbered 0 to m, where m is one less than the number of lines on the screen. Columns are numbered 0 to n, where n is one less than the number of character positions in a line. (If the line width is selected to 0, the AT function cannot be used.)

For example, the following statement displays the value of A$ starting at Row 3, Column 5, and the value of X starting at Row 3, Column 30:

```
      PRINT AT (3,5); A$; AT(3,30); X
```

The erase–count parameter specifies the number of characters to be erased from the screen. After the AT function positions the cursor according to the row and column parameters, the system erases the specified number of characters, beginning with the character located at the cursor position. If the number of characters to be erased exceeds the number of characters on the current screen line, characters on the next line are also erased.

For example, the following statement positions the cursor at Row 3, Column 32, and erases the characters in Columns 32 and 33.

```
PRINT AT (3,32,2);
```

If a comma follows the column parameter and the erase–count parameter is omitted, PRINT AT erases the screen from the specified position to the end of the screen.

For example, assuming SELECT LINE = 24, the following statement erases Lines 8 through 24 from the screen:

```
PRINT AT(8,0,)
```

*Examples of valid syntax:*

```
PRINT AT(5,10); X
PRINT AT(X,Y); A$; AT(X+1,Z); B$
```

## PRINT BOX

*Format:*

```
    BOX (height, width)

  where:

    height, width  =  numeric-expressions
```

The BOX function, which is legal only within a PRINT statement, draws or erases boxes or lines of specified dimensions on the screen. The BOX function uses the current cursor position as the upper left corner of the box; the BOX function does not move the cursor.

> ***Note:*** *The PRINT BOX statement is functional only with terminals that support box graphics. If the terminal does not support box graphics, PRINT BOX has no effect (refer to the appropriate terminal manual).*

The height expression specifies the number of lines the box occupies; the width expression indicates the number of character positions the box occupies. The signs of the expressions determine whether the box is drawn or erased. You can combine the parameters of the BOX function to produce the following results:

- If the sign of each expression is positive (e.g., PRINT BOX(4,6)), the BOX function draws a box of the specified dimensions.
- If the sign of each expression is negative (e.g., PRINT BOX(–2,–4)), the BOX function erases a box of the specified dimensions.
- If one expression is zero and the sign of the other expression is positive (e.g., PRINT BOX(0,5)), the BOX function draws a line of the specified dimensions.
- If one expression is zero and the sign of the other expression is negative (e.g., PRINT BOX(–7, 0)), the BOX function erases a line of the specified dimensions.

If one expression is positive and the other expression is negative, an error occurs. Results are undefined if the height expression exceeds the number of lines available on the screen after the current cursor position or if the width expression exceeds the available screen width.

PRINT BOX is useful for boxing fields of characters or highlighting fields of characters. The vertical line unit has the height of a character space. Vertical lines are drawn through the middle of a character space; the line coexists with the character at that location.

Whenever fields of characters are boxed, the box must be one character wider than the length of the field, and the left edge of the box must be one character position to the left of the field to be enclosed. To box a field, use the following statement where A$ is the given field, LEN(A$) is the length of the field A$, and the symbol b represents one space:

```
PRINT BOX(1, LEN(A$)+1): "b"; A$
```

*Example:*

```
PRINT BOX(1,17); " SYSTEM UTILITIES"
10 PRINT "PROMPT"; BOX(1, 24);: LINPUT A$
```

*Examples of valid syntax:*

```
PRINT BOX(4,7)
PRINT BOX(-4,-9)
PRINT BOX(0,3)
PRINT BOX(-12,0)
```

## PRINT HEXOF

*Format:*

```
                         literal-string
    HEXOF
                         alpha-variable
```

The HEXOF function, which is legal only within a PRINT statement, prints the value of an alpha–variable or literal–string in hexadecimal notation. The HEXOF function prints trailing spaces (HEX 20) in the value of the alpha–variable.

*Example:*

```
:10 A$ = "ABC"
:20 PRINT "HEX VALUE OF A$ = "; HEXOF(A$)
:RUN
HEX VALUE OF A$ = 414243202020202020202020202020202020
```

If the printed value of the alpha–variable exceeds one line on the screen or printer, the system automatically prints the value on the next line or lines. To format the output from lengthy print–elements more precisely, you can use the SELECT PRINT statement to set the desired line width for printing operations.

*Examples of valid syntax:*

```
PRINT HEXOF(STR(A$,X,Y));
PRINT HEXOF(A$( ))
```

## PRINT TAB

*Format:*

```
     TAB (numeric-expression)

  where:

     0 ≤ value of expression < 256
```

The TAB function, which is legal only within a PRINT statement, produces tabulated formatting of printed output. The value of the expression in the TAB function specifies the TAB column. Columns are numbered 0 to n, where n is one less than the line width of the screen or printer. The TAB function positions the cursor or print–element to the specified column in the output line by inserting spaces in the line up to the specified column. On the screen, all intervening values displayed on the line are erased. (The AT function, which also positions the cursor in the screen display, does not output spaces when moving the cursor and therefore does not erase intervening output.)

If the specified column has already been passed, the system ignores the TAB. If the TAB value is greater than the defined width of the output line, the system positions the cursor or print–element at the beginning of the next line.

*Example:*

```
:10 FOR I = 0 TO 5
:20 PRINT TAB(I); "X ="; I
:30 NEXT I
:RUN
X = 0
X = 1
X = 2X = 3X = 4
X = 5
```

*Examples of valid syntax:*

```
PRINT TAB(10)
PRINT HEX(030D0A); TAB(33); "SYSTEM MENU"
```

## PRINTUSING

*Format:*

```
                                          ,
  PRINTUSING image-specification [           print-element] ... [;]
                                          ;

  where:

                                alpha-variable containing image
  image-specification =          line-number of Image statement
                                literal-string specifying image


                                 alpha-variable
       print-element =           numeric-expression
                                 literal-string
```

The PRINTUSING statement creates 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 (refer to the discussion of the SELECT statement in Chapter 7).

A PRINTUSING image defines the format of the output print line. The image can be specified in one of the following ways:

- As the value of an alpha–variable specified in the PRINTUSING statement

*Example:*

```
100 A$ = "##.##"
110 PRINTUSING A$, N
```

- As a separate Image statement whose line–number is specified in the PRINTUSING statement

*Example:*

```
100 %##.##
110 PRINTUSING 100, N
```

- As a literal string specified in the PRINTUSING statement

*Example:*

```
100 PRINTUSING "##.##", N
```

The line–number form of the PRINTUSING statement cannot be used in Immediate mode. (Refer to the discussion of the Image (%) statement earlier in this section.)

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. If the image is specified as the value of an alpha–variable or as a separate Image statement, 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. A format–specification has the following general form:

```
                                                  +
   +
                                                  -
         [$] [#[,]]...[.][#]... [↑↑↑↑]
   -
                                                  ++
                                                  --
```

A format–specification consists of at least one digit–selector character (#) and one or more of the following characters: plus sign (+, ++), minus sign (–, —), decimal point (.), comma (,), dollar sign ($), and up arrow (↑). A format–specification cannot contain embedded spaces or other alpha characters except the designated special characters. Print–elements edited into the print line can be either numeric or alphanumeric. 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.

*Example:*

```
:90 N = 55.25
:100 PRINTUSING "$##.##", N
$55.25
```

**Numeric Print–Elements**

The following 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 smaller than the integer format (i.e., the format contains more digit–selector characters than the value has digits), the value is extended 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 zeros to fit the format. If the integer portion is smaller than the format, it is extended with leading spaces. 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 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. However, if a dollar sign is also specified in the format–specification, the sign of the value is edited into the print line immediately preceding the dollar sign.
5. 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 that appears to the left of the decimal point. When a dollar sign also appears in the format–specification, the sign of the value is edited into the print line immediately preceding the dollar sign.
6. 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 that appears to the left of the decimal point. However, if a dollar sign also appears in the format–specification, 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.
7. 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.

8. 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.
9. If the format contains either a comma (,) or a decimal point (.), the specified character is edited into the print line in the corresponding location.
10. If the format ends with a plus sign (+), the true sign of the value (+ or –) is edited into the print line following the last digit.
11. If the format ends with a 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.
12. If the format ends with a pair of plus signs (++), two space characters are edited into the print line following the last digit (for nonnegative values), and the characters CR are edited into the print line following the last digit (for negative values).
13. If the format ends with a pair of minus signs (—), the characters DB are edited into the print line following the last digit (for negative values), and two space characters are edited into the print line following the last digit (for 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 occurs 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.

*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 either by a space character or by any alphanumeric character other than a digit–selector character (#) or one of the designated special characters. 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. (Refer to the discussion entitled "Suppression of the Carriage Return Code" in this section.)

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.

*Example:*

```
:100 A$ = "### ABC ### DEF ###"
:110 PRINTUSING A$, 123, 456
:RUN
123 ABC 456 DEF
```

If there are more print–elements specified than format–specifications, 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.

*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, usually 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 usually be issued. This feature is useful for repeatedly reusing a single image with a series of print–elements.

*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 usually 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 (screen or printer).

### Selecting a Pause and the Meaning of Device-Types

The rate at which PRINTUSING output is generated on the screen or printer can be slowed by invoking a pause with the SELECT P 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.

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 for a full explanation of the significance of device–type selection on printed or displayed output.

*Example:*

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

*Example:*

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

*Example:*

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

*Example:*

```
: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
```

*Example:*

```
: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

*Format:*

```
PRINTUSING TO alpha-variable,image-spec [ , print-element]...[;]
                                          ;

where:

                  line-number of Image statement
 image-spec    =  alpha-variable containing image
                  literal-string specifying image

                   numeric-expression
 print-element  =  alpha-variable
                   literal-string
```

The PRINTUSING TO statement stores formatted print output in an alpha–variable. The program can then process the output when it is convenient. The PRINTUSING TO statement is the same as the PRINTUSING statement except that the formatted print line is stored in an alpha–variable rather than sent directly to an output device. The following rules govern the storage of formatted output in an alpha–variable when using the PRINTUSING TO statement:

- 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.
- If an alphanumeric array is used as the receiving variable, array element boundaries are ignored and information is contiguously packed.
- 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.
- 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 (refer to 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 usually issued when the print line exceeds the maximum line width is suppressed. Other device–types and line widths can be used with a PRINT statement when the line is ultimately printed.

- 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
```

*Examples of valid syntax:*

```
PRINTUSING TO A$, "ERROR ##.##", E
PRINTUSING TO B$(), 100, X,Y,A$
```

## READ

*Format:*

```
READ variable [,variable ] ...
```

The READ statement, in conjunction with DATA statements, assigns values to variables. The first–executed READ statement searches the program for the first DATA statement and sequentially assigns the next available element in a DATA list to the corresponding variable in the READ list. The assignment process continues until all variables in the READ list receive values or until all elements in the DATA list are used.

The READ variable list can include both numeric and alphanumeric variables. Numeric variables must reference numeric data and alphanumeric variables must reference alphanumeric data. If data types do not match, the system signals an error.

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, the system displays an error message and terminates program execution. If a READ statement contains fewer variables than there are values in a DATA statement, the next READ statement begins with the first unused value in the DATA statement.

You can use the RESTORE statement to reset the DATA list pointer, which allows you to reuse the values in a DATA list. Refer to the discussion of the RESTORE statement later in this section. The READ statement cannot be used in Immediate mode.

*Examples of valid syntax:*

```
READ A,B,C
DATA 4, 315, -3.98

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

FOR I = 1 TO 10
READ A(I)
NEXT I
DATA 7.2, 4.5, 6.921, 8, 4
DATA 11.2, 9.1, 6.4, 8.52, 27
```

## REM

*Format:*

```
    REM[[%] [↑]] text-string

  where:

    text-string = any characters, except colons, that indicate
                  the end of the statement
```

The REM statement is a documenting tool that you can use to insert comments or explanatory remarks into a program. Since the REM statement is a nonexecutable statement that is ignored when encountered during program execution, it can be inserted anywhere in a program. You can combine the parameters of the REM statement to insert headings and subheadings into a program listing. The following features of the REM statement are significant only when listing the program with the LIST D command:

- REM followed by a percent sign (e.g., REM% LOAD FILE) skips a line after printing REM%, prints the specified test string (highlighted) on a separate line, then skips a line before resuming the listing.

*Example:*

The following lines exist in memory:

```
10 REM% PERFORM ADDITION
20 X = 5: Y = 2: PRINT X + Y
```

If you issue a LIST D command, the system lists the program as follows.

```
0010 REM%

   PERFORM ADDITION

0020 X = 5
   : Y = 2
   : PRINT X + Y
```

- REM followed by a percent sign and an up arrow (e.g. REM%↑ VARIABLE) issues a top–of–form of a listing to a printer. It prints the text string (highlighted) at the top of a new page, then skips a line before resuming the listing. When the REM%↑ form is used, the up arrow (↑) must immediately follow the percent sign (%).

*Examples of valid syntax:*

```
REM THE NUMBER MUST BE LESS THAN 1
REM%↑ FILE OPEN ROUTINE
```

## RESTORE

*Format:*

```
                      numeric-expression
    RESTORE
                      LINE line-number [,numeric-expression]
```

```
  where:

    0 ≤ numeric-expression ≤ 65535
```

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 reads DATA values beginning with the specified value.

You can combine the parameters of the RESTORE statement to produce the following results:

- RESTORE followed by no parameters (e.g., RESTORE) resets the DATA pointer to the first DATA value in the first DATA statement in the program.
- RESTORE followed by an expression (e.g., RESTORE (A+B)) resets the DATA pointer to the nth DATA value in the program, where n is the integer portion of the expression.
- RESTORE followed by the LINE parameter and a line–number (e.g., RESTORE LINE 500) 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.
- RESTORE followed by the LINE parameter, a line–number, a comma, and an expression (e.g., RESTORE LINE 200, 4) resets the DATA pointer to the nth DATA value on the specified program line, where n is the integer portion of the expression following the line–number.

  ***Note:** RESTORE 0 is equivalent to RESTORE 1.*

*Examples of valid syntax:*

```
RESTORE
RESTORE LINE 105
RESTORE (X-Y)/2
```

## RETURN

*Format:*

```
RETURN
```

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 stack along with the loop parameters of any FOR/NEXT loops contained within the subroutine.

If entry was made to a marked subroutine by a special function key, 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 is pressed 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, eventually resulting in a Not Enough Memory error.

*Example:*

Using RETURN 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
```

*Example:*

Using RETURN 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
```

The RETURN statement can be used to terminate incomplete FOR/NEXT loops that were initiated after the transfer to the subroutine occurred.

*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 statement after the RETURN statement is executed causes an error.

## RETURN CLEAR

*Format:*

```
RETURN CLEAR [ALL]
```

The RETURN CLEAR statement clears return address information in the last–executed subroutine call. In addition, the FOR/NEXT loop information contained within the subroutine is also cleared. 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.

RETURN CLEAR followed by the ALL parameter, (e.g., RETURN CLEAR ALL) clears all subroutine return address and FOR/NEXT loop information from the internal stacks. The normal sequence of execution continues with the statement following the RETURN CLEAR statement.

In some cases, you may want to start and continue a program with a special function key without returning to the keyboard or the main program. You use the RETURN CLEAR statement to clear the subroutine parameters from the internal stacks without returning from the subroutine.

Repeated subroutine calls without executing either a RETURN CLEAR (or RETURN) statement cause return information to accumulate in the stack, eventually resulting in a Not Enough Memory error.

*Examples of valid syntax:*

```
RETURN CLEAR
RETURN CLEAR ALL
DEFFN' 15: RETURN CLEAR
```

## ROTATE

*Format:*

```
    ROTATE [C] (alpha-variable, numeric-expression)

  where:

     -8 ≤ numeric-expression ≤ 8
```

The ROTATE statement rotates the bits of each character in the value of the alphanumeric–variable the specified number of bits. ROTATE operates on all characters in the alphanumeric–variable, including trailing spaces.

The value of the expression is an integer specifying the number of bits to rotate in each character. If the value is positive, the bits of each character are rotated to the left; the high–order bits replace the low–order bits. If the value is negative, the bits of each character are rotated to the right; the low–order bits replace the high–order bits.

*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 the C parameter is included in the ROTATE statement, 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; if the value of the expression is negative, the rotation proceeds to the right.

*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
```

The ROTATEC statement 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.

*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

BCDEAEABCD
```

**Examples of valid syntax:**

```
ROTATE (A$,3)
ROTATE (B$( ),X)
ROTATEC (STR(C$,X,Y), 2*Z)
ROTATEC (A$,-8)
```

## STOP

*Format: (Program mode)*

```
STOP [literal-string] [#]
```

*Format: (Immediate mode)*

```
STOP [line#]
```

The STOP statement suspends program execution. When a STOP statement is encountered during program execution, program execution stops and word STOP is displayed. If specified, the literal string and/or the line number of the STOP statement is also displayed. The system then displays the colon prompt and awaits user entry.

To continue program execution at the statement immediately following the STOP statement, do one of the following:

- Press HALT to step through the program execution one step at a time
- Enter a CONTINUE command to resume program execution

*Examples of valid syntax:*

```
STOP
STOP "Mount data disk"
STOP #
STOP "Error" #
```

STOP, when used in Immediate mode, sets a program stop point at the specified program line. Subsequently, when a program is run, that program's execution will stop just before the specified line is to be executed, as if the STOP statement were the first statement of that line. When the program stops, the word STOP followed by the line number is displayed. Only one stop point can be set; entering a new Immediate mode STOP replaces the previous stop point setting. To clear a stop point, enter Immediate mode STOP without a line number, or RESET, or enter a CLEAR command. Immediate mode STOP is especially useful when debugging programs since STOP need not be edited into the program itself.

*Examples of valid syntax:*

```
STOP 100
STOP
```

## $TRAN

*Format:*

```
   $TRAN (alpha-variable, translation-table) [mask] [R]

 where:

                                alpha-variable
 translation-table =
                                literal-string


              mask   =          two hex digits
```

The $TRAN (translate) instruction performs code conversion (for example, converting EBCDIC to ASCII). You can use the $TRAN instruction whenever byte substitutions are needed to perform operations such as hex conversions, character verification, and initialization.

$TRAN replaces each character (byte) of the alpha–variable by a character in a translation table according to one of two table lookup procedures. The $TRAN instruction represents either a replacement or a displacement procedure. You specify the desired procedure by the inclusion or omission of the R parameter.

If a mask is specified, $TRAN logically ANDs each character of the alpha–variable with the specified mask character prior to the translation.

### Replacement Procedure

If the R parameter is specified in a $TRAN statement, the table lookup is a replacement procedure. The translation table must represent a list of "to–from" translation characters. The list must consist of pairs of bytes to be interpreted as follows:

- The second byte in each pair of bytes is a "translate from" character (i.e., the even–numbered bytes in the translation table define the set of characters to be translated).
- 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 the translation table 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:

- If a mask is specified, $TRAN (...) R masks the next successive character in the alpha–variable.
- $TRAN (...) R looks up the original (or masked) character in the translation table by comparing the character with each successive "translate from" character in the list.

- When a match is found, $TRAN (...) R returns the corresponding replacement character (the preceding "translate to" character) to the alpha–variable character position currently being processed.
- If no match is found, $TRAN (...) R returns the original (or masked) character to the alpha–variable character position currently being processed.

*Example:*

The following example uses the $TRAN statement with the R parameter to translate selected codes. This example assumes that data is stored in A$ 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(0D 11 00 07)) R
```

### Displacement Procedure

If the R parameter is omitted in a $TRAN statement, the table–lookup is a displacement procedure. The sequential position of each character in a translation table is extremely important; you must specify the translation table as a set of consecutive characters.

During execution of a $TRAN statement (no R parameter specified), the following events occur:

- If a mask is specified, $TRAN masks the next successive character in the alpha–variable.
- The equivalent decimal system value of the original (or masked) character is calculated by treating the byte (the 2–hexdigit–code) as an 8–bit binary number rather than an ASCII character. For example, in the ASCII character set, an unmasked uppercase G is represented by HEX(47) = binary 01000111 = decimal 71.
- The decimal system value becomes the displacement that $TRAN uses to locate the proper translation character in the translation table. The displacement can be defined as the movement of an imaginary pointer that 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, the decimal system value of an ASCII G is 71. If you do not use a mask in the translation operation, the translation character must appear in position 72 of the translation table (refer to the example).
- After a translation character is located in the translation table, $TRAN stores the character in the alpha–variable character position currently being processed.
- If the translation table is too short for the required displacement, $TRAN returns the original (or masked) character to the alpha–variable character position currently being processed.

*Example:*

The following program sequence extracts and prints the low–order hex digit from each ASCII character stored in X$.

```
:10 DIM X$4
:20 T$ = "0123456789ABCDEF"
:30 X$ = "*GO#"
:40 PRINT HEXOF(X$)
:50 $TRAN (X$,T$) 0F
:60 PRINT X$
:RUN
2A474F23
A7F3
```

Line 20 assigns a table containing the characters corresponding to the 16 hexadecimal symbols to T$. Line 30 stores a sample set of ASCII characters in X$ to demonstrate the effect of the actual $TRAN operation when the program is executed. Upon execution, Line 40 prints the hexadecimal notation for each character (byte) stored in X$ prior to the translation operations. In Line 50, the mask (HEX(0F)= binary 00001111) replaces the four high–order bits in each character 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 pointer from the first to the eleventh position in the T$ table, where the character A is located. Due to this condition, 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, and so on.

*Examples of valid syntax:*

```
$TRAN (X$, A$)
$TRAN (D$, HEX(2A 6F)) R
```

## UNPACK

*Format:*

```
UNPACK (image) alpha-variable TO numeric-variable [,numeric-vari-
able] ...

where:

                      +  [#] ... [.][#] ... [↑↑↑↑]
     image   =        -
                   alpha-variable containing image


length of image < 255
```

The UNPACK statement unpacks 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. You specify the format of the packed data in the image (refer to the discussion of the PACK statement in this section). The same image originally used to pack the data should be used in the UNPACK statement. UNPACK sequentially unpacks and assigns values to the receiving numeric variables or arrays; the number of receiving variables determines the number of values to be unpacked. If there are not enough packed values in the alpha–variable to fill all the receiving numeric–variables, the system signals an error.

*Examples of valid syntax:*

```
UNPACK (####)A$ TO X, Y, Z
UNPACK (+#.##)STR(A$,4,2) TO X
UNPACK (+#.##     ) A$( ) TO N( )
UNPACK (######) A$( ) TO X, Y, N( ), M( )
```

## $UNPACK

*Format:*

```
                    F        alpha-variable
  $UNPACK     (         =                            )      alpha-variable
                    D        literal-string

                         variable              variable
             TO           array        [,      array          ]...
```

The $UNPACK statement extracts data from a buffer and stores 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.

*Example:*

In the following statement, the variable A$ represents the buffer containing data. The variables X, A$, Y( ), B$( ) receive the data.

```
$UNPACK A$( ) TO X, A$, Y( ), B$()
```

The following three forms of $UNPACK are available:

- Delimiter Form (specified by the D= parameter)
- Field Form (specified by the F= parameter)
- Internal Form (assumed if neither the D= nor the F= parameter is specified)

### The Delimiter Form of the $UNPACK Statement

The delimiter form of $UNPACK unpacks data values that 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 is satisfied.

The first two bytes of the alpha–variable or literal string following the D= parameter serve as the delimiter specification.

*Example:*

In the following statement, A$ is the delimiter specification variable:

```
$UNPACK (D = A$) B$( ) TO X, Y, Z
```

In the following statement, HEX(002C) is the delimiter specification specified as a hex literal:

```
$UNPACK (D = HEX(002C)) B$( ) TO X, Y, Z
```

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 the system displays an error message and terminates program execution or whether the remaining variables retain their current values. Also, if there is more than one delimiter character between data values in the buffer, the atom specifies whether successive delimiters are ignored or whether variables in the variable list are 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 11–4.

**Table 11-4. 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:*

```
  Alphanumeric Values

      C  C                        C
where:

      C  =  any character except a delimiter
  Numeric Values (ASCII free format)
```

The data can be any valid BASIC representation of a number; spaces are ignored.

```
      s  d  d  ...  d  .   d   d  ... d  E  s  d  d

      sign          mantissa                exponent
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:*

The following routine unpacks these values and stores them in the variables A, B, C, D, and E (this example assumes that B$ contains five numeric data values separated by commas):

```
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:*

The following examples assumes that B$ contains three alphanumeric values separated by spaces.

```
B$ = "ABC  DEF  GHI"
```

| **Statements** | **Results** |
|---|---|
| :100 D$ = HEX(0320)<br>:110 $UNPACK (D = D$) B$ TO W$, X$, Y$, Z$ | W$ = "ABC"<br>X$ = "DEF"<br>Y$ = "GHI"<br>Z$ = unchanged |
| :100 D$ = HEX(0220) | Results in an error because there are not enough :110 $UNPACK (D = D$) B$ TO W$, X$, Y$, Z$ values to satisfy all the variables in the list. |
| :100 D$ = HEX(0020)<br>:110 $UNPACK (D = D$) B$ TO W$, X$, Y$, Z$ | W$ = "ABC"<br>X$ = "DEF"<br>Y$ = "GHI"<br>Z$ = unchanged |

## The Field Form of the $UNPACK Statement

The field form of $UNPACK unpacks data that 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 when the entire variable list is 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; the second byte specifies the field width (i.e., the number of characters in that field).

*Example:*

The following examples illustrate that the field specifications for a buffer can be either contained within an alphanumeric–variable or expressed as a hexadecimal literal string:

```
$UNPACK (F = F$) B$( ) TO X, Y, Z
$UNPACK (F = HEX(1008)) B$( ) TO X, Y, Z
```

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 to 6 identifies a unique numeric format. (Refer to Table 11–5.) 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 results if a numeric field is unpacked into an alphanumeric–variable or if an alphanumeric field is unpacked into a numeric–variable.

**Table 11-5. 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 |
| 7d0y | packed decimal with binary overflow format |
| 8d0y | signed binary format |
| 9d0y | unsigned binary format |
| A0xx | alphanumeric field |
| A1xx | compressed alphanumeric format |

where:

| | | |
|---|---|---|
| xx | = | field width in binary (xx > 0) |
| y | = | field width in binary (< y <=4) |
| d | = | implied decimal position in binary |

You must supply a separate field specification for every variable or array in the variable list. All elements in an array use the field specification for that array.

*Example:*

The following statement requires three field specifications:

```
$UNPACK (F = F$) B$( ) TO A$, B( ), C$
```

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$
```

You can also mnemonically define a field specification in a $FORMAT statement. $FORMAT permits the use of simple mnemonics rather than hex codes to specify field formats. Refer to the discussion of the $FORMAT statement in this section.

*Example:*

The field specification defined for F$ above could be defined as follows in a $FORMAT statement:

```
$FORMAT F$ = A8, F6, A16
```

*Buffer format:*

```
field 1    field 2         ield 3      ...      field n
```

*Data format:*

Alphanumeric Fields (A0xx)

```
      C      C     ...        C
  where:
      C  =  any character in an alphanumeric field to be unpacked.
```

Numeric Fields

*ASCII free format (10xx)*

The data can be any valid BASIC representation of a number; spaces are ignored.

```
      s  d   d  ...   d   .   d   d  ...  d    E  s  d   d

      sign               mantissa                exponent
  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 the following formats, each number can 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.

*ASCII integer format (2dxx)*

```
      s     d      d  ...  d
  where:

      s  =  sign (ASCII + or -), required
      d  =  ASCII digit
```

*IBM display format (3dxx)*

```
      Fd   Fd        ...   Fd       sd
  where:

      s  =  sign (C = +, D = -)*
      d  =  digit (0-9)
```

*IBM USASCII-8 format (4dxx)*

```
      5d   5d     ...        5d           sd
  where:

      s  =  sign (A = +, B = -)*
      d  =  digit (0-9)
```

*IBM packed decimal format (5dxx)*

```
      dd       dd        ...        ds
  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 (+).

*Unsigned packed decimal format (6dxx)*

```
      dd      dd     ...      dd
  where:

      d  =  digit (0-9)
```

The above formats are shown in hexadecimal notation.

Decimal addition and subtraction can be performed on unsigned packed decimal numbers. (Refer to the discussion of the DAC and DSC operators in Chapter 6.)

*Examples:*

The following examples assume that B$ = "+12345678901234567890".

| **Statements** | **Results** |
|---|---|
| :100 F$ = HEX(A005) | |
| :110 $UNPACK (F = F$) B$ TO A$ | A$ = "+1234" |
| :100 F$ = HEX(1010) | |
| :110 $UNPACK (F = F$) B$ TO X | Results in an error because B$ contains more than 13 digits. |
| :100 F$ = HEX(2015) | |
| :110 $UNPACK (F = F$) B$ TO X | X = 1.234567890123E19 |
| :100 F$ = HEX(2206) | |
| :110 $UNPACK (F = F$) B$ TO X | X = 123.45 |
| :10 B$ = HEX(F1F2F3D4) | |
| :20 F$ = HEX(3304) | |
| :30 $UNPACK (F = F$) B$ TO X | X = –1.234 |

```
:10 B$ = HEX(51525354A5)
:20 F$ = HEX(4005)
:30 $UNPACK (F = F$) B$ TO X          X = 12345
```

| Statements | Results |
|---|---|
| :10 B$ = HEX(000012345C) | |
| :20 F$ = HEX(5105) | |
| :30 $UNPACK (F = F$) B$ TO X | X = 1234.5 |

*Packed Decimal with Binary Overflow Format (7d0y)*

The Packed Decimal with Binary Overflow Format is used to unpack numeric values that were stored with $PACK format 7d0y. The maximum field length allowed is 4. The last hexdigit of the packed value identifies the value as being either packed decimal or binary. If the last hexdigit is hex(C–F), the value is packed decimal (same as format 5dxx). If the value is hex(0–B), the value is binary.

For binary values, the upper 3–bits of the low hexdigit of the packed value are the high 3–bits of the binary value. The lowest bit of the last hexdigit is the sign of the value: zero for nonnegative and one for negative values.

*Example:*

```
10 D$=HEX(12 34 56 7C)
20 B$=HEX(2D 68 72)
30 $UNPACK (F=HEX(7004)) D$ TO X
40 $UNPACK (F=HEX(7003)) B$ TO Y
```

Results in X = 1234567 and Y = 1234567

*Signed Binary Format (8d0y)*

The Signed Binary Format is used to unpack numeric values that were packed with $PACK format 8d0y. The maximum field length allowed is 4. The value to be unpacked is a signed binary value. Negative values are stored in 2's complement.

*Example:*

```
10 P$=HEX(00 2D 68 72)
20 N$=HEX(FF D2 97 8E)
30 $UNPACK (F=HEX(8004)) P$ TO X
40 $UNPACK (F=HEX(8004)) N$ TO Y
```

Results in X = 1234567 and

Y = –1234567

*Unsigned Binary Format (9d0y)*

The Unsigned Binary Format is used to unpack numeric values that were packed with $PACK format 9d0y. The maximum field length allowed is 4. The value to be unpacked is an unsigned binary value.

*Example:*

```
10 P$=HEX(00 2D 68 72)
20 $UNPACK (F=HEX(8004)) P$ TO X
```

Results in X = 1234567

*Compressed Alphanumeric Format (A1xx)*

The $UNPACK Compressed Alphanumeric Format is used to decompress alphanumeric data compressed with $PACK format A1xx. Each 6–bits of the field is converted to an ASCII character. Thus, each three bytes of the field unpacks into four ASCII characters. Values hex(00) through hex(3F) are converted to the ASCII characters with values hex(20) through hex(5F). These include the uppercase characters, digits, space, and certain symbols.

If the compressed value is shorter than the receiver variable, the result is padded with trailing spaces. If the receiver is too short, the unpacked value is truncated.

*Example:*

```
10 P$=HEX(86 28 E4)
20 $UNPACK (F=HEX(A103)) P$ TO S$
```

Results in S$ = "ABCDEFGH"

## 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 results 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):*

```
80    01    SOV  value    SOV    value  SOV  value   EOB
  16    16

control
bytes
```

- The 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.

```
                                   binary count (number of
                                   bytes in value)
                                   0 if numeric, 1 if alpha-
                                    numeric
```

- The EOB (End–of–Block, HEX(FD)) character indicates the end of valid data in the record.
- $UNPACK ignores the first two control bytes.

*Data format:*

```
Alphanumeric Values

      C   C  ...    C

  where:

      C  =  any character of the alphanumeric value to be unpacked.

Numeric Values
```

Numeric values must be in Wang Internal Numeric Format.

```
      se      e  d    dd    dd    dd   dd   dd   dd
        L     H

  where:

        s   =  sign:  0 if mantissa +, exponent +
                     1 if mantissa -, exponent +
                     8 if mantissa +, exponent -
                     9 if mantissa -, exponent -

      e e   =  exponent (2 digits)
       L H

        d   =  mantissa digit (always 13)
```

Leading zeros 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.**

```
B$ = 80 01 83 41 42 43 08 02 01 45 00 00 00 00 FD AB CD
          sov   alpha  sov    numeric value    EOB
                 value

:100 $UNPACK B$ TO A$, X
:110 PRINT A$; X
:RUN
ABC 123.45
```

*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)
```

12

# Disk I/O Statements

## Overview

BASIC-2 disk I/O statements instruct the system to store and retrieve data and programs on disk files. Each disk unit is divided into one or more disks. The system provides two modes of disk operation for recording information on disks: File Catalog mode, and Sector Address mode. Both modes of operation apply both to disks and to diskettes.

## File Catalog Mode

The File Catalog mode statements enable a programmer to create and access program files and data files on the disk by name, without reference to specific sector locations. Each newly created file is placed in an available location by the system, and the file's name and location are recorded for future reference. In addition, a number of auxiliary file operations (such as skipping records within a file, creating backup copies of files, and providing essential file parameters) are supported.

The structure used to keep track of where each file is stored is called the catalog. The catalog consists of two sections, a Catalog Index and a Catalog Area. Files are stored sequentially in the Catalog Area, and their names and locations are recorded in the Catalog Index.

A file's location is expressed in terms of a sector address. The storage area of each disk is segmented into a number of storage blocks called sectors. Each sector has a total storage capacity of 256 bytes. The sectors on a disk are numbered sequentially, starting at zero; the number assigned to each sector is referred to as its address. Files typically occupy a number of sequential sectors, and the files themselves are always stored sequentially in the Catalog Area. For example, FILE#1 may occupy 50 sectors, starting at sector #100 and proceeding sequentially to sector #149. A second file, FILE#2, would automatically be stored beginning at sector #150. If FILE#2 also is 50 sectors in length, it occupies sectors #150 to #199. The location of a file is simply the address of the first sector in the file. As an example, the system might make the following entries in the Catalog Index for the two files, FILE#1 and FILE#2:

| Name | Starting Sector Address | Ending Sector Address |
|---|---|---|
| FILE#1 | 100 | 149 |
| FILE#2 | 150 | 199 |

To access one of the existing files, FILE#1 or FILE#2, you need only provide the system with the desired file name. If you request FILE#2, the system looks for the name FILE#2 in the Catalog Index and finds its location in the Catalog Area. Because the system updates the Catalog Index whenever it stores a new file, and the system consults the Index when it accesses an existing file, you not need be concerned about where a cataloged file is actually located.

File Catalog mode includes statements that can do the following actions:

1. Establish a catalog on a designated disk
2. Save and resaving programs on disk by name, without reference to a sector location, and load programs from disk into memory by name, without referencing a sector location
3. Open and reopen data files on disk by name, without reference to a sector location
4. Store data in an open data file without reference to a sector location
5. Write multi-sector records on the disk
6. Skip forward and backward over data records within a data file
7. List the contents of the Catalog Index
8. Scratch unwanted files on disk and reuse the space occupied by such files
9. Copy the entire contents of the catalog onto a second disk
10. Change the name of files

The remainder of this section provides a description of when to use the various File Catalog mode statements. The syntax for each statement is listed in the section entitled "RAM Disk".

## Initializing the Catalog

The SCRATCH DISK statement establishes a Catalog Index and Catalog Area on a specified disk. The SCRATCH DISK statement overwrites the new catalog index area. The following items of information are included in a SCRATCH DISK statement:

1. The disk on which the catalog is to be established
2. The number of sectors that are to be reserved for the Catalog Index (any number between 1 and 255 is allowed)
3. The address of the last sector to be used for the Catalog Area; usually the highest sector address on the disk

In deciding how many sectors you should allocate for the Catalog Index, keep in mind that the first sector of the Index (sector 0) can hold 15 file names, and each subsequent sector (up to sector 254) can hold 16 file names. If you do not specify the number of sectors to be reserved in your SCRATCH DISK statement, the system automatically reserves the first 24 sectors on the disk for a Catalog Index.

## Saving Cataloged Programs on Disk: The SAVE Statement

Once the catalog is initialized, you can begin storing cataloged information on the disk. In Catalog mode, all information must be stored in a named file on the disk. Catalog disk files may be of two types: program files and data files. A data file may contain a large collection of data; a program file always contains only one BASIC program or program segment.

Each time a SAVE statement is executed, it creates a single named program file on disk. A program file consists of the BASIC program or program segment being saved, as well as certain file control information automatically included in the file by the system when the program is stored on disk.

In order to record a cataloged program on disk, the SAVE statement must specify the name of the program and the disk where the program is to be stored.

## Saving Modified Programs on Disk: The RESAVE Statement

The RESAVE statement saves the currently loaded program, or specified part of the program, on disk. The file in which to store the program must be specified and may be a program or data file. Like the SAVE statement, the disk on which the program is to be saved must be specified when using the RESAVE statement.

## Changing the Names of Files: The RENAME Statement

The RENAME statement changes the name of a file. The contents of the file are not changed, only the Catalog Index is updated.

## Retrieving Programs Stored on Disk: The LOAD Command

The LOAD command retrieves catalog programs from the disk and loads them into memory. The LOAD command, unlike the LOAD statement, is not executable in program mode; it is executed in Immediate mode only. The system first checks the Catalog Index for the specified program name and then, upon locating the name, determines the program's location in the Catalog Area and moves to that location to load the program.

Following execution of the LOAD command, the newly loaded program is appended to existing program text in memory. New program lines, which have the same numbers as program lines already stored in memory, replace the currently stored lines in memory. Currently stored program lines that do not have the same line numbers as new program lines are not cleared. However they remain as lines in the new program. (For example, if the old program has lines numbered 5, 15, 25, etc., and the newly-loaded program lines are numbered 10, 20, 30, etc., the new program in memory has lines numbered 5, 10, 15, 20, etc.) For this reason, you should clear memory prior to loading the new program. You can clear all memory by executing a CLEAR command prior to executing the LOAD command. Alternatively, a CLEAR P command causes only program text to be cleared from memory.

The LOAD command must include the following two items of information:

- The disk parameter
- The program name

## Listing the Catalog Index: The LIST DC Statement

Executing the LIST DC statement provides a listing of all entries in a Catalog Index. This listing gives the names of all files stored on a particular disk, along with the following information:

- Information on the status of the catalog itself
  - The number of sectors reserved for the Catalog Index on the specified disk
  - The address of the last sector reserved for the Catalog Area
  - The current end of the Catalog Area

- Information on the status of each cataloged file
  - The name of each cataloged file
  - The file type (program or data) of each file, and its status ("S" if scratched)
  - The starting and ending sector addresses of each file
  - The number of sectors used in each file
  - The number of free sectors available (not used) in each file

Figure 12-1 illustrates a sample catalog listing.

```
  INDEX SECTORS = 00024
  END CAT. AREA = 01000
  CURRENT END   = 00132


Name        Type       Start       End         Used          Free

PROG#2       P         00051       00112       00062         00000

PROG#3       P         00113       00132       00020         00000

PROG#1       P         00024       00050       00027         00000
```

**Figure 12-1. The Catalog Index Listing**

## The LOAD RUN Command

The LOAD RUN command loads and immediately executes a program stored on disk. It produces an automatic combination of the following operations:

- Clears from memory all program text (including both common and non-common variables)
- Loads the named program from the designated disk
- Runs the program

## Saving Data Files

Unlike a program file, which always contains only a single program or program segment, a data file normally contains many different items of data. In order to facilitate fast, efficient retrieval of data from the disk, data stored on disk is organized into a well-defined structure or hierarchy.

The hierarchy of data is organized in the following way: items of data relating to a single subject (such as a particular customer or a particular item in the inventory) are organized into a data record (also known as a logical record); a number of related data records are then stored in a data file (in this case, an inventory file or customer file). An inventory file, for example, would contain a number of inventory records, each of which contains information about an individual item in the inventory (such as model number, name, price, number in stock, etc.). Whenever a particular piece of information about one of the items in the inventory is needed, the procedure is to first locate the inventory file and then read the desired record from the file.

You can establish either a number of different files, or a large single file, to occupy the entire Catalog Area. Within each file, the individual records can be as long as necessary (but each record must have a minimum length of at least one sector, unless special techniques are used to block records in a sector). In Catalog mode, the system automatically keeps track of where each file is located on the disk. However, you should keep track of the locations of records within the file by using appropriate DSKIP and DBACKSPACE instructions.

Because the system itself has no way of knowing how many records will be stored in a file, or how long those records will be, you must estimate how many sectors each file requires. The system must then be instructed to reserve adequate space for the file on a disk. In effect, two steps are required to save data on the disk.

1. First, the data file must be cataloged, or "opened," with a special statement, DATASAVE DC OPEN. In this statement, the new data file is named, and the number of sectors to be reserved for the file are specified. No data is actually stored in the file at this point.
2. Once the file is opened, data records can be stored in the file with the DATASAVE DC statement.

## The DATASAVE DC OPEN Statement

The DATASAVE DC OPEN statement opens a data file. This statement must include the following information:

- The disk on which the data file is to be opened
- The maximum number of sectors to be reserved for the data file (a data file cannot extend beyond the limits of the Catalog Area)
- The name of the data file

When the DATASAVE DC OPEN statement is executed, the specified number of sectors are reserved for the newly-opened file in the Catalog Area on the designated disk. The last sector of the file is used by the system for a special control record, which marks the absolute end of the file. No data can be written in the file beyond that point. The file's name and location are also automatically entered in the Catalog Index.

> ***Note:*** *The system automatically allocates the last sector in each data file exclusively for the system control record. The system control record contains control information and pointers used by the system in maintaining the data file, and no data should be stored in this sector. You should also write an end-of-file trailer record in a data file after all data has been stored; the trailer record also occupies one sector, which cannot be used for data. It is always good programming practice to reserve at least two more sectors than are actually required for a data file in order to account for the two sectors used for control information and end-of file. For example, if you wish to store 24 sectors of data in a file, you should reserve at least 26 sectors (24 + 2) in the DATASAVE DC OPEN statement.*

## The DATASAVE DC Statement

Once a data file has been opened on a disk, you can easily store data in that file with a DATASAVE DC statement. In the DATASAVE DC statement, you indicate only the data that is to be saved or the alpha or numeric variables or arrays containing the data. This information is known as the DATASAVE DC argument list. The system automatically groups information from the argument list sequentially in a logical data record and stores this record sequentially in the currently open file on the disk.

Each DATASAVE DC statement creates a single logical record (or data record) in a file on the disk. The logical record contains all of the data included in the DATASAVE DC argument list. For example, the following statements might be used to open a cataloged data file and store one logical data record in it:

```
DATASAVE DC OPEN T (200) "DATFIL-1"
DATASAVE DC "PETER RABBIT", 01121,B$,N,A()
```

The first statement opens DATFIL-1. The second statement creates one logical record in DATFIL-1 containing all the data from the DATASAVE DC argument list. There are several different types of data in the argument list. The first item is a literal string "PETER RABBIT." The second item, 01121, is a numeric value, which need not be set in quotes. The third item, B$, is an alphanumeric variable. The fourth, N, is a numeric variable, and the fifth, A(), is a numeric array. Empty parentheses indicate that the entire array is to be saved. Each individual item in the DATASAVE argument list (including each array element) is considered to be a single argument. If the array A() is dimensioned to contain four elements, the DATASAVE DC argument list in the second statement 20 consists of a total of eight data values.

When the DATASAVE DC statement is executed, the arguments are taken in sequence from the argument list and stored in a logical record on the disk (if a two-dimensional array is included in the argument list, the array elements are transferred row by row). If the following assignments are assumed, the logical record created by the second statement resembles Figure 12-2.

```
B$    = "10 OAK DRIVE"
N     = 2222
A(1)  = 123
A(2)  = 456A(3)  = 789
A(4)  = 100
```

```
                                  B$           N    A(1) A(2) A(3)  A(4)

PETER RABBIT   01121   10 OAK DRIVE   2222   123   456   789   100
```

**Figure 12-2. Logical Record Consisting of One Sector**

Records created with a DATASAVE DC or DATASAVE DA statement are automatically formatted by the system to contain control information. This control information includes two sector control bytes at the start of the record, a start of value control byte before each record, and an end of data control byte at the end of the record. When a data record is read from the disk with a DATALOAD DC or DATALOAD DA statement, the system expects to find the control information; a record that does not contain the expected control information cannot be read with a DC or DA statement.

The arguments saved in a logical record on disk are commonly referred to as fields within the record. For example, in the previous record, "PETER RABBIT" is the first field in the record, while '100' is the last field. When a logical record is read back into memory from disk, each field must be read into a single variable or array element; it is never possible to read two or more fields into a single variable or array element, even if the receiving variable or element is large enough to contain more than one field. Also, alphanumeric fields must be read back into alphanumeric variables or array elements, while numeric fields must be read back into numeric variables or array elements.

In the present example, the logical record occupies somewhat less than one sector. The remainder of the sector contains meaningless data, which is ignored by the system. If another logical record is created (with a second DATASAVE DC statement), the new record begins at the beginning of the next sector. The remaining unused portion of the first sector is not used for the second record. A logical record always begins at the beginning of a sector. This is the case even if the logical record occupies only a very small portion of a sector. For example, consider the following statements:

```
A = 34.2
B = 100
DATASAVE DC A
DATASAVE DC B
```

Each of these statements creates a single logical record containing a single numeric value and each occupies an entire sector.

```
34.2        UNUSED              100         UNUSED
```

**Figure 12-3. Two One-Sector Logical Records**

A more efficient way to store both values in a single record would be to use the following single DATASAVE DC statement:

```
DATASAVE DC A, B
```

On the opposite end of the spectrum, a single logical record can occupy several sectors – as many sectors, in fact, as are required to store all the data in the DATASAVE DC argument list. The system automatically appropriates as many sectors as it needs to store a logical record.

## Opening a Second Data File on Disk

Most applications require the maintenance of two or more data files on disk. An update operation may require three files (old master, new master, and transaction file). The process of opening subsequent data files on disk is identical to that of opening the first. The DATASAVE DC OPEN statement is used. For example, the following statement could be used to open a second data file, named "DATFIL-2":

```
250 DATASAVE DC OPEN T (500) "DATFIL-2"
```

However, when the system opens DATFIL-2 with the above DATASAVE DC OPEN statement, it closes DATFIL-1, and DATFIL-2 now becomes the currently open file on disk. To specify that the first file remains open on the disk, you would include the file# parameter in the DATASAVE DC statement. With this parameter, any of the 16 slots in the Device Table can open data files (up to 16 files open at once). Before you can use a slot to open a new file, you must store the disk address in it with a SELECT statement such as the following example:

```
SELECT #3/D10, #5/D10
```

This statement instructs the system to store disk address D10 in the Device Table opposite File Numbers #3 and #5. To use one of these slots, you need only to specify its file number in a DATASAVE DC OPEN or DATALOAD DC OPEN statement. The following example uses file #3 to open a second data file on the disk:

```
DATASAVE DC  OPEN T #3, (50) "DATFIL-3"
```

Since the Device Table does not include the file names, the system can identify each file only by its associated file number. The file number associated with a file must therefore be used in any subsequent disk statement or command which accesses that file. The statement

```
DATASAVE DC T  #3,A$()
```

causes A$() to be saved in DATFIL-3, since DATFIL-3's parameters are stored opposite #3.

If it is convenient, a numeric variable can store a file number. Subsequently, the variable can be included in a disk statement or command to reference the file number. For example, the statements

```
A = 3
SELECT #A/D10
DATALOAD DC  OPEN T #A, "DATFIL-1"
```

cause the system to reopen DATFIL-1 and store its parameters opposite #3 in the Device Table (since A=3).

## The DATALOAD DC OPEN Statement

After a data file has been created on disk and subsequently closed, the data in the file can be accessed by reopening the file with a DATALOAD DC OPEN statement. The DATASAVE DC OPEN statement used to open a new file initially cannot be used to reopen an existing file; any attempt to use this statement to reopen a file will result in an error.

In the DATALOAD DC OPEN statement, the system must be provided with the following information:

- The disk on which the file is cataloged.
- The name of the file. The name may be specified as a literal string ("DATFIL-1") or as the value of an alpha variable (A$ = "DATFIL-1").

When a DATALOAD DC OPEN statement is executed, the system searches the Catalog Index on the designated disk for the specified file name. The file's location is then recorded in memory for future reference to the file. The file name specified in the DATALOAD DC OPEN statement must be the name of a data file currently cataloged on the specified disk. If the system cannot locate the file name in the Catalog Index, an error is signaled.

Once a file has been reopened with a DATALOAD DC OPEN statement, you can both store new data in the file with a DATASAVE DC statement and read existing data from the file with a DATALOAD DC statement.

## The DATALOAD DC Statement

In Catalog mode, data is read from a currently open file on disk with a DATALOAD DC statement. When loading data from the disk into memory, you must tell the system which variable(s) and/or array(s) in memory are to receive the data. The list of receiving variables and arrays is specified in a DATALOAD DC statement and is known as the argument list for that statement. The system reads one or more logical records from the currently open file on disk (if no file is currently open, an error is signaled) and sequentially stores the data in the variable(s) and array(s) specified in the argument list. The system continues to read data from the file until all variables in the argument list have been filled or until there is no more data remaining in the file.

If the argument list contains more receiving variables than there are fields in a record, the first fields of the next sequential record are automatically read to satisfy all unfilled variables. The remainder of the second record is then read and ignored. If only the first few fields in a record are read (i.e., if the argument list contains fewer receiving arguments than there are fields in the record), the remainder of the record is read but ignored.

The quantity of data read by a DATALOAD DC statement is determined solely by the number of receiving arguments in the DATALOAD DC argument list, and not by the length of a logical record in the file. DATALOAD DC reads exactly enough data to satisfy its argument list, whether that means reading only a portion of one record, more than one record, or exactly one record.

If the DATALOAD DC argument list requires less than an entire logical record, the remaining unused data in the record is read but ignored, and the system ends up positioned at the beginning of the next logical record. This is true even for multi-sector records; if only the first sector of a multi-sector record is read, the remaining sectors are skipped, and cannot be retrieved unless the entire record is reread. The next DATALOAD DC statement automatically begins reading with the next sequential logical record.

If, on the other hand, the DATALOAD DC argument list requests more than an entire logical record, data is appropriated from the next sequential logical record to satisfy the argument list. In this case as in the preceding case, if only a portion of the second record is read, the remaining unread portion is ignored, and the system is positioned at the beginning of the third logical record.

While the versatility of the DATALOAD DC statement may be important for specialized access routines, it is generally good programming procedure to read back exactly one logical record with each DATALOAD DC statement. To accomplish this, the DATALOAD DC argument list used to read a record must correspond to the DATASAVE DC argument list used in writing the record. It is not required that the two argument lists be identical. The only requirement is that the same number of fields be read as they were written in the record initially.

## The DSKIP and DBACKSPACE Statements

An existing data file on the disk is generally reopened (with a DATALOAD DC OPEN statement) for one of three reasons.

- To read data from the file
- To store additional data in the file
- To change or update existing data in the file

In any of these three cases, you must usually access one or more specific logical records within the file. Two catalog statements, DSKIP and DBACKSPACE, enable you to move through a file and access particular records within the file.

Assume that the file "TEST-1" occupies 50 sectors. Five logical records have been stored in TEST-1, and a trailer record has been written following the last logical record. Each logical record consists of two sectors; therefore, the five records occupy ten sectors (see Figure 12-4).

```
one sector

Record #1   Record #2   Record #3   Record #4    Record #5
```

**Figure 12-4. Logical Records in TEST-1**

Consider that TEST-1 is closed and subsequently reopened with a DATALOAD DC OPEN statement. When the file is reopened, the system automatically positions itself at the beginning of the file. In order to access any record other than Record #1, you must instruct the system to skip ahead through the file to the desired record. You can skip over logical records in a data file with a DSKIP statement. In the DSKIP statement, you must tell the system how many records to skip. Suppose, for example, you wish to read Record #3 in the file. Since the system is currently positioned at Record #1, you must skip two records. The statement DSKIP 2 instructs the system to skip two logical records (Records #1 and #2) and reposition itself at the beginning of Record #3. A DATALOAD DC statement such as DATALOAD DC T C() now loads Record #3 from the file into memory.

The statement DSKIP END positions the system to the current end of the file. This statement is necessary in order to store additional data in a file that has just been reopened, as the system can save only new data at the end of a file. If no end-of-file trailer record has been written in the file, an Error 87 (No End-of-File) is returned following execution of the DSKIP END statement.

After a logical record has been loaded, the system is positioned at the beginning of the next logical record. Suppose you want to load and check logical Record #1 from TEST 1. If the system is currently positioned at the beginning of Record #4 (having just loaded Record #3), you must backspace three logical records. You can do so with a DBACKSPACE statement. The statement DBACKSPACE 3 backspaces over three logical records in the currently open file (TEST-1) on disk. Since the system is currently positioned at the beginning of Record #4, it is repositioned to the beginning of Record #1 following statement execution. Record #1 can now be loaded with a DATALOAD DC statement such as DATALOAD DC A().

You can also backspace to the beginning of a file from any point in the file with a DBACKSPACE BEG statement.

The S parameter is used in a DSKIP or DBACKSPACE statement to inform the system that it is to skip a specified number of sectors rather than logical records. For example, the statement DSKIP #1, 20S instructs the system to increment the Current Address for the file associated with Slot #1 in the Device Table by 20. If the old Current Address was equal to X, the new Current Address is equal to X+20. If each logical record consists of four sectors, this statement has the effect of skipping over five logical records.

## Scratching Unwanted Files

The SCRATCH statement sets the status of the named file to a scratched condition. A scratched file is not physically removed from the disk. The file's name and location remain listed in the Catalog Index, but the file is flagged as a scratched file. A scratched file has three significant characteristics.

- A scratched data file cannot be reopened with DATALOAD DC OPEN, and a scratched program file cannot be loaded into memory with LOAD or LOAD RUN. The danger of accidentally accessing a scratched file is therefore removed.
- A scratched file can be renamed and reopened with a DATASAVE DC OPEN or SAVE statement. In this case, a new file is created in the space previously occupied by the scratched file.
- When a disk is backed up by copying it to another disk with a MOVE command, all scratched files are automatically removed. This provides the overall capability of freeing all scratched disk space in a contiguous manner.

## Making Backup Copies of Cataloged Files

MOVE and COPY operations allow the transfer of information between disks. The MOVE statement can copy an individual cataloged file onto a separate disk (if the file name is specified) or copy the entire contents of the catalog onto a second disk (if no individual file name is specified). In the latter case, MOVE performs the additional function of deleting all scratched files from the catalog when it is copied to the new disk.

MOVE effectively has two separate forms.

- Form 1 scratches the destination disk, then copies all active files to the destination disk and deletes all scratched files.
- Form #2 copies a specified file from the catalog on the origin disk to the catalog on the destination disk.

## Closing a Data File

A data file is closed when its parameters are removed from the Device Table, either by writing over the parameters with another set of parameters or by zeroing out the parameters. There are four methods of closing a currently open data file.

- Assigning the file number currently associated with the file to another file.
- Executing a CLEAR command with no parameters.
- Master Initialization of the system.
- Executing a DATASAVE DC CLOSE statement. DATASAVE DC CLOSE causes all sector address parameters for the specified file or files in the Device Table to be zeroed out, thereby closing the file(s). The disk device address stored in a slot is not zeroed out by DATASAVE DC CLOSE, however.

You should close a data file once processing is complete. In this way, you can prevent someone else from accidentally saving data into the file over currently stored data, and thus destroying your data. You should also always be sure to write a data trailer record in the file prior to closing it, since you will then be able to reopen the file, skip to the end, and continue storing data in it at a later date.

When a file is closed (using any method) its three sector address parameters are removed from the Device Table. When the file is subsequently reopened with a DATALOAD DC OPEN statement, the Current Sector Address is automatically set equal to the Starting Sector Address.

## Sector Address Mode

Sector Address mode is comprised of BASIC-2 statements and commands that enable you to read or write information in specific sectors on the disk. No catalog or Catalog Index can be established or maintained in Sector Address mode (except by user-supplied software), nor is it possible to name programs or data files. Files are identified only by reference to their starting sector addresses. Similarly, individual records must be saved into or loaded from a file by specifying a starting sector address. You must maintain all file addressing information; such information is not maintained automatically by the system. Because the disk statements in Sector Address mode provide direct access to individual sectors, they are referred to as "direct addressing" statements.

It is sometimes useful to use direct addressing statements in conjunction with automatic file cataloging statements. The DC statements can provide a framework for file access and control while the direct addressing statements provide you with a means of writing customized disk operating systems and special access procedures within a file. Some examples are binary searches and sorting routines, which cannot be done efficiently – and, in some cases, which cannot be done at all – with catalog procedures alone. Three classes of statements are available in Absolute Sector Addressing mode: the DA statements (where "DA" is a mnemonic for "direct address"), the BA statements (where "BA" is a mnemonic for "block address"), and the BM statement (where "BM" is a mnemonic for "block mode"). All permit direct access to specific sectors of the disk.

The DA statements can be used to write or read programs or data records beginning at a specified sector. Multisector programs and data records are automatically read or written, just as they are with DC statements. Records created by DA statements or commands are identical in format to records created by DC (catalog) statements or commands, and records saved in one mode may be retrieved in the other.

The BA statements comprise a special class of statements that read and write exactly one sector (256 bytes) of unformatted data. When a record is created with a DATASAVE BA statement, the control information present in records written with DATASAVE DA or DATASAVE DC statements is not included. In this special case you are free to write your own control information in each record and to format your records in a way best suited for your application. Records with a nonstandard format can be read with a DATALOAD BA statement; they cannot be read with DC or DA statements. DATALOAD BA can also be used to read sectors (program or data) written originally with a DC or DA statement.

The BM statements comprise a special class of statements that read and write multiple sectors of unformatted data. It can be thought of as performing multiple BA statements.

No Catalog or Catalog Index is established or maintained in Sector Address mode and the Device Table is used only to obtain the disk-address.

The DA and BA statements do not modify any file start, end, or current sector address information in the Device Table.

In addition to reading and writing information on the disk, Sector Address mode also enables the system to perform disk-to-disk copy operations and verify the transferred data. The Sector Address mode statements and commands are shown in Table 12-1.

**Table 12-1. Sector Address Mode Statements and Commands**

| Command | Function |
|---|---|
| COPY | Copies sectors from one disk to another. |
| DATALOAD DA | Reads a logical record. |
| DATALOAD BA | Reads one sector. |
| DATALOAD BM | Reads one or more sequential sectors. |
| DATASAVE BA | Writes one sector. |
| DATASAVE BM | Writes one or more sequential sectors. |
| DATASAVE DA | Writes a logical record, starting at the specified sector address. |
| LOAD DA (command) | Loads a BASIC program starting at the specified sector address. |
| LOAD DA (statement) | Overlays a BASIC-2 program with a program segment starting at the specified address. |
| SAVE DA | Saves a program on the disk beginning at a specified location. |
| VERIFY | Ensures that information has been written correctly to disk. |

## RAMdisk

### What is RAMdisk?

RAMdisk allows a portion of user memory to be used as a high-speed disk. All BASIC-2 disk statements can be used with RAMdisk. Since user memory is used to emulate disk storage and there is no physical disk access, RAMdisk access is considerably faster than access to an actual disk. However RAMdisk provides only temporary storage; all information stored is lost when the system is powered off or reconfigured.

### Setting up RAMdisk

When the system is configured by executing @GENPART, RAMdisk is created utilizing all available user memory beyond the memory partitions defined for the current configuration. The size of the RAMdisk is determined by the number and size of memory partitions that you define. Each 1K of memory not used for partitions provides 4 RAM disk sectors. If the configuration defined allocates all of memory for partitions, there is no RAM disk available.

Device address /340 is used to access RAMdisk. The address /340 must be put into the Master Device Table using the @GENPART program. As with other devices, RAMdisk can be assigned exclusively to a specified partition or made available to all partitions.

### Accessing RAMdisk

After Master Initialization, RAMdisk is available but initially contains no useful information. RAMdisk must be scratched and needed files must be copied or written to RAMdisk. For example,

```
SCRATCH DISK T/340, LS=5, END=350
MOVE T/D10, "PROGRAM" TO T/340,
MOVE T/D10, "OVERLAY" TO T/340,
```

SCRATCH DISK established a catalog index of 5 sectors for a 350 sector RAMdisk. Then, the files "PROGRAM" and "OVERLAY" are copied so that they can be accessed quickly from the RAMdisk.

All BASIC-2 disk statements can be used with RAMdisk. Access to the RAMdisk can be controlled using the $OPEN and $CLOSE statements. $GIO statements cannot be used to access RAMdisk.

Before powering off or reconfiguring the system, copy all needed information to a real disk unit. All information stored in the RAMdisk is lost when the system is powered off, loses power, or is reconfigured.

## General Forms of the Disk I/O Statements

This section discusses the general forms of the disk I/O statements in alphabetical order.

The following terms are used frequently in the general forms of disk statements and commands in this chapter. They are defined here rather than repeated for each statement or command.

| | | |
|---|---|---|
| BA | = | Specifies Sector Address mode with unformatted data. |
| BM | = | Specifies Sector Address mode with unformatted data. |
| DA | = | Specifies Sector Address mode and standard BASIC-2 dataformat. |
| DC | = | Specifies Disk Catalog mode. |
| disk | = | Disk-address in the form /Dab<br>a must be 1, 2, or 3; b must be a hex digit. |
| i | | If neither a file # nor a disk-address is specified in a disk statement, the default Device Table Slot (#0) is used. |
| filename | = | alpha-variable or literal-string<br><br>Length of file name must be from one to eight characters. The filename specifies a cataloged file. |
| file# | = | # integer or # numeric-variable<br><br>The value of the integer or value of the variable must be from 0 to 15. The file # specifies the Device Table Slot to be used by the disk statement. The Device Table Slot contains the disk address (assigned by a SELECT statement) and file start, end, and current sector address information for certain Catalog mode statements. If neither a file # nor a disk address is explicitly specified in a disk statement, the default Device Table Slot (#0) is used. |
| T | = | platter specification. T indicates that the disk-address itself specifies which disk in the disk unit is to be used. T must be used whenever device type D is used to access a disk. In most state ments, platter specifications F and R are supported for compatibility with older version of BASIC-2 and Wang BASIC when using device type 3 or B. |
| $ | = | A parameter specifying read-after-write verification is to be performed on all data written to disk. Read-after-write detects improperly written information, but it also effectively doubles the time required for the write operation. |

## COPY

*Format:*

```
        file#,                                       file#,
COPY T       [(start-sector, end-sector)] TO T            [(sector)]
                                                      disk,
        disk,
```

```
where:

 sector  =  numeric-expression
```

The COPY statement copies information from one disk to another. All sectors from the specified start-sector through the specified end-sector are copied. If the start-sector and end-sector are not specified, all sectors from the beginning of the disk through the last cataloged file are copied.

The specified sectors are copied to the destination disk beginning at the specified sector. If a destination sector is not specified, the destination sector defaults to the start-sector.

Following the COPY, you may wish to execute a VERIFY statement to ensure the copied sectors were recorded correctly.

*Examples of valid syntax:*

```
COPY T/D10, TO T/D11,
COPY T#1, (S,S+99) TO T#2, (D)
COPY T/D10, (0, 1023) TO T /D20
```

## DATALOAD BA

*Format:*

```
                 file#,
DATALOAD BA T              (sector [,[next-sector]]) alpha-variable
                 disk,
```

```
where:

  sector      =  numeric-expression or alpha-variable
  next-sector  =  numeric or alpha-variable
```

The DATALOAD BA statement reads one sector from the specified disk and stores the entire 256 bytes in the designated alpha-variable. An error results if the alpha-variable is not large enough to hold at least 256 bytes. If the alpha-variable is larger than 256 bytes, the additional bytes of the array are not affected by the DATALOAD BA operation.

After the statement is executed, the system returns the address of the next consecutive sector, either as a decimal value if a numeric return-variable is specified, or as a two-byte binary value if an alphanumeric return-variable is specified. This address can be used in a subsequent disk statement to provide sequential access to data stored on the disk.

The BA parameter specifies Sector Address mode and is not normally used when the referenced file is a cataloged file. Execution of the DATALOAD BA statement does not alter the sector address parameters in the Device Table.

*Examples of valid syntax:*

```
DATALOAD BA T #2, (B$,B$) B$()
DATALOAD BA T #A, (A,B) STR(A$(),X,256)
DATALOAD BA T/D13, (30) A$()
```

## DATALOAD BM

*Format:*

```
                  file#,
   DATALOAD BM T           (sector [,[next-sector]]) alpha-variable
                  disk,

 where:

   sector       = numeric-expression or alpha-variable
    next-sector = numeric or alpha-variable
```

DATALOAD BM reads one or more sectors of unformatted data from the specified disk. The data read is stored in the designated alpha variable. The amount of data read is determined by the size of the alpha variable. If less than or equal to 256, then one sector is read and the appropriate amount of data is moved into the alpha-variable. If greater than 256 bytes, multiple sectors will be read.

The starting sector address of the data to be read is specified by a numeric expression or alpha variable. In the case of the alpha variable, the binary value of the first 2 bytes is used as the sector address. After the statement is executed, the system returns the address of the next consecutive sector, either as a decimal value if a numeric return-variable is specified, or as a two-byte binary value if an alphanumeric return-variable is specified. This address can be used in a subsequent disk statement to provide sequential access to data stored on the disk.

Execution of the DATA LOAD BM statement does not alter the sector address parameters in the Device Table default slot (or in one of the other slots, #1 - #15, if a file number is used in the statement).

*Examples of valid syntax:*

```
DIM A$120, A$(16), B$(2,16), C$(4,16)
DATA LOAD BM T (20) A$(): REM Read 256 bytes of data
DATA LOAD BM T#2,(B$,B$) B$(): REM Read 512 bytes of data
DATA LOAD BM T/320, (C,C) A$: REM Read 120 bytes of data
DATA LOAD BM T #A, (A,B) C$(): REM Read 1024 bytes of data
DATA LOAD BM T/D13, (30) STR(C$(),X,512)
```

## DATALOAD DA

*Format:*

```
                       file#,
  DATALOAD DA T                  (sector [,[next-sector]]) argument-list
                       disk,

 where:

 sector         = numeric-expression or alpha-variable
 next-sector     = numeric or alpha-variable

                  variable      ,variable
 argument-list =                                      ...
                  array         , array
```

DATALOAD DA reads one or more logical records from a data file, starting at the sector address specified, and assigns the values read to the variables and arrays in the argument-list. The data to be read must be in standard format, including the necessary control information (i.e., the data must have been written onto the disk by a DATASAVE DA or DATASAVE DC statement). An error results if numeric data is assigned to an alpha-variable, or vice versa. If data assigned to an alpha-variable is shorter than the length of the variable, the value is padded with trailing spaces; if the value is longer, it is truncated. Arrays are filled row by row.

It should be noted that alpha-arrays (e.g., A$()) receive a separate data value for each element of the array. However, the STR() of an array receives only a single value.

If the argument list requires more data than is contained in the logical record being read, data is read from the next logical record until the argument list is satisfied. The remainder of the next record is then read but ignored. If an end-of-file (trailer record) is encountered while executing a DATALOAD DA statement, no additional data is read, the next available sector is set to the sector address of the trailer record, and the remaining variables in the argument list remain at their current values. An IF END THEN statement will then cause a valid program transfer.

After the DATALOAD DA statement is executed, the system returns the address of the next sequential logical record, either as a decimal value if a numeric return-variable is specified or as a binary value if an alphanumeric return-variable is specified. This address can be used in a subsequent disk statement to provide sequential access to data stored on the disk.

The DA parameter specifies direct addressing mode and generally is not used when the referenced data file is a cataloged file. Execution of the DATALOAD DA statement does not alter the sector address parameters in the Device Table.

*Examples of valid syntax:*

```
DATALOAD DA T #3, (20) A$, B2$(), M2
DATALOAD DA T #A, (E,D) STR(X$(),Y,200)
DATALOAD DA T /D11, (C$,C$),R,S,T()
```

## DATALOAD DC

*Format:*

```
DATALOAD DC [file#,] argument-list
```

```
where:

                      variable     ,variable
argument-list =                                        ...
                      array        , array
```

The DATALOAD DC statement reads logical data records from a cataloged data file and assigns the values read to the variables and/or arrays in the argument list. An error results if numeric data is assigned to an alpha-variable, or vice versa. If data assigned to an alpha-variable is shorter than the length of the variable, the value is padded with trailing spaces; if the value is longer, it is truncated. Before data can be read from a cataloged file, the file must be opened by a DATALOAD DC OPEN or DATASAVE DC OPEN statement. Thereafter, each time a DATALOAD DC statement is executed, the system begins reading data from the file at the next sequential logical record in the file. Arrays are filled row by row.

If the argument list is not filled by one logical record, the next logical record is read. If the logical record being read contains more data than is required to fill all receiving variables in the argument list, data not used is read but ignored. Each time the DATALOAD DC statement is executed, the Current Sector Address associated with the file in the Device Table is updated to the Starting Sector Address of the next consecutive logical record. If an end-of-file trailer record is read, an end-of-file condition is set, the Current Sector Address is set to the address of the trailer record, and no data is transferred. The end-of-file condition can be tested by a subsequent IF END THEN statement. An attempt to read beyond the final sector address for the file results in an error.

*Examples of valid syntax:*

```
DATALOAD DC S(), Y, Z
DATALOAD DC #2, A$(), B()
DATALOAD DC #B2, B(), C, D$
```

## DATALOAD DC OPEN

*Format:*

```
DATALOAD DC OPEN T [file#,] filename
```

The DATALOAD DC OPEN statement opens data files that have been previously cataloged on the disk. When the statement is executed, the system locates the named file on the specified disk and sets up the starting, current, and ending sector addresses of the file in the Device Table (the current address is set equal to the starting address). Any subsequent use of the same file number in other catalog (DC) statements accesses this file. If no file number is included, the file is assumed to be associated with the default file number (#0) and can be accessed by subsequent DC statements with the file number omitted or by specifying file number = #0.

An error will result if the file name cannot be located in the Catalog Index of the specified disk or if the file has been scratched.

The DATALOAD DC OPEN statement must be used when reopening an existing cataloged data file. As a result, DATALOAD DC OPEN is used to reopen a cataloged file irrespective of whether data is to be written in the file or read from the file.

*Examples of valid syntax:*

```
DATALOAD DC OPEN T #2, A$
DATALOAD DC OPEN T "PARTFIL"
```

## DATASAVE BA

*Format:*

```
                     file#,                                alpha-variable
 DATASAVE BA T [$]           (sector [,[next-sector]])
                     disk,                                 literal-string
```

```
 where:

   sector        =  numeric-expression or alpha-variable

   next-sector   =  numeric or alpha-variable
```

The DATASAVE BA statement writes one sector to the disk. The alpha-variable or literal-string contains the data to be written (trailing spaces are also written). If the data to be written is longer than 256 bytes, only the first 256 bytes are written. If the data is shorter than 256 bytes, the remainder of the sector is filled with zeros.

After the statement is executed, the system returns the address of the next sequential sector, either as a decimal value if a numeric return variable is specified, or as a two-byte binary value if an alphanumeric return variable is specified. This address can be used in a subsequent disk statement to permit sequential storage of data on the disk.

The $ parameter specifies that a read-after-write verification be made on the sector written. This verification provides added insurance that data is written accurately, but it also substantially increases the execution time of the DATASAVE BA statement.

The BA parameter specifies Sector Address mode. Execution of the DATASAVE BA statement does not alter the sector address parameters in the Device Table default slot (or in one of the other slots, #1 - #15, if a file number is used in the statement).

*Examples of valid syntax:*

```
DATASAVE BA T (L$,L$) A$()
DATASAVE BA T #2, (Q,Q) HEX (438D9247)
DATASAVE BA T /D12, (50) B$()
```

## DATASAVE BM

*Format:*

```
                    file#,                              alpha-variable
DATASAVE BM T [$]        (sector [,[next-sector]])
                    disk,                               literal-string

where:

sector      =  numeric-expression or alpha-variable
next-sector  =  numeric or alpha-variable
```

DATASAVE BM writes one or more sectors of unformatted data on the specified disk. The data written is the value of the designated alpha variable or literal string. The size of the value determines the number of sectors written. If less than or equal to 256, then one sector is written. If greater than 256 bytes, multiple sectors will be written. Values that do not completely fill a sector are left justified; the remainder of the sector is filled with zeros. All data is written without the control bytes normally found in data records.

The starting sector address of the data to be written is specified by a numeric expression or alpha variable. In the case of the alpha variable, the binary value of the first 2 bytes is used as the sector address. After the statement is executed, the system returns the address of the next consecutive sector, either as a decimal value if a numeric return-variable is specified, or as a two-byte binary value if an alphanumeric return-variable is specified. This address can be used in a subsequent disk statement to provide sequential access to data stored on the disk.

Execution of the DATA SAVE BM statement does not alter the sector address parameters in the Device Table default slot (or in one of the other slots, #1 - #15, if a file number is used in the statement).

*Examples of valid syntax:*

```
DIM A$120, A$(16), B$(2,16), C$(4,16)
DATA SAVE BM T (20) A$(): REM Write 256 bytes of dataDATA SAVE
BM T#2,(B$,B$) B$(): REM Write 512 bytes of data
DATA SAVE BM T/320, (C,C) A$: REM Write 120 bytes of data
DATA SAVE BM T $ #A, (A,B) C$(): REM Write 1024 bytes of data
DATA SAVE BM T/D13, (30) STR(C$(),X,512): REM Write 512 bytes-
DATA SAVE BM T (100) "Boston, Mass."
```

## DATASAVE DA

*Format:*

```
                        file#,                             END
 DATASAVE DA T [$]            ( sector [,[next-sector]])
                        disk,                              argument-list

 where:

 sector       =  numeric-expression or alpha-variable
 next-sector  =  numeric or alpha-variable

                   variable              ,variable
 argument-list  =  literal-string        ,literal-string        ...
                  numeric-expression     ,numeric-expression
                   array                 ,array
```

The DATASAVE DA statement writes a logical record to a data file, starting at the sector-address specified.

The data in the argument-list is written in standard format, including the necessary control information. Each DATASAVE DA statement writes a logical record consisting of one or more sectors. The DATASAVE DA statement causes the values of variables, expressions, and array elements to be written sequentially onto the specified disk. Arrays are written row by row. However, the STR() of an alpha-array represents a single data value. Alphanumeric values must be less than or equal to 124 bytes in length.

The number of sectors written depends on the argument-list. Each numeric value in the argument-list requires nine bytes on disk; each alphanumeric variable requires the maximum number of characters for which the variable is dimensioned plus 1. Each 256-byte sector also requires three bytes of control information.

If the END parameter is used, a data trailer record is written for the file. This record can be used to test for the end of a file during processing with an IF END THEN statement.

After the DATASAVE DA statement is executed, the system returns the address of the next available sector, either as a decimal value if a numeric return-variable is specified or as a two-byte binary value if an alphanumeric return-variable is specified. This address can be used in subsequent disk statements to provide sequential access to data.

The $ parameter specifies that a read-after-write verification be made on all sectors written. This verification provides added insurance that data is written accurately, and it also increases the execution time of the DATASAVE DA statement.

The DA parameter indicates a direct addressing operation; the statement therefore generally is not used when the referenced data file is a cataloged file. The END parameter in a DATASAVE DA statement should never be used for records stored in a cataloged file.

There are two important considerations that must be kept in mind when writing a record into a cataloged file with DATASAVE DA. First, the system provides no automatic boundary checking; as a result, records can be written past the end of one file and into the beginning of the next without system detection. Second, the "number of sectors used" is not updated in the Catalog Index when a trailer record is written with DATASAVE DA END. As a result, DSKIP END cannot be used to skip to the end of the file. Execution of the DATASAVE DA statement does not alter the sector address parameters in the Device Table default slot (or in one of the other slots, #1 - #15, if a file number is used in the statement).

*Examples of valid syntax:*

```
DATASAVE DA T (20) X, Y(), Z$()
DATASAVE DA T $ #2, (B$,B$) M$(), "J.CARTER"DATASAVE DA T
(Q,Q) ENDDATASAVE DA T #A, (A,B) END
DATASAVE DA T $/D13, (A,A)P$(),R()
```

## DATASAVE DC

*Format:*

```
                                END
DATASAVE DC [$] [file#,]
                                argument-list

where:

                  variable             ,variable
argument-list  =  literal-string       ,literal-string         ...
                 numeric-expression    ,numeric-expression
                  array                 ,array
```

The DATASAVE DC statement writes one logical record, consisting of all the data in the DATASAVE DC argument list, to a data file. The file must previously have been opened with a DATASAVE DC OPEN or DATALOAD DC OPEN statement. The record is written starting at the current sector address associated with the specified file number (#n) in the Device Table. If no file number is specified in the DATASAVE DC statement, the data is written into the file currently associated with the default file number (#0) in the Device Table. No data can be saved into an unopened file.

The data in the argument list is written as one logical record in standard format, including the necessary control information. The values in the argument list are stored sequentially on the specified disk. Arrays are written row by row. Each element of an array is written as a separate data value. However, the STR () of an alpha array represents a single data value. Alphanumeric values must be ≤ 124 bytes in length. Each single logical record may consist of one or more sectors on the disk. Each numeric value in the argument list requires nine bytes of storage on disk. Each alphanumeric variable requires the maximum length to which the variable is dimensioned plus one byte; e.g., if the length of A$ is set to 24 characters in a DIM A$24 statement, then A$ requires 25 (24 + 1) bytes of storage. Each 256-byte sector also requires three bytes of sector control information.

If the END parameter is specified, a data trailer record is written in the file, and the Catalog Index entry for the file is updated so that the number of sectors used by the file includes all sectors up to the trailer record just written. A cataloged file always should be ended by a trailer record. A new data record can be stored in the file by writing over the trailer record and subsequently creating a new trailer record. (A DSKIP END statement positions the system to the beginning of the trailer record; a DATASAVE DC statement can be executed at that point to store the new record over the trailer record, and a subsequent DATASAVE DC END statement can be executed to create a new trailer record.)

The $ parameter specifies that a read-after-write verification be made on all sectors written. This verification provides added insurance that data is written accurately, and also increases the execution time of the DATASAVE DC statement.

*Examples of valid syntax:*

```
DATASAVE DC A,X, "CODE#4"
DATASAVE DC $ #2, M$, P2(), F1$()
DATASAVE DC $ #1, "ADDRESS", (3*1)/100, J$()
DATASAVE DC #3, END
DATASAVE DC #A, A$()
```

## DATASAVE DC CLOSE

*Format:*

```
                                file#
DATASAVE DC CLOSE
                                 ALL
```

The DATASAVE DC CLOSE statement closes an individual data file or all data files that are currently open, if they are no longer needed in the current or subsequent programs. The DATASAVE DC CLOSE statement closes a file by setting to zero the starting, ending, and Current Sector Addresses associated with its file number in the Device Table. When the file is closed, a disk statement referencing that file causes an Error 80 (File Not Open) to be displayed.

If the file # parameter is used, the single file associated with that file number is closed. If the ALL parameter is used, every open file is closed. If neither parameter is used, the currently open file associated with the default file number (#0) is closed.

The DATASAVE DC CLOSE statement should not be confused with DATASAVE DC END. The latter writes an end-of-file record at the end of a newly written file. The end-of-file record should always be written prior to executing DATASAVE DC CLOSE.

Closing a file with DATASAVE DC CLOSE upon completion of processing ensures that subsequent users will not erroneously access the file and possibly destroy data. Also, DATASAVE DC CLOSE can be used at the beginning of a program to initialize file parameters to zero before they are set by DATASAVE DC OPEN or DATALOAD DC OPEN. DATASAVE DC CLOSE does not remove disk device addresses from the Device Table.

*Examples of valid syntax:*

```
DATASAVE DC CLOSE
DATASAVE DC CLOSE #3DATASAVE DC CLOSE ALL
DATASAVE DC CLOSE #A
```

## DATASAVE DC OPEN

*Format:*

```
                                          old-filename
  DATASAVE DC OPEN T [$] [file#,]                           filename
                                           size

  where:

    size  =  numeric-expression
```

The DATASAVE DC OPEN statement creates and opens a new data file. Space is reserved for the file in the Catalog Area, and a file entry is made in the Catalog Index.

Each data file must be created initially with a separate DATASAVE DC OPEN statement; if multiple files are to be opened simultaneously, each file must be assigned a different file number. Since there are 16 file numbers available (0-15), a total of 16 data files can be opened simultaneously.

The file# parameter is the file number that identifies the newly-opened file in the Device Table. The disk on which the file is stored, along with the file's starting, ending, and Current Sector Addresses, are entered in the Device Table. The information in the Device Table is identified only by the file number assigned to the file in the DATASAVE DC OPEN statement. A file number must be included in the DATASAVE DC OPEN statement if more than one file is to be open at one time. If no file number is specified, or if file # = #0, the system automatically assigns the newly opened file to the default slot, #0, in the Device Table. Subsequent reference to a file number in a disk catalog statement or command automatically provides access to the Current Sector Address of the associated file.

The filename parameter is the name of the new data file being opened. If the new file is being stored in space previously occupied by a scratched cataloged file (old), then new can be identical to old. Otherwise, new must be unique.

If the size parameter is used instead of old-filename, the new file is appended at the current end of the Catalog Area and given a total number of sectors equal to the value of the size expression. The last sector of each cataloged data file is reserved for systems information. Therefore, the number of sectors available for data storage is always at least one less than the number of sectors reserved for the file.

The old-filename parameter specifies the name of a previously scratched cataloged file (either program or data) that is to be renamed and reused. The new file is given the space previously occupied by the scratched file.

The $ parameter specifies that a read-after-write verification be made to ensure that information is written accurately. This also increases the execution time of the operation.

*Examples of valid syntax:*

```
DATASAVE DC OPEN T (100) "DATFIL1"
DATASAVE DC OPEN T #1, (A*2) "I/O DATA"
DATASAVE DC OPEN T #2, ("DATFIL1") "DATFIL2"
DATASAVE DC OPEN T$#4, (200) A$
```

## DATALOAD AC (CS/386 Only)

*Format*

```
    DATALOAD AC [file #,][record-number] alpha-variable

  where:

    alpha-variable = 512 bytes or larger
    record-number = numeric-expression
```

The DATALOAD AC statement reads one record from a specified file of a specified disk and stores the entire 512 bytes in the designated alpha-variable. (Record = one sector = 512 bytes).

An error results if the alpha-variable is not large enough to hold at least 512 bytes. If the alpha-variable is larger than 512 bytes, the additional bytes of the array are not affected by the DATALOAD AC operation. An error will also occur if the record-number is beyond the total sectors of the file.

When a record-number is specified, the system runs Random Read mode. The statement reads one sector of the specified record which is relative to the start of the file opened. After execution, the current record-number is not affected.

If the record-number is not specified, the system is in Sequential Read mode. The statement gets the record-number from the Device Table and performs read operations. After execution, the record-number in the Device Table is automatically increased by one.

*Examples of valid syntax*

```
      DATALOAD AC #2,(A) B$( )
      DATALOAD AC #1,(20) X$( )
      DATALOAD AC #1, A$( )
```

## DATALOAD AC OPEN (CS/386 Only)

*Format*

```
DATALOAD AC OPEN T [file #,] filename
```

The DATALOAD AC OPEN statement opens data files that have been previously stored on a MS-DOS diskette. When the statement is executed, the system finds the named file on the specified disk and sets up the starting cluster, current record-number, and file length in sectors in the Device Table. (The current record-number is set to zero, one record = one sector = 512 bytes). Any request use of the same file number in other AC statements access this file. If no file number is included, the file is assumed to be associated with the default file number (0).

An error will result if the filename cannot be found in the directory area of the specified disk or the diskette is not in MS-DOS format.

*Examples of valid syntax*

```
DATALOAD AC OPEN T#2, A$
DATALOAD AC OPEN T#1, "Part.Dat"
```

## DATASAVE AC (CS/386 Only)

*Format*

```
    DATASAVE AC [file#,] [(record-number)] (literal-string)
                                            (alpha-variable)

  where:

    record-number = numeric-expression
```

The DATASAVE AC statement writes one record to the disk. The alpha-variable or literal-string contains the data to be written. (Record = one sector = 512 bytes). If the data is longer than 512 bytes, the first 512 bytes are written. If the data is shorter than 512 bytes, the remainder of the sector is filled with zeros.

If record-number is specified, the system runs Random Write mode. The data is written into the record-number which is relative to the start of the opened file. After execution, no information in the Device Table will be altered.

When record-number is not specified, the system is in Sequential Write mode. The statement gets the record-number from the Device Table and performs write operations. The record-number in the Device Table automatically increases by one after execution.

*Examples of valid syntax*

```
      DATASAVE AC #2,(1) A$( )
      DATASAVE AC #1, A$( )
```

## DATASAVE AC OPEN (CS/386 Only)

*Format*

```
    DATASAVE AC OPEN T [file #] (old-filename)
                                ((size in sector) new-filename)

  where:

    Size = numeric-expression
```

The DATASAVE AC OPEN statement creates a new DOS file or rewrites an existing file.

If creating a new file, space is reserved in the disk and a file entry is made in the directory. (Space = sector number specified in size). The disk on which the file is stored, along with the file starting cluster, record-number (initially set to zero), and total sectors are entered in the Device Table. An error will occur if there is not enough space.

When rewriting an old file, the operations are the same as the DATALOAD AC OPEN statement.

*Examples of valid syntax*

```
DATASAVE AC OPEN T#2, (10) "Data1.Dat"
DATASAVE AC OPEN T#3, "Data1.Dat"
```

## DATASAVE AC CLOSE (CS/386 Only)

*Format*

```
    DATASAVE AC CLOSE file#
```

The DATASAVE AC CLOSE statement closes an individual data file if it is no longer used in the current or sequential program.

*Example of valid syntax*

```
      DATASAVE AC CLOSE #1
```

## DATASAVE AC END (CS/386 Only)

*Format*

```
DATASAVE AC [file#,] END
```

After execution of the DATASAVE AC OPEN statement, the disk space is allocated for use. Completing the DATASAVE AC operations, you may find unused space. The DATASAVE AC END statement enables you to update the file length and return the unused disk space.

The DATASAVE AC END statement runs in sequential write mode. The statement takes the record-number from the Device Table as the total file sector number.

Restrictions

```
File# must be 0-15
You can shorten file length only; i.e., no expanded file
length
```

*Example of valid syntax*

```
DATASAVE AC #1, END
```

## DBACKSPACE

*Format:*

```
                                     BEG
DBACKSPACE [file#,]
                                   numeric-expression [S]
```

The DBACKSPACE statement backspaces over logical records or sectors within a cataloged disk file. If a value is specified with a numeric expression and S is not specified, the system backspaces over the number of logical records equal to the value of the numeric-expression, and the Current Sector Address of the file in the Device Table is updated to the starting sector of the new logical record. For example, if numeric-expression = 1, the Current Sector Address is set equal to the starting address of the previous logical record.

If the BEG parameter is used, the Current Sector Address is set equal to the Starting Sector Address of the file (i.e., the starting address of the first logical record in the file).

If the S parameter is used, the value of the expression equals the total number of sectors to backspace. The Current Sector Address of the file in the Device Table is decreased by the number of sectors specified. If the amount specified is too large, the Current Sector Address is set to the starting Sector Address of the file. The S parameter is particularly useful in files where all the logical records are of the same length (i.e., have the same number of sectors per logical record). Backspacing with the S parameter is much faster than backspacing over logical records in a file since the system merely decreases the Current Sector Address in the Device Table by the specified number of sectors and no disk accesses are required. However, you must be certain that you know exactly how many sectors are in each logical record.

*Examples of valid syntax:*

```
DBACKSPACE BEG
DBACKSPACE 2*X
DBACKSPACE #2, 5S
DBACKSPACE #1, BEG
DBACKSPACE #A, 10
```

## DSKIP

*Format:*

```
                                        END
  DSKIP [file#,]
                                        numeric-expression [S]
```

The DSKIP statement skips over logical records or sectors in a cataloged disk file. If a value is specified with a numeric expression and S is not specified, the system skips over a number of logical records equal to the value of 'expression,' and the Current Sector Address for the file is updated to the starting address of the new logical record.

If the END parameter is used, the system skips to the end of the file, i.e., the Current Sector Address for the file is updated to the address of the end-of-file trailer record. Once a DSKIP END statement has been executed, data can be added to the end of the file using DATASAVE DC statements. Note that the DSKIP END statement cannot be used unless a trailer record has previously been written in the file with a DATASAVE DC END statement. DSKIP END results in an Error 87 (No End-of-File) if no trailer record can be located in the file.

If the S parameter is used, the value of the expression equals the total number of sectors to be skipped. The Current Sector Address of the file is increased by the number of sectors specified. If the amount specified is too large, the Current Sector Address is set to the Ending Sector Address of the file. The S parameter is particularly useful in files where all logical records are of the same length (i.e., have the same number of sectors per logical record). Skipping with the S parameter is much faster than skipping logical records in a file since the system merely increases the current address by the specified number of sectors and no disk accesses are necessary. However, you must know exactly how many sectors are in each logical record.

*Examples of valid syntax:*

```
DSKIP 4
DSKIP #2, ENDDSKIP END
DSKIP #3, 4*X
DSKIP #A, 20S
```

## $FORMAT DISK

*Format:*

```
                                   file#
  $FORMAT DISK T
                                   disk
```

$FORMAT DISK is used to format a disk. Before a disk can be used for the storage and retrieval of information, the disk must be formatted. Formatting involves recording a unique address for each sector on the disk, along with other control information needed to ensure proper disk recording and retrieval. Formatting also verifies the media and bypasses flawed areas on the disk.

> ***Caution:*** *Formatting overwrites all data previously recorded on the disk. Zeros are written in each sector.*

$FORMAT DISK can only be used with Wang disk units that support formatting under software control.

> ***Note:*** *Since formatting is catastrophic to previously recorded data, it is recommended that you use the Wang Format Utility supplied with BASIC-2. This utility reduces the chance of accidentally formatting the wrong disk.*

*Examples of valid syntax:*

```
$FORMAT DISK T /D10
$FORMAT DISK T #1
```

## LIMITS

*Format 1: (get limits from disk file)*

```
    LIMITS T [file#,] filename, start, end, used [,status]
```

*Format 2: (get limits from Device Table)*

```
    LIMITS T [file#,] start, end, current
```

```
  where:

    start, end, used, current, status  =  numeric-expressions
```

The LIMITS statement obtains the beginning and ending sector address and Current Sector Address or number of sectors used for a cataloged file. In addition, LIMITS determines the status of the specified file.

*Format 1: Limits of a Cataloged File ("filename" Specified)*

If a filename is specified, the LIMITS statement finds the named file on the specified disk and sets the start variable equal to the starting sector address of the file, the end variable equal to the ending sector address of the file, the used variable equal to the number of sectors currently used by the file, and the status variable equal to a value that specifies the status of the file. The number of sectors currently being used by the file is accurate only if an end-of-file record has been written in the file. An end-of-file record is written in a data file with a DATASAVE DC END statement. Therefore, in order to be able to tell how many sectors are used in a data file, the file must be ended with an end-of-file record.

The status variable receives a value indicating the status of the file. If the status variable is not specified, an error will result if the specified file does not exist. The following values can be assigned to the status variable:

```
                         2 if active data file
                         1 if active program file
"status" variable  =   0 if file name not in index (in this
                         casevariables "start", "end", and "
                         used" will also be set to zero)-1 if
                         scratched program file
                        -2 if scratched data file
```

*Examples of valid syntax:*

```
     LIMITS T "PAYROLL", A,B,C,D
     LIMITS T A$, S,E,A
     LIMITS T #A, "DATFIL 1", X,Y,Z(3)
     LIMITS T #1, "SAM", A,B,C
```

*Format 2: Limits of a Currently Open File ("file-name" Not Specified)*

If a file name is not specified, the LIMITS statement gives the starting, ending, and current sector addresses of the file currently open at the specified file #; if a file is not specified, #0 is used. The disk is not accessed; the data is read directly from the Device Table.

*Examples of valid syntax:*

```
LIMITS T #A(1), A1,A2,A3
LIMITS T #5, A,B,C
LIMITS T X,Y,Z(2)
```

## LIST DC

*Format:*

```
                        file#,
LIST [title] DC T                [filename-mask] [file-type] W
                        disk,
```

```
where:

       title = alpha-variable or literal-string

filename-mask = alpha-variable or literal-string

                P
    file-type = D   ...
                SP
                SD
```

LIST DC lists the contents of the specified disk. LIST DC first shows information regarding the size of the Catalog Index and Catalog Area followed by a listing of files on the disk. For example,

```
:LIST DC T
INDEX SECTORS = 0005'
END CAT. AREA = 1231
CURRENT END   = 0089

NAME      TYPE   START     END    USED     FREE
DATA-L1    P     00006    00027  00022    00000
2231W      P     00028    00030  00003    00000
XPRINT    SP     00031    00033  00003    00000
JUNK      SD     00034    00043  00009    00001
INVTORY    D      00044   00089  00020    00026
```

INDEX SECTORS is the number of sectors allocated for the Catalog Index. The single quote (') after the number of index sectors indicates that the index was created with the SCRATCH DISK' statement; the (') is not displayed for indexes created with SCRATCH DISK. END CAT. AREA is the sector address of the last sector reserved for storing files. CURRENT END is the last sector that has been used.

For each cataloged file LIST DC shows the file name, file type, starting and ending sector addresses of file, the number of sectors used in the file, and the number of sectors not used. The file type is either:

- P for program file
- D for data file
- SP for scratched program file
- SD for scratched data file
- ? for an invalid entry in the index

For data files, the number of sectors currently used in the file is originally set to one, and is updated only when an end-of-file record is written to the file.

The W (wide) option directs the system to use a different output display format. This option results in only the file names being displayed across the screen. For example,

```
LIST DC T W
DATA-LL  2231W XPRINT JUNK INVTORY
```

LIST DC follows the general rules of the LIST statement regarding the title and control of the list output (refer to Chapter 10).

The filename-mask and file type parameters in the LIST DC statement are used to specify a subset of the files to be listed. The filename-mask contains up to 8 characters which are compared against each file name in the Catalog Index. If the current file name 'matches' the mask, then the entry is displayed; otherwise, the search moves on to the next Catalog Index entry. The filename-mask supports the use of two special characters: '*' and '?'. The '?' character is a single character placeholder and allows any character to be in this position of the file name. For example,

```
LIST DC T "BOOT???"
```

lists only those files that have 7 character names beginning with "BOOT". The '*' character is a multi-character placeholder and allows any number of characters to be in this position of the filename. For example,

```
LIST DC T "*JS"
```

lists all files ending with "JS". Any character other than '*' and '?' must match exactly in the file name.

The file-type parameter is used to list only files of the specified type. More than one type can be specified. For example,

```
LIST DC T PD
```

lists only active program and data files; scratched files are not listed. The filename–mask and the file type may be used together to list a group of files of the specified type.

*Examples of valid syntax:*

```
LIST DC T
LIST DC T #2
LIST "Program Title" DC T
LIST DC T "JS*"
LIST DC T #1, "*MVP?"
LIST DC T/D14, P
LIST "Title" DC T/D10, "*MVP?" D
LIST DC T W
LIST DC T P W
```

## LOAD (Immediate mode)

*Format:*

```
                  file#,
LOAD [DC] T                  filename
                  disk,
```

The LOAD command loads a BASIC program or program segment from the specified disk. If neither disk address nor file# is specified, #0 is used.

LOAD can be used to add to program text currently in memory or, if executed following a CLEAR command, to load a new program. An error results if the requested file is not a program file or if it is not present in the catalog.

*Examples of valid syntax:*

```
LOAD T "PROG1"
LOAD T #2, "TESTI/0"
LOAD T A$
LOAD DC T #A1, B$
LOAD T/D11, "PARFILE"
```

## LOAD (Program mode)

*Format:*

```
                file#,    filename
  LOAD [DC] T
               disk,    <numeric-expression> alpha-variable

   [starting-line-number] [,ending-line-number] [BEG line-number]
```

The LOAD statement loads a BASIC-2 program or program segment into memory from the disk and begins executing it. LOAD is a BASIC-2 statement that, in effect, produces an automatic combination of the following BASIC-2 statements and commands:

| | |
|---|---|
| STOP | Stops current program execution. |
| CLEAR P | Clears program text from memory.<br>– If the starting and ending line numbers are not specified, all program text is cleared.<br>– If only a starting line number is specified, all program lines from the specified line to the highest numbered line are deleted.<br>– If only an ending line number is specified, all lines from the lowest numbered line through the specified line are deleted.<br>– If both starting and ending line numbers are specified, the two lines and all intervening lines are deleted. |
| CLEAR N | Clears all non-common variables from memory and the stack. |
| LOAD DC | Loads new program or program segment from disk. |
| RUN | Runs new program, beginning at BEG line-number. If the BEG parameter is not specified, program execution begins at starting-line-number (or at the lowest program line in memory, if start-ing-line-number is not specified). |

The LOAD statement permits segmented programs to be run automatically without normal user intervention. Common variables are passed between program segments. If LOAD is included on a multistatement line, it must be the last executable statement on the line.

The LOAD statement can be used to load several programs from disk when the <numeric-expression> parameter and an alpha-variable are specified. The numeric-expression specifies the number of programs to load. The alpha- variable contains the names of the files to load, each name taking eight bytes in the variable. This feature allows a reduction in program overlay time (since the program is resolved only after all overlays have been loaded, rather than after loading each overlay) and provides more flexibility for program assembly from several program modules.

*Example of multiple file LOAD statement:*

```
COM A$(5)8
A$(1) = "PROG1": A$(2) = "PROG2": A$(3) = "PROG3"
LOAD F<3> A$()
```

**The third line specifies that three program files (PROG1, PROG2, PROG3) are to be loaded, in that order.**

*Examples of valid syntax:*

```
LOAD T "PROG1"
LOAD T #2, "I/OMSTR"
LOAD DC T/D20, "I/OSUB1" 250, 299LOAD T "I/OCNTRL" 500LOAD DC
T #X, B$
LOAD T "R. NIXON" 10, 500 BEG 300
LOAD T/D12, "INFILE"
```

## LOAD DA (Immediate mode)

*Format:*

```
                file#,
  LOAD DA T                (sector [,[next-sector]])
                disk,

  where:

          sector = numeric-expression or alpha-variable
     next-sector = numeric or alpha-variable
```

When the LOAD DA statement is executed in Immediate mode, the program that begins at the specified sector address is read and appended to the current program in memory. (The sector address must be the address of the first sector of the program file.) The LOAD DA command can be used to add program text to a program currently in memory or, if entered after a CLEAR command, to load a new program.

After the LOAD DA command is executed, the system returns the address of the next available sector, either as a decimal value if a numeric return variable is specified or as a two-byte binary value if an alphanumeric return variable is specified. This address can be used in a subsequent disk statement or command to permit sequential access to programs on the disk.

Execution of the LOAD DA command does not alter the sector address parameters in the Device Table.

*Examples of valid syntax:*

```
LOAD DA T (24)
LOAD DA T (A$,B$)
LOAD DA T /D20, (L$,L$)LOAD DA T #2, (A,B)LOAD DA T #A, (C,D)
LOAD DA T /D10, (100)
```

## LOAD DA (Statement)

*Format:*

```
                file#,
  LOAD DA T               (sector [,[next-sector]])
              disk,

    [starting-line-number] [,ending-line-number] [BEG line-number]

  where:

          sector = numeric expression or alpha-variable
     next-sector = numeric or alpha-variable
```

When the LOAD DA statement is executed in Program mode, the program starting at the specified sector address is loaded from the specified disk. The sector address specified must be the address of the first sector of the program file. LOAD DA is a BASIC-2 program statement that, in effect, produces a combination of the following:

| | |
|---|---|
| STOP | Stops current program execution. |
| CLEAR P | Clears program text from memory.<br>– If the starting and ending line numbers are not specified, all program text is cleared.<br>– If only a starting line number is specified, all program lines from the specified line to the highest numbered line are deleted.<br>– If only an ending line number is specified, all lines from the lowest numbered line through the specified line are deleted.<br>– If both starting and ending line numbers are specified, the two lines and all intervening lines are deleted. |
| CLEAR N | Clears all non-common variables from memory and the stack. |
| LOAD DC | Loads new program or program segment from disk. |
| RUN | Runs new program, beginning at BEG line-number. If the BEG parameter is not specified, program execution begins at starting-line-number or at the lowest program line in memory, if starting-line-number is not specified. |
| LOAD DA | Permits segmented programs to be run automatically without normal user intervention, with common variables passed between program segments. If included on a multistatement line, LOADDA must be the last executable statement on the line. |

After the program is loaded, the system returns the address of the next sequential sector either as a decimal value, if a numeric return-variable is specified, or as a two-byte binary value, if an alphanumeric return-variable is specified. This address can be used in a subsequent statement to permit sequential access to programs on the disk.

Execution of the LOAD DA statement does not alter the sector address parameters in the Device Table.

***Note:*** *When a LOAD DA statement is executed, the system stacks are cleared of all subroutine and loop information.*

*Examples of valid syntax:*

```
LOAD DA T (40)
LOAD DA T /D20, (L$,L$) 310,450
LOAD DA T #2, (N$,L$) 570
LOAD DA T /D20, (L,L)
LOAD DA T (2*I+1,L$) 400
LOAD DA T #B, (C,D)
```

## LOAD RUN

*Format:*

```
                    file#,
    LOAD RUN [T]              [filename]
                    disk,
```

The LOAD RUN statement loads and initiates execution of a program. This statement produces a combination of the following operations:

- Clears program text and variables from memory.
- Loads the named program from the designated disk. If a name is not specified, the program START is loaded.
- Runs the program that was loaded.

Since default values are provided for the disk, disk-address, and filename parameters, a system initialization program named START can be loaded and run by pressing three keys on the keyboard: LOAD, RUN, and RETURN. The START program can then provide a menu and entry points of available programs.

*Examples of valid syntax:*

```
LOAD RUN
LOAD RUN "BEGIN"
LOAD RUN /D20, "NAMES"
LOAD RUN /D13, "CITIES"
```

## MOVE (file)

*Format:*

```
          file#,                      file#,    old-filename
MOVE T              filename TO T
          disk,                       disk,     extra-sectors

where:

      extra-sectors  =  numeric-expression
```

The MOVE (file) statement copies the specified file from one disk to another. If extra-sectors are specified, the system reserves an additional number of sectors for the named file.

If an old-filename is included within the parentheses, it is the scratched file to be overwritten by the new file when MOVE is executed. If a scratched file with the same name as the file to be moved is to be overwritten, the old-filename can be omitted from the parentheses.

Following a MOVE, you can execute a VERIFY statement to ensure that the information was copied correctly.

*Examples of valid syntax:*

```
MOVE T/D20, "PRICES" TO T/D10,
MOVE T/D11, "BONDS" TO T/D10, ("STOCKS")
```

## MOVE (disk)

*Format:*

```
          file#,         file#,
MOVE T             TO T             [LS = index-size,] [END = sector]
          disk,          disk,

where:

   index-size  =  an expression such that 1 <= value <= 255
       sector  =  expression
```

The MOVE (disk) statement copies all active files from the first disk specified to the second disk specified. Scratched files and temporary files are not copied. In this format, MOVE provides a means to recover space lost to scratched files by creating a new disk containing only the active files from the old disk.

Before files are copied, MOVE (disk) scratches the destination disk. The LS parameter specifies the size of the Index to be created on the destination disk. If LS is not specified, the size of the Index will be the same as the Index on the source disk. The END parameter specifies the size of the Catalog Area on the destination disk; the value is the highest sector address in the Catalog Area. If END is not specified, the highest sector address is the same as on the source disk. If there is insufficient space on the destination disk, an I93 Format Error is displayed before the disk is scratched.

If the LS and END parameters are not specified, the type of Index created is the same as that of the source disk (ie, a SCRATCH DISK or SCRATCH DISK ' index). Note that if LS and/or END is specified, a SCRATCH DISK ' index is created.

Following a MOVE, the VERIFY statement can be used to ensure that the files were recorded without error. Note that MOVE does not modify the source disk.

To execute a MOVE, approximately 800 bytes of memory must be available for buffering (not occupied by a BASIC-2 program or variables); otherwise an error A03 results and the MOVE is not performed. The large buffer minimizes the time required for the MOVE operation.

*Examples of valid syntax:*

```
MOVE T/D65, TO T/D60,
MOVE T#1, TO T#2, LS=4, END=1279
MOVE T TO T/B20, END=10000
```

## MOVE END

*Format:*

```
                 file#,
MOVE END T                    = sector
                  disk,
```

```
where:

     sector  =  numeric-expression
```

The MOVE END statement changes the size of the Catalog Area. The upper limit of the Catalog Area is initially defined by the END parameter in the SCRATCH DISK statement (see SCRATCH DISK). Once the limit of this area has been set, it can be altered using the MOVE END statement. The value of the numeric-expression specifies the sector address of the new end of the Catalog Area. An error results if a previously cataloged file resides at this address or if the address is higher than the highest legal address on the disk. MOVE END does not alter the size of the Catalog Index.

*Examples of valid syntax:*

```
MOVE END T = 4799
MOVE END T#12, = X+Y
MOVE END T/D20, = 2399
```

## RENAME

*Format:*

```
                file#,
  RENAME T                old-filename TO filename
                disk,
```

RENAME changes the name of an existing file to some other name. After the file has been renamed, the file can only be accessed by using the new file name. The file itself is not modified, occupying the same sectors as it did before the rename; only the index has been updated.

*Examples of valid syntax:*

```
RENAME T "OLDFILE" TO "NEWFILE"
RENAME T/320, O$ TO N$
RENAME T/D10, "TEST001" TO A$
```

## RESAVE

*Format:*

```
                                            file#, !
RESAVE [DC] [<[W][S][R]>] T [$]                      filename [start-line-num
                                             disk, P                      ber]
                                                       [,[end-line-number]]
```

```
where:

     filename   =  alpha-varialbe or literal-string
        start   =   starting line number
          end   =   ending line number
```

RESAVE saves the program in memory over an existing file. The file name specifies the name of the file to be overwritten and becomes the name of the program file on disk. The file to be overwritten can be active or scratched, and can be either a data or a program file. If the file does not exist or is not large enough to hold the program, an error results.

RESAVE operates as a combination of the two statements SCRATCH file and SAVE over an existing file. The parameters in the RESAVE statement are the same as for SAVE. See SAVE for a description of the save parameters.

*Examples of valid syntax:*

```
RESAVE T "CONVERT"
RESAVE T ! "MONEY" 1000,2000
RESAVE T/D22, F$
RESAVE T#1, "LAYOVER" 100
RESAVE <SR> T $ "PROG"
RESAVE DC <S> T $ /D25, P A$ 100,900
```

## SAVE

*Format:*

```
                                   file#,   old-filename
  SAVE [DC] [<[W][S][R]>] T [$]                               ! filename
                                   disk,    extra-sectors     P

        [start-line-number] [,[end-line-number]]

  where:

    extra-sectors = numeric-expression
```

The SAVE statement records on the the specified disk the program, or a portion of the program, currently residing in memory. The file name and file location on the disk are entered into the Catalog Index. The line number parameters specify the portion of the program to save as follows:

- SAVE with no line numbers (e.g., SAVE T "PROG") saves all program text.
- SAVE with only a starting line number (e.g., SAVE T "PROG" 100,) saves all program lines from the specified line through the end of the program.
- SAVE with only an ending line number (e.g., SAVE T "PROG" ,1000) saves all program lines from the lowest numbered line up to and including the specified ending line.
- SAVE with starting and ending line numbers (e.g., SAVE T "PROG" 100,500) saves the specified lines and all intervening lines.

The <[W][S][R]> parameter specifies that the program is to be saved in a more compressed format on disk. The <W> parameter allows program lines to wrap from one disk sector to the next. Without the <W> parameter specified, each program line is required to fit within a single sector. If the next program line to be saved does not fit within the current sector, it is saved in the next sector, leaving unused space in the current sector. Programs saved with SAVE <W> can only be loaded by Release 3.0 (or later) of BASIC-2.

When saving a program on disk, nonessential spaces can be deleted by using the <S> parameter. Spaces in literal strings or REM and % statements are not deleted. Note that when a program is displayed, the system inserts spaces after line numbers, statement separators (:), and most BASIC-2 words. These spaces are not saved with the program.

The <R> parameter causes REM statements to be deleted when the program is saved. All REM statements are deleted, except those that begin with an exclamation point (!). Since REM ! statements are saved, they can be used to contain essential program documentation, such as program titles, instructions, and copyright messages.

The $ parameter specifies that read-after-write verification be performed to ensure that all program text is recorded correctly. The read-after-write check increases the time required for the save operation.

The extra-sectors parameter specifies the number of extra sectors to be included in the program file beyond the actual number required for saving the program text. Saving with extra sectors is useful if the program may need to be updated later. If the program is modified and becomes larger, it can then still be resaved if extra sectors have been included in the program file.

### Overwriting an Existing File

RESAVE is used to record an updated program. The program must already reside on disk and be large enough to accommodate the new program. If the program file on disk is not large enough, RENAME the file on disk to something else and use SAVE to record the updated program.

A new program can also be saved over a scratched program or data file of a different name. The old file name parameter specifies the name of the scratched file to be overwritten. The file to be overwritten must be scratched and large enough to store the program being saved. If the old file name is omitted but the () are included in the SAVE statement, the name of the file to be overwritten defaults to the new program name; however, it is more convenient to use RESAVE than this form of SAVE.

### Protecting a Program

The ! parameter helps protect a program against modification by preventing examination of the program. SAVE with ! scrambles the program while recording it on disk. After subsequently loading the program, an error results if an attempt is made to list, resave, or alter the program. This protection remains in effect until a CLEAR command or LOAD RUN statement is executed.

Programs saved with SAVE <W> ! can only be loaded by Release 3.0 (or later) of BASIC-2. In order to save scrambled programs that can be used by earlier versions of BASIC-2, do not use <W>. However, note that programs saved by SAVE! W are saved more efficiently and typically occupy less disk space than those saved by SAVE !.

The P parameter also helps protect against accidental program modification by preventing programs saved with SAVE P from being listed, resaved, or modified. However, SAVE P does not scramble the program when recording it on disk. This enables the program to be read from disk and examined by using DATALOAD BA statements.

LIST of a protected program only lists REM ! statements. Use REM ! statements for essential program documentation that you want to be visible to the operator. Such information may include the program title and version number, instructions, and copyright messages.

*Examples of valid syntax:*

```
SAVE T "PROGRAM"
SAVE <S> T/D21,"NoSpaces"
SAVE <SR> T#1,"NoREMs"SAVE T $ "VERIFY"
SAVE T (10) "10EXTRA"
SAVE T ("OLD") "NEW"
SAVE T ! W  "SCRAMBLE"
SAVE T "RANGE" 100,999SAVE T "PROGRAM"
SAVE <S> T1021 "No Spaces"
```

## SAVE DA

*Format:*

```
                                  file#,
  SAVE DA [<[W][S][R]>] T [$]                  !  (sector [,next-sector])
                                  disk,        P

     [start-line-number] [,[end-line-number]]

  where:

         sector  =  numeric-expression or alpha-variable
     next-sector  =  numeric or alpha-variable
```

SAVE DA is used to save a program on the the specified disk starting at the specified sector. Because SAVE DA specifies Absolute Sector Addressing mode, a program saved with SAVE DA is not named and is not entered into the Catalog Index. Programs saved by SAVE DA can only be loaded with LOAD DA.

The starting sector address of the program is specified by a numeric expression or alpha variable. In the case of the alpha variable, the binary value of the first 2 bytes is used as the sector address. After the statement is executed, the system returns the address of the next consecutive sector after the program, either as a decimal value if a numeric return-variable is specified, or as a two-byte binary value if an alphanumeric return-variable is specified. This address can be used in a subsequent disk statement as the next sector to record to.

Execution of the SAVE DA does not alter the sector address parameters in the Device Table default slot (or in one of the other slots, #1 - #15, if a file number is used in the statement).

The remaining SAVE DA parameters operate the same as in SAVE. See SAVE for a description of these parameters.

*Examples of valid syntax:*

```
SAVE DA T (1000)
SAVE DA <S> T (X)
SAVE DA <SR> T (S,E)
SAVE DA T#3, (S,S)
SAVE DA T (S) 100,500
SAVE DA T#F, ! (900) ,999
```

## SCRATCH

*Format:*

```
               file #,
  SCRATCH T               filename [,filename] ...
               disk,
```

The SCRATCH statement sets the status of the named disk file(s) to a scratched condition. The SCRATCH statement does not remove the files from the catalog; a subsequent listing of the catalog shows the normal information for both scratched and non-scratched files, as well as showing which files have been scratched. The program text or data in the scratched files is not altered or destroyed by the SCRATCH statement. Once files have been scratched, they cannot be accessed by DATALOAD DC OPEN or LOAD statements. The space allocated to a scratched file can be used by a DATASAVE DC OPEN or SAVE statement for a new file.

The SCRATCH statement is generally used prior to a MOVE statement. When a MOVE statement is executed, information concerning all scratched files is deleted from the Catalog Index, and the corresponding program text or data is deleted from the Catalog Area (refer to the section entitled "MOVE" for further information).

> ***Note:*** *Until a MOVE is executed, all scratched file names remain in the Catalog Index, even if the space occupied by the files in the Catalog Area has been renamed and reused. In the latter case, the scratched file name no longer appears in a listing of the Catalog Index, but it continues to occupy space in the Index. A scratched file name is removed from the Index only when a MOVE is executed.*

*Examples of valid syntax:*

```
SCRATCH T "HEADER"
SCRATCH T #2, "FLD4/15", "FLD10/7"
SCRATCH T/D20, "COLHDR"SCRATCH T A$, B$, C$
SCRATCH T #2, "TEMP 1", A$
```

## SCRATCH DISK

*Format:*

```
                         file#,
SCRATCH DISK [′] T                     [LS = index-size,] END = sector
                         disk,

where:

     index-size  =  an expression such that 1 <= value <= 255
         sector  =  expression
```

SCRATCH DISK reserves space on a disk for the Catalog Index and the Catalog Area. This must be done prior to saving programs or data files on the disk.

The LS parameter specifies the size of the Index to be created on the disk. If LS is not specified, 24 sectors are reserved for the Index. The entry in the Index for each catalogued file consists of the file name, file type, and file location.

The END parameter specifies the size of the Catalog Area used for storing files; the value is the highest sector address in the Catalog Area. The end of the Catalog Area can be altered with the MOVE END statement (see MOVE END).

> ***Note:*** *The Catalog Area can be expanded when necessary with the MOVE END statement. However, once the size of the Catalog Index is specified it cannot be altered without reorganizing the entire catalog. Take special care to provide ample space for future expansion when specifying the size of the Catalog Index in the LS parameter. The number of entries in the Index equals 16 times the number of index sectors.*

SCRATCH DISK' is the preferred format because it uses a more efficient method of locating files in the Catalog Index of a disk than does SCRATCH DISK. The SCRATCH DISK' method improves the performance of locating files within the index and is particularly effective on disks containing many files. Both index types can exist on the same system. The SCRATCH DISK structure is supported for compatibility with existing disks set up by SCRATCH DISK. If the catalog was created with SCRATCH DISK', the first byte of sector 0 equals hex 01. In a disk index created without the ' parameter (SCRATCH DISK), byte 1 of sector 0 equals hex 00. File entries in both index structures are identical.

It is not a recommended practice to directly access the disk index with DATALOAD BA and DATASAVE BA. However, if such program is written, it should support both index types. Older programs that were written making direct use of the SCRATCH DISK index method may not operate properly with the SCRATCH DISK' index since the method of locating files has changed. Careful analysis should be done before converting to the SCRATCH DISK' index since there is a risk that some software will not operate properly after the conversion.

To convert a disk from the SCRATCH DISK index structure to the SCRATCH DISK' index structure, do the following:

- Establish a new disk index on a disk using the SCRATCH DISK' statement.
- Move the files from the existing disk to the new disk with a MOVE disk statement.

*Examples of valid syntax:*

```
SCRATCH DISK T END = 9791
SCRATCH DISK' T LS = 4, END = 1000
SCRATCH DISK T/D20, END = X*2
SCRATCH DISK T #X, LS = L, END = E
```

## VERIFY

*Format:*

```
                file#,
  VERIFY T                  [(start-sector, end-sector)] [numeric-variable]
                disk,

  where:

      sector  =  numeric-expressions
```

The VERIFY statement reads all sectors within the specified range from the designated disk to ensure that information has been written correctly to those sectors. The value of start-sector specifies the address of the first sector to be verified, and the value of end-sector specifies the address of the last sector to be verified. If the start-sector and end-sector parameters are omitted, the entire catalog area is defined.

When an error is detected during the verify operation, one of two things happens. If a numeric-variable is specified, the variable is set equal to the sector address +1 of the sector in error and the verify operation terminates. If there are no errors, the numeric-variable is set equal to zero. If the numeric-variable is omitted, an error message showing the sector address of the bad sector is displayed on the Console Output device and the verify operation continues. The HALT key can be used to terminate the printout of erroneous sectors, but the verify operation cannot be continued.

*Example of error Output:*

```
ERROR IN SECTOR 1097
ERROR IN SECTOR 8012
```

*Examples of valid syntax:*

```
VERIFY T #2,
VERIFY T #A(1), (100,200)
VERIFY T (0,1023)
VERIFY T/D20, (0,2000)
```

# 13

# Math Matrix Statements

## Overview

The math matrix statements provide the operations summarized inTable 13-1.

**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 | MAT INPUT A, B$ from keyboard |
| MAT INV,d,n | matrix = inverse of matrix<br>d = determinant of matrix<br>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$ |

(continued)

**Table 13-1. Matrix Operations (continued)**

| Operation | Description | Example |
|---|---|---|
| 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 = transposition 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. MAT operations on alphanumeric arrays can be used for simple and rapid Input/Output (I/O) and printing of alphanumeric material.

## Array Dimensioning

Both numeric and alphanumeric arrays can be manipulated with the matrix statements. The rules of the BASIC-2 language require that each array be dimensioned. An array is usually 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 characters (bytes).

## Array Redimensioning

The dimensions of an array can be changed explicitly during the execution of the following MAT statements by supplying the new dimensions, enclosed in parentheses, following the array name.

```
MAT CON
MAT IDN
MAT INPUT
MAT READ
MAT REDIM
MAT ZER
```

For example, assume the array A( ) is initially dimensioned as a 10 x 8 array. The following statement redimensions A( ) to a 5 x 5 array:

```
MAT A = CON(5,5)
```

Arrays can also be implicitly redimensioned. In the following example, the array A( ) is redimensioned from a 10 x 10 array to a 2 x 2 array because of the matrix addition performed in Line 20:

```
10 DIM A(10,10), B(2,2), C(2,2)
20 MAT A = B + C
```

For alphanumeric arrays, you can change the maximum length of each element by specifying the new length after the dimension specification. For example, the following 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 characters (bytes):

```
MAT REDIM A$(2,3)10
```

## Redimensioning Rules

- When explicit or implicit redimensioning occurs, the space that the newly dimensioned array requires cannot 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.
- An array cannot be redimensioned to have a different number of dimensions.
- Caution must be exercised when using DIM or COM statements in programs where redimensioning occurs. When a program is overlaid, the entire program is resolved. At this time, any existing common variable that is explicitly dimensioned must agree in size and shape with its original specifications. In particular, arrays that have been redimensioned usually must 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.

## General Forms of the Math Matrix Statements

General forms of the math matrix statements summarized in Table 13-1 are presented in alphabetical order on the following pages.

## MAT Addition: MAT +

*Format:*

```
    MAT c = a + b

  where:

    c, a, and b  =  numeric-array-names
```

The MAT + statement adds two arrays having the same number of dimensions (up to 255). 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 MAT C = A + B is equivalent to the following program lines:

```
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
```

*Example:*

This example assumes that arrays A( ) and B( ) have the following values:

```
        1  2  3                    .5   2
A=                          B=
        4  5  6                    -1  -2
```

The statement MAT C = A + B produces the following results in C( ):

```
           1  2.5  5
C=         5  4    4
```

*Example of valid syntax:*

```
MAT A = A + D
```

## MAT Constant: MAT CON

*Format:*

```
    MAT c = CON [(dim1[,dim2])]

  where:

       c  =  a numeric-array-name

     dim  =  numeric-expression

             For 1-dimensional arrays:  1 ≤ dim ≤ 65535
             For 2-dimensional arrays:  1 ≤ dim ≤ 255
```

The MAT CON statement sets all elements of the specified array equal to 1. You can redimension a matrix to a specified size by using the dimension expressions. 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, then the statement MAT A = CON is equivalent to the following program lines:

```
10 FOR I = 1 TO N: FOR J = 1 TO M
20 A(I,J) = 1
30 NEXT J,I
```

*Examples of valid syntax:*

```
MAT A = CON
MAT A = CON(10)
MAT C = CON(5,7)
MAT B = CON(5*Q,S)
```

## MAT Equality: MAT =

*Format:*

```
    MAT c = a

  where:

    c, a   =   numeric-array-names
```

The MAT = 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 (arrays can contain up to 255 dimensions).

```
10 FOR I = 1 TO N: FOR J = 1 TO M
20 A(I,J) = B(I,J)
30 NEXT J,I
```

*Example:*

This example assumes that the arrays A( ) and B( ) have the following values:

```
               1  1  1            9  8  7
A=                         B=
               1  1  1            6  5  4
```

The statement MAT A = B produces the following results in A( ):

```
               9  8  7
A=
               6  5  4
```

*Example of valid syntax:*

```
MAT A = B
```

## MAT Identity: MAT IDN

*Format:*

```
    MAT c = IDN [(dim1,dim2)]

  where:

      c  =  a numeric-array-name
    dim  =  numeric-expression

            For 2-dimensional arrays:  1 ≤ dim ≤ 255
```

The MAT IDN statement causes the specified array to assume the form of the identity matrix. If the specified matrix is not a square matrix, the system displays an error message and terminates execution.

You can redimension a matrix to a specified size by using the dimension expressions (dim1 and dim2). 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 C( ) is a matrix with N rows and columns, then the statement MAT C = IDN is equivalent to the following program lines:

```
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
```

*Example:*

In the following example, C( ) is a matrix with three rows and three columns. The statement MAT C = IDN produces the following results in C( ):

```
         1 0 0
C   =    0 1 0
         0 0 1
```

*Examples of valid syntax:*

```
MAT A = IDN
MAT B = IDN(4,4)
MAT C = IDN(X,Y)
```

## MAT INPUT

*Format:*

```
     MAT INPUT array-name (dim1[,dim2]) [length]] [,...]

  where:

      dim =  numeric-expression

             For 1-dimensional arrays:  1 ≤ dim ≤ 65535
             For 2-dimensional arrays:  1 ≤ dim ≤ 255

   length = a numeric-expression specifying the maximum length of
            each alpha-array-element such that: 1 ≤ length < 125,
             default = 16
```

The MAT INPUT statement allows you to supply values from the keyboard for an array during program execution. When the system encounters a MAT INPUT statement, it displays a question mark (?) and waits for you to supply values for the array(s) specified in the MAT INPUT statement.

The dimensions of the array(s) are those that 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 you do not specify the length, the system uses a default value of 16 characters (bytes).

The system assigns entered values 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 quotation marks (" "). 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). If no data is entered on an input line (i.e., only a carriage return is entered by pressing RETURN), the system terminates the MAT INPUT statement and ignores the remaining elements of the array currently being filled.

*Example:*

This example uses the MAT INPUT statement to accept values for three numeric arrays. It assumes that the following lines exist in memory:

```
      10 DIM A(2), B(3), C(3,4)
      20 MAT INPUT A, B(2), C(2,3)
```

When the program is run, the following values can be entered in response to the MAT INPUT request:

```
?-3, -5, .612, .41
```

Pressing RETURN at this point assigns these values to array elements A(1), A(2), B(1), and B(2). Because array C( ) remains to be filled, the system issues another request. The following values can be entered:

```
?-6.4, -5.6, 98
```

Pressing RETURN assigns these values to array elements C(1,1), C(1,2), and C(1,3). If RETURN is pressed 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:

```
          -3             .612                -6.4   -5.6   98
A=                 B=                C=
          -5             .41                  0      0      0
```

*Examples of valid syntax:*

```
MAT INPUT A
MAT INPUT A$
MAT INPUT B(2,3), C
MAT INPUT B$(6)10, C, N
MAT INPUT A$(4)3, C$
```

## MAT Inverse: MAT INV

*Format:*

```
    MAT c = INV(a)   ,  d    ,n   ,d

  where:

    c, a  =   numeric-array-names

       d  =   a numeric-variable that equals the value of the
              determinant of array a

       n  =   a numeric-variable that equals the value of the nor
              malized determinant of array a
```

The MAT INV statement replaces matrix c 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. If matrix a is not a square matrix, the system displays an error message and terminates program execution.

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. The variable n (if specified) equals the value of the normalized determinant of matrix a.

The following types of errors can occur during matrix inversion:

- Singular Matrices
- Computational Errors
- Round-Off Error
- Ill-Conditioned 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 signals an error and terminates program execution. If a determinant variable is included in the MAT INV statement, the system sets it equal to zero and continues program execution; the resultant matrix contains meaningless values. You must check the value of the determinant after each inversion.

Computational errors can occur during the calculation of the inverse. Generally, you should let underflow default to zero. However, you should check for overflow (SELECT ERROR > 60). If overflow is detected, the matrix being inverted can be scaled down by dividing each element by a constant before performing the inversion.

Round-off error accumulation results from the successive calculations performed during the inversion of the matrix. By using the maximum pivot strategy, BASIC-2 reduces this type of error. However, some loss of precision is inevitable, especially with large matrices. To detect round-off error, calculate the residuals defined by the following equation:

```
R = I - A * C

where:     I = identity matrix
         A = matrix being inverted
         C = computed inverse
```

For an exact inverse, each element of R is zero. If C is a close approximation of the inverse, each element of R is small.

Ill-conditioned matrices are matrices for which the residual does not provide a reasonable measure of the accuracy of the resultant inverse. Loss of accuracy is not due to round-off error accumulation or the algorithm chosen to perform the inversion, but to the nature of the data itself. Typically, the size of the determinant is used to detect ill-conditioned matrices. Small determinants usually indicate potential problems. For an accurate measurement of the condition of the matrix, the determinant must be normalized relative to the matrix being inverted. BASIC-2 provides the following normalized determinant for a matrix A:

```
             a11      a12      ...    a1n
             a21      a22      ...    a2n
              .
if A=         .
              .
             an1       an2      ...    ann

and k           =   2    +    2  + ... + 2
                   ak1       ak2        akn

then NORM |A| = a11/  1    a12/  1         ... a1n/  1
                a21/  2    a22/  2         ... a2n/  2
                 .
                 .
                 .
                an1/    n  an2/   n           ... ann/   n

                =          |A|
                       1  2  ...  n
```

If the normalized determinant of matrix A is small relative to 1, 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 4 x 4 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 4 x 4 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:*

```
MAT A = INV(B)
MAT Z1 = INV(P),D
MAT B = INV(B),D,N
MAT M = INV(W),,N
```

## MAT Multiplication: MAT *

*Format:*

```
    MAT c = a * b

  where:

    c, a, and b  =  numeric-array-names
```

The MAT * statement multiplies array a by array b and then stores the product 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, the system displays an error message and terminates execution. The resulting dimensions of array c are determined by the number of rows in array a and the number of columns in array b. Arrays a, b, and c can have a maximum of two dimensions.

If array A( ) has N rows and M columns and array B( ) has M rows and P columns, then the statement MAT C = A * B creates an array C( ) that has N rows and P columns. The statement MAT C = A * B is equivalent to the following program lines:

```
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
```

*Example:*

This example assumes that arrays A( ) and B( ) have the following values:

```
          0  1  4              5  1  0  4
A=                       B=    4  1  0  4
          7  7  7              3  4  3  4
```

The statement MAT C = A * B produces the following results in C( ):

```
          16  17  12  20
C  =
          84  42  21  84
```

*Examples of valid syntax:*

```
MAT G = E * F
MAT C = A * B
```

## MAT PRINT

*Format:*

```
                                  ,                 ,
  MAT PRINT array-name                array-name   ...
                                  ;                 ;
```

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. (Refer to the discussion of the PRINT statement in Chapter 11.) 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 following program accepts as input nine alphanumeric literal strings, each having a maximum length of 16 characters, and prints them as a 3 x 3 array.

```
:10 REM Massachusetts Cities and Exchanges
:20 DIM Z$ (3,3)
:30 MAT INPUT Z$
:40 MAT PRINT Z$,
:RUN

? ATHOL-249, AYER-772, CANTON-828, CHELMSFORD-256, ESSEX-768
? GROTON-448, HOPKINTON-435, PEPPERELL-433, ROCKPORT-546

ATHOL-249        AYER-772       CANTON-828
CHELMSFORD-256   ESSEX-768      GROTON-448
HOPKINTON-435   PEPPERELL-433   ROCKPORT-546
```

*Examples of valid syntax:*

```
MAT PRINT A;B,C$
MAT PRINT A,B$
MAT PRINT N,
MAT PRINT A$;
```

## MAT READ

*Format:*

```
MAT READ array-name (dim1[,dim2]) [length]]  [, ...]

where:

  dim   =   numeric-expression

            For 1-dimensional arrays:  1 ≤ dim ≤ 65535
            For 2-dimensional arrays:  1 ≤ dim ≤ 255

length =   a numeric-expression specifying the maximum length of
          each alpha-array-element such that: 1 ≤ length < 125,
           default = 16
```

The MAT READ statement assigns values contained in DATA statements to array-variables, without referencing individual elements of the array. Referenced arrays are sequentially filled, row by row, with the values from the DATA statement(s). 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, the system signals an error and terminates program execution.

You can also specify alphanumeric arrays 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 that were last specified in the program by a COM, DIM, or MAT statement unless new dimensions are specified. You can specify the maximum length for alphanumeric array elements by including the length after the dimension specification. If you do not specify a length, the system uses a default value of 16 characters (bytes).

*Example:*

```
: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
```

## MAT Redimension: MAT REDIM

*Format:*

```
    MAT REDIM array-name (dim1[,dim2]) [length] [, ...]

  where:

        dim = numeric-expression

              For 1-dimensional arrays:  1 ≤ dim ≤ 65535
              For 2-dimensional arrays:  1 ≤ dim ≤ 255

     length = a numeric-expression specifying the maximum length of
              each alpha-array-element such that: 1 ≤ length < 125,
              default = 16
```

The MAT REDIM statement redimensions the specified arrays. The new dimension(s) appears 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 you do not specify a length, the system uses a default value of 16 characters (bytes). The following program redimensions an array:

```
:10 DIM A(3,3)
:20 MAT INPUT A
:30 PRINT HEX(0A), "Original Array"
:40 MAT PRINT A;
:50 MAT REDIM A(2,2)
:60 PRINT HEX(0A 0A), "Redimensioned Array"
:70 MAT PRINT A;

:RUN
?9,8,7,6,5,4,3,2,1

Original Array

9 8 7
6 5 4
3 2 1

Redimensioned Array

9 8
7 6
```

An array cannot be redimensioned to have a different number of dimensions. A redimensioned array cannot be larger than the originally defined array. Any attempt to produce a redimensioned array that is larger than the originally defined array yields an error.

*Examples of valid syntax:*

```
MAT REDIM A(4,5)
MAT REDIM B$(20)8, C1$(3*X,4), D$(8)
```

## MAT Scalar Multiplication: MAT( ) *

*Format:*

```
    MAT c = (k) * a

  where:

    c, a  =  numeric-array-names
       k  =  a numeric-expression
```

The MAT( ) * statement multiplies each element of array a 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, then the statementMAT C = (F1↑2) * A is equivalent to the following program lines:

```
10 FOR I = 1 TO N: FOR J = 1 TO M
20 C(I,J) = (F1↑2) * A(I,J)
30 NEXT J,I
```

*Example:*

This example assumes that the expression (F1↑2) = 5 and the array A( ) have the following values:

```
                5  3  1
       A  =     2  2  2
                1  1  2
```

The statement MAT C = (F1↑2) * A produces the following results in C( ):

```
               25  15   5
       C  =    10  10  10
               5   5    5
```

*Examples of valid syntax:*

```
MAT C = (SIN(X)) * A
MAT D = (X+Y↑2) * A
MAT A = (5) * A
```

## MAT Subtraction: MAT -

*Format:*

```
   MAT c = a - b

 where:

     a, b, and c  =  numeric-array-names
```

The MAT - statement subtracts array b from array a and stores the difference between each pair of elements in array c. Array c can appear on both sides of the equation. If the dimensions of a and b are not the same, the system displays an error message and terminates program execution. 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 MAT C = A - B is equivalent to the following program lines:

```
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
```

*Example:*

This example assumes that the arrays A( ) and B( ) have the following values:

```
                3  3  3                1  1  1

A=              3  3  3        B =     2  2  2

                3  3  3                3  3  3
```

The statement MAT C = A - B produces the following results in C( ):

```
              2  2  2
C =           1  1  1
              0  0  0
```

*Example of valid syntax:*

```
MAT C = A - B
```

## MAT Transposition: MAT TRN

*Format:*

```
     MAT c = TRN(a)

  where:

    a, c   =   numeric-array-names
```

The MAT TRN statement replaces array c by the transpose of array a. Array c is redimensioned to the same dimensions as the transpose of array a. Arrays a and c cannot have more than two dimensions. Array c cannot appear on both sides of the equation.

If array A( ) has N rows and M columns, then the statement MAT C = TRN(A) creates an array C( ) that has M rows and N columns. The statement MAT C = TRN(A) is equivalent to the following program lines:

```
10 FOR I = 1 TO M: FOR J = 1 TO N
20 C(I,J) = A(J,I)
30 NEXT J,I
```

*Example:*

This example assumes that the array A( ) has the following values:

```
        9  8
  A =   6  5
        3  2
```

The statement MAT C = TRN(A) produces the following results in C( ):

```
        9  6  3
  C =   8  5  2
```

*Example of valid syntax:*

```
MAT C = TRN(A)
```

## MAT Zero: MAT ZER

*Format:*

```
    MAT c = ZER [(dim1[,dim2])]

  where:

       c  =  a numeric-array-name


     dim  =  numeric-expression

             For 1-dimensional arrays:  1 ≤ dim ≤ 65535
             For 2-dimensional arrays:  1 ≤ dim ≤ 255
```

The MAT ZER 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 original array.

If array A( ) has N columns and M rows, then the statement MAT A = ZER is equivalent to the following program lines:

```
10 FOR I = 1 TO N: FOR J = 1 TO M
20 A(I,J) = 0
30 NEXT J,I
```

*Examples of valid syntax:*

```
MAT A = ZER
MAT B = ZER(20)
MAT C = ZER(5,2)
MAT D = ZER(F,T+2)
```

14

# Sort Statements

## Overview

The BASIC-2 language provides a unique built-in sorting capability with the MAT MERGE, MAT MOVE, and MAT SORT statements. These statements perform the following functions:

- The MAT MERGE statement merges data from two or more sorted input files into a sorted output file.
- The MAT MOVE statement moves data from one array (the move-array) to a second array (the receiver-array). MAT MOVE can also convert numeric data to sort format or restore data from sort format to numeric format.
- The MAT SORT statement sorts data in an alpha array.

Neither the MAT SORT nor the MAT MERGE statement rearranges the data of the original array. Instead, each statement produces as its output a locator-array, which contains the subscripts of the elements in the data array arranged in order of ascending value. The MAT MOVE statement moves 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.

## Sorting Numeric Data

Before a MAT SORT or MAT MERGE statement can sort numeric data, you must convert numeric data from internal numeric format to sort format. After the data is sorted, you must restore the numeric data to internal numeric format before performing any computations. If the input array and output array are of different types, the MAT MOVE statement 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 (move-array) in a MAT MOVE statement is numeric and the output array (receiver-array) is alphanumeric, the numeric data is automatically converted to sort format as it is transferred into the alphanumeric array. You must perform this operation prior to sorting the numeric data.

If the input array is alphanumeric and the output array is numeric, the data is automatically restored from sort format to internal numeric format as it is transferred to the numeric array. This operation is usually performed after a file of numeric data is 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 without converting any data.

## Representation of Array Subscripts in the Locator-array

The MAT SORT and MAT MERGE statements store array subscripts in a locator array. Array subscripts are 2-byte binary values that identify elements in the input array. The 2-byte binary values have different meanings, depending upon the number of dimensions in the array.

A 1-dimensional array has one column with one or more rows. The subscript for each element is a 2-byte binary value identifying the row that element occupies. For example, the statement DIM N(5) dimensions a 1-dimensional array consisting of one column with five rows (one element per row). The subscript for element N(1) is equal to HEX(0001) in hexadecimal notation. The subscript for N(5) is HEX(0005). Because the maximum value of a 2-byte binary number is 65,535 in decimal notation, a 1-dimensional array can theoretically have up to 65,535 elements.

A 2-dimensional array has one or more rows and columns. In this case, the 2-byte binary subscript is divided into two separate values: the first byte identifies the row; the second byte identifies the column in which the element is located. For example, the statement DIM N(5,3) dimensions a 2-dimensional array with five rows and three columns. The subscript for element N(1,1) is equal to HEX(0101) in hexadecimal notation and the subscript for element N(4,3) is HEX(0403). Because the maximum value of a 1-byte binary number is 255 in decimal notation, a 2-dimensional array can have up to 255 rows and 255 columns.

## General Forms of the Sort Statements

The general forms of the MAT MERGE, MAT MOVE, and MAT SORT statements are discussed in alphabetical order on the following pages.

## MAT MERGE

*Format:*

```
MAT MERGE merge-array [(x[,y])] TO control-variable,
work-variable, locator-array

where:

   merge-array = a 2-dimensional alpha-array

          x,y = optional field-designators defining a field within
                each alpha-array element so that:

                x = a numeric-expression whose value specifies the
                     starting position of the field within each
                      element

                y = a numeric-expression whose value specifies the
                     number of characters in the field (assumes
                     the remainder of the element if not
                      specified)

 control-variable = an alpha-variable that stores merge status in
                    formation

   work-variable = an alpha-variable that the system uses as work
                   space

   locator-array = an alpha-array, with elements of length 2, that
                  stores subscripts of elements in the merge-array
```

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 is required when the input file to be sorted is larger than available memory. In this case, the file cannot be completely sorted in one pass; it must be sorted in segments. Typically, the segments are stored on disk in several temporary files. When the entire original file is sorted in this manner, the result is a number of sorted subfiles that must be merged into a single output file.

MAT MERGE operates on the following variables:

- The *merge-array* is a 2-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. 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.
- The *control-variable* stores information on the status of the merge following each execution of MAT MERGE. The control-variable is an alpha-variable with at least one more byte than the number of rows in the merge-array. As a result, if the merge-array has n rows, the control-variable must have at least n+1 bytes.

- The *work-variable* does not assume a role in the operation of the program. When performing a MAT MERGE, the system uses the work-variable as work space. The work variable is an alpha-variable with at least twice as many bytes as the number of rows in the merge-array.
- The *locator-array* stores the subscripts of elements in the merge-array. The locator-array represents the output of a MAT MERGE operation. The locator-array is 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 must 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. The program must load data into these rows from the input files. 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. As a result, the first subscript placed in the locator-array is that of the element with the lowest value in the merge-array; the next subscript is that of the next-higher data value, and so on. However, MAT MERGE does not actually move any data in the merge-array during this process. You must use the MAT MOVE statement to move merged data from the merge-array to an output array (refer to Figure 14-1).

```
                  1 2 3               1,1
                                       2,1                       A B C
Subfile #1      1 A D F   MAT MERGE  2,2      MAT MOVE
Subfile #2      2 B C E                1,2                       D E F
                                       2,3
                                      1,3

                Merge-Array           Locator-Array             Output-Array
                                     (contains subscripts
                                      pointing to elements
                                      of merge-array in
                                      sorted sequence)
```

**Figure 14-1. Simplified Merge Sequence**

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. 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 you omit the second expression, the field is assumed to occupy the remainder of the element.

*Example:*

This statement performs a merge comparison on a 12-byte field in each element of A$( ), starting at the fifth byte of the element.

```
MAT MERGE A$( ) (5,12) TO B$( ), C$( ), D$( )
```

MAT MERGE terminates the process of scanning the merge-array and storing subscripts in the locator-array when one of the following conditions occurs:

- MAT MERGE fills the locator-array with subscripts
- MAT MERGE completely empties a row of the merge-array (i.e., merges all elements in that row)

The merged data must now be processed and, if necessary, the empty row in the merge-array must be replenished. Processing the merged data usually involves executing MAT MOVE to move the merged data elements into an output buffer. Replenishing a row of the merge-array usually involves loading 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.

### 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, you can determine why the MAT MERGE operation terminated 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 that is set by MAT MERGE following statement execution. This code can have the following values:

| Code | Meaning |
|---|---|
| HEX(0F) | Indicates that the locator-array is full |
| HEX(01) to HEX(FF | Specifies the row number of an empty row in the merge-array |

When MAT MERGE terminates, each of the first n bytes of the control-variable contains the column number of 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. Therefore, the first n bytes of the control-variable must be initialized to HEX(01) prior to beginning the merge. You can easily perform this initialization procedure with the ALL function. Since the n+1st byte of the control-variable specifies the termination status code, its initial value is not important (refer to Figure 14-2).

```
                                    Control-Variable

                                    1  01  Pointer to First Element in
              Merge-Array                  Merge-Array Row One

              1  A  B  Q  T  X       2  01

              2  C  D  E  F  U       3  01

         Rows 3  G  H  I  J  K      4  01   Pointer to First Element in
                                            Merge-Array Row Four
              4   L  M  R  V  Y

              5   N  O  P  S  W    5  01

                                  6  01    Status Code (Initial Value
                                           Not Important)
```

**Figure 14-2. 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, 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. However, if MAT MERGE terminates with one of the merge-array rows empty, the corresponding element of the control-variable receives a HEX(FF) code, and the status code (n+1st byte) points to the row number of the empty row (refer to Figure 14-3).

```
                                 Control-Variable

                                 1  03  Points to Next Element to be
         Merge-Array                      Scanned in Row One

         1  A  B  Q  T  X         2  05

         2  C  D  E  F  U         3  FF   Indicates Empty Row

    Rows 3  G  H  I  J  K        4  03   Pointer to First Element in
                                          Merge-Array Row Four
         4   L  M  R  V  Y

         5   N  O  P  S  W        5  04

                                 6  03   Status Code Contains Row
                                          Number of Empty Row
```

**Figure 14-3. Control-Variable Following Termination of MAT MERGE Due to Empty Row**

Currently, there are two distinct courses of action following an execution of MAT MERGE.

- If the n+1st byte of the control-variable equals HEX(00) following MAT MERGE execution, statement execution terminates because the locator-array is filled. You must use MAT MOVE to merge 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, you must reexecute MAT MERGE to resume merging the unmerged data in the merge-array.
- If the n+1st byte of the control-variable equals HEX(01) to HEX(FF) following MAT MERGE execution, statement execution terminates with an empty row. You must use MAT MOVE to merge data to an output buffer. Before reexecuting MAT MERGE, 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, which destroys the subscripts stored by the previous MAT MERGE operation. Data merged on one pass must always be moved (with MAT MOVE) into an output buffer prior to reexecuting MAT MERGE on the next pass.*

## Replenishing Empty Rows in the Merge-Array

To replenish an empty row in the merge-array, you must first load data from the appropriate input file into the empty row, then you must 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 simply to load data from an input file directly into its row in the merge-array whenever that row becomes empty. However, this technique cannot accommodate the situation in which insufficient data remains in the file to fill the row completely. If the row cannot be filled, use a LET STR(...) = STR(...) statement to right-justify the data within the row. The recommended procedure involves first loading data from the input file to an input buffer and then transferring the data from the input buffer to the appropriate row of the merge-array which is 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, you 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, you must right-justify the data and set the pointer 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, those five values occupy elements 6 to 10, right-justified in the row. Elements 1 to 5 of this row contain invalid data and should not participate in the next merge pass. Thus, you must set the pointer to this row in the control-variable to HEX(06), which instructs 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 2-byte HEX(0000) code in the first element of the locator-array and terminates. The merge operation is then complete.

*Example:*

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 MAT MERGE statement uses the following arrays:

```
M$( )  =  merge-array     C$( )  =  control-variable
W$( )  =  work-variable   L$( )  =  locator-array
```

The following statement defines arrays M$( ), C$( ), W$( ) and L$( ):

```
10 DIM M$(3,6)1, C$(4)1, W$(3)2, L$(12)2
```

Note that the merge-array M$( ) has three rows (since there are three input files) and six columns. Each element is one byte in length. The control-variable C$( ) has one more element than M$( ) has rows, and each element in the control-variable is one byte in length. The work-variable W$( ) 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(HEX(01))
```

The initial contents of the control variable C$( ) appear as follows:

```
    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
```

This example sorts the characters of the alphabet. After data from each file is loaded 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 INPUT 1

    Rows  2    B  C   F   J   M   Q       Merge-Buffer for INPUT 2

          3    E  G   K   N   O   R       Merge-Buffer for INPUT 3
```

The following statement executes the merge operation:

```
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. The subscript of A is the first subscript stored in L$( ), and the subscript of B is the second, and so on. MAT MERGE also rewrites the first three elements of the control-variable C$( ) 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 Line 60 are as follows:

```
            M$ ()                      C$ ()           L$ ()

       01 02 03 04 05 06     1  06  P (Row 1)    0101    A
   01   A  D  H  I  L  P     2  05  M (Row 2)    0201    B
   02   B  C  F  J  M  Q     3  04  N (Row 3)    0202    C
   03   E  G  K  N  O  R      4   00  Status     0102    D
                                      Code
                                                 0301    E
          Indicates Merged                        0203   F
              Data
                                                  0302   G
                                                  0103   H
                                                  0104   I
                                                  0204   J
                                                      0303   K
                                                      0105  L
```

*Example:*

The following program performs the 3-file merge described in the previous example. This program assumes that there are three sorted data files on disk (INPUT1, INPUT2, and INPUT3), each with 50 elements of length 8 in each record. These three files are merged into a fourth file called OUTPUT.

```
10 REM ARRAY DEFINITIONS
20 DIM M$(3,50)8: REM MERGE-ARRAY:  DIM I$(50)8: REM INPUT
   BUFFER
30 DIM O$(50)8: REM OUTPUT BUFFER:  DIM C$(4)1: REM CONTROL-
   VARIABLE
40 DIM W$(3)2: REM WORK-VARIABLE:  DIM L$(50)2: REM LOCATOR-
   ARRAY
50 REM OPEN THE DATA FILES ON DISK
60 OPEN #1, "INPUT1": OPEN #2, "INPUT2"
70 OPEN #3, "INPUT3": OPEN #4, "OUTPUT"
80 REM FILL THE MERGE-ARRAY
90 FOR I = 1 TO 3: GOSUB '40(I): NEXT I
100 M = 1
110 REM MERGE: MAT MERGE M$( ) TO C$( ), W$( ), L$( )
120 IF L$(1) = HEX(0000) THEN 370: REM EXIT IF DONE
130 REM MOVE THE MERGED DATA TO THE OUTPUT BUFFER
140 S = 1
150 N = 50
160 MAT MOVE M$( ), L$(S), N TO O$(M)
170 M = M+N: REM M = NUMBER OF ELEMENTS IN OUTPUT BUFFER
180 REM PUT DATA INTO OUTPUT FILE IF OUTPUT BUFFER FULL
190 IF M <= 50 THEN 230
200 WRITE #4, O$( )
210 M = 1: S = N+1
220 IF S < 51 THEN 150: REM BRANCH IF MORE DATA TO MOVE
230 REM CHECK MERGE TERMINATION FLAG
240 T = VAL(C$(4))
250 IF T = 0 THEN 110: REM BRANCH IF NO ROWS OF M$( ) EMPTY
260 GOSUB '40(T): REM REFILL EMPTY ROW OF M$( )
270 GOTO 110
280 DEFFN'40(R): REM READ THE NEXT BLOCK FROM SPECIFIED
290 REM INPUT FILE AND PUT INTO MERGE-ARRAY
300 READ #R, I$( )
310 IF END THEN 350
320 STR(M$( ),(R-1)*400+1,400) = I$( )
330 C$(R) = HEX(01): REM RESET APPROPRIATE CONTROL-VARIABLE
ELEMENT
340 RETURN
350 C$(R) = HEX(FF): REM ROW EMPTY
360 RETURN
370 END
```

*Examples of valid syntax:*

```
MAT MERGE A$( ) TO C$( ), W1$( ), L$( )
MAT MERGE B$( ) (1,5) TO C$,W$,N$( )
MAT MERGE M$( ) TO X$( ), Y$,Z$( )
```

## MAT MOVE

*Format:*

```
   MAT MOVE move-array [,locator-array] [,n] TO receiver-array

 where:
                         move-alpha-array [(x[,y])]
   move-array =
                         move-numeric-array


                         locator-array
 locator-array =
                         locator-array-element

             n =         a numeric-scalar-variable representing the
                         maximum number of elements to be moved

                         receiver-alpha-array-element [(x[,y])]
                          receiver-numeric-array-element
 receiver-array =
                          receiver-numeric-array
                          receiver-alpha-array [(x[,y])]

           x,y =         optional field-designators defining a field
                          within each alpha-array-element such that:

                          x = a numeric-expression whose value specifies
                              the starting position of the field within
                               each element
                          y =  a numeric-expression whose value speci-
                                fies the number of characters in the
                                field (assumes the remainder of the
                                element if not specified)
```

**The MAT MOVE statement transfers data element by element from one array to another. In its complex form, MAT MOVE also can automatically convert numeric data in Wang internal format to alphanumeric data in sort format and, conversely, can restore alpha data in sort format to numeric data in Wang internal format. MAT MOVE operates on the following variables:**

- **The *move-array* is an alpha- or numeric-array of up to 255 dimensions, with no dimension of two-dimensional arrays exceeding 255. If the move-array is 2-dimensional, the first byte of the subscript specifies the row subscript, and the second byte specifies the column subscript. If the move-array is 1-dimensional, the subscript of each element is expressed as a 2-byte binary number.**
- The *locator-array* is an alpha-array containing a series of 2-byte array subscripts that 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.)

- The *counter-variable* is a numeric-scalar variable specifying the maximum number of elements to move. After the move, the counter-variable is set equal to the number of elements actually moved.
- The *receiver-array* is an alpha- or numeric-array of up to 255 dimensions, with no dimension in a two-dimensional array exceeding 255.

You can specify a locator-array in addition to the move-array and receiver-array. If you specify a locator-array, the MAT MOVE statement transfers data 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 is used, the starting element is assumed to be the first.

If you do not specify a locator-array, the MAT MOVE statement transfers data sequentially from the move-array, starting with the first element. Each element proceeds sequentially to 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 – 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 can be moved to and from designated fields of each element.

MAT MOVE continues transferring data from the move-array to the receiver-array until one of the following conditions occurs:

- MAT MOVE reaches the end of the move-array and no locator-array is specified
- MAT MOVE reaches the end of the array of subscripts (locator-array)
- MAT MOVE finds a binary (0000) in the locator-array (refer to the discussion of MAT SORT following this section)
- MAT MOVE moves the number of elements specified by the counter-variable
- MAT MOVE fills the receiver-array

When MAT MOVE finishes data transfer, a count of the number of elements moved is returned to the counter-variable, if it is specified.

If the move-array and receiver-array are of different types, the MAT MOVE statement automatically performs conversion to or from sort format. If both the move-array and the receiver-array are the same type, MAT MOVE moves the data and a conversion does not occur. When MAT MOVE 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, but format conversion is automatic before individual elements are moved and sorted.

Wang sort format organizes a number within the receiver-array in the following manner:

```
                        BYTES

             1      2      3      4      5      6     7    8

        SIGNS EXPONENT                     MANTISSA
```

The first four bits of the alpha-array-element represent the signs of the mantissa and the exponent. Table 14-1 summarizes the decimal value of these bits for each case of signing.

**Table 14-1. 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 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 14-2 summarizes conditions when the decimal and decimal complement forms are used to express the digits of the exponent.

**Table 14-2. 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 |

The remaining bytes of the alpha-array-element 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 the system truncates or pads the number 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:*

The following example uses a locator-array in a MAT MOVE statement to create a sorted output array. This example assumes that a 2-dimensional array of data D$( ) and an associated locator-array S$( ) with the following values are created in a MAT SORT operation:

```
                    1     2     3     4

              1     B     A     C     D

D$()   =      2     C     B     E     A

              3     A     F     A     E


              0102      0303     0301      0204

S$()   =      0202      0101     0103      0201

              0104     0203      0304      0302
```

The data must be moved from D$( ) to a receiver-array E$( ) in the order specified by S$( ), which creates an output array E$( ) that contains the data from D$( ) in sorted order.

```
100 DIM E$(3,4)1
110 MAT MOVE D$( ), S$(1) TO E$(1)
```

Following execution of statements 100 and 110, an array E$( ) with the following values is produced:

```
               A     A     A     A

E$()   =       B     B     C     C

               D     E     E     F
```

*Examples of valid syntax:*

```
MAT MOVE A$( ) TO B$(1,1)
MAT MOVE A$( ) (5,3), L$(1) TO B(1)
MAT MOVE A( ), L$(3,1), X TO B$(Y)
MAT MOVE A( ), W TO B$(1,5) (2,5)
MAT MOVE A$( ) TO N( )
MAT MOVE A$( ), N TO B$( )
```

## MAT SORT

*Format:*

```
   MAT SORT sort-array [(x[,y])] TO work-variable, locator-array

 where:

   sort-array = n alpha-array containing the data to be sorted
         x,y = optional field-designators defining a field within
                each alpha-array element such that:
                x =  a numeric-expression whose value specifies the
                     starting position of the field within each
                      element
                y = a numeric-expression whose value specifies the
                    number of characters in the field (assumes the
                    remainder of the element if not specified)work-
        variable = an alpha-variable that the system uses as work
                     space
   locator-array =  an alpha-array, with elements of length 2,
                    that stores subscripts of elements in the sort-
                     array in sorted sequence
```

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 can be used as the sort value. Once the locator-array is created by MAT SORT, it can be used to create a sorted output-array with MAT MOVE, which moves elements from the sort-array to a specified receiver-array in the order the locator-array specifies. MAT SORT operates on the following variables:

- The *sort-array* is an array consisting of up to 255 dimensions, with no dimension in a 2-dimensional array greater than 255. If the sort-array is 2-dimensional, the first byte of the subscript specifies the row subscript, and the second byte specifies the column subscript. If the sort-array is 1-dimensional, the subscript of each element is expressed as a 2-byte binary number.
- The *work-variable* does not assume a role in the operation of the program. When performing a MAT SORT, the system uses the work-variable as work space. The work-variable is an alpha-variable with at least twice as many bytes as the number of elements in the sort-array.
- The *locator-array* stores the subscripts of elements in the sort-array. The locator-array must have at least as many elements as the sort-array, and each element must be two bytes. If the locator-array has more elements than the sort-array, the first element of the locator-array that does not receive a subscript from the sort-array receives a 2-byte value of binary zero. The remainingelements of the locator-array are unchanged.

If the sort-array contains the same value more than once, the duplicate values might not occur in the same order as in the sort-array when they are sorted. Figure 14-4 graphically portrays the sorting process. The input array is a 1-dimensional array of five elements, with each element containing a single letter.

```
        1      D                 02                  A

        2      A                 05                  B

        3      E                 04                  C

        4      C   MAT SORT      01   MAT MOVE       D

        5      B                 03                  E

     Sort-Array             Locator-Array         Output-Array
                             (contains            (sorted sequence)
                              subscripts
                              pointing to
                              elements of
                              input array)
```

**Figure 14-4. Simplified Sort Sequence**

*Example:*

The following example uses the MAT SORT statement to sort data in an alpha-array. The MAT SORT statement uses the following arrays:

```
S$( ) = sort-array
W$( ) = work-variable
L$( ) = locator-array
```

This example assumes that the sort-array S$( ) is a matrix of 12 elements containing the following data values:

| | | Columns | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| Rows 1 | B | A | C | G |
| 2 | E | L | C | F |
| 3 | B | F | H | O |

The following statement defines S$( ), W$, and L$( ).

```
100 DIM S$(3,4)1, W$24, L$(3,4)2
```

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.

The following statement executes the sort operation:

```
110 MAT SORT S$( ) TO W$, L$( )
```

Execution of this statement fills the locator-array L$( ) with subscripts specifying the order of the sorted values from S$( ). The resultant locator-array L$( ) is as follows:

| | | Columns | | | |
|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 |
| Rows | 1 | 0102 | 0101 | 0301 | 0203 |
| | 2 | 0103 | 0201 | 0302 | 0204 |
| | 3 | 0104 | 0303 | 0202 | 0304 |

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). The subscript (1,2) is stored in the first element of the locator-array as a 2-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$( ) in elements (1,1) and (3,1). These subscripts, HEX(0101) and HEX(0301), are stored in the next two elements of L$( ). 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:*

```
MAT SORT A$( ) TO W$, S$( )
MAT SORT G$( ) (6,10) TO F$, L1$( )
MAT SORT B$( ) (5,3) TO W$, B1$( )
```

15

# General I/O Statements

## Overview

BASIC-2 provides the user with unique control over and access to I/O devices. The $IF ON/OFF statement determines when a given peripheral device is ready to supply data or receive data. A programmer can design a program to perform another task if the device is not in the desired state, thus saving time. The $GIO statement allows the programmer 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.

## Considerations for the Use of $GIO

There are certain circumstances in which Wang-supplied I/O routines are insufficient to satisfy user I/O requirements. For example, a user may wish to attach a non-Wang peripheral device to a system with a Model 2250 Controller. Since I/O routines do not exist for this device, a programmer may have to write one.

The $GIO statement enables a programmer to write such routines using a series of microcommands that control the electronic signals on the I/O bus. A programmer can write various microcommand sequences to perform I/O functions, such as reading data from devices. User-written $GIO routines can archive data transfers up to 100,000 characters per second.

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 programmer should not write $GIO routines for Wang-supported devices. For several such devices (e.g., nine-track tape), Wang-written $GIO microcommand sequences exist. When these microcommand sequences are available for a device, they are provided in the appropriate reference manual. The programmer must use the Wang-written microcommand sequences rather than creating user-written routines.

In Tables 15-1 to 15-13, the shaded areas indicate recommended microcommands. A programmer should be familiar with the I/O bus and BASIC-2 programming before attempting to write a $GIO routine. Additionally, a programmer should only use $GIO when necessary and should verify carefully the accuracy of all routines to avoid unrecoverable errors.

## The I/O Bus

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.

The microcommand tables at the end of this chapter list the signal sequences generated by each available $GIO I/O microcommand. Within these signal sequences, reference is made to four types of strobes: address strobes (ABS), output strobes (OBS and CBS) and input strobes (IBS).

A strobe is a short-duration change in the voltage level of a direct-current circuit. For example, in the circuitry of a Model 2250 I/O Interface Controller, a high-level signal between +2.4 and 3.6 volts DC represents a logic 0, and a low-level signal between 0 and +0.4 volts DC represents a logic 1. As shown in Figure 15-1, output strobes from a Wang 2200 CPU to an external device using a Model 2250 I/O Interface Controller have a pulse width of five microseconds ±10%. Input strobes from an external device to the Model 2250 must have a pulse width between 5 and 20 microseconds.

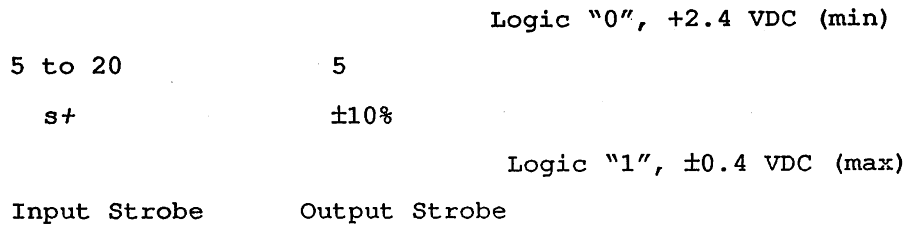

Figure 15-1. Schematic of Input and Output Strobes for the Model 2250 I/O Interface Controller

## Address Bus Strobes

A programmer enables peripheral devices by using an Address Bus Strobe (ABS), which transmits the address of the desired peripheral along the bus. The address is an 18-bit code representing the device-address rather than a data or control character. The controller with the corresponding address is then enabled; all other devices are disabled at that time. A programmer can use an additional ABS to change the enabled device at some other point during program execution.

## Output Bus Strobes

Data is output from the CPU to a device selected with an ABS by one of two methods: an Output Bus Strobe (OBS) or a Control Bus Strobe (CBS). An output strobe sent from the CPU 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: output data to be stored and control data (e.g., instructions). 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; on the other circuit, an output strobe is designated as a CBS. A byte of information on the eight parallel data circuits can be identified by using either an OBS or a CBS to indicate whether it is output data or control data.

If a microcommand table contains a code for a signal sequence that sends information with 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. 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 CRTs and printers. For peripherals that are interfaced to a system with the Model 2250 I/O Interface Controller, both OBS and CBS strobes are supported.

When executing a $GIO routine to output data, the programmer should use an OBS, unless a CBS is necessary. While both strobes carry data from the CPU to a device, most controllers interpret them differently. The OBS and the CBS are not generally interchangeable, and the programmer must know how an interfaced device is connected to such controllers before making decisions regarding the two types of output strobes.

### Wait For Ready

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 Wait for Ready (WR) at the beginning of a microcommand assures the programmer that the device is ready before the CPU makes an attempt to send data. Since it is possible to send a byte over the bus while the device is not able to receive it, the byte is then lost. Therefore, a programmer should always use a WR command unless certain that communication with the device can proceed without this precaution.

On multiuser systems, a programmer must use the WR command since methods of communication with peripherals that function on single-user systems fail on multiuser systems. A programmer using a multiuser system must understand the constraints that 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.

### Input Bus Strobes

When the CPU receives data from a device, the device sends an Input Bus Strobe (IBS) with the data. An IBS indicates to the CPU that there is data (8 bits in parallel) coming in from a peripheral device, awaiting transfer into the CPU. 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. When the CPU wants data from a device, it sets CPU READY and waits for the IBS. If this wait is longer than approximately 8 milliseconds and a I-92 occurs, data may be lost. Many devices avoid the error by using the READY signal to indicate that they are indeed prepared to send the IBS. If the WR takes too long, the CPU will breakpoint with no errors and no loss of data.

## General Forms of the I/O Statements

The general forms of the $IF ON/OFF and $GIO statements are described in the following sections.

## $IF ON/OFF

*Format*

```
        ON      device-address,       line-number
 $IF
         OF      File#,

 where:

    device-address =  /taa, where taa is the device-address of the
                      peripheral device to be tested.

             file# =  #n, where n is either an integer or a numeric-
                      variable whose truncated value lies within the
                       range 0-15, inclusive.  A file# 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.
                       (Refer to Chapter 7.)
```

The $IF ON/OFF statement determines the ready/busy status of a given device attached to the CPU and branches when ready or busy, depending upon which form of the statement is 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 BASIC-2 I/O statements. When it is necessary to test this signal, $IF ON/OFF allows conditional branching, depending upon the results of the test.

The $IF ON command tests the device ready/busy signal for the ready condition. If device ready 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 statement.

The $IF OFF statement also tests the ready/busy signal for device busy. If device busy is sensed, the program branches to the specified line. If device ready is sensed, program execution continues at the next statement.

The following list summarizes the device classes and their ready/busy characteristics:

*Console CRT Class* – $IF ON/OFF determines if the partition executing the statement is in the foreground or the background. If the terminal is attached to the partition, it is in the foreground and the signal is set to ready. If the terminal is not attached, the partition is in the background and the signal is set to busy. Example: console CRTs..

*Keyboard Class* – This class indicates the device is ready if one or more characters are ready to be sent to the CPU. Examples: keyboard, 2236 MXE Terminal Processor, 2250 Controller - input (even) address, 2252A Controller, 2227 Controller, and 2207 Controller.

*Note: $IF ON to the keyboard (/001) senses a ready status if the terminal is attached to the partition (foreground job) and a key is pressed. It senses a busy status if the terminal is detached (background) or if the terminal is attached (foreground) but no key is pressed.*

*Printer/Plotter Class* – This class indicates the device is ready when the controller is ready to receive at least one more byte. Examples: printers, plotters, typewriters and output writers, and 2254 Controller – main address during data output operations.

*Disk Class* – $IF ON/OFF is not recommended as a method for testing the state of a disk drive. This is because it cannot determine the READY/BUSY state of a multiplexed disk. Instead, $OPEN with the optional line number parameter should be used.

*Special Devices Class* – This class indicates the device is ready when the special device completes the previous operation. Examples: 2209A Nine-Track Tape Drive and 2227 B and 2228B Communications Controllers.

*Model 2250 8-bit Parallel I/O Controller (at odd addresses)* – This board has no built-in ready/busy signal. The 2250 defines a pin on the board's connector to be the origin of this signal. The meaning of the signal at this pin depends upon the external device. It is recommended that the device attached to this controller use the ready/busy signal in the same way as devices in the Printer/Plotter Class.

*Address /000* – This address is always ready with BASIC-2. Since device-address /000 is always busy on the 2200T, a programmer can use this address to test whether the system is a 2200T or a BASIC-2 system. A programmer can also use Address /000 as a dummy address for undesired output by selecting it for PRINT operations.

The programmer can use the following methods to specify a device-address in a $IF ON/OFF statement.

- Direct address specification using a slash followed by a 3-hexdigit device-address

  ```
  10 $IF OFF/028, 1000
  ```

- Indirect address specification using a file-number to which the desired device-address has been previously assigned

  ```
  10 SELECT #3/02B
  20 $IF ON #3, 250
  ```

- Omitting an address, thereby implying that the device-address currently selected for TAPE-class operations should be used

  ```
  10 SELECT TAPE/02A
  20 $IF ON 200
  ```

*Examples of Valid Syntax:*

```
$IF ON/028, 450

SELECT #7/02B
$IF ON #7, 100

SELECT TAPE/02A
$IF ON 400
```

## $GIO

*Format:*

```
                 device-address[,]
 $GIO [comment]                (     microcommand-sequence [,registers])
                  file#,
                  buffer   ;buffer    ...

 where:

                comment = character string consisting of uppercase
                          letters, digits, and spaces that identi-
                          fies the particular operation performed
                          by the $GIO sequence.

         device-address = /taa, where taa is the three-hexdigit
                          device-address of a specified I/O device.

                  file# = #n, where n is either an integer or a nu
                          meric-variable whose truncated value lies
                          within the range 0-15, inclusive.

   microcommand-sequence = A customized microcommand sequence which
                           defines the I/O operation as follows:

              registers = An alpha-variable whose individual bytes
                          (registers) store special characters and
                          error/status information; dimensioned
                          length must be at least 10 bytes.

                 buffer = An alpha-variable that serves as the
                          data buffer for multiple-character input
                          and output operations.
```

The $GIO statement uses a series of microcommands represented by a code that is two bytes (four hexadecimal digits) to write I/O routines. 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.

### Comment Parameter

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 is helpful when reviewing or revising a program (e.g., WRITE, READ, and CHECK READY). The comment has no functional purpose; the system ignores it during execution of the $GIO statement. The comment can consist of any combination of uppercase letters, digits, and blanks. Other characters are illegal in a comment.

### Device-Address Parameter

The first hex digit (t) represents the device-type. It has no significance during a $GIO operation and can be set to zero. The next two hex digits (aa) represent the unit-device address; they must correspond to the address that is preset on the I/O controller board of the device being accessed. If a programmer neither specifies an indirect nor absolute value, the address currently assigned to TAPE is the default.

The programmer can use the following methods to specify a device-address for a $GIO operation.

- Direct address specification using a slash followed by a 3-hexdigit device-address

  ```
  200 $GIO READ /03A (M$,R$) B$( )
  ```

- Indirect address specification using a file-number to which the desired device-address has been previously assigned

  ```
  300 SELECT #2/03A
  310 $GIO READ #2, (M$,R$) B$( )
  ```

- Omitting an address, thereby implying that the device-address currently selected for TAPE-class operations should be used

  ```
  400 SELECT TAPE/03A
  410 $GIO READ (M$,R$) B$( )
  ```

### File-Number Parameter

A file-number identifies a slot in the Device Table to which the device-address of the specified device is assigned with a SELECT statement.

### Microcommand-Sequence Parameter

Each microcommand in a sequence consists of a 4-hexdigit code of the form hhhh, where h is a hexadecimal digit (0 to 9 or A to F). The first two hex digits in a microcommand code usually identify the type of operation to be performed, such as single-character output with echo or multicharacter verify. The last two hex digits supply information, such as a character to be stored or a register containing a character to be transmitted. There is no practical limit on the number of microcommands that can be specified in a single $GIO statement.

The 21 different categories of microcommands available for use in a $GIO statement are of two basic types: I/O microcommands and control microcommands (refer to Table 15-2 for a summary of the 21 microcommand categories and their functions).

I/O microcommands send one or more characters to an external device or receive one or more characters from an external device. 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 (OD) character (carriage return) to the specified external device. Therefore, specifying a sequence of I/O microcommands is equivalent to programming the signal sequence required to perform the desired I/O operation.

Control microcommands provide the capability to program complete I/O routines, including testing and branching within a single $GIO statement. Control microcommands do not directly perform input or output operations, but they can perform the following important supplementary programming functions.

- Move, compare, and test specified characters; and set the condition code, depending upon the result of a test.
- Check error/status information in a register and set the condition code.
- Enable and disable the timeout condition.
- Enable and disable a delay before sending all output strobes.
- Branch to a specified microcommand within the $GIO sequence or terminate $GIO execution on condition code true or false.
- Set up the next specified data buffer.
- Move characters between the data buffer and the register.
- Increment or decrement the specified register or register pair.

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 (on) and false (off). Initially, the CPU sets the condition code to false. The condition code remains false until a condition occurs that alters the status. In general, the condition code is automatically set to true by a variety of special conditions, including certain error and termination conditions. Additionally, by using control microcommands, the programmer can 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 a branch instruction that tests the status of the condition code and initiates a branch within the $GIO routine. These two special microcommands have the following forms where "hhh" is the 3-hexadecimal-digit address of a microcommand within the microcommand sequence.

```
Dhhh (branch to hhh if condition code is true)
Ehhh (branch to hhh if condition code is false)
```

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 and so on.

*Example:*

If the condition code is true, the following instruction generates a branch to the second microcommand in the $GIO statement. If the condition code is false, a branch does not occur, and the next microcommand in the sequence is executed.

```
D001
```

If the condition code is false, the following microcommand generates a branch to the sixth microcommand in the $GIO statement. If the condition code is true, a branch does not occur, and the next microcommand in the sequence is executed.

```
E005
```

Both branch instructions also automatically reset the condition code to false.

In conjunction with the condition code, the branch instructions provide a means of responding to special conditions in an I/O operation without terminating $GIO statement execution. Additionally, branch instructions 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.

A programmer can specify the microcommand sequence defining a desired I/O operation either directly or indirectly in a $GIO statement. Direct specification of the microcommand sequence can occur by using one of the following methods.

- Hex digits

  *Example:* 10 $GIO (A000,R$) A$
- Hex literal

  *Example:* 10 $GIO (HEX(A000) , R$) A$

Indirect specification of the microcommand sequence is accomplished by assigning the microcommand codes to an alpha-variable and specifying the alpha-variable in a $GIO statement.

*Example:*

```
10 A$ = HEX(A000)
20 $GIO (A$,R$) B$
```

Line 10 and Line 20 are equivalent to the following statement.

```
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.

The alpha-variable specified as the registers parameter of a $GIO statement serves as a multipurpose memory area where special characters and error/status information are stored. Each byte of the 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 10 bytes because the system stores special information in bytes (or registers) 8, 9, and 10. The maximum number of registers that can be accessed is 15 (bytes 1to 15). If the register-variable contains more than 15 bytes, the programmer cannot use the additional bytes as registers.

The registers parameter 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 BASIC-2 program can neither read nor modify the values of particular registers.

The BASIC-2 program uses the registers for purposes such as storing data characters or special termination characters. It stores a binary value defining the duration of an implemented delay or timeout condition, or serves as a counter controlling the execution of a loop. By using certain control microcommands, the programmer can compare the contents of two registers, increment or decrement a specified register or register pair by a fixed amount, and test for specific error bits in the error register.

Although all 15 registers are available for use by the programmer, the system uses several registers to store information during certain operations. In particular, register 8 stores error/status information (each bit representing a specific error condition), and registers 9 and 10 maintain a count of the characters processed during a multicharacter input operation. Registers 5 and 6, respectively, 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 programmer must realize that the system alters the values of those registers during certain operations (refer to Table 15-13 for a summary of 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.

The data buffer parameter 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 restricted to single-character input or output.

A programmer can use alphanumeric scalar and array variables as data buffers. 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. A programmer uses the STR function to define the length of this specified portion.

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 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 uses that buffer.

For nonsequential processing of buffers, two control microcommands move the pointer to a specified buffer in the data buffer 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. Hex (00) is the address of the first buffer, hex (01) is the address of the second buffer, and so on. The address of the buffer to be used can be specified immediately in an 18hh microcommand, where <u>hh</u> is the address. Alternatively, the address 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. (Refer to Table 15-4.)

The data buffer pointer is moved only by one of the three microcommands (i.e., 1A00, 18hh, or 1Ah0). 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.

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 2-byte binary number stored in registers 9 and 10. The low-order 8 bits of the count are stored in register 10, and the high-order 8 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 that use the same buffer. The count is reset to binary zero only when the data buffer pointer is moved to a new buffer. 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 stores data in the remaining unused portion of the buffer and continues 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 is filled, the count continues to be updated to reflect the total number of characters received. However the additional characters are lost, and not stored.

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 is sent.

A multicharacter input operation, however, can be terminated when one of the following three conditions occurs: an ENDI character is received, a special termination character is received, or the character count equals the input buffer length.

- *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 register 6. However, only the eight data bits are stored; the ENDI bit is removed prior to storage. The count does not include the ENDI byte.

*Example:*

Termination upon reception of a special function character (i.e., by an ENDI bit that is sent) can be achieved in the following manner.

```
    60 $GIO /001 (C320, R$) A$
```

The system checks each incoming character to determine if it carries a special ninth bit (ENDI). If an ENDI character is received, the system saves the character in byte 6 of R$ and terminates the input operation according to LEND. Otherwise, the character is saved in A$ and the sequence continues. The character count is maintained in bytes 9 and 10 of R$.

- *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. If this termination condition is specified, the system compares each incoming character with the character stored in register 1/. The system terminates the input operation when the specified termination character is received. The special character can either be stored with the data characters in the data buffer or discarded.

*Example:*

The following example illustrates a variable length input that is terminated by the input of a specific character.

```
50 STR(R$,1,1) = HEX(0D)
60 $GIO /001 (C310, R$) A$)
```

Line 50 stores a carriage-return character in the first byte of R$. The third digit of the microcommand instructs the system to check byte 1 of R$ 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, the following statement calculates the number of characters input in the $GIO statement.

```
70 N = VAL(STR(R$,9,2),2)
```

- *Termination on Character 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).

*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 if the buffer is full. If so, the input sequence terminates according to the last digit in the microcommand (LEND).

If the character count is not specified as a termination condition, the number of characters received prior to termination can 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 or not these characters are stored .

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 as follows.

```
1.  ENDI Character
2.  Special Termination Character
3.  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. If only one termination condition is specified in the input command, the system does not set a bit in register 8.

Selection of combinations of termination conditions and LEND sequences can be accomplished by varying the last two digits of the C3 type microcommands according to the desired combination. For instance, the C330 microcommand terminates the input sequence when either a special character or an ENDI bit is received.

The following examples illustrate the use of different $GIO routines to output one or more characters to a device. Usually, a programmer displays information to the CRT by using standard BASIC-2 statements, such as PRINT or LIST rather than the $GIO statement. To simplify the illustration of these routines, the following examples use the CRT as the output device.

*Example:*

The following $GIO sequence displays the word WANG on the CRT screen using Immediate Mode output microcommands.

```
$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 that holds the data.

*Example:*

The following program will output WANG on the screen by using variables with the microcommand.

```
:10 R$="WANG"
:20 $GIO /005 (4210 4220 4230 4240, R$)
```

The first byte of each microcommand (42) waits for device ready. Then 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 the data buffer.

*Example:*

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 displayed, but only the first 100 bytes.

The following examples illustrate $GIO usage for data input from a single device. Usually the keyboard is accessed with the BASIC-2 statements KEYIN, INPUT, and LINPUT, rather than the $GIO statement. The following examples use the keyboard as the input device to provide examples that illustrate typical programming situations.

*Example:*

The following $GIO sequence receives three characters from the keyboard using single-character input microcommands. The first three bytes of R$ contain the three characters input from the keyboard.

```
:10 $GIO /001 (8701 8702 8703, R$)
```

If more than a few characters are to be input, multicharacter input commands should be used.

*Example:*

The following example accepts a string of characters from the keyboard and displays each character on the CRT as it is entered.

```
:10 $GIO (010D 7101 8702 7105 4220 1B22 1C12 E001, R$) A$
```

The microcommand sequence in Line 10 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$.
- The second microcommand (7101) sends an ABS to the device whose address is equal to the second byte of the microcommand (device type is omitted). In this example, the keyboard is now enabled
  `(address 01).`
- The third microcommand (8702) waits for the device to be ready and then receives an IBS (and, hence, one character) from the keyboard, storing it in the register specified by the last digit of the command.
- The fourth microcommand (7105) sends an ABS to the device whose address is equal to the second byte of the microcommand. In this example, the CRT is now enabled (address 05).
- The fifth microcommand (4220) waits for CRT ready and performs an OBS sending the character in register 2 back to the CRT.
- The sixth microcommand (1B22) takes the character in register 2, places it in the data buffer (A$), and increments the count maintained in registers 9 and 10. Additionally, this command sets the condition code and terminates the $GIO if the data buffer is already full.
- The seventh microcommand (1C12) compares the character in register 1 to that in register 2 and sets the condition code if the two are equal.
- The eighth microcommand (E001) tests the condition code. If it is false, a branch to the second microcommand takes place (001); otherwise, the sequence is ended.

The sequence can be terminated if either the data buffer becomes full or a carriage return is received. The data buffer A$ contains the characters received (including the carriage return), and the count is stored in registers 9 and 10.

**Table 15-1. Legend***

| 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 Special Character and full buffer 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. |

| 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 |

* Mnemonics used to describe signal sequences for I/) microcommands.

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 | |
| $1h_1h_2h_3$ | | Control – general | |
| $75h_1h_2$ | | Control – delay immediate | |
| 77r0 | | Control – delay immediate | Table 15–4 |
| $Da_1a_2a_3$ | | Branch Control | |
| $Ea_1a_2a_3$ | | Branch Control | |
| $4ch_1h_2$ | | Single Character output | |
| $5ch_1h_2$ | | Single Character output with acknowledge | Table 15–5 |
| $6ch_1h_2$ | | Single Character output with echo | |
| $8ch_1h_2$ | (c = 0–3, 8–B) | Single Character input with verify | |
| $8ch_1h$ | (c = 0–3, 8–B) | Single Character input | Table 15–6 |
| $9ch_1h_2$ | (c = 0–3, 8–B) | Single Character input with echo | |
| Ac0l | | Multicharacter output | |
| Bctl | (c = 0, 1, 4, 5 | Multicharacter output with acknowledge | Table 15–7 |
| 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 | |
| Cctl | (c = 0, 1, 4, 5) | Multicharacter input with echo | Table 15–8 |
| 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 |
|---|---|
| 0r$h_1h_2$ | 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. |
| 11$r_1r_2$ | Move contents of register $r_1$ to register $r_2$.. |
| 12r1 | Set a "course" 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 subsequeant 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.) |
| 13$d_1d_2$ | 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 $GIO sequence begins, this delay is set to HEX (OA) (50 microseconds).) |
| 14$r_1r_2$ | If contents of register $r_1$ ≠ contents of register $r_2$, set compare error bit (bit 08, register 8) to 1. |
| 15$r_1r_2$ | 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. |
| 16$a_1a_2$ | 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). |
| 17$a_1a_2$ | 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). |
| 18$a_1a_2$ | 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 an 10) to zero. |
| 19rc | Increment/decrement binary value stored in register r or in pair of registers r and r + 1, depending upon the 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 |

(continued)

**Table 15-4 Control Microcommands (continued)**

| Code | Operation |
|---|---|
| | 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. |
| 1Ar0 | 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 $r_2$. |
| $1Dr_1r_2$ | Set condition code if register pair $r_1$ $r_{1+1}$ = register pair $r_2$. $r_2$. + 1. |
| $1Er_1r_2$ | Set condition code if register $r_1$ register $r_2$.. |
| $1Fr_1r_2$ | Set condition code if register pair $r_1$ $r_2$. + 1 is greater than register pair $r_2$. $r_2$. + 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**

**Single Character Output**

| Code | Signal Sequence | Character to be Sent | Character to be Saved |
|---|---|---|---|
| 40$h_1h_2$ | WR, OBS/IMM | HEX ($h_1h_2$) | |
| 41$h_1h_2$ | OBS/IMM | HEX ($h_1h_2$) | |
| 42r0 | WR, OBS/IND | from register r | |
| 43r0 | OBS/IND | from register r | |
| 44 $h_1h_2$ | WR, CBS/IMM | HEX ($h_1h_2$) | |
| 45 $h_1h_2$ | CBS/IMM | HEX ($h_1h_2$) | |
| 46r0 | WR, CBS/IND | from register r | |
| 47r0 | CBS/IND | from register r | |

**Single Character Output with Acknowledge**

| Code | Signal Sequence | Character to be Sent | Character to be Saved |
|---|---|---|---|
| 50$h_1h_2$ | WR, OBS/IMM, W5, CPB, IBS | HEX ($h_1h_2$) | |
| 51$h_1h_2$ | OBS/IMM, W5, CPB, IBS | HEX ($h_1h_2$) | |
| 52$r_1r_2$ | WR, OBS/IND,W5, CPB, IBS, SAVE | from register r | into register $r_2$ |
| 53$r_1r_2$ | OBS/IND,W5, CPB, IBS, SAVE | from register r | into register $r_2$ |
| 54 $h_1h_2$ | WR, OBS/IMM, W5, CPB, IBS | HEX ($h_1h_2$) | |
| 55 $h_1h_2$ | CBS/IMM W5, CPB, IBS | HEX ($h_1h_2$) | |
| 56$r_1r_2$ | WR, OBS/IND,W5, CPB, IBS, SAVE | from register r | into register $r_2$ |
| 57$r_1r_2$ | OBS/IND,W5, CPB, IBS, SAVE | from register r | into register $r_2$ |

**Single Character Output with Echo**

| Code | Signal Sequence | Character to be Sent | Character to be Saved |
|---|---|---|---|
| 60$h_1h_2$ | WR, OBS/IMM, W5, CPB, IBS, VERIFY | HEX ($h_1h_2$) | |
| 61$h_1h_2$ | OBS/IMM, W5, CPB, IBS, VERIFY | HEX ($h_1h_2$) | |
| 62$r_1r_2$ | WR, OBS/IND,W5, CPB, IBS, SAVE, VERIFY | from register r | into register $r_2$ |
| 63$r_1r_2$ | OBS/IND,W5, CPB, IBS, SAVE, VERIFY | from register r | into register $r_2$ |
| 64 $h_1h_2$ | WR, CBS/IMM, W5, CPB, IBS, VERIFY | HEX ($h_1h_2$) | |
| 65 $h_1h_2$ | CBS/IMM W5, CPB, IBS, VERIFY | HEX ($h_1h_2$) | |
| 66$r_1r_2$ | WR, CBS/IND,W5, CPB, IBS, SAVE, VERIFY | from register r | into register $r_2$ |
| 67$r_1r_2$ | CBS/IND,W5, CPB, IBS, SAVE, VERIFY | from register r | into register $r_2$ |
| 68$h_1h_2$ | WR, OBS/IMM, W5, CPB, IBS, VERIFY | | |
| | SET CC (if VFY bit set) | HEX ($h_1h_2$) | |
| 69$h_1h_2$ | OBS/IMM,W5, CPB, IBS, VERIFY | | |
| | SET CC (if VFY bit set) | HEX ($h_1h_2$) | |
| 6A$r_1r_2$ | WR, OBS/IND, W5, CPB, IBS, SAVE, VERIFY | | |
| | SET CC (if VFY bit set) | from register r | into register $r_2$ |
| 6B$r_1r_2$ | OBS/IND, W5, CPB, IBS, SAVE, VERIFY | | |
| | SET CC (if VFY bit set) | from register r | into register $r_2$ |

(continued)

**Table 15-5. Single Character Output Microcommands (continued)**

**Single Character Output with Echo**

| Code | Signal Sequence | Character to be Sent | Character to be Saved |
|---|---|---|---|
| 6C$h_1h_2$ | WR, CBS/IMM,W5, CPB, IBS, VERIFY | | |
| | SET CC (if VFY bit set) | HEX ($h_1h_2$) | |
| 6D$h_1h_2$ | CBS/IMM,W5, CPB, IBS, VERIFY | | |
| | SET CC (if VFY bit set) | HEX ($h_1h_2$) | |
| 6E$r_1r_2$ | WR, CBS/IND,W5, CPB, IBS, SAVE, VERIFY | | |
| | SET CC (if VFY bit set) | from register r | into register $r_2$ |
| 6F$r_1r_2$ | CBS/IND,W5, CPB, IBS, SAVE, VERIFY | | |
| | SET CC (if VFY bit set) | from register r | into register $r_2$ |

**Table 15-6. Single Character Input Microcommands**

**Single Character Input**

| Code | Signal Sequence | Verify Character | Character to be Saved |
|---|---|---|---|
| 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**

| Code | Signal Sequence | Verify Character | Character to be Saved |
|---|---|---|---|
| 80$h_1h_2$ | CPB, IBS, VERIFY/IMM | HEX ($h_1h_2$) | |
| 81$h_1h_2$ | WR, CPB, IBS, VERIFY/IMM | HEX ($h_1h_2$) | |
| 82$r_1r_2$ | CPB, IBS, SAVE, VERIFY/IMM | in register $r_1$ | into register $r_2$ |
| 83$r_1r_2$ | WR, CPB, IBS, SAVE, VERIFY/IND | in register $r_1$ | into register $r_2$ |
| 88 $h_1h_2$ | CPB, IBS | | |
| | SET CC (if VFY bit set) VERIFY//IMM | HEX ($h_1h_2$) | |
| 89 $h_1h_2$ | WR, CPB, IBS | | |
| | SET CC (if VFY bit set) VERIFY//IMM | HEX ($h_1h_2$) | |
| 8A$r_1r_2$ | CPB, IBS, SAVE VERIFY/IND | | |
| | SET CC (if VFY bit set) | in register $r_1$ | into register $r_2$ |
| 8B$r_1r_2$ | WR, CPB, IBS, SAVE VERIFY/IND | | |
| | SET CC (if VFY bit set) | in register r | into register $r_2$ |

(continued)

**Single Character Input with Echo**

| Code | Signal Sequence | Verify Character | Character to be Saved |
|---|---|---|---|
| 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**

**Multicharacter Output**

| Code | Signal Sequence * |
|---|---|
| A001 | (WR, DATAOUT/OBS), LEND |
| A101 | ( DATAOUT/OBS), LEND |
| A201 | 25 microsecond version of a001; no timeout or delay |
| A401 | (WR, DATAOUT/CBS), LEND |
| A501 | ( DATAOUT/CBS), LEND |
| A601 | SCAN DATA BUFFER, CALCULATE LRC, LEND |

**Multicharacter Output with Acknowledge**

| Code | Signal Sequence * | |
|---|---|---|
| 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**

| Code | Signal Sequence * |
|---|---|
| B2tl | (WR, DATAOUT/OBS, W5, CPB, IBS, VERIFY, CHECK T), LEND |
| B3tl | ( DATAOUT/OBS, W5, CPB, IBS, VERIFY, CHECK T), LEND |
| B6tl | (WR, DATAOUT/CBS, W5, CPB, IBS, VERIFY, CHECK T), LEND |
| B7tl | ( DATAOUT/CBS, W5, CPB, IBS, VERIFY, CHECK T), LEND |

**Multicharacter Output with Each Character Requested**

| Code | Signal Sequence * |
|---|---|
| B8tl | ( CPB, IBS, CHECK T, WR, DATAOUT/OBS), LEND |
| B9tl | ( CPB, IBS, CHECK T, DATAOUT/OBS), LEND |
| BCtl | ( CPB, IBS, CHECK T, WR, DATAOUT/OBS), LEND |
| BDtl | ( CPB, IBS, CHECK T, DATAOUT/OBS), LEND |

(continued)

Table 15-7. **Multicharacter Output Microcommands (continued)**

| | Multicharacter Verify |
|---|---|
| **Code** | **Signal Sequence *** |
| BAt0 | (CPB, IBS, VERIFY, CHECK T) |

*A sequence in parentheses is repeated for each characater in the data buffer.

**Table 15-8. 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 eithe condition, set cc | | 3 | | B |

*These termination conditions end the microcommand, not the $GIO, and do not set the condition code.

**Table 15-9. Valid LEND Codes for Table 15–7**

| LRC** End Sequence / Microcommand | Actl | B0tl through 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 15-10. Multicharacter Input Microcommands***

**Multicharacter Input**

| Code | Signal Sequence |
|---|---|
| C22l | (CPB, IBS, no timeout or delay, CHECK ENDI, SAVE DATA), LEND |
| C3tl | (WR, CPB, IBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |
| C6tl | (CPB, IBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |
| C7tl | (Delay 50 , CPB, IBS, CHECK $T_1$, SAVE DATA , CHECK $T_2$), LEND |

**Multicharacter Input with Echo**

| Code | Signal Sequence |
|---|---|
| C0tl | (CPB, IBS, WR, ECHO/OBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |
| C1tl | (CPB, IBS, ECHO/OBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |
| C4tl | (IBS, WR, ECHO/OBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |
| C5tl | (CPB, IBS, ECHO/OBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |

**Multicharacter Input with Each Character Requested**

| Code | Signal Sequence |
|---|---|
| C8tl | (WR, OBS, W5, CPB. IBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |
| C9tl | (OBS, W5, CPB. IBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |
| CAtl** | (CPB, WR, OBS, IBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |
| CBtl** | (CPB, OBS, IBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |
| CCtl | (WR, CBS, W5, CPB, IBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |
| CDtl | (CBS, W5, CPB, IBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |
| CEtl** | (CPB, WR, OBS, IBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |
| CFtl** | (CPB, CBS, IBS, CHECK $T_1$, SAVE DATA, CHECK $T_2$), LEND |

*(A sequence in parentheses is repeated until a valid termination condition occurs. In each command, the t–value is the termination code; the 1–value the LEND code.)

**The four indicated microcommands cannot be used with a 2200 multiuser operating system. An illegal microcommand error results.

Table 15-11. Valid CHECK T Codes for Table 15–10

| Termination Conditions* (order of checking from left to right) | | | |
|---|---|---|---|
| t | 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 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.

Table 15-12. Valid LEND Codes for Table 15–10

| 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 15-13. 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 characater. |
| 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 | 1 = Buffer overflow. |
|  | 02 | 1 = LRC error. |
|  | 04 | 1 = Echo/Verify error. |
|  | 08 | 1 = Compare error. |
|  | 10 | 1 = Timeout. |
|  | 20 | 1 = ENDI–level termination |
|  | 40 | 1 = Special character termination. |
|  | 80 | 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:

80 40 20 10 08 04 02 01

Low–order hexdigit 8–4–2–1 bit positions.

High–order hexdigit 8–4–2–1 bit positions.

***Note:*** *Registers 8, 9, and 10 are zeroed by the system when $GIO begins execution. Registers 9 and 10 are zeroed again whenever an 18a1a2, 1A00, or 1Ar0 command is executed.*

16

# Multiuser Operation

## Overview

Multiuser BASIC-2 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, Multiuser BASIC-2 is loaded. Power-on causes the "MOUNT SYSTEM PLATTER" message to be displayed at terminal #1; the operator at that terminal uses the appropriate function key to load the BASIC-2 from the system platter.

The utility program @GENPART is now loaded and executed at terminal #1. This program prompts 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 BASIC-2 statement $INIT. This statement directs the system to allocate resources in the prescribed manner and create the specified system configuration. The system is then configured as a multiuser system with each terminal assigned one or more memory partitions.

## Functional Description

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 use the system in turn while preventing individual users from interfering with each other's computations.

With Multiuser BASIC-2, each memory partition behaves much like a single-user system. 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, 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 maintains control of the terminal for as long as desired. Because of this, messages from other partitions cannot appear and disturb the CRT display at undesirable times. A background partition may 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.

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 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, with Multiuser BASIC-2, 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 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 entitled "Peripheral Allocation" and "Multiuser Language Features".

Regarding the system as a whole, it appears that all partitions are executing simultaneously. Because all partitions share a single CPU, 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 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-2 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, 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 Multiuser BASIC-2 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 low operating system overhead when compared with most other multiuser systems. More importantly, breakpoints may occur in the middle of BASIC-2 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. For example, 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 system 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 system can give several timeslices to other partitions between operator keystrokes. Similarly, several partitions can be serviced during a carriage return on a printer.

To accomplish the overlap of CPU and I/O processing, I/O devices must be buffered and microprocessors must frequently be used to relieve the CPU of responsibility for controlling the peripheral. The most sophisticated of these "intelligent" peripheral controllers is the 2236MXE terminal controller, which handles many I/O operations to the terminals which otherwise would require the attention of the CPU. For example, the CPU does not perform INPUT or LINPUT statements; instead, it asks the microprocessor in the 2236MXE 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 function key. The 2236MXE also performs line-editing functions at the terminal CRT to move the cursor and to insert and delete characters. In dealing with the 2236MXE, the 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 Multiuser BASIC-2 in some detail. These discussions are followed by the general formats of all BASIC-2 statements designed for multiuser operation. The final section of this chapter discusses programming considerations and suggests some good programming practices.

## User Memory Allocation

Memory is divided into two distinct kinds. The operating system, BASIC-2 interpreter, and other system data are contained in *control memory*. The control memory is completely separate from memory available for user programs and data. When referring to the memory size of a system, only the user memory is considered.

User memory is divided into areas known as "banks". A bank contains a maximum of 64K bytes of user memory. A system containing more than 64K bytes of user memory contains more than one bank, depending on the amount of memory purchased.

Multiuser BASIC-2 uses 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 system 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; so a partition may not extend from one bank to another. Figure 16-1 illustrates memory bank organization.

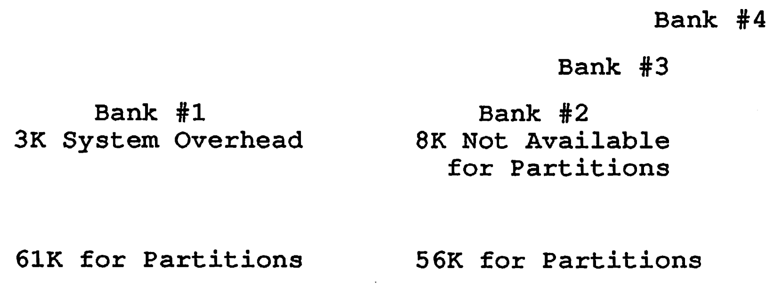

**Figure 16-1. Memory Bank Organization**

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. In effect, 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. 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.

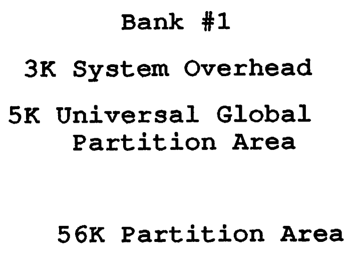

**Figure 16-2. The Universal Global Area**

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.

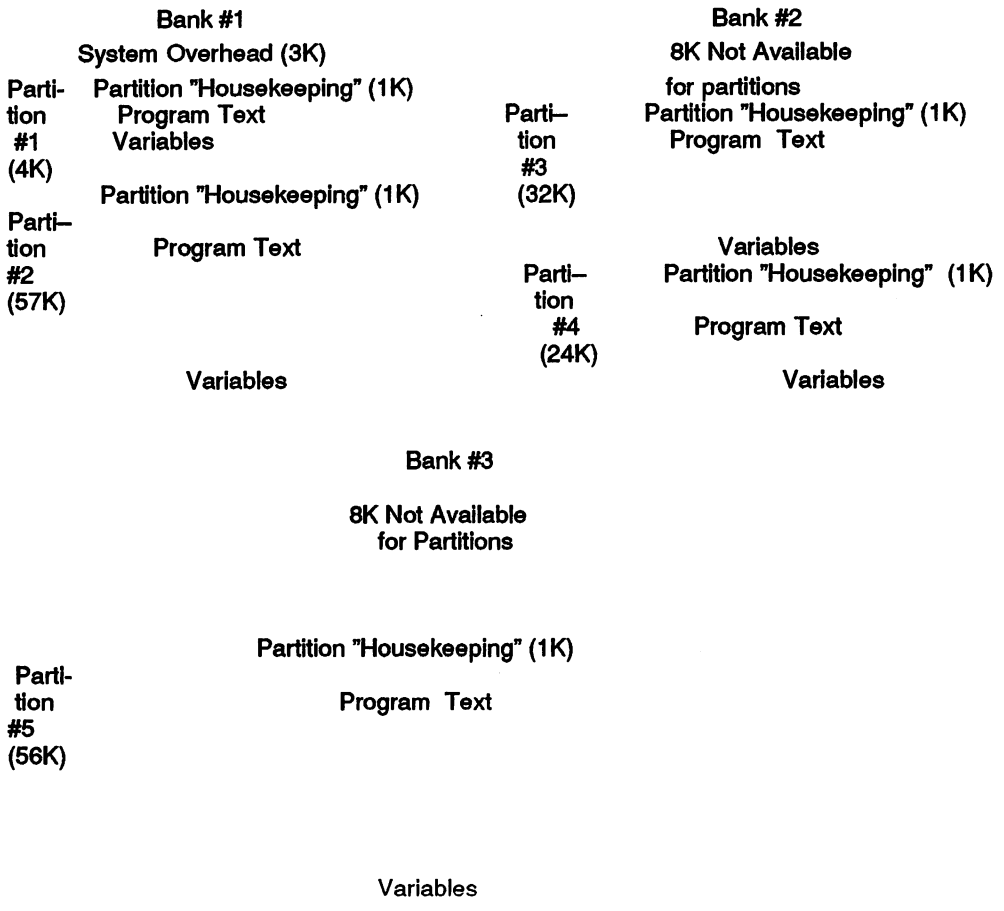

**Figure 16-3. A Multibank System Configuration**

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 in Figure 16-3) 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.

## Peripheral Allocation

By default, all peripherals attached to the system 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 operating system enables a partition to 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. 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. 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 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 multi-address 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:  MXE addresses
40:RAM Disk address
80:  memory selection (operating system use only)
```

## 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 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). Multiuser BASIC-2 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.

## Disabled Programming

Multiuser BASIC-2 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 al-
waysimplied).
```

## 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 follows:

```
*** 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.

## 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 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 desired, 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 the section entitled "Global Partitions".)

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. 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.

## Background Terminal Printer Operation

Printers can be physically attached to either a printer controller in the CPU or directly to a 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 operating system 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 2236MXE 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.

## Global Partitions

Multiuser BASIC-2 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 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.

```
       Calling Partition                        Global Partition
  Calling Partition
    10 SELECT @PART                   10 DEFFN @PART                    10 SELECT @PART
          "COMTEXT"                            "COMTEXT"
"COMTEXT"
    20  GOSUB' 100                          20 DEFFN' 100
     30



            40 GOSUB' 100
    50 END                                 50 RETURN

            50 END
```

**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. As a result of this, 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 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, references to local variables in the global subroutine actually refer to variables in the original calling partition. Because of this, 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.

```
 Calling Partition            Level 1                    Level 2
                           Global Partition          Global Partition

10 SELECT @PART  "@P1"     10 DEFFN @PART  "@P1"     10  DEFFN @PART "@P2"
20  GOSUB' 1               20 DEFFN' 1
30                          30
40                         40  SELECT @PART "@P2"
50                         50  GOSUB'2               100  DEFFN' 2
60                         60 RETURN                 110
70                         70 DEFFN' 3               120 RETURN
80 GOSUB' 3                  80
90 END                     90 RETURN
```

**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. 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. As a result, a global subroutine can select another global partition without affecting the pointers of the originating partition. (See the section "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, 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 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
                 ↑ ERROR C62:DIVISION BY 0
```

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, Multiuser BASIC-2 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, 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. 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. 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.

```
  Calling Partition           Global Partition             Calling Partition

10 SELECT @PART "GLOB"       10 DIM @X, @Y              10 SELECT @PART "GLOB"
20 A$ = "ABCDE"              20 A = 1                    20 A$ = "FGHIJ"
30 A = 1000                 30 @X = 123                   30 a = 50
40 Q = @X + @Y               40 @Y = 456                 40 R = @X + @Y
50 PRINT "Q = ";Q            50 DEFFN @PART "GLOB"       50 PRINT "R = ";R
                             60 STOP
                           70 B = 2

   Variable                   Variable                    Variable
    Table                      Table                       Table

    Q   579                  @X   123                    R     579
    A  1000                  @Y   456                    A     v 50
   A$  ABCDE                   A     1                  A$  FGHIJ
```

**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", 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 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. 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 should be 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 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 the following:

```
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 C62 (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                 40 GOSUB' 1 (10,5)
10 DEFFN'1 (A,B)          50 PRINT C
20 C = A/B                60 END
30 RETURN                 70 DEFFN'1 (A,B)
40 GOSUB'1 (10,5)         80 C = A/B
50 PRINT C                90 RETURN
60 END
```

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, 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. 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". For example, 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.

```
Program
   .
   .
   .
100 GOSUB '1
110 PRINT A
   .
   .
   .
1000 DEFFN'1
2000 RETURN
```

| Step 1 | Step 2 | Step 3 |
|---|---|---|
| GOSUB' executed. Current value of text pointer saved on stack. | DEFFN' located. Its location replaces current value of text pointer and execution is passed to that point. | RETURN executed. Old value of text pointer is removed from stack and replaces current value of text pointer. Execution resumes at that point. |
| Text Ptr<br>110, 1 | Text Ptr<br>1000, 1 | Text Ptr<br>110, 1 |
| Stack<br>110, 1 | Stack<br>110, 1 | Stack<br>undefined |

**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, 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.

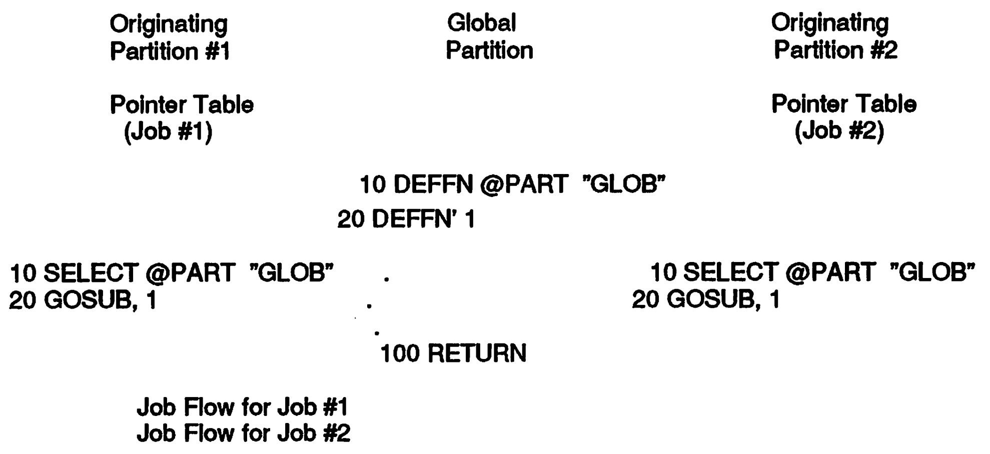

**Figure 16-9. Job Flow Between Originating Partitions and Global Partition**

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 follow:

- *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.

**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 GOSUB' | Global variable reference, |
| 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. 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 or GOSUB' modify the DATA Partition#, nor does RETURN affect its value. Following a global subroutine call, 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. 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.

**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#<br>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<br>Text Partition# | Retrieves old values of text pointer, Text Partition#, and Global Partition# from stack and places them back in Pointer Table. |
| RESTORE | DATA Partition# | Sets DATA Partition# equal to Text Partition#. |

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.

## Terminal Connect/Disconnect Detection

The terminal connect/disconnect detection facility enables BASIC-2 programs to be written to monitor system users and their use. With this facility, the operating system can detect when a terminal is connected or disconnected. Upon disconnection, the system can initiate procedures specified by the programmer. The $DISCONNECT statement enables or disables disconnect detection.

The following list summarizes the features of the terminal connect/disconnect detection facility:

- Connect detection for partition allocation and initiation of a user log on program.
- Forced disconnection if the user does not log on within the program specified time. This prevents a user from tying up a port without completing the log on procedure.
- Program control of the connect/disconnect facility through BASIC-2.
- Disconnect detection for initiation of user log off programs.
- TIME and DATE function for logging system use.

The connect/disconnect facility requires the 2236 MXE Terminal Processor; it does not operate with the 2236 MXD.

## Multiuser Language Features

Table 16-3 explains the functions provided to control multiuser operations.

Table 16-3. **Multiuser Functions**

| Function | Action |
|---|---|
| $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. |
| $BREAK use | Relinquishes a specified amount of the current partition's execution time for 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. |

## $BREAK

*Format:*

```
     $BREAK     numeric-expression
                  !

  where:

     0  <= numeric-expression  < 256, default = 1
```

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. $BREAK! can be used to end the execution of a global partition running in the background without the program requiring interaction with the terminal. Note that subsequent execution of "$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

*Format:*

```
                file#              ,file#
    $CLOSE                                                 ...
                device-address     ,device-address
```

```
  where:

    device-address = /taa, where t = device-type and aa = physical
                     device-address.
           file# = #n, where n is an integer or numeric-variable
                     whose value is 0-15.
```

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

*Format:*

```
                alpha-variable
DEFFN @PART                              FOR terminal#   ,terminal#    ...
                literal-string

where:

               terminal# = numeric-expression
```

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 the section entitled "User Memory Allocation".) 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
```

## $DISCONNECT

*Format:*

```
                    ON    [numeric-expression]
    $DISCONNECT
                     OFF
```

```
  where:

      0 ≤ numeric-expression ≤ 65534
```

The $DISCONNECT statement enables or disables terminal disconnection detection. Once enabled, the operating system can detect when a terminal is disconnected. A local terminal is considered to be disconnected when the terminal is powered off. A remote terminal is disconnected when the phone communication is terminated. The operating system cannot distinguish between disconnections caused by timeout, modem disconnection, or terminal power off.

A $DISCONNECT ON statement enables disconnect detection. A $DISCONNECT OFF statement disables disconnect detection; this is the default state of the system. Issuing a subsequent $DISCONNECT statement *always* overrides the previous command and sets a new state.

The optional expression in the $DISCONNECT ON statement represents the time in seconds after which the operating system forces a disconnection. If specified, the operating system forces a disconnection of the terminal when the value of the expression equals zero. The counter remains at zero until another $DISCONNECT ON statement sets the time expression. After the terminal is disconnected, the next terminal at the same port is not disconnected until the execution of another $DISCONNECT ON statement.

$DISCONNECT is a command to the port attached to the partition issuing the statement and not to the partition. Therefore, changing partitions or terminals does not affect the way $DISCONNECT affects a specific port (i.e., $DISCONNECT remains in effect after the execution of a $RELEASE PART or $RELEASE TERMINAL statement).

### Terminal Connect Detection

A local terminal is considered to be connected when it is powered on. A remote terminal is considered to be connected when the phone link between the terminal and the terminal processor is established.

When the operating system detects a terminal connection or a RESET from a terminal that does not control any partitions, the operating system automatically assigns an available partition to that terminal. All partitions assigned to the null terminal (i.e., terminal 0) and waiting for a terminal (i.e., having accessed a statement that performs I/O to the terminal, such as PRINT or LINPUT) are available partitions. If the terminal already controls one or more partitions, or if no available partitions exist, the operating system does not perform any action.

Once a partition is assigned to a terminal, this partition can then execute a log on program that issues a $DISCONNECT ON statement and sets a time limit for the user to complete the log on sequence. (Refer to Example 1: Sample Log On Program). If the user does not complete the log on sequence within the program specified time, the operating system automatically disconnects the user. Since a user can initiate but cannot complete the log on procedure, forced disconnection prevents the user from tying up a terminal port. If, however, the user properly completes the log on procedure, the forced disconnection can be overridden by executing a $DISCONNECT ON statement with no specified disconnect time.

*Example 1: Sample Log On Program*

```
0010 REM $CONNECT - Sample Log On Program
0020 REM To log system users, run this program (or similar
: REM program) in all released partitions.  When a terminal is
: REM connected, the operating system assigns a partition to
  the
: REM terminal and begins execution of this program.
0025 REM For protection, the partition should be nonprogram
  mable.
0030 REM%
```

Disconnect terminal if not logged on within 2 minutes.

```
: $DISCONNECT ON 120
0040 REM%
```

Title

```
: PRINT HEX(03); AT(0,30); "LOG ON"
0050 REM%
```

Obtain user's password

```
: PRINT AT(10,20,8); "Enter password: ";
: FOR I = 1 TO 8
: KEYIN STR(P%,I,1)
: PRINT HEX(8B);
: NEXT I00
60 REM%
```

Verify user's password and log on user

```
: REM Perform any required log on procedures
0070 REM%
```

Turn off the disconnect timeout

```
: $DISCONNECT ON
0080 REM%
```

Load menu

```
: LOAD RUN T "START"
: REM START should check that user has logged on
```

**Terminal Disconnect Detection**

The operating system is informed of disconnections only if disconnect is enabled. Upon detection of a disconnect, the operating system forces all partitions assigned to that terminal to run a user-written BASIC program called $DISCNCT. (Refer to Example 2: Sample Log Off Program.) The process is functionally equivalent to typing RESET followed by LOAD RUN "$DISCNCT". However, running the program occurs independently of the terminal.

The $DISCNCT program can perform any required log off or accounting procedures concerning system use. Typically, this program makes the partition available to other users by executing a $RELEASE PART statement. After it is initiated, the released partition can execute a log on program that interacts with the user assigned this partition when terminal connections are detected.

If disconnect detection is not enabled for the terminal port, the operating system does not perform any action following the disconnection.

*Example 2: Sample Log Off Program*

```
0010 REM%
  $DISCNCT - Sample Log Off Program
0020 REM If terminal disconnect detection is enabled and a
  terminal
: REM disconnects, the operating system automatically runs the
: REM $DISCNCT program in all partitions assigned to that ter
  minal
0030 REM%
```

Release the partition

```
: $RELEASE PART
0040 REM%
```

Log off the user

```
: REM Perform any required log off procedures
0050 REM%
```

Load a log on program

```
: LOAD T "$CONNECT"
```

*Examples of valid syntax:*

```
$DISCONNECT ON
$DISCONNECT ON 60
$DISCONNECT OFF
```

## $INIT

*Format 1 (Program mode):*

```
$INIT (alpha-1, alpha-2, alpha-3, alpha-4, alpha-5 [, alpha-6]
[,alpha-7]
```

*Format 2 (Immediate mode):*

```
$INIT password

where:

                literal string
 alpha     =
                alpha-variable


 password =   system reconfiguration password, which must be a
             literal string of one to eight characters in length.
```

The special-purpose statement $INIT passes the system configuration parameters to the BASIC-2 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"; which requires the operator on terminal #1 to 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 a program is to be
                  bootstrapped into 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 = optional reconfiguration password.

    Length of string  >= 8 bytes.
    1st eight bytes are the password.

alpha-7 = optional printer driver associations

    Length of string > = 10 bytes

    Where:

    bytes 0-7 = printer driver table file name
    byte 8 = device address in hex
    byte 9 = terminal numbers in hex, if byte 8 is 04; otherwise,
            0 total length of string > = 150 bytes (unused bytes
             must be set to hex 20)
```

*Examples of Valid Syntax:*

```
$INIT "SYSTEM"
10 $INIT (S$,T$,M$,N$,D$)
20 $INIT (S$,T$,M$,N$,D$, P$)
```

## $MSG

*Format:*

```
$MSG = alpha-expression
```

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

*Format:*

```
                      file#            ,file#
$OPEN   line-number,                                    ...
                      device-address   ,device-address

where:

   device-address =  /taa, where t = device-type and aa =
                    physical device-address.

            file# =  #n, where n is an integer or numeric-variable
                     whose value is 0-15.
```

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 the device is a disk, a branch to a specified line number will be made if it is not ready. This will happen when the disk is not powered on, is multiplexed, or is being used by some other CPU. 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 /D12 or /D13; the entire disk unit must be hogged.

*Examples of Valid Syntax:*

```
10 $OPEN /215
20 $OPEN 100, /215, /02A, /02B
30 $OPEN #3
```

## $PSTAT

*Format (as a statement):*

```
   $PSTAT = alpha-expression

  where:
          alpha-expression is the same as for LET

  Format (as a function within an alpha-expression):
        $PSTAT (numeric-expression)

  where:

         1 < = numeric-expression < = number of partitions
```

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 Multiuser BASIC-2, "V"
         if VP BASIC-2)

byte 10 = operating system release number
```

The release number is stored in packed decimal; HEX(12) represents 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 = programming status
```

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

*Format:*

```
$RELEASE PART
```

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 pseudo terminal 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 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 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

*Format:*

```
                              numeric-expression
$RELEASE TERMINAL   TO                                  ,STOP
                              partition-name

where:

                           literal-string
partition-name =                                  (1-8 bytes in length)
                           alpha-variable
```

The $RELEASE TERMINAL statement detaches the terminal from the current partition. The 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 the section entitled "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
```

*Format*

```
SELECT @PART partition-name

where:

                       alpha-variable
partition-name =                               (1-8 bytes in length)
                       literal-string
```

**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. 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 now 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
```

## Programming Considerations

This section consists of two parts: the first subsection discusses general programming considerations for Multiuser BASIC-2, while the second subsection enumerates specific areas of incompatibility between Multiuser BASIC-2 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 Multiuser BASIC-2 and other Wang systems. The discussion of compatibility focuses on those differences which would prevent a program written for another Wang system from running with Multiuser BASIC-2. This discussion is intended primarily for programmers who must modify existing software to run with Multiuser BASIC-2.

### 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.

## Considerations for Time-Dependent Programs

1. The execution time for a given program 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. Multiuser BASIC-2 terminal communication speed is slower than the speed of the 2226CRT and other CRT models. As a result, programs written to update the entire screen may perform slowly when run with Multiuser BASIC-2. 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 MXE controller rather than the CPU and 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 become minimum delay loops with Multiuser BASIC-2. The preferred method of implementing time delays requires the use of a SELECT P statement. SELECT P is timed by the 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.

## 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 MXE controller; 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 only the terminal. 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.

## I/O Statement Restrictions

The following table defines the devices with which the operating system permits individual statements to communicate. ERR P48 results if a statement addresses an illegal device.

**Table 16-4. Devices to Which BASIC-2 Statements Communicate**

| Statement Or Operation | Terminal Keyboard | Terminal CRT | Terminal Printer | Devices Other than 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 | | | | X |
| Disk Statements | | | | X |

*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.

## Default Disk Address

The default disk address is set to the address of the drive from which the system was loaded. For example, loading by pressing function key '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.

## Special Features of the 2236MXE Controller

1. The 2236 MXE 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 MXE 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 with Multiuser BASIC-2.

A

# Key Codes and Character Sets

## Standard Key Codes

Table A-1 shows the standard keys and the expected ASCII key codes produced in hexadecimal notation.

**Table A-1.** **BASIC-2 Standard Key Codes**

| CHARACTERS | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **HEX** | **CHAR** | **HEX** | **CHAR** | **HEX** | **CHAR** | **HEX** | **CHAR** | **HEX** | **CHAR** | **HEX** | **CHAR** |
| 20 | SPACE | 30 | 0 | 40 | @ | 50 | P | 60 | ° | 70 | p |
| 21 | I | 31 | 1 | 41 | A | 51 | Q | 61 | a | 71 | q |
| 22 | ” | 32 | 2 | 42 | B | 52 | R | 62 | b | 72 | r |
| 23 | • | 33 | 3 | 43 | C | 53 | S | 63 | c | 73 | s |
| 24 | $ | 34 | 4 | 44 | D | 54 | T | 64 | d | 74 | t |
| 25 | ” | 35 | 5 | 45 | E | 55 | U | 65 | e | 75 | u |
| 26 | & | 36 | 6 | 46 | F | 56 | V | 66 | f | 76 | v |
| 27 | ’ | 37 | 7 | 47 | G | 57 | W | 67 | g | 77 | w |
| 28 | ( | 38 | 8 | 48 | H | 58 | X | 68 | h | 78 | x |
| 29 | ) | 39 | 9 | 49 | I | 59 | Y | 69 | i | 79 | y |
| 2A | # | 3A | : | 4A | J | 5A | Z | 6A | j | 7A | z |
| 2B | + | 3B | ; | 4B | K | 5B | [ | 6B | k | | |
| 2C | ’ | 3C | < | 4C | L | 5C | \ | 6C | l | | |
| 2D | – | 3D | = | 4D | M | 5D | ] | 6D | m | | |
| 2E | . | 3E | > | 4E | N | 5E | ↑ | 6E | n | | |
| 2F | /\ | 3F | | 4F | O | 5F | ← | 6F | o | | |

## Special Key Codes

Table A-2 shows the special keys and the expected key codes shown in hexadecimal notation. The (’) indicates that the key produces a function code.

**Table A-2. BASIC-2 Special Key Codes**

| Key | Unshifted | Shifted |
|---|---|---|
| INDENT | '00 | '10 |
| PAGE | '01 | '11 |
| CENTER | '02 | '12 |
| DECTAB | '03 | '13 |
| FORMAT | '04 | '14 |
| MERGE | '05 | '15 |
| NOTE | '06 | '16 |
| STOP | '07 | '17 |
| SRCH | '08 | '18 |
| REPLC | '09 | '19 |
| COPY | '0A | '1A |
| MOVE | '0B | '1B |
| COMMAND | '0C | '1C |
| Up/down arrows (↑↓) | '0D | '1D |
| Blank Function key | '0E | '1E |
| GO TO | '0F | '1F |
| PREV | '42 | '52 |
| NEXT | '43 | '53 |
| South cursor control key (↓) | '44 | '55 |
| North cursor control key (↑) | '46 | '56 |
| DELETE | '49 | '59 |
| INSERT | '4A | '5A |
| East cursor control key (→) | '4C | '5C |
| West cursor control key (←) | '4D | '5D |
| ERASE | '48(or E5) | E5 |
| DTAB/RCALL | '4F | '5F |
| ERASE | '48 | 'E5 |
| GL | '7C | '7D |
| TAB/FN | '7E | '7F |
| CANCEL | 'F0 | '50 |
| BACKSPACE | 08 | 08 |
| CLEAR | 81 | 82 |
| RUN | 82 | 82 |
| EXEC | 82 | A1 |
| CONTINUE | 84 | 84 |
| LOAD | A1 | A1 |
| _ (underscore) | FF 'A0 | FF 'A0 |

## Character Sets

The figures in this section illustrate two BASIC-2 character sets. Figure A-1 shows the character set the default font produces. The system selects this font by default when it enters BASIC-2. Figure A-2 shows the alternate font character set; this character set differs from the default font character set in the values from HEX(90) through HEX(FF).

To change the character set from the default font character set to the alternate font character set, you need only enter the Immediate mode statement PRINT HEX (0202020F). To change the character set from the alternate font character set to the default font character set, enter the Immediate mode statement PRINT HEX (0202000F).

|  | High–order HEX Digit |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |
| **0** | a | space | 0 | @ | P | ° | p | space | a | _ | 0 | @ | P | ° | p |
| **1** | e | ! | 1 | A | Q | a | q |  | e | ! | 1 | A | Q | a | q |
| **2** | i | " | 2 | B | R | b | r |  | i | " | 2 | B | R | b | r |
| **3** | o | # | 3 | C | S | c | s |  | o | # | 3 | C | S | c | s |
| **4** | u | $ | 4 | D | T | d | t | → | u | $ | 4 | D | T | d | t |
| **5** | a | % | 5 | E | U | e | u |  | a | % | 5 | E | U | e | u |
| **6** | e | & | 6 | F | V | f | v | \| | e | & | 6 | F | V | f | v |
| **7** | i | ' | 7 | G | W | g | w |  | i | ' | 7 | G | W | g | w |
| **8** | o | ( | 8 | H | X | h | x | ; | o | ( | 8 | H | X | h | x |
| **9** | u | ) | 9 | I | Y | i | y | ; | u | ) | 9 | I | Y | i | y |
| **A** | a |  | : | J | Z | j | z |  | a |  | : | J | Z | j | z |
| **B** | e | + | ; | K | [ | k |  |  | e | ± | ; | K | [ | k |  |
| **C** | u | , | < | L | \ | l |  | ‖ | u | , | ≤ | L | \ | l |  |
| **D** | A | – | = | M | ] | m | e | ↑↓ | A | – | = | M | ] | m | e |
| **E** | O | . | > | N | ↑ | n |  | β | O | . | > | N | ↑ | n |  |
| **F** | U | / | ? | O | ← | o |  | U | / | ? | O | ← | o |  |  |

**Figure A-1. The Default Font Character Set**

In the default font, the characters from HEX(90) through HEX(FF) are the characters HEX(10) through HEX(7F) underscored.

**High–order HEX Digit**

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | a | space | 0 | @ | P | ° | p | | | | | | | | |
| **1** | e | ! | 1 | A | Q | a | q | | | | | | | | |
| **2** | i | ” | 2 | B | R | b | r | | | | | | | | |
| **3** | o | # | 3 | C | S | c | s | | | | | | | | |
| **4** | u | $ | 4 | D | T | d | t | → | ↓ | | | | | | |
| **5** | a | " | 5 | E | U | eu | | | | | | | | | |
| **6** | e | & | 6 | F | V | f | v | \| | | | | | | | |
| **7** | i | ' | 7 | G | W | g | w | ° | | | | | | | |
| **8** | o | ( | 8 | H | X | h | x | | \| | | | | | | |
| **9** | u | ) | 9 | I | Y | i | | y | \| | | | | | | |
| **A** | a | | : | J | Z | j | z | | | | | | | | |
| **B** | e | + | ; | K | [ | k | | | | | | | | | |
| **C** | u | , | < | L | \ | l | | !! | | | | | | | |
| **D** | A | – | = | M | ] | m | e | ↑↓ | | | | | | | |
| **E** | O | . | > | N | ↑ | n | | | | | | | | | |
| **F** | U | | ? | O | ← | o | | | | | | | | | |

**Figure A-2. The Alternate Font Character Set**

B

# Error Messages and Recovery

## Overview

BASIC-2 provides an extensive set of error detection features designed to automatically detect and report a wide range of error conditions. The system automatically scans program text for errors during program entry, resolution, and execution.

When the system encounters an error, it displays the erroneous line with an arrow pointing to the approximate position of the error. The error number and a descriptive error message is displayed on the next line. For example,

```
100 DATALOAD DC #1, X
                    ↑
   ERROR D80:  File Not Open
```

If the system discovers an error during text entry, it stores the erroneous line in memory. If the system encounters an error during program resolution or execution, it immediately terminates resolution or execution. The system stops error scanning when it encounters the first error. If a line contains more than one error, the system detects and reports only the first error.

The following pages contain a list of all BASIC-2 error messages, including an explanation of each error and suggested recovery procedures.

## Miscellaneous Errors

**A01 Not Enough Memory**

Not enough free space remains in memory to enter the program line or to accommodate the defined variable. You can still execute system commands (e.g., SAVE) and some Immediate mode statements. To make memory space available, enter a CLEAR P, CLEAR N, or CLEAR V command to shorten the program or reduce the number of variables defined.

**A02 Not Enough Memory**

Not enough free space remains in memory to execute the program or Immediate mode statement. You can still execute system commands (e.g., SAVE) and some Immediate mode statements. To make memory space available, enter a CLEAR P, CLEAR N, or CLEAR V command to shorten the program or reduce the number of variables defined.

**A03 Not Enough Memory**

Not enough free space remains in memory to accommodate the program text. To make memory space available, enter a CLEAR P, CLEAR N, or CLEAR V command to shorten the program or reduce the number of variables defined.

**A04 Operator Stack Overflow**

FOR/NEXT loops, subroutines, or expressions are nested too deeply. Often this error occurs because the program repeatedly branches out of subroutines or loops without executing a terminating RETURN or NEXT statement. Correct the program, possibly by using a RETURN CLEAR statement to clear subroutine or loop information.

**A05 Line Too Long**

The program line being entered cannot be saved in one disk sector because its length exceeds 253 bytes. The line can be executed, but it cannot be saved on disk. Shorten the line by breaking it up into two or more smaller lines.

**A06 Program Protected**

Protected program text in memory cannot be listed, saved, or modified, except with a LOAD RUN or CLEAR command. A LOAD RUN or CLEAR command deactivates protect mode, but it also clears program text from memory.

**A07 Illegal in Immediate Mode**

The statement cannot be executed in Immediate mode.

**A08 Statement Illegal Here**

The indicated statement cannot be used in the current context.

**A09 Program Not Resolved**
The system cannot execute an unresolved program. Resolve the program by executing a RUN command.

## Syntax Errors

**S10 Missing '('**
BASIC-2 syntax requires a left parenthesis.

**S11 Missing ')'**
BASIC-2 syntax requires a right parenthesis.

**S12 Missing '='**
BASIC-2 syntax requires an equal sign (=).

**S13 Missing ','**
BASIC-2 syntax requires a comma (,).

**S14 Missing '*'**
BASIC-2 syntax requires an asterisk (*) in the statement.

**S15 Missing '>'**
BASIC-2 syntax requires the (>) character.

**S16 Missing Letter**
BASIC-2 syntax requires a letter.

**S17 Missing Hex Digit**
BASIC-2 syntax requires a hex digit (digit from 0 to 9 or letter from A to F).

**S18 Missing Relational Operator**
BASIC-2 syntax requires a relational operator (<, =, >, <=, >=, <>).

**S19 Missing Word**
BASIC-2 syntax requires a required word (such as THEN or STEP).

**S20 End of Valid Syntax**
Although syntax is correct up to the point of the error message, the system cannot comprehend the remainder of the statement.

**S21 Missing Line Number**
The BASIC-2 syntax requires a line number.

**S22 Illegal PLOT argument**
An argument in the PLOT statement is illegal.

**S23 Invalid Literal**
The syntax or length of the literal is invalid. A literal string must be 1 to 255 characters in length.

**S24 Illegal Expression or Missing Variable**
The expression syntax is illegal or a variable is missing.

**S25 Missing Numeric Scalar Variable**
BASIC-2 syntax requires a numeric-scalar-variable.

**S26 Missing Array Variable**
BASIC-2 syntax requires an array-variable.

**S27 Missing Numeric Array**
BASIC-2 syntax requires a numeric-array.

**S28 Missing Alpha Array**
BASIC-2 syntax requires an alpha-array.

**S29 Missing Alpha Variable**
BASIC-2 syntax requires an alpha-variable.

## Program Errors

**P31 DO Not Matched with ENDDO**
DO and ENDO statements are not properly matched.

**P32 Start > End**
The starting value exceeds the ending value.

**P33 Line Number Conflict**
The system cannot execute the RENUMBER command because the renumbered program text cannot fit between existing program lines. Adjust the RENUMBER command parameters.

**P34 Illegal Value**
The value exceeds the allowed limit.

**P35 No Program**
Memory contains no program statements. Prior to issuing a RUN command, enter program statements or load a program.

**P36 Undefined Line Number or CONTINUE Illegal**
If the program references a line number that does not exist, ensure that all referenced lines exist. If the system aborts a CONTINUE command, rerun the program using a RUN command. The following circumstances prevent continuation of terminated program execution: the occurrence of a stack or

memory overflow, the entry of a new variable, the execution of a CLEAR command, the modification of program text, or a reset operation.

**P38 Undefined DEFFN' Subroutine**
The program references a nonexistent DEFFN' subroutine.

**P39 FN's Nested Too Deeply**
The system encountered more than five levels of nesting when evaluating an FN function.

**P40 NEXT without FOR**
The program contains a NEXT statement without a companion FOR statement or it branches into the middle of a FOR/NEXT loop.

**P41 RETURN without GOSUB**
The program executes a RETURN statement without first executing a GOSUB or GOSUB' statement. Either a companion GOSUB or GOSUB' does not exist, or the program branches into the middle of a subroutine.

**P42 Illegal Image**
The indicated image is illegal in the current context. For example, a PRINTUSING statement refers to an image that does not contain a format specification.

**P43 Illegal Matrix Operand**
The same array name appears on both sides of the equation in a MAT multiplication or MAT transposition statement.

**P44 Matrix Not Square**
The dimensions of a MAT inversion or identity operand are not equal.

**P45 Incompatible Operand Dimensions**
The dimensions of the operands in a MAT statement are not compatible.

**P46 Illegal Microcommand**
A microcommand in the specified $GIO sequence is illegal or undefined. An illegal escape sequence was sent to the Generialized Printer Driver.

**P47 Missing Buffer Variable**
A data input, output, or verify microcommand omits a $GIO statement buffer. Include a buffer in the $GIO statement.

**P48 Illegal Device Specification**
The statement refers to an undefined file number or device address not entered into the Master Device Table. Execute a SELECT statement defining the file number, or correct the device address. This error occurs when the system tries to communicate with the device, not when the device is selected. P48 is a recoverable error.

**P49 Interrupt Table Full**
The program can define interrupts for a maximum of eight devices.

**P50 Illegal Array Dimensions or Variable Length**
The array dimension or alpha-variable length exceeds the legal limits.

**P51 Variable or Value Too Short**
The length of the indicated variable or value is too short for the specified operation.

**P52 Variable or Value Too Long**
The length of the indicated variable or value is too long for the specified operation.

**P53 Noncommon Variables Already Defined**
Noncommon variables cannot be defined in a program before a COM statement. Either move all COM statements to the beginning of the program, or clear noncommon variables with a CLEAR N command.

**P54 Common Variable Required**
A multiple-file LOAD command requires a common variable.

**P55 Undefined Variable**
The indicated variable is not defined elsewhere in the program. This error usually results because a referenced array in a DIM or COM statement is improperly defined.

**P56 Subscript Out of Range**
The variable subscripts exceed the defined array dimensions, or the number of subscripts does not agree with the array dimensions.

**P57 Illegal STR Argument**
The STR function arguments exceed the maximum defined length of the alpha-variable. Correct the STR function arguments, or redefine the alpha-variable.

**P58 Illegal Field/Delimiter Specification**
The $PACK or $UNPACK statement specifies an illegal field or delimiter specification.

**P59 Illegal Redimension**
The space required to redimension the array exceeds the space initially reserved for the array. Redimension the array to fit in the required space, or adjust the DIM or CON statement to reserve additional space.

## Computational Errors

**C60 Underflow**

The absolute value of the result was less than 1E-99 but greater than zero. The statement SELECT ERROR >60 suppresses this error and moves a default value of zero into the result.

**C61 Overflow**

The absolute value of the result was greater than 9.999999999999E+99. The statement SELECT ERROR >61 suppresses this error and moves a default value of ±9.999999999999E+99 into the result.

**C62 Division by Zero**

Division by zero is mathematically undefined. The statement SELECT ERROR >62 suppresses this error and moves a default value of ±9.999999999999E+99 into the result.

**C63 Zero Divided by Zero or Zero Raised to Zero Power**

Zero divided by zero or zero raised to the zero power is mathematically undefined. The statement SELECT ERROR >63 suppresses this error and moves a default value of zero into the result.

**C64 Zero Raised to Negative Power**

Zero raised to a negative power is a mathematically undefined operation. The statement SELECT ERROR >64 suppresses this error and moves a default value of ±9.999999999999E+99 into the result.

**C65 Negative Number Raised to Non-Integer Power**

Raising a negative number to a noninteger power is a mathematically undefined operation. The statement SELECT ERROR >65 suppresses this error and moves a default value of the absolute value of the number raised to the negative power into the result.

**C66 Square Root of Negative Value**

The square root of a negative value is mathematically undefined. The statement SELECT ERROR >66 statement suppresses this error and moves a default value of SQR(ABS(X)), where X is the negative value, into the result.

**C67 LOG of 0**

The LOG of zero is mathematically undefined. The statement SELECT ERROR >67 suppresses this error and moves a default value of -9.999999999999E±99 into the result.

**C68 LOG of Negative Value**

The LOG of a negative value is mathematically undefined. The statement SELECT ERROR >68 suppresses this error and moves a default value of the LOG of the absolute value of the number into the result.

**C69 Argument Too Large**

The absolute value of the SIN, COS, or TAN function is greater than or equal to 1E+10 and the system cannot evaluate this function; or, the absolute value of the ARCSIN, ARCCOS, or ARCTAN argument is greater than 1.0, and the value of this function is mathematically undefined. The statement SELECT ERROR >69 suppresses this error and moves a default value of zero into the result.

## Execution Errors

**X70 Insufficient Data**

The DATA statement does not contain enough data values to satisfy READ or RESTORE statement requirements. Correct the program to supply additional data, or modify the READ or RESTORE statement.

**X71 Value Exceeds Format**

The PACK or CONVERT image does not specify enough integer digits to express the value of the number being packed or converted.

**X72 Singular Matrix**

A MAT inversion operand is singular and cannot be inverted. Include a normalized determinant parameter in the MAT INV statement, and check the determinant following the inversion.

**X73 Illegal INPUT Data**

The value requested by an INPUT statement is in an illegal format. Reenter the data in the correct format, or stop program execution by pressing the RESET key and then RUN the program again. Alternately, to avoid the entry of illegal data, substitute a LINPUT statement for the INPUT statement, and verify operator-entered data within the program.

**X74 Wrong Variable Type**

The variable type (alpha or numeric) and the data type do not correspond. Correct the program or data, or ensure that the proper file is being accessed.

**X75 Illegal Number**

The format of the indicated number is illegal.

**X76 Buffer Exceeded**

The buffer variable is too small or too large for the specified operation.

**X77 Invalid Partition Reference**

The partition referenced by SELECT @PART or $RELEASE TERMINAL is not defined or the name specified by DEFFN @PART has already been used. Use the proper partition name and wait for the global partition to be defined.

**X78 Print Driver Error**

An error was detected with the print drivers. The error also results from an invalid driver table name. The error is also returned if you attempt to associate more than 15 device addresses with printer driver tables or when an address associated with the printer driver tables is used more than once. To recover, change the incorrect address parameter.

**X79 Invalid Password**

The password entered does not match the password set when the system was configured with the @GENPART utility.

## Disk Errors

**D80 File Not Open**

The file operation cannot be performed upon a closed file.

**D81 File Full**

No more information can be written into the indicated file. Correct the program, or transfer the file to another platter, reserving additional space on the new platter for this file.

**D82 File Not Found**

The file name does not exist, or a data file was loaded as a program file or a program file as a data file. Ensure that the file name is entered correctly; ensure that the proper disk is mounted; and ensure that the correct disk drive is being accessed.

**D83 File Already Exists**

The file name already exists in the Catalog Index. Use a different name, or catalog the file on a different platter.

**D84 File Not Scratched**

A file must be scratched before it can be renamed or written over.

**D85 Index Full**

The Catalog Index contains no space for new names. Scratch unwanted files and compress the catalog using a MOVE statement, or mount a new disk platter and create a new catalog.

**D86 Catalog End Error**

The defined Catalog Area ends within the Catalog Index or has no more available space to store information. This usually occurs because a MOVE END statement tries to move the end of the Catalog Area to the area already occupied by cataloged files. Correct the SCRATCH DISK or MOVE END statement, or increase the size of the Catalog Area by executing a MOVE END statement. Alternately, scratch unwanted files and compress the catalog using a MOVE statement, or mount a new disk platter and create a new catalog.

**D87 No End-of-File**

Because neither a DATASAVE DC END nor a DATASAVE DA END statement recorded an end-of-file record in the file, the DSKIP END statement cannot locate an end-of-file record. Write an end-of-file trailer after the last data record in the file.

**D88 Wrong Record Type**

The system encountered a program record when a data record was expected or vice versa. Ensure that the proper drive and file is being accessed.

**D89 Sector Address Beyond End-of-File**

A DATALOAD DC or DATASAVE DC statement accesses a sector address beyond the end-of-file. This error can be caused by a bad disk platter. Press RESET and run the program again. If the error persists, use a different platter or reformat the platter. If the error still exists, inform a Wang Service Representative.

## I/O Errors

**I90 Disk Controller Error**

The system aborts the disk operation because the controller responded improperly at the beginning of the operation. Press RESET and rerun the program. If the error recurs, make certain that the disk unit is on and all cables are properly connected. If the error persists, contact a Wang Service Representative.

**I91 Disk Drive Not Ready**

The disk unit is not ready for access. Make certain that the program addresses the correct disk. Also, make sure that the disk unit is on and in run mode, and all cables are properly connected. Press RESET and rerun the program. If the error recurs, power the disk unit off and then back on and rerun the program. If the error persists, call a Wang Service Representative.

**I92 Timeout Error**

A device did not respond to the system. If the device is a disk, the system aborts the disk operation. Press RESET and run the program again. If the error recurs, ensure that the disk has been formatted. If the error persists, inform a Wang Service Representative.

**I93 Format Error**

The system detects invalid sector control information of the disk platter. If a disk operation is in progress, the platter may need to be reformatted. If the formatting is in progress, the surface of the platter may be flawed. Reformat the disk platter; if the error recurs, replace the platter. If the error persists, contact a Wang Service Representative. The error can also occur if the user attempts to access a disk formatted for use on a different type of system.

**I94 Disk Controller Error**

The system aborts the disk operation because the controller did not receive the disk command correctly. Press RESET and rerun the program. If the error recurs, make certain that the disk unit is on and all cables are properly connected. If the error persists, contact a Wang Service Representative.

For those disks with a format key, this error message also can indicate that the format key is engaged. Disk operations cannot be performed until formatting is turned off with the format key.

For SCSI controllers, the controller aborts the operation due to an illegal SCSI command or request. Verify that the SCSI unit is on and that all disks are configured correctly.

**I95 Device Error**

The disk cannot perform the requested operation. Repeat the operation. If performing a write operation, make certain that the disk is not write-protected. If the error recurs, power the disk off and back on, and again perform the operation. If the error persists, contact a Wang Service Representative.

**I96 Data Error**

For read operations, the checksum calculations (CRC or ECC) indicate that the data read is incorrect. For disk drives that perform ECC, the attempt to correct errors was unsuccessful. Rewrite the data; the read sector may have been written incorrectly. If read errors recur, reformat the platter.

For write operations, the LRC calculation indicates that the data sent to the disk is incorrect. The data has not been written. Repeat the write operation. If write errors recur, make certain that all disk cables are properly connected.

If either error persists, inform a Wang Service Representative.

**I97 LRC Error**

A longitudinal redundancy check error occurred while a sector was being written or read. An LRC error usually indicates a transmission error between the disk and the CPU. Press RESET and rerun the program. If the error recurs, rewrite the flawed sector; the sector may have been previously written incorrectly. If the error persists, contact a Wang Service Representative.

**I98 Illegal Sector Address or No Platter**

The indicated sector is not on the disk platter, or the specified drive contains no platter. Ensure that the correct drive is being accessed. Correct the program statement, or ensure that the diskette is inserted into the drive.

**I99 Read-After-Write-Error**

The comparison of read-after-write to a disk sector failed, usually indicating a defective platter. Rewrite the information; the data may have been previously written incorrectly. If the error recurs, replace the platter. If the error persists, contact a Wang Service Representative.

C

# Compatibility With 2200 Series Cpu's

## Overview

Wang Laboratories, Inc., has introduced a number of different CPU models in the 2200 series, including the 2200B, 2200C, 2200S, 2200T, 2200VP, and 2200MVP. The 2200VP and 2200MVP central processors support the BASIC-2 language. The BASIC-2 language is fundamentally compatible 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 earlier version of BASIC is referred to as "Wang BASIC" to distinguish it from BASIC-2.) In order to provide downward compatibility with the other 2200 series CPU's, the BASIC-2 interpreter is programmed to recognize the syntax of itbothatt the BASIC-2 instruction set and the Wang BASIC instruction set.

Most programs written for Wang BASIC can be loaded and run with BASIC-2 itwithout modificationatt. The remaining programs require slight modification. Because the routines which interpret Wang BASIC syntax are a resident part of the BASIC-2 interpreter, the programmer can intermix BASIC-2 and Wang BASIC instructions within the same program.

The areas of incompatibility between BASIC-2 and Wang BASIC are: Wang BASIC language features not supported by the BASIC-2, and language incompatibilities resulting from improved BASIC-2 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 newer systems.

## Wang Basic Language Features not Supported in BASIC-2

There are two types of features not supported in BASIC-2 which limit the ability of Wang BASIC programs to be transferred from other 2200 series systems.

Teletype, Tape Cassette and Manual-Feed Mark Sense Card Reader statements are not supported with BASIC-2. Versions of the LOAD, DATALOAD, and DATALOAD BT statements designed to access these devices produce syntax and/or execution errors when the program is run with BASIC-2. (In some situations, a $GIO statement can be used to emulate the nonsupported statement.)

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. 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.

## Incompatibilities Resulting From Design Changes

Several incompatibilities between Wang BASIC and BASIC-2 arise as a result of changes in the design of the internal architecture and in implementation of certain BASIC-2 language features:

1. ON ERROR GOTO Statement

   The ON ERROR GOTO statement in Wang BASIC in not recommended in BASIC-2. BASIC-2 error codes are different from those of Wang BASIC. BASIC-2 employs a three-byte error code. This consists of a letter prefix and two digits, as opposed to the two-byte code. 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. If a two-byte variable is used for the error code, 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, 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)

   BASIC-2 uses an improved technique for representing the subscripts of one-dimensional arrays. This leads to incompatibilities in interpretation of subscript arrays produced by the MAT SORT statement and used by the MAT MOVE statement. Array subscripts are represented internally as two-byte binary values.

   With Wang BASIC, one-dimensional arrays are 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.

   With BASIC-2 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. With BASIC-2, 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 BASIC-2 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.

   These are two statements 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:," is used as input to a MAT MOVE statement to produce an output data array. Ordinarily, there is itnoatt compatibility problem. When a BASIC program containing these statements is run with BASIC-2, the locator-array created by MAT SORT automatically receives subscripts in BASIC-2 format, and MAT MOVE interprets them correctly.

   When the user has written a custom routine to either interpret the MOVE, or use a locator-array generated by Wang BASIC, the program must be modified to correctly represent and interpret the subscripts of one-dimensional arrays with BASIC-2. 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 is 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. 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. Instead, the colon is interpreted as a statement separator. For example, the line

   50 %+##.##: GOTO 100

   is interpreted to contain two statements, the Image statement "%+##.##" and the GOTO statement "GOTO 100". In BASIC-2, a colon is allowed as a legal character in a PRINTUSING Image statement. It is not interpreted as a statement separator in this case. 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, 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 with BASIC-2.

4. 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 itonlyatt 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 itnotatt cause the end-of-file flag to be set. In order to test for end-of-file with an IF END THEN statement, it was necessary to be certain that a software trailer is written in the file following the last data record. In BASIC-2, itbothatt a software trailer itandatt a hardware trailer cause the end-of-file even if no software trailer has been written in the file.

5. Reading Past the End-of-File With DATALOAD DC 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 itunchangedatt, retaining the values of the last valid data record read.

## Language-Independent Incompatibilities

There is a third class of incompatibility problems arising from changes and improvements to the system 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 with BASIC-2. There are two potential problem areas to consider:

- Problems in Time-Dependent Programs
- Problems Arising From Improved Math Accuracy

The significantly improved processing speed of systems with BASIC-2 and the variations in speed on with Multiuser BASIC-2 due to multiprogramming processing may cause problems for time-dependent programs. Prompts which remain on the screen long enough to be easily read when a program was run on an earlier model may flash by at unreadable speed when the same program is run on a system with BASIC-2. 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.

Math package functions and the MAT INVerse statement are generally more accurate with BASIC-2. This increased accuracy may affect a few programs which rely on a specific value being returned by a certain expression.

## Incompatibilities With Vp BASIC-2

The following Multiuser BASIC-2 features are not available in VP BASIC-2.

- Lowercase program entry
- Descriptive error messages
- Immediate mode STOP
- List COM/DIM
- 3 and 4-byte BIN and VAL functions
- DO groups
- Select functions
- ERR$ function
- RAM Disk
- Filename mask and file type selection in list DC
- SAVE W
- RESAVE
- RENAME

- DATALOAD/SAVE BM
- Move with index size and catalog area size specifications
- Printer drivers

  Also, the output from the following statements has been improved in Multiuser BASIC-2.
- TRACE
- LIST DT
- LIST HEX literals

## Restrictions Imposed by Multiuser BASIC-2

Even if an existing program runs correctly on Multiuser BASIC-2, it may 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 use special Multiuser BASIC-2 features to facilitate cooperation among programs can enhance system performance significantly. The following is a list of restrictions imposed by Multiuser BASIC-2:

- Some programs use KEYIN to input atom codes from the BASIC Keyword keys. This technique works properly on the Multiuser BASIC-2, but the standard 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. Refer to Table A-2 for key codes and character sets.
- Many programs test for the existence of printers and disks before attempting to access them. Such tests present a problem on the Multiuser BASIC-2 because several lines of buffering separate the terminal printer from the 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.
- 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.
- Multiuser BASIC-2 does not permit CI or CO to be selected to any device other than the 2236D terminal. The output of a TRACE command 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.

D

# Series 2200 Device-Addresses

This appendix lists the standard and alternate device-types and device-addresses for various classes of I/O devices, as well as for several specialized controllers.

| Device or Class | Device-Type Usual, (Alternates) | Usual Address | Alternate Addresses |
|---|---|---|---|
| CRT | 0 | 05 | - |
| Keyboard | 0 | 01 | - |
| Terminal Printer | 2,(0) | 04 | - |
| Terminal Plotter | C,(4) | 04 | - |
| Printers | 2,(0) | 15 | 16 |
| Plotters | C,(4) | 13 | 14,15,16 |
| Disk Drives | D,(3,B) | 10 | 20,30 |
| Second drive | D,(3,B) | 50 | 60,70 |
| Nine-Track Tape Transport | 0 | 7B | 7D,7F |
| 2258 | 0 | 6D-6F | 7D-7F |

# Index