List of Software Versions

BOOT	1.1
FDOS	1.1
CONMON	1.1
BASIC	1.1
FUP	1.0
SET	1.0
TIME	1.0

1720A
Instrument Controller

Fluke BASIC Programming Manual

P/N 518670
May 1980

List of Sections

Table of Contents

(continued on page ii)

TABLE OF CONTENTS, *continued*

(continued on page iii)

TABLE OF CONTENTS, *continued*

(continued on page iv)

TABLE OF CONTENTS, *continued*

(continued on page v)

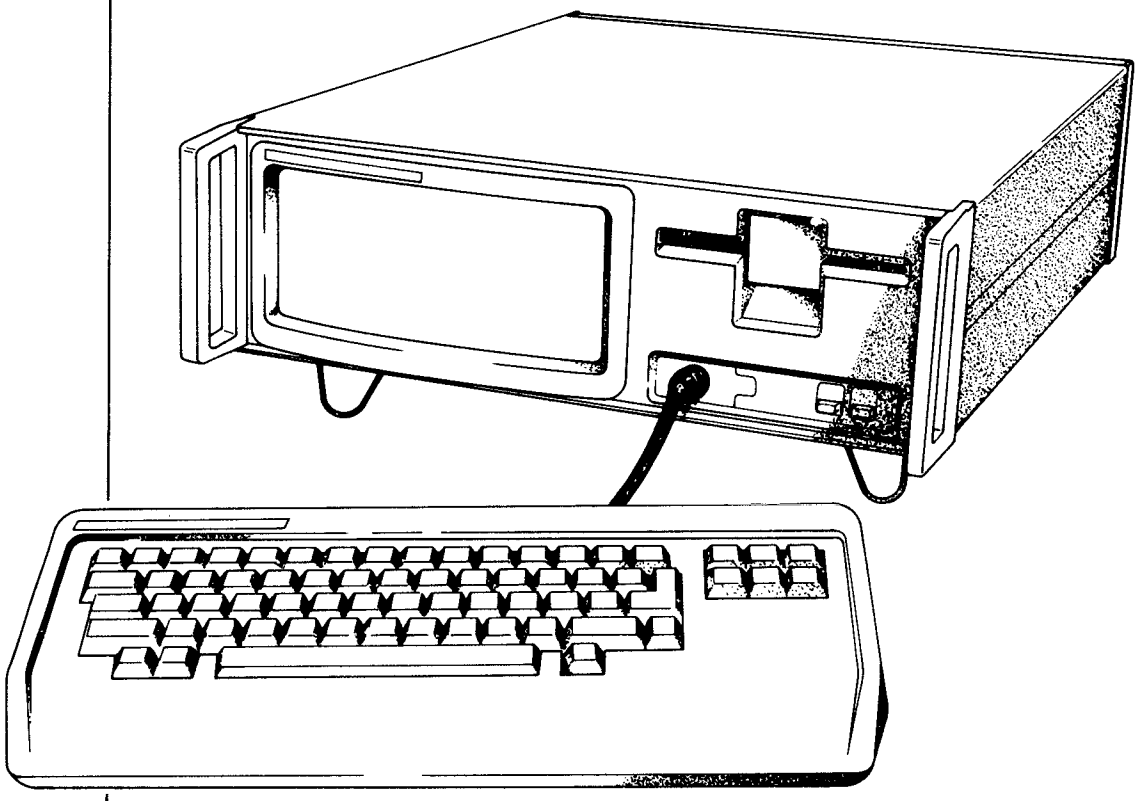
TABLE OF CONTENTS, *continued*

(continued on page vi)

TABLE OF CONTENTS, *continued*

List of Illustrations

List of Tables



1720A Instrument Controller

Section 1
Introduction

1-1. DESCRIPTION OF THE 1720A DOCUMENTATION

1-2. Several documents are available to serve the needs of a variety of users. Figure 1-1 illustrates the manual set. To obtain additional copies of these manuals, or others when developed, check with a Fluke Sales Office, listed at the end of this manual.

1-3. 1720A User Manual

1-4. The User Manual is an introductory manual for the programmer or system designer who is using the 1720A to set up an instrumentation system. It presents both the purpose of and the interaction between the various software and hardware resources in the 1720A. In addition, it serves as a reference guide for the Console Monitor, the File Utility Program, and the other utility programs. Order Fluke Part Number 518654.

1-5. 1720A Fluke BASIC Programming Manual

1-6. The Programming Manual provides a description of the Fluke-Enhanced ANSI-standard BASIC language developed for the 1720A Instrument Controller. It is arranged by subject, and combines readable syntax diagrams with numerous examples. Emphasis is on instrumentation system control using effective programming techniques. This manual presumes familiarity with the 1720A User Manual. Order Fluke Part Number 518670.

1-7. 1720A BASIC Reference Guide

1-8. The BASIC Reference Guide is a pocket-size quick reference guide that summarizes the most often used contents of the User and BASIC Programming manuals. Order Fluke Part Number 526210.

1-9. HPL to Fluke BASIC Handbook

1-10. A specialized manual for the programmer who is familiar with the Hewlett-Packard 9825 Calculator using the HPL language. Building upon this familiarity, the handbook describes functional similarities and differences, and leads the programmer through the task of converting existing HPL programs for operation on the 1720A Controller. Order Fluke Part Number 546341.

1-11. 1720A Display Worksheet Pads

1-12. The 1720A display worksheets have a grid pattern that shows both normal and double size character positions, and the touch sensitive areas of the display. This allows the programmer to design effective display layouts. They are available in pads of 50. Order Fluke Part Number 533547.

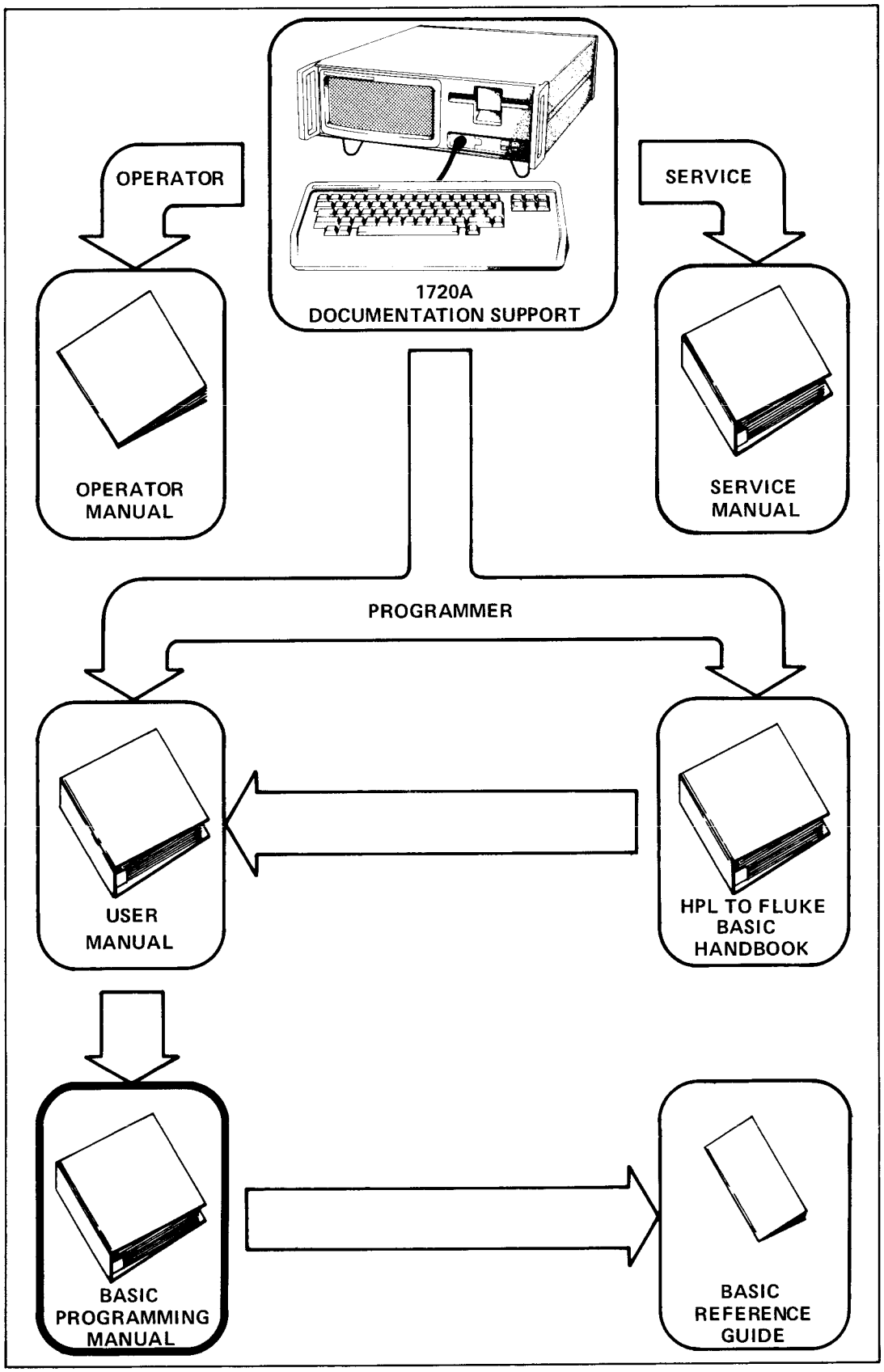


Figure 1-1. 1720A Manual Set

1-13. 1720A Operator Manual

1-14. The Operator Manual is a brief manual for the operator of a programmed 1720A-controlled system who will not normally use the keyboard. It discusses such subjects as proper disk handling and error interpretation, and includes an error and trouble-incident log. Order Fluke Part Number 518647.

1-15. 1720A Service Manual

1-16. The Service Manual includes component level theory of operation for each hardware module, with diagnostic procedures that can resolve failures to the modular level. Schematics, parts lists, and appropriate appendices are also provided. Order Fluke Part Number 518662.

1-17. INTRODUCTION TO THIS MANUAL

1-18. This manual considers the 1720A Instrument Controller to be part of a programmable system consisting of the Controller and the instruments that it controls. Its contents result from these considerations:

- The user configures this system to suit specific applications by developing programs written in Fluke-Enhanced BASIC.

- The controller executes these programs from memory.

- The user will build a library of programs on floppy disks and load the programs into the Controller memory as necessary.

1-19. Section Headings

1-20. The sections that make up this manual are groupings of information by function. The heading for any given section describes the functions covered by that section. A list of section headings appears just ahead of the table of contents. A particular subject, such as a given BASIC statement, may appear in more than one section. In this case, each section covers one aspect of the subject and refers to the other sections.

1-21. Statement Definitions

1-22. In this manual, BASIC program statements are introduced with a syntax diagram that defines the different legitimate statement constructs. A complete definition of the various forms of the statement, including implications and limitations, follow the syntax diagram. These definitions use the following conventions:

- The short indented paragraphs (with a "bullet" in place of a paragraph number) following the syntax diagram are part of the definition of the statement.

- This arrangement of points allows the reader to quickly skip to the question at hand without searching through text.

- When a statement is covered elsewhere in the manual, the definition is partial, focussing on one or more of the statement constructs relevant to the subject matter of the section.

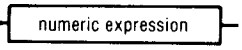
- The last point in each definition list is a reference to the other sections in this manual where that statement appears.

1-23. Syntax Diagrams

1-25. Syntax diagrams are used throughout this manual to define correct spelling, punctuation, sequences of words, symbols, and expressions for Fluke-Enhanced BASIC statements. Refer to the following guidelines when using the syntax diagrams:

- Any path through a diagram starting from the left that does not run contrary to an arrowhead forms a legitimate Fluke BASIC statement. There are a few exceptions to this. The semantic requirements of the language dictate that some syntactically correct sentences are illegal, and cannot be used in a program. In all cases, the accompaning text explains legal usage.

- BOLDFACE words in a circular enclosure are to be entered exactly as shown. Example:

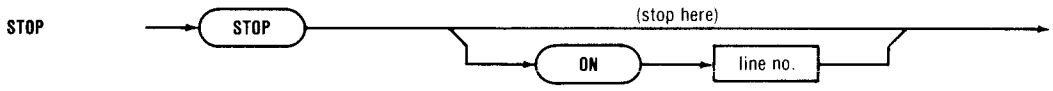
- Lower case words in a box enclosure represent other information to be supplied. Example: 

- Words outside the path of the diagram, usually in parentheses, are supplementary information that add to the usefulness of the diagram. These words are not part of the definition of the statement. Example:

- Where the discussion concentrates on a particular path through a diagram, that path is indicated by shading. Other paths through the diagram are discussed in other section(s) referenced by the last definition point in each definition.

- The line length of any program statement, including the line number if used, cannot exceed 80 characters.

1-26. The following syntax diagram of the STOP statement illustrates these points. The diagram shows that two alternate constructions of the statement are acceptable: STOP, and STOP ON line number. The supplementary note shows that the stop occurs at the location in the program where the statement is placed when no other line number is referenced.



1-27. Appendices

1-28. A set of appendices is included at the back of this manual. These provide a repetition of some of the material presented throughout the manual, summarized for quick reference. In addition, supplemental information is provided on internal software structure. Refer to the Table of Contents for a list of these appendices.

1-29. FLUKE ENHANCED BASIC

1-30. The BASIC language interpreter developed by Fluke for the 1720A Controller is based upon Standard X3J2/77-26 of the American National Standards Institute. A comprehensive set of additional command statements for control of IEEE-488 Bus instrumentation has been added as well as string data operation, mass storage control commands, debugging features, and other extensions. Also included are constructs for chaining multiple programs in sequence and passing variables between programs.

1-31. Exceptions to ANSI Standard BASIC

1-32. Fluke BASIC is an enhancement of ANSI standard minimal BASIC. The enhancements include many statements for instrument control. An ANSI Minimal BASIC program will run on the 1720A unless there is conflict in one of the following areas:

- Dimensioned arrays must be defined by a dimension statement prior to use in a program.

- LOG is the logarithm to base 10. LN is an additional Fluke BASIC function for natural logarithms (base e).

- A GOTO within a FOR-NEXT loop is not allowed if the GOTO tranfers control permanently outside the loop.

- The print format used for floating point numbers greater than 7 digits in length is .1 to 1, instead of 1 to 10 as specified in the ANSI standard. For example, .01234567 is printed as .1234567E-01, instead of 1.234567E-02.

- TAB(X) will position the cursor at X+1, instead of X.

- Variables are assigned in sequence, upon validation, in a multiple-variable input list. The ANSI standard allows variable assignment only upon validation of the entire input list.

- The optional base of 1 for a dimensioned array is not supported. Fluke BASIC arrays have a base of 0. For example, A(10) has eleven elements numbered 0 through 10.

- User defined functions must have arguments. For example, DEF FNA(X) = expression is allowed while DEF FNA = expression is not.

- Multiple commands are allowed on a single line when separated by the \ character.

- Spaces may not appear between GO and SUB of GOSUB.

1-33. Gaining Access to BASIC

1-34. BASIC is entered through the Console Monitor (CONMON, prompt #) by typing B or BASIC. Refer to the 1720A User Manual for information on how to enter the Console Monitor. The following display is produced by BASIC when it is entered.

```
BASIC (time) (date)
BASIC Version n.n

Ready
```

NOTE

Verify that the version number given for "n.n" agrees with the version number at the beginning of this manual. If not, contact a Fluke Customer Service Center for advice.

1-35. In addition, BASIC may be directly entered at Power-On, or when RESTART is pressed. This is done with an active Command File that contains the command B (or BASIC). Refer to the 1720A User Manual for information on the Command File, COMMND.SYS.

1-36. Exiting BASIC

1-37. There are four ways to exit BASIC. Each returns control to the Console Monitor, resulting in the CONMON prompt #.

- From Immediate Mode type EXIT and press RETURN. This allows input/output operations to finish, closes files, deletes the BASIC program in memory, and loads the Console Monitor.

- Execute an EXIT statement within a running program. This allows input/output operations to finish, closes files, deletes the BASIC program in memory, and loads the Console Monitor.

- Press RESTART. This transfers control to the self-test and load routine, terminates all operations of the 1720A, loads the Console Monitor, and processes the Command File (COMMND.SYS), if there is one. The 1720A User Manual contains information on the Command File.

NOTE

Any I/O in progress is aborted and the file directory may be destroyed.

- Enter CTRL P. This closes all files immediately, deletes the BASIC program in memory, and loads the Console Monitor. Output operations are not finished before the files are closed.

CAUTION

If a BASIC program is in memory and control is returned to CONMON, that program will be deleted from memory.

1-38. BASIC Operating Modes

1-39. BASIC has four modes: Immediate, Edit, Run, and Single-Step. The "Ready" prompt is displayed whenever BASIC is in Immediate Mode and ready to accept keyboard input. This is the mode that BASIC starts in whenever it is entered from CONMON. Other modes are accessable only from Immediate Mode.

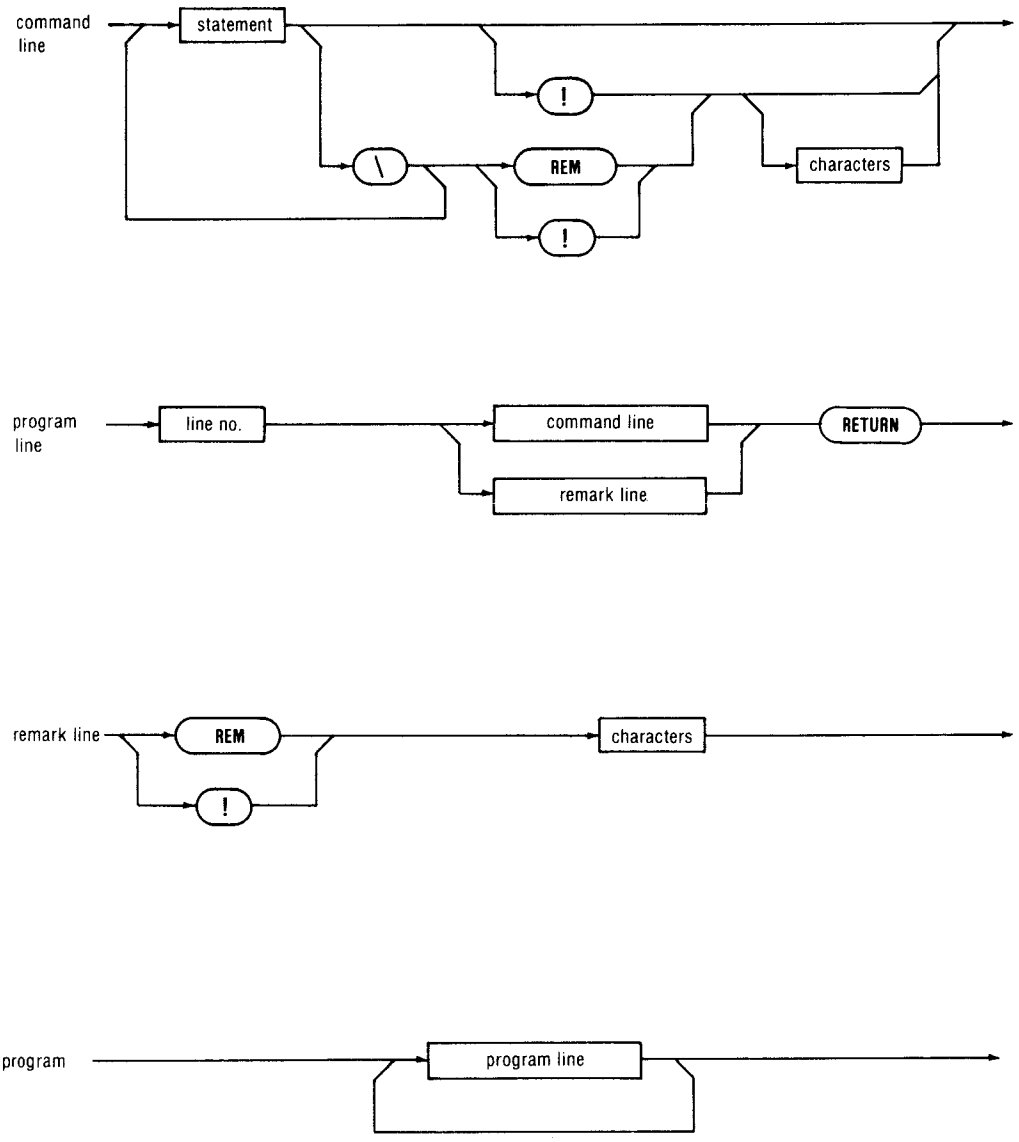
1-40. Immediate Mode

1-41. An entry to BASIC is always into Immediate Mode, identified by the prompt "Ready". Immediate mode may be accessed from any other BASIC mode by entering CTRL C. Also, a running program will return to Immediate Mode whenever the program executes a STOP or END statement, or if a fatal error occurs.

1-42. Exit from Immediate Mode to the Console Monitor is accomplished by typing EXIT, or by entering CTRL P. Any program that was in main memory is erased by the exit to CONMON.

1-43. In Immediate Mode, command lines or program lines may be entered. A command line consists of one or more BASIC statements; these statements are entered either without a line number or a line number of zero. Multiple BASIC statements are separated by a backslash character (\). An Immediate Mode command is executed immediately upon pressing RETURN. Some BASIC commands can be executed only in Immediate Mode. Subsequent sections cover them in detail.

1-44. A program line is a command line preceded by a line number between 1 and 32767. A program is a meaningful sequence of program lines. The following diagrams illustrate these points:



1-45. Edit Mode

1-46. The entry to Edit Mode is made from Immediate Mode by entering EDIT, or EDIT line number. Editing begins with the lowest numbered line of the program memory unless a line number is specified. The exit from Edit Mode to Immediate mode is accomplished by typing CTRL C. Refer to the Writing A Program discussion on the CTRL Key-Modifier (in this section) for additional information.

1-47. The editor provided as a part of Fluke BASIC is an easy to use character oriented editor. Edit Mode allows the user to insert, delete, or modify the characters that make up program lines in main memory. The editing keys in the upper right corner of the keyboard plus the BACK SPACE, CTRL U, Carriage Return, and LINE FEED keys control the movement of the cursor and the deletion of text. The remaining keys are used for text entry. Edit Mode stores program lines in main memory for subsequent use by other modes. A complete discussion of using Edit Mode to write a program appears later on in this section.

1-48. Run Mode

1-49. The entry to Run Mode is made from Immediate Mode by entering RUN, or RUN "file name". Refer to the Input and Output section for a discussion of file name prefixes that may be used to identify the location of the program file to be run. If a file name is specified, the file is first located on the specified or default file storage device and it is then loaded into main memory.

1-50. When a program is present in main memory and BASIC is in Immediate Mode, Run Mode may also be entered by an Immediate Mode branch command: GOTO, GOSUB, ON GOTO, or ON GOSUB. In addition, when a running program has been stopped by CTRL C, STOP, STOP ON, or CONT TO, Run Mode may be resumed by typing CONT or CONT TO line number. These concepts are discussed in the section Program Debugging.

1-51. BASIC exits RUN Mode and returns to Immediate Mode under any of the following conditions:

- The program executes a STOP or END statement.
- CTRL C is entered at the programmer keyboard.
- The front panel ABORT button is pressed.
- A fatal error occurs.

1-52. In Run Mode, program lines are automatically executed in the sequence defined by line numbers, except as directed by branch instructions in statements.

1-53. Step Mode

1-54. Step Mode is entered from Immediate Mode by entering the STEP command after a breakpointed stop set by CONT TO or STOP ON. A RUN, CONT, CONT TO, GOTO, GOSUB, ON GOTO, or ON GOSUB command will cause BASIC to enter Run Mode; entry of any other command will cause BASIC to enter Immediate Mode.

1-55. In Step Mode, one program line is executed after each RETURN key entry, followed by a "STOP AT LINE line number" message. Step Mode is discussed further in the Program Debugging section.

1-56. File Names

1-57. Programs and data files on the 1720A are stored on the floppy or electronic disk by name. File names consist of 1 to 6 letters, numbers, spaces, or $ characters. File names may be extended by a period character, followed by up to 3 letters, numbers, spaces, or $ characters. Most usage of file names in program statements requires that they be enclosed in single or double quotes. Some examples of file names are:

TEST
DATA.473
$25795.98
TESTB2.H44

1-58. Some file name extensions have special meaning to the 1720A:

.BAL Lexical form BASIC programs
.BAS ASCII-text form BASIC programs
.CIL Binary utility programs (Core Image Load)
.HLP System data files (Help)
.SYS System binary programs, and the Command File

1-59. Following are points to remember about file names for programs:

- A program is normally saved in a file with a ".BAS" extension when an extension is not specified, and the SAVE command is used.

- A program stored via the SAVEL command is saved with a ".BAL" extension. See the SAVEL statement.

- The system will look for a file with the extension ".BAL" when seeking a program with a file name that has no specified extension.

- The system will look for a file with the extension ".BAS" when seeking a program with a file name that has no specified extension for which a ".BAL" version could not be found.

- The location of the file may be specified as a prefix to the file name. Use "MF0:" for the floppy disk, or "ED0:" for the optional electronic disk. For example: SAVE "MF0:TEST.T12".

- The default file location when not specified is called the System Device. The Input and Output section includes a discussion of this concept.

1-60. WRITING A PROGRAM

1-61. A user program in the 1720A is any meaningful sequence of Fluke BASIC statements that will, in Run Mode, direct the controller and its associated instrumentation to accomplish a desired task. Line numbers and branch instructions (such as GOTO) define the sequence of statement execution.

1-62. Special Purpose Keys

1-63. This section explains the use of the DELETE, PAGE MODE, NEXT PAGE, and CTRL keys. These keys may be used in Immediate Mode or Edit Mode. The discussion on Edit Mode explains the use of additional keys which are active only in the Edit Mode.

1-64. The DELETE Key

1-65. The DELETE key deletes the character immediately to the left of the cursor. If held down, it will repeatedly backspace and delete characters until all characters to the left of the cursor are deleted.

1-66. The PAGE MODE Switch and the NEXT PAGE Key

1-67. PAGE MODE is an alternate action switch that selects and cancels Page Mode. The Page Mode indicator on the switch is ON when Page Mode is selected.

- The scroll feature is disabled.

- Display is limited to one full screen.

- When NEXT PAGE is pressed, the display is cleared, and an additional full screen is displayed from the top line down.
- Page Mode is available in Immediate Mode and Run Mode.
- Page Mode is not available in Edit Mode.

NOTE

When the display is full, the keyboard will appear "dead" until PAGE MODE or NEXT PAGE is pressed.

1-68. For example, to study the portion of the program in memory between lines 100 and 5000, one page at a time, follow this procedure:

- Type LIST 100-5000.
- Select PAGE MODE (indicator on).
- Press the RETURN key.
- 16 lines of the program are displayed, beginning with Line 100.
- Press the NEXT PAGE key.
- 16 additional lines of the program are displayed.
- To scroll, press PAGE MODE again (indicator off).
- After the screen fills from the top down, the display will scroll upward one line at a time.
- To stop scrolling, press PAGE MODE again (indicator on).

1-69. The CTRL Key-Modifier

1-70. The CTRL key, like the SHIFT key, is a key modifier which must be held down while another key is pressed. The A through Z keys, when modified by CTRL, generate character codes of which only the C, P, Q, S, T, U, or Z keys have meaning to BASIC. CTRL Z is described in the 1720A User Manual. The remaining CTRL functions are described only as used by BASIC.

1-71. CTRL P

1-72. CTRL P stops any input/output operation in progress, loads the Console Monitor from the System Device, and transfers control to it. This action is taken regardless of what the 1720A was doing previously. See the Input and Output section for a discussion of the System Device.

CAUTION

CTRL P deletes the user program in memory. Also, CTRL P might not properly terminate a file transfer operation in progress (the buffers are not "flushed").

1-73. CTRL C

1-74. CTRL C returns BASIC to Immediate Mode, resulting in a "Ready" display prompt. In addition, a common use of CTRL C is to terminate a program test run during program development. Some further considerations are:

- If an IEEE-488 Bus operation is in progress, CTRL C stops the transfer immediately.
- In Edit Mode, CTRL C will store the current line as edited, if it is syntatically correct. If not, the line is stored as it was before editing.
- The ABORT button is treated by BASIC as a CTRL C; in addition, a Device Clear message is sent to both IEEE-488 instrument ports.
- In Run Mode, execution of an ON CTRL/C GOTO statement inhibits a return to Immediate Mode. Instead, control is transferred to the program line referenced. See the Interrupt Processing section.

1-75. CTRL S and CTRL Q

1-76. CTRL S stops the display from scrolling. CTRL/Q allows the display to continue scrolling. If a running program displays enough data to make the display scroll, CTRL S will stop the display for study. The program will suspend output to the display until CTRL Q is entered.

1-77. There are some differences between using CTRL S and CTRL Q, and using Page Mode with the NEXT PAGE key for control of display scrolling:

- NEXT PAGE erases the display and begins filling it again from the top. CTRL Q allows scrolling to continue from the bottom.
- After a CTRL S has been pressed, there is no indication of the status of the system, e.g., no indicator as on the PAGE MODE key. Thus, CTRL S can cause the keyboard to appear "dead". CTRL C will clear this condition.

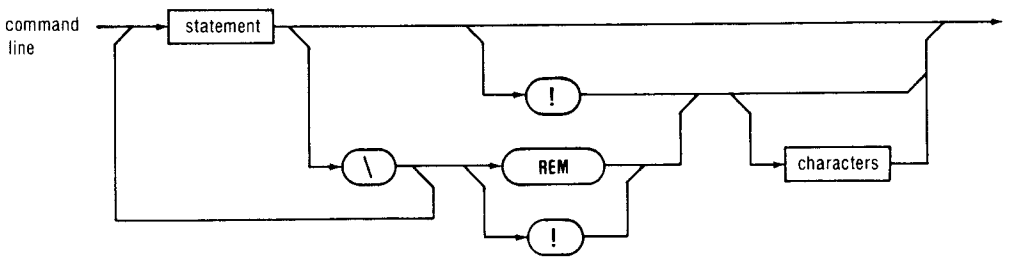
1-78. CTRL T

1-79. CTRL T erases the display, moves the cursor to the upper left corner, enables single-size characters, disables graphics mode and all other display enhancements, and enables the keyboard. It then sends a Carriage Return and Line Feed. The section on The Touch Sensitive Display describes display enhancements.

1-80. CTRL U

1-81. CTRL U deletes the current line containing the cursor.

1-82. Using Immediate Mode Statements

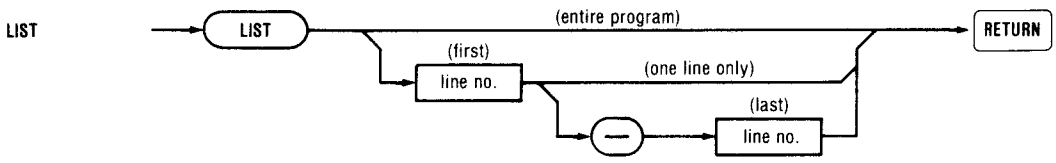


1-83. To create a program while in Immediate Mode, type in the individual program lines using the format of the syntax diagram above. The following rules also apply:

- Each program line must contain a line number.
- Each line may contain remarks only, or may contain one or more statememts optionally followed by remarks.
- Each program line must not exceed 79 characters in length, including line number.
- Regardless of the sequence that program lines are entered, BASIC will store and execute them in line number order.
- To insert a line, use any line number between the two lines where it needs to be placed. BASIC will store it in proper sequence. This is why BASIC line numbers are normally incremented by 10.
- To replace a line, type the new line with the line number of the old line.
- To delete a line, type the line number only. Then press RETURN. See also DELETE below.
- When all program lines have been entered, type RUN to run the program.
- Program lines may contain multiple statements (commands) which are separated by the backslash character (\) but must not exceed the 79 character length restriction mentioned earlier.
- In general, BASIC executes all the statements on a multiple statement line before going on to the next line. The IF-THEN statement discussed later is one important exception to this general rule.

1-84. This discussion presents Immediate Mode commands which aid in creating or modifying a program. They are: LIST, DELETE, REN, and EDIT. A syntax diagram is included for each command.

1-85. The LIST Command



1-86. The LIST command displays a program or a portion of a program in line number order.

- Display starts at the first line of a program and proceeds to the last line, if line numbers are not specified.
- One line is displayed if a single line number is specified. The command is ignored if the line does not exist.

- A portion of a program is displayed if two line numbers are specified. The display will be the lines with numbers between and including the specified lines numbers if they exist.

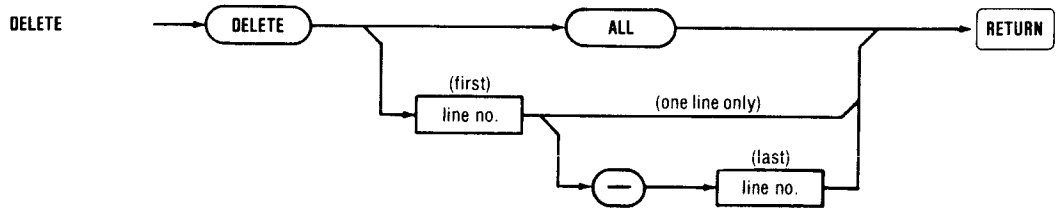
- If the portion to be listed is larger than one display page (16 lines) the display will scroll upwards until the last line specified has been displayed.

- Use Page Mode and the NEXT PAGE key, or CTRL S and CTRL Q to stop and restart the display. These functions are discussed earlier in this section.

1-87. The following examples illustrate common uses of the LIST command:

	RESULTS
`LIST`	Displays the entire program in memory from the first line.
`LIST 500`	Displays only line 500 of the program, if it exists.
`LIST 600-800`	Displays a program segment beginning with the first line after 599 and ending with last line before 801.

1-88. The DELETE Command



1-89. The DELETE command deletes part or all of a program from memory.

- The entire program is deleted when ALL is specified.

- DELETE ALL also de-allocates Common Variables (see the COM statement in the Program Chaining section).

- One line is deleted if a single line number is specified. The command is ignored if the line does not exist.

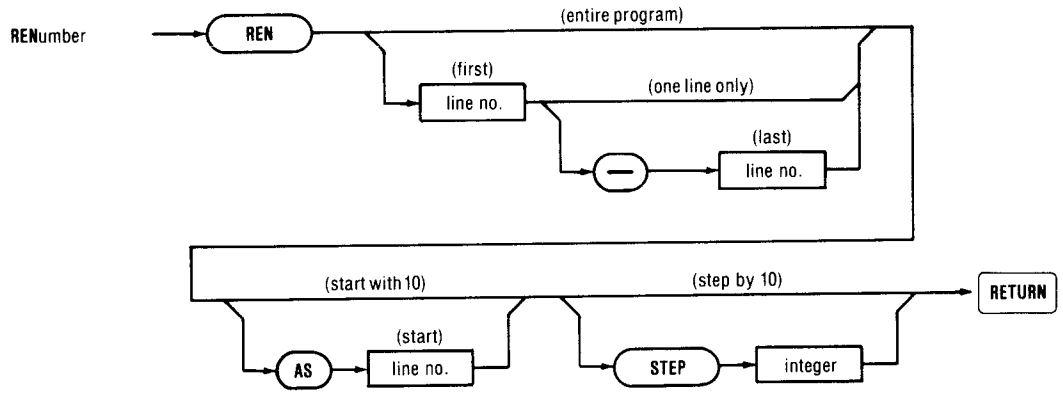
- One line may also be deleted by typing the line number only, followed by pressing RETURN.

- The portion of the program between and including specified lines is deleted if two line numbers are specified.

1-90. The following examples illustrate common uses of the DELETE command.

COMMAND	RESULTS
`DELETE`	No action
`DELETE ALL`	Deletes entire program
`DELETE 100`	Deletes only line 100
`DELETE 200-300`	Deletes lines 200 through 300
`400 <RETURN>`	Deletes line 400

1-91. The REM Command



1-92. The REN command changes the line numbers of some or all of the program lines in memory. Renumbering is useful to make room for additional program lines.

- REN cannot change the order of program lines.
- REN changes all references to line numbers in the program to reflect the new line numbers.
- All items but REN are optional.
- The entire program is renumbered when no line numbers are specified.
- One line is renumbered when a single line number is specified. The command is ignored if the line does not exist.
- A portion of the program is renumbered when two line numbers are specified.
- The line number following AS specifies the new starting number of the segment being renumbered. If this would re-arrange the sequence of the program, a fatal error occurs and the line numbering remains unchanged.
- The new starting line number is 10 when AS is not specified, and a line number or range of line numbers is not specified following REN.

- The new starting line number is the same as the old first line number of the range, when AS is not specified, and a line number or range of line numbers is specified following REN.

- The value of the integer expression following STEP must be positive. It defines the difference between any two consecutive, renumbered lines.

- If there is no STEP keyword, the line increment is 10.

- If the vlaue of the integet expression following STEP is so large that the new line numbers would force the program to be re-arranged, a fatal error occurs and the lines are not changed.

- The command REN is equivalent to REN 1-32767 AS 10 STEP 10.

CAUTION

Renumbering from lower to higher line numbers (with more digits) may cause the renumbered lines to exceed the 79-character maximum line length allowed by BASIC. Restricting program lines to 74 characters maximum length will generally eliminate this problem. This exception is on long lines which include line number references (e.g. ON expression GOTO, IF-THEN-ELSE with line numbers, etc).

NOTE

Program lines containing the ERL (error line) function may have statements such as IF ERL = 200 THEN RESUME 400. The expression, the constant 200, is used as a line number reference. It is not changed during renumbering. It may need to be changed to the correct line number manually.

1-93. The following program is used in the renumbering examples below:

```
10  A = 1
20  PRINT A + A
30  A = A + 1
40  IF A <= 2 THEN 20
50  PRINT "Done!"
60  END
```

COMMAND	RESULTS
`REN 60 AS 32767`	Change line 60 to read: `32767 END`

`REN 10-50 AS 5 STEP 5`	Change lines 10 through 50 to read:
	`5  A = 1`
	`10  PRINT A + A`
	`15  A = A + 1`
	`20  IF A <= 2 THEN 10`
	`25  PRINT "Done!"`
	Note the changed reference in line 20.

`REN`	This would in this case restore the program back to its original form.

`REN 60 AS 1000`	Renumber only line 60 as line 1000. An error results if any lines are numbered between 60 and 1001, since this would re-arrange program sequence.
`REN 60 AS 1000 STEP 5`	Same as the previous example. STEP is ignored when only one line is renumbered.
`REN 10-500 AS 1000`	Renumber lines 10 through 500 to start at 1000, in steps of 10.

1-94. Changing the Sequence of Program Lines

1-95. REN cannot be used to change the sequence of program lines. The simplest way to change the sequence of program lines is to use Edit Mode (discussed below) and change line numbers individually. The process must also include changing line number references (GOTO, etc.) so they will refer to the correct lines. Edit Mode, however, is not best suited for the task when larger portions of a program are to be moved, because of the time required.

1-96. As an alternate method:

1. Delete part of the program.
2. Renumber the remaining portion.
3. SAVE it under a temporary file name.
4. Retrieve the original program again, using OLD.
5. Repeat this procedure as needed.
6. Use the File Utility Program to merge these files together in the desired order. Refer to the 1720A User Manual for details.

1-97. Following are some points to keep in mind while changing the order of program lines in this manner:

- FUP cannot merge lexical files. Use SAVE (not SAVEL) to save the program segments to be merged.
- A BASIC program cannot have more than one line for each line number. If two or more program segments are merged which have internal line number conflicts, the last occurrance of each line number will be all that remains when the merged file is processed by the BASIC interpreter. Ensure that all program segments use different blocks of line numbers, in the re-arranged sequence that you need.
- Some of the line number references may be incorrect. Check and correct branching line number references before running the program.

1-98. Using the BASIC Editor

1-99. The editor provided as a part of Fluke Enhanced BASIC is an easy to use character oriented editor.

- Edit Mode is entered from Immediate Mode by typing EDIT, or EDIT line number.

- Editing begins with the lowest numbered line of the program in memory unless another line is specified.

- No line number specification is used when beginning the edit of a new program when no other program is in memory.

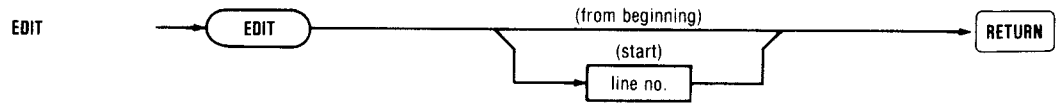
- Program entry procedure is the same as in Immediate Mode.

- Immediate Mode commands and program statements cannot be executed while in Edit Mode.

- Exit from Edit Mode to Immediate mode by entering CTRL C.

1-100. Edit Mode allows the user to create, delete, or modify the characters that make up program lines in main memory. Program lines are stored in main memory for subsequent use by other modes. The editing keys in the upper right corner of the keyboard plus the CTRL U, BACK SPACE, RETURN, and LINE FEED keys control the cursor and delete text. The remaining keys are used for text entry. This section describes the edit keys and their use along with other editor features.

1-101. The EDIT Command



1-102. This Immediate Mode command selects Edit Mode.

- The special edit keys on the programmer keyboard are enabled.

- Up to 15 lines of the program in memory are displayed, beginning at the first line or at the line number given with the command.

- Edit Mode enables the user to scroll the cursor forward or backward in a program as well as right or left on program lines.

- Edit Mode also enables the user to delete characters, portions of lines, or entire lines.

1-103. The following examples illustrate the two different uses of the EDIT command.

COMMAND	RESULTS
`EDIT`	Select Edit Mode and display up to 15 lines of the program in memory, beginning with the first line. If no program exists, the display is cleared and the cursor is positioned to the upper left corner of the display.
`EDIT 1000`	Select Edit Mode and display up to 15 lines of the program in memory, beginning with the first line greater than 999. If no program exists or the last line number is less than 1000, the display is cleared and the cursor positioned to the upper left corner of the display.

1-104. Edit Mode Keys

1-105. Some of the keys on the programmer keyboard have special functions that are enabled or modified in Edit Mode. Any key, if held down, performs its function repeatedly. Figure 1-2 describes the special functions of the Edit Mode Keys.

NOTE

Any edit command that will move the cursor from the current line is not accepted if the line does not pass a check for correct syntax. A blinking error message (e.g.: "Mismatched Quotes") will be displayed until the line is corrected.

←	Move one position left. Ignored if already at the left margin.
→	Move one position right. Ignored if already at the right end of the line.
↑	Move one position up. If the line above is shorter than the current column position, move left to the end of that line. Scroll down one line if the cursor is on the top line of the display, and another line as available. This action will not be done if the line does not pass a syntax check.
↓	Move one position down. If the line below is shorter than the current column position, move left to the end of that line. Scroll up one line if the cursor is on the bottom line of the display, and another line is available. This action will not be done if the line does not pass a syntax check.
DEL LINE	Delete from the cursor position to the end of the line. If the cursor is at the left margin, delete the entire line and move the rest of the program up one line to fill the deletion.
DEL CHAR	Delete the character at the cursor position and move the remaining characters left one position to fill the deletion. When held down for repeat, the portion of the line to the right of the cursor will progressively disappear.
DELETE	Delete the character to the left of the cursor position and move the remaining characters left one position to fill the deletion. When held down for repeat, the portion of the line to the left of the cursor will progressively disappear as the portion to the right moves to the left margin. This key function is also available in Immediate Mode.
CTRL + U	Delete the current line.
BACK SPACE	Move to the left margin.
LINE FEED	Move to the right end of the line.

Figure 1-2. Edit Mode Keys

RETURN	When the cursor is at the right end of the line, open a new empty line below, and move to its left margin. When the cursor is not at the right end of the line, break the line into two lines. The cursor position identifies the first character of the new (second) line. This action will not be done if the portion of the line that was to the left of the cursor does not pass a syntax check.
Character Keys	Insert characters at the current cursor position. Each character entry moves the cursor right one position along with any text to the right of the cursor. Entries that would result in a line length greater than 79 characters are not accepted, and produce a beep sound.
CTRL C	Return to Immediate Mode.

Figure 1-2. Edit Mode Keys (cont.)

1-106. Additional Editor Features

1-107. It is not necessary to insert a line in correct sequence. Regardless of the order in which program lines are entered, the editor will store them in memory in the correct line number sequence.

1-108. When the cursor is instructed to move from a line, the editor checks some syntax errors, such as omitting a quote, parenthesis, or line number. If a line does not pass the check, an error message is displayed and the cursor is not allowed to leave the line until the error is corrected.

1-109. If CTRL C is entered when the current line will not pass the syntax check, the blinking error message is displayed in Immediate Mode and the line is not stored in memory.

1-110. There are many errors the editor will not detect, such as forgetting to dimension an array, or specifying GOTO with a nonexistent line number. Such errors will be detected only when the program is run.

1-111. The cursor will not scroll above the lowest line number nor below the highest line number in the program. If the cursor is in the middle of the program and a new last line is entered at that position, the cursor will not scroll down past that line. There are two ways to correct this condition:

1. The cursor may be scrolled in the opposite direction until the line entered out of sequence disappears from the display. Reverse scroll direction again and the line will then be in proper sequence.

2. Enter CTRL C, and then type EDIT, followed by the line number that needed editing. Lines will then be displayed in correct sequence, allowing access to all lines.

1-112. When the cursor leaves a line, the editor stores that line in memory with the line number shown on the display, replacing any line currently in memory with the same line number. This feature will result in duplicate lines if the user attempts to move a line by changing only the line number and moving the cursor off the line. The line with the

previous line number is not deleted by this process. The display, however, will show only the most recent line number entered. To see both resulting lines, scroll the entered line off the display and back on.

1-113. When a line is scrolled off the display with the same line number as a line previously stored, the original line in memory will be replaced by the one which is scrolled off. In order to prevent this from occuring, assign each line a unique line number.

1-114. SAVING A PROGRAM

1-115. The procedures discussed above produce a program in main memory. However, whenever the user exits BASIC for the Console Monitor (using EXIT or CTRL P) the program in memory is deleted. To save a program for later use, it should be stored on a floppy disk.

1-116. Programs may also be stored on the optional electronic disk. Electronic disk procedures are identical to those for the floppy disk. It should be remembered however that the electronic disk will lose its contents whenever power is turned off unless the internal battery is activated, or an external battery is provided. Consult the 1720A User Manual for details.

CAUTION

Use of the internal battery for back-up of electronic disk contents has time limitations. Consult the 1720A User Manual for details.

1-117. To save a program currently in memory onto a floppy disk, follow the procedure below. Refer to the file names discussion in this section. Refer to the General Purpose Fluke BASIC Statements section for a discussion of SAVE and SAVEL.

1. Ensure that a disk is properly loaded with the door latched. Consult the 1720A User Manual.

2. Place BASIC in Immediate Mode as described in this section.

CAUTION

Do not exit BASIC, or the program will be deleted from memory.

3. Type SAVE "MF0:file name" followed by the RETURN key. A file name is 1 to 6 characters, consisting of letters, numbers, spaces, or $ signs.

1-118. LOADING AND EXECUTING A PROGRAM

1-119. BASIC provides two ways to retrieve a program from a floppy disk or the optional electronic disk. The OLD command loads a program into memory and remains in Immediate Mode. The RUN command loads a program into memory and immediately transfers control to the program and places BASIC into the Run Mode.

1-120. The OLD Command

1-121. The OLD command is used to load a program into memory from the floppy disk or the optional electronic disk.

- The file name, including optional storage device prefix and name extension, must be enclosed in quotes.

- BASIC will look for the file on the System Device if the floppy disk or electronic disk is not specified as a file name prefix. Refer to the Input and Output section for a discussion of the System Device. This section discusses file names earlier.

- BASIC will look on the specified device if it is included as a file name prefix (MFO: for the floppy disk, and EDO: for the electronic disk).

- This command assumes that the file named is a valid BASIC program in either ASCII or lexical form. A discussion of lexical form is included under SAVEL in General Purpose Fluke Basic Statements.

- If the file name extension is .BAS or .BAL, it does not need to be specified in the file name.

- If no extension is specified, BASIC looks for a file with .BAL name extension and loads it if it exists.

- If the file named does not exist with a .BAL extension, BASIC looks for the file with a .BAS extension and loads it if it exists.

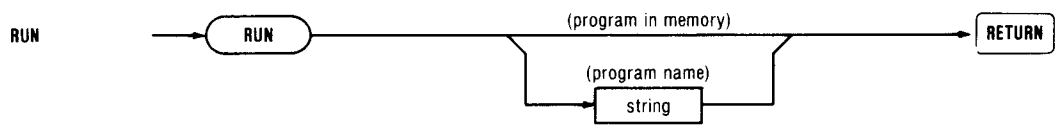
- If the file exists in both lexical (.BAL) and ASCII (.BAS) form, BASIC will load the lexical form unless the command specifies otherwise directly.

- Other references in this manual: None.

1-122. The following examples illustrate use of the OLD command:

COMMAND	RESULTS
`OLD "TEST"`	Load the file named TEST.BAL (if present) or TEST.BAS (if TEST.BAL is not present) from the default System Device into memory.
`OLD "MF0:TEST.5"`	Load the file named TEST.5 from the floppy disk.

1-123. The RUN Command



1-125. The RUN command begins execution of a user program at the lowest numbered line.

- The program in memory is run if no other file is specified.

- The file name, including optional storage device prefix and name extension, must be enclosed in quotes.

- BASIC will look on the default System Device for the file if the floppy disk or electronic disk is not specified as a file name prefix. Refer to the Input and Output section for a discussion of the System Device. File names are covered in this section.

- BASIC will look on the specified device if it is included as a file name prefix (MF0: for the floppy disk, and ED0: for the electronic disk).

- This command assumes that the file named is a valid BASIC program in either ASCII or lexical form. A discussion of lexical form is included under SAVEL in General Purpose Fluke Basic Statements.

- If the file name extension is .BAS or .BAL, it does not need to be specified in the file name.

- If no extension is specified, BASIC looks for a file with .BAL name extension and loads it if it exists.

- If the file named does not exist with a .BAL extension, BASIC looks for the file with a .BAS extension and loads it if it exists.

- If the file exists in both lexical (.BAL) and ASCII (.BAS) form, BASIC will load the lexical form unless the command specifies otherwise directly.

- Other references in this manual: Program Chaining.

Section 2
Data Types, Operators, And Expressions

2-1. INTRODUCTION

2-2. BASIC is designed with the ability to compute with numbers and strings. This section covers the data, data types and operators used by the programmer to build expressions. These expressions are a precise list of directions which BASIC follows to compute a result.

2-3. DATA TYPES

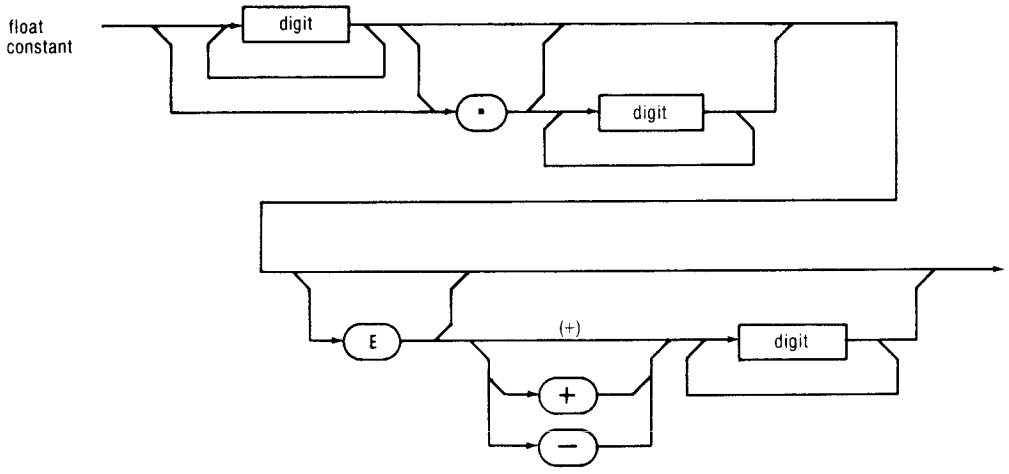
2-4. Floating Point Data

2-5. Floating point data has the following characteristics:

- Decimal exponent range from +308 to -308.
- Exact range 2.2225074E-308 to 3.595386E+308.
- Resolution 15 decimal digits.
- Inexactness in the numeric representation.
- Memory requirement (per data item): 8 bytes.
- Represented internally in binary in accordance with proposed standard "IEEE Floating Point Arithmetic for Microprocessors". Copies of this standard are available from The Institute of Electrical and Electronic Engineers, 345 East 47th Street, New York, New York, 10017.

2-6. Unless modified by a PRINT USING statement, floating point data is displayed with a leading space or sign, and a trailing space. It is printed out to seven significant digits. A value from .1 to 9999999 inclusive is printed out directly. A number less than .1 is printed out without E notation if all of its significant digits can be printed. All other values are printed in E notation (+0.dxxxxxxE+yyy), where d is a non-zero digit, x is any digit, and trailing zeros are dropped.

2-7. Floating Point Constants



2-8. Floating point constants, often called real numbers, are represented in a program in decimal or possibly scientific notation. The syntax diagram illustrates the proper representation of floating point numbers. A number in scientific notation, with an exponent following "E", represents a number multiplied by a power of 10. Examples of floating point constants are:

```
.005
6354.33
-134.7
-12E2          Represents -1200
0.13E-05       Represents .0000013
0.1E 6         Represents 100000
-.1E-400       Floating point number outside the
               legal range. Returns error 602.
```

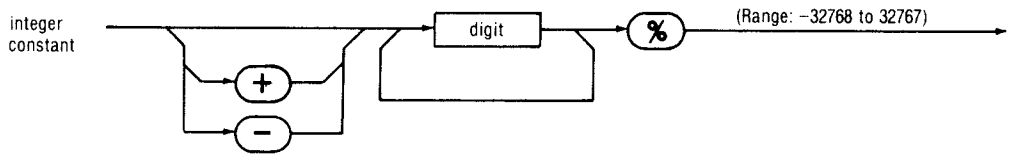
2-9. Integer Data

2-10. Integer data has the following characteristics:

- Range: -32768 to +32767
- Resolution: Integers
- Exactness.
- Memory requirement (per datum): 2 bytes
- Integer data is represented internally in binary but displayed by the 1720A in decimal without the modification process described in Floating Point Data.
- Operations that call for an integer result are rounded to an integer, if necessary.

2-11. Integer Constants

2-12. Integer constants are whole numbers identified by a "%" suffix on the number.



Examples of integers are:

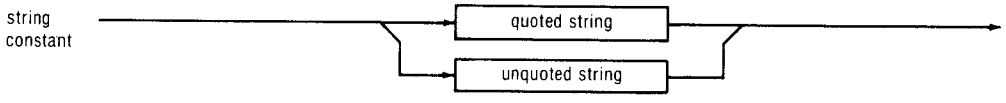
```
-0%
 5%
-32000%
-40000%    Outside the allowed range.
```

2-13. String Data

2-14. Strings are sequences of 8-bit positive integers that are normally interpreted as ASCII characters. Strings are used to store characters for messages to instruments and to the display, as well as for storage of binary data taken from instruments. String data has the following characteristics:

- Maximum length limited only by available memory or 16383 characters.
- Memory requirement: each string of 16 or less characters occupies an 18-byte memory segment and an additional 18 bytes for each additional 16 characters.
- String data is normally displayed by the 1720A in ASCII. See the Touch Sensitive Display section for exceptions.
- When interpreted as ASCII, the value of the most significant (8th) bit is ignored. See Appendix I, ASCII/IEEE-488 Bus Codes.

2-15. String Constants



2-16. String constants are expressed as a sequence of printable characters (numerics, upper case alphabetic characters, lower case alphabetic characters, printable symbols, e.g., *, –, [, etc). In most cases, string constants must be enclosed in either single or double quotes. Enclosing the statement in single quotes allows the use of double quotes in the constant and vice versa. String constants need not be expressed in quotes when part of a

DATA statement or when entered after an INPUT statement. Some examples of strings are:

```
A$ = "The result of 3.8 * PI is  "
IN$ = 'Reply with "YES" or "NO" '
```

2-17. System Constants

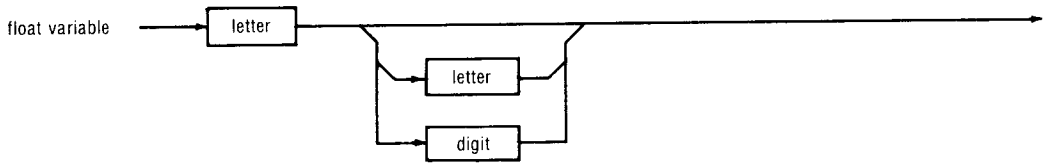
2-18. Fluke BASIC makes available two floating point constants. Under the name PI is stored the value 3.14159265358979. The mathematical function EXP(X) computes the result of raising the base (e) of the natural logarithm to a power expressed by the value of X. The function EXP(1) produces the stored value of e, 2.71828182845905. Refer to the Functions section for further information.

2-19. Introduction To Variables

2-20. Variables are named data items that may be changed by the actions of a running program.

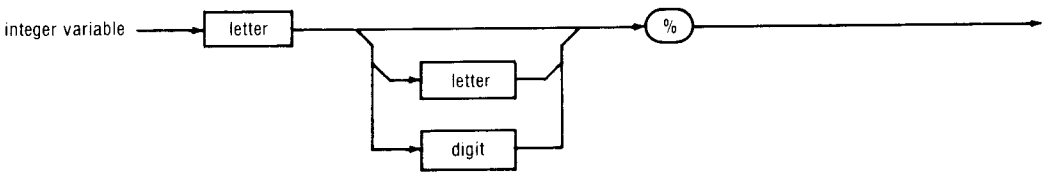
- They may be defined by the program itself, or they may be stored by the 1720A operating software.
- User variables are assigned a value by the assignment statement, by the READ statement, or by an INPUT statement.
- An assignment statement assigns a value (the result of evaluating an expression), to a variable. One form of an expression is a constant. For example, A=2.
- The READ statement and statements which input data, associate a variable name with a constant.
- System variables store changing event information, such as time of day or length of last file opened, for use as required by a program.

2-21. Floating Point Variables



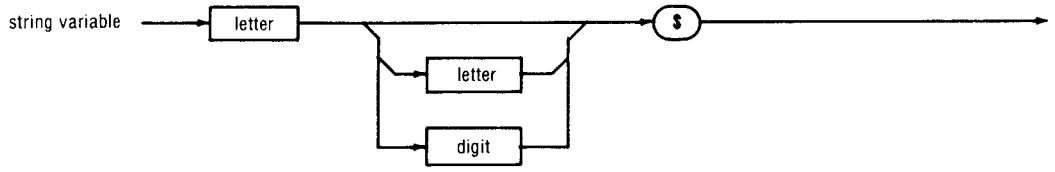
2-22. Floating point variables are designated by a letter followed by an optional second character. The second character can be a letter or a number. The following variable names are not allowed, since they are keywords of Fluke BASIC: AS, FN, IF, LN, ON, OR, PI, TO.

2-23. Integer Variables



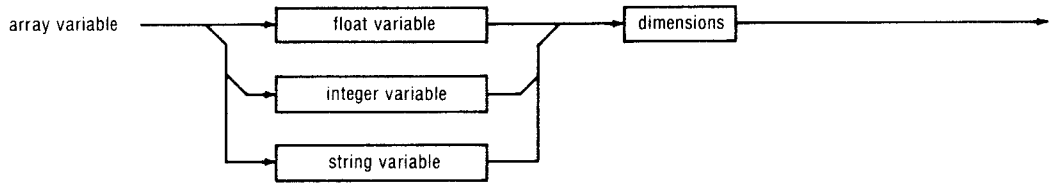
2-24. Integer variables are designated by a floating point variable name followed by a "%" character.

2-25. String Variables



2-26. String variables are designated by a floating point variable name followed by a "$" character.

2-27. Array Variables



2-28. An array variable is a collection of variable data under one name.

- Arrays consist of floating point, integer, or string variables.
- The variable name has either one or two subscripts to identify individual items within the array.
- Subscripts are enclosed in parentheses.
- When two subscripts are used, they are separated by a comma.
- It is helpful to view two dimensional arrays as a matrix. The first subscript is the row number, and the second subscript is the column number. For example, FT%(3,18) identifies the integer in row 3, column 18 of the array FT%(m,n).
- A subrange (portion) of an array can be designated by specifying a first and last subscript separated by two periods.

For Example:

```
A$(3..7)
```

Strings 3 through 7 of the string array A$.

```
FT%(2..4, 5..15)
```

Rows 2 through 4 in columns 5 through 15 of the integer array FT%.

- In the second example above, the second subscript is incremented or decremented before the first. For example, the statement PRINT FT%(2..4,5..15) will display the range FT%(2, 5 through 15) before displaying FT%(3, 5 through 15).

- Array variables are distinct from simple variables. A and A(0) are two different variables.

- Only one array variable can be associated with an identifier. A%(n) and A%(m,n) are not simultaneously allowed.

- Memory space must be reserved for an array variable before it can be used. See the discussion of the DIM statement in the General-Purpose Fluke BASIC Statements section.

- Virtual arrays are array variables accessible through a channel to the floppy disk, or the optional electronic disk. This feature allows a program to take advantage of the much greater storage space available on these mass storage devices. Refer to the Virtual Arrays section.

- Some examples of array variables are:

```
A%(3)
B1%(2%, 3%)
A$(5)
C(3%)
D(2 + A * B, C)
D( D(2) )
```

2-29. System Variables

2-30. System variables store changing event information for use as required by a program. They are accessed by name and return a result in floating point, integer, or string form as appropriate. Table 2-1 lists the system variables and gives their meaning and form.

Table 2-1. System Variables

NAME	TYPE	EXAMPLE	MEANING
DATE$	String	08-Feb-81	Current date in the format DD-Mnn-YY
ERL	Integer	1120	Line number at which the most recent BASIC program error occurred.
ERR	Integer	305	Error code of the most recent error the BASIC interpreter found in the program being executed.
FLEN	Integer	6	Length of the last file opened in 512-byte blocks.
KEY	Integer	20	Position number of the last Touch Sensitive Display region pressed.
MEM	Integer	29302	Amount of unused main memory, expressed in bytes.
RND	Floating Point	0.2874767	Pseudo-random number greater than 0 and less than 1. Repeatable if not preceded by RANDOMIZE.
TIME	Floating Point	0.5491405E+08	Number of milliseconds since the previous midnight.
TIME$	String	17:45	Current time of day in 24-hour format.

2-31. OPERATORS AND EXPRESSIONS
2-32. The Assignment Operator

=

2-33. When used as a program statement, with a variable to its left, and an expression to its right, = assigns the result of evaluating the expression to the variable. No equality is implied. This is the default form of the LET statement (see the General Purpose Fluke BASIC Statements section). In the following example, the integer N% is incremented by 1.

```
N% = N% + 1%
```

2-34. Arithmetic Operators

2-35. Arithmetic operators act upon or between numbers or numeric expressions to produce a numeric result. A numeric expression is composed of an arithmetic operator and one or more operands. Table 2-2 summarizes the arithmetic operators. The following guidelines apply:

- The + and - operators can act upon a single number or numeric expression (unary operation).

- All arithmetic operators act between two numbers or numeric expressions.

- Numeric variables can be used as numbers in expressions if they have previously been assigned a value.

- Floating point numbers and integers may be intermixed. Integers will automatically be converted to floating point, if necessary.

- When one integer is divided by another, the result is a truncated integer quotient (the fraction or remainder is truncated). For example:

```
2% / 5%      is  0%
15% / 3%     is  5%
17% / (-3%)  is  -5%.
```

- When a result is assigned to a numeric variable, it is automatically converted to the assigned data type.

- When a floating point number is assigned to an integer variable it is rounded, not truncated.

- Computation speed is significantly faster when floating point and integer data types are not intermixed.

- The result of evaluating an arithmetic expression may be used in a larger expression, or assigned to a variable for later use.

- Except for +, arithmetic operators cannot be used on strings.

Table 2-2. Arithmetic Operators

OPERATOR	NAME	MEANING AND EXAMPLES
+	Positive	Unary plus operator. Does not change sign. + 58.4 + D I% = + KR%
+	Add	Add two numeric quantities. .382 + .046 A + B RE% = CK + T% IF C% = K% + 1% THEN RESUME
-	Negative	Unary minus operator. Changes the sign. - 73.138 - KV C% = - ST%
-	Subtract	Subtract two numeric quantities. 46332.33 - 473.88 C - D T% = RE% - CK IF D% = T% -10% THEN WAIT FOR KEY
*	Multiply	Multiply two numeric quantities. 44.21 * 3.992 3.582 * RR K% = 4 * I% IF X > 4.77 * CK THEN Y = Y * 5.76
/	Divide	Divide first numeric quantity (dividend) by the second numeric quantity (divisor) to produce a quotient. 3.584 / 0.338 KV / 2% MA = V / KO PRINT C / PI
^	Exponentiate	Raise the first numeric quantity to a power equal to the second numeric quantity. PI * R^2% LS = VR^2.886E-6

NOTE

Exponentiation is left associative, meaning that A^B^C is evaluated as (A^B)^C.

2-36. Relational Operators

```
=     <     >     <>     <=     >=
```

2-37. Relational operators compare numeric values or character strings. A relational expression returns an integer Boolean result of 0 for false, and -1 for true. The structure of a statement determines whether the = operator is used for a relational comparison or an assignment. Two examples:

`PRINT A = B` Displays -1 if A and B are equal.

`A = B = C` Assigns -1 to A if B and C are equal.

2-38. Numeric Comparisons

2-39. Numeric comparisons are made as follows:

- All negative numbers are "less than" zero or any positive number.
- Integers are converted to floating point when a comparison is between mixed numeric data types. This conversion requires additional processing time.
- When an operator checks for equality or inequality of numeric expressions, use integers wherever possible. This is due to the inexactness, and rounding and truncation errors, of floating point values.
- To check equality of floating point numbers, compare the absolute value of their difference to a small enough limit. For example, use ABS(A - B) < 1E-15 instead of A = B.

2-40. String Comparisons

2-41. Strings are compared on the basis of character matching, order within the standard ASCII code set, and overall length.

- A string character value is its relative position in the ASCII code table (see Appendix I.). This means, for example, that all capital letters are less than any lower-case letter.
- String comparisons are done left to right, character for character. The first inequality determines "less than" or "greater than".
- If no inequality is found, the shorter string is "less than".
- Strings must be identical to be equal.

2-42. The following examples illustrate the rules for string comparisons:

TRUE COMPARISON	REASON
`"" < any other string`	"" is a string with no elements.
`" Z" < "AZ"`	Space is less than "A".
`" Z" < "A"`	Space is less than "A".
`"LONG STRING" < "LONGER STRING"`	Space is less than "E".
`"1999" < "20"`	"1" is less than "2".
`"ON" < "ONE"`	"ON" has fewer characters.

2-43. String Concatenation Operator

+

2-44. When used between character strings, + concatenates (connects) the strings together. The following examples illustrate the use of this operator.

EXPRESSION(S)	RESULT
`"BEGIN" + "OPERATION"`	`BEGINOPERATION`
`"BEGIN " + "OPERATION"`	`BEGIN OPERATION`
`A$ = " Volts"` `"ENTER" + A$`	`ENTER Volts`
`A$ = "Millivolts"` `"ENTER" + A$`	`ENTER Millivolts`

2-45. Logical (Bitwise) Operators

2-46. Logical operators AND, OR, XOR, NOT, operate on the binary digits that make up an integer. NOT is a unary operator, acting upon one integer. AND, OR, and XOR are binary operators, using the bits of one integer to act upon another integer. Logical operators allow examination or modification of integer bit patterns when they have been used to store binary data, such as status or binary readings from instrumentation.

2-47. Binary Numbers

2-48. To use logical operators effectively, it is necessary to understand the binary number system, and how the 1720A Instrument Controller uses binary numbers to represent integers.

2-49. The 1720A Instrument Controller uses a 16-bit (2-byte) word to store each integer. Each bit position represents a weighted power of 2. The sum of the weight column in the following chart is the number of separate integers that can be represented with 16 bits less one because zero is an additional integer.

Bit Position	Weight
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
SUM:	65535

2-50. Integer numbers are represented by setting appropriate bit positions to 1. Adding the weighted values of each position that is set to 1 gives the decimal value of the integer. For example, the number 305 is represented, by the binary pattern 0000 0001 0011 0001, as follows:

Position:	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
Setting:	0 0 0 0	0 0 0 1	0 0 1 1	0 0 0 1

2-51. Since each bit that is set carries a binary weight, it is possible to verify that the binary pattern is correct. Adding the numbers 256 + 32 + 16 + 1 gives the decimal value 305.

2-52. Decimal numbers can be converted to binary by continuously dividing by 2 and keeping track of the remainders (always 1 or 0). For example, the number 305 is converted to binary as follows:

305 / 2 = 152	R = 1
152 / 2 = 76	R = 0
76 / 2 = 38	R = 0
38 / 2 = 19	R = 0
19 / 2 = 9	R = 1
9 / 2 = 4	R = 1
4 / 2 = 2	R = 0
2 / 2 = 1	R = 0
1 / 2 = 0	R = 1

2-53. Reading the remainder bits, from the bottom up, gives the result 100110001, the binary representation of 305.

2-54. Twos Complement Binary Numbers

2-55. By using the most significant bit (15) to identify a negative integer, the 1720A divides the pattern into 32767 positive numbers, the number 0, and 32768 negative numbers. Negative numbers are represented in a form called two's complement. To change either to or from two's complement form, the following steps are required:

1. Replace every 1 with a 0.
2. Replace every 0 with a 1.
3. Add 1.

2-56. For example, to change the pattern 0000 0001 0011 0001 (+305) to two's complement form, first reverse 1's and 0's: 1111 1110 1100 1110, and then add 1: 1111 1110 1100 1111 (-305). To change it back, first reverse 1's and 0's: 0000 0001 0011 0000, and then add 1: 0000 0001 0011 0001.

2-57. The AND Operator

2-58. AND returns an integer bit pattern with a 1-bit in every position where both of two input integers have a 1-bit. The AND operator is useful to check for the setting of particular bit(s) to 1 by operating on an unknown status word with a mask word (number) having the appropriate bit(s) set to 1. The following examples illustrate the results of AND operations:

```
33% AND 305%          33% 0000 0000 0010 0001
                     305% 0000 0001 0011 0001
                     ---------------------
Result:  33%              0000 0000 0010 0001

-74% AND 305%        -74% 1111 1111 1011 0110
                     305% 0000 0001 0011 0000
                     ---------------------
Result:  304%             0000 0001 0011 0000
```

2-59. The OR Operator

2-60. OR returns an integer bit pattern with a 1-bit in every position where either of two input integers have a 1-bit. The OR operator can be used to check for the resetting of particular bit(s) to 0 by operating on an unknown status word with a mask word (number) having the appropriate bit(s) set to 0. The following examples illustrate the results of OR operations:

```
33% OR 305%           33% 0000 0000 0010 0001
                     305% 0000 0001 0010 0001
                     ---------------------
Result:  305%             0000 0001 0011 0001

-74% OR 305%         -74% 1111 1111 1011 0110
                     305% 0000 0001 0011 0001
                     ---------------------
Result:  -73%             1111 1111 1011 0111
```

2-61. The XOR Operator

2-62. XOR (Exclusive OR) returns an integer bit pattern with a 1-bit in every position where the bits of two input integers are opposite. A mask word applied to an unknown integer through XOR will invert (1 to 0, and 0 to 1) all bit positions where the mask contains a 1, and leave unchanged all bit positions where the mask contains a 0. The following examples illustrate the results of XOR operations:

```
33% XOR -1%           33% 0000 0000 0010 0001
                      -1% 1111 1111 1111 1111
                     ---------------------
Result:  -34%             1111 1111 1101 1110

-74% XOR 0%          -74% 1111 1111 1011 0110
                       0% 0000 0000 0000 0000
                     ---------------------
Result:  -74%             1111 1111 1011 0110
```

2-63. The NOT Operator

2-64. NOT is a unary operator that operates upon a single integer. NOT returns a 1-bit in every position where the input integer had a 0 bit, and a 0 bit in every position where the input integer had a 1-bit. The following examples illustrate the results of XOR operations:

```
NOT 33%               33% 0000 0000 0010 0001
                     ---------------------
Result:  -34%             1111 1111 1101 1110
```

```
NOT -74%                -74% 1111 1111 1011 0110
Result: 73%                  0000 0000 0100 1001
```

2-65. Operator Hierarchy

2-66. As long as the results of individual operations are compatible with each other, operators can be combined in any order within an expression. However, BASIC follows certain internal rules of hierarchy when evaluating expressions. Table 2-3 lists the operators discussed within this section in seven levels of hierarchy.

- An expression is scanned left to right for level 1 operations (exponentiation).
- After performing these, the expression is scanned left to right for level 2 operations (+ sign, - sign).
- This sequence is continued until level 7 operations have been scanned and performed, if present.
- Within the same priority level, operations are performed in left to right sequence.
- This sequence can be modified by the use of parentheses (discussed below).

2-67. The following example illustrates these concepts:

The expression: `1 + 2 * 7 ^ 4 / 3 - 5`

is evaluated as `(7 ^ 4) * (2 / 3) + 1 - 5` (hierarchal order)

not as `((((1 + 2) * 7) ^ 4) / 3) - 5` (left-to-right order)

Table 2-3. Operator Priority

PRIORITY	OPERATOR	FUNCTION
1	^	Exponentiate
2	+	Positive Sign (Unary plus)
2	-	Negative Sign (Unary minus)
3	*	Multiply
3	/	Divide
4	+	Add
4	-	Subtract
5	<	Less Than
5	>	Greater Than
5	=	Equals
5	<>	Not Equal to
5	<=	Less Than or Equal To
5	>=	Greater Than or Equal To
6	NOT	Logical Not (Unary)
7	AND	Logical And
7	OR	Logical Or
7	XOR	Logical Exclusive Or

2-68. Use of Parentheses in Expressions

2-69. After the rules of operator hierarchy are satisfied, expression evaluation normally proceeds from left to right. Parentheses can be used to organize and change the evaluation sequence of expressions. The following rules govern the interpretation of parentheses by Fluke BASIC:

- Parentheses have priority over all operators.
- Each left parenthesis "(" must have a corresponding right parenthesis ")".
- A pair of parentheses may be nested within another pair. Line length is the only limit to this nesting.
- Evaluation of nested parentheses proceeds from the innermost pair outwards.
- Pairs of parentheses that are not nested within each other are evaluated in left to right sequence.
- Parentheses may also be used where they have no effect except clarity to the programmer.

2-70. The following example illustrates these concepts. The variables have the values A = 2, B = 3, C = 4, D = 5, and E = 6.

The expression `A + B * C + D / E` will evaluate as:

```
2 + 3 * 4 + 5 / 6
2 + 12 + .833333....
14.83333....
```

The expression `(A + B) * (C + D) / E` will evaluate as:

```
(A + B) * (C + D)/E
(5) * (9)/6
45/6
7.5
```

. . a very different result.

Section 3
Functions

3-1. INTRODUCTION

3-2. Functions are predefined operations available in Fluke BASIC by applying a function name to an appropriate set of arguments. The availability of a wide range of functions can significantly simplify a programming task. Mathematical functions perform calculations on numeric quantities. String functions create, manipulate, measure, and extract portions of character strings. In addition, Fluke BASIC allows the user to define specialized functions to meet the needs of particular data processing and instrumentation control tasks. Refer to the discussion of the DEF FN statement in the section General Purpose Fluke BASIC Statements.

3-3. OVERVIEW

3-4. This section is divided into two subject areas: String Functions and Mathematical Functions. Effective use of functions requires a clear understanding of proper data types and formats for input and expected results. A function may be used in a BASIC expression or statement in place of the same data type that the function produces. Each subsection includes definitions and examples to clarify these concepts.

3-5. STRING FUNCTIONS

3-6. String functions create, manipulate, and extract portions of character strings.

- String functions operate only upon strings, but can have floating point numbers, integers, or character strings in their argument list.

- Integer results from string input include:

  ASCII (decimal) value of a character.

  Length of a string.

  Number of characters within a string at which a substring was located.

- String results from integer input include:

  Character corresponding to an ASCII (decimal) value.

  Specified left, right, or center substrings.

  Space character strings, of either specified length or to a specified column position.

  Escape code sequence for positioning the display cursor.

- A string result is available in the format that PRINT or PRINT USING would display a number.

- Floating point numbers may be used in place of integer input. BASIC will truncate as necessary and convert to integer form (requires additional processing time). Note that truncation to integer form may cause unexpected results.

- Numeric expressions may be used in place of floating point or integer numeric input.

- Table 3-1 summarizes the character string functions. The discussions that follow detail the characteristics of these functions.

- See also Appendix I, ASCII/IEEE-488 Bus Codes, for a chart of ASCII characters and corresponding decimal numbers.

Table 3-1. String Functions

FUNCTION	DESCRIPTION
ASCII (A$)	Integer position of the character A$ in the ASCII code set.
CHR$ (I%)	String character corresponding to position number I% in the ASCII code set.
CPOS (L%, C%)	Eight-character escape-code string that will, if sent to the display, position the cursor to line L% and column C%.
INSTR (N1%, A$, B$)	Integer location of the first character of string B$, if it is found completely within string A$. Search begins at the N1%th character of string A$.
LEFT (A$, N%)	String composed of leftmost N% characters within string A$.
LEN (A$)	Integer number of characters in A$.
MID (A$, N1%, N2%)	String taken from A$, starting at the N1%th character and N2% characters long.
NUM$ (N)	Numeric string formatted as PRINT would display the value of floating point number N.
NUM$ (N, A$)	Numeric string formatted as PRINT USING A$, N would display the value of floating point number N.
RIGHT (A$, N%)	String composed of rightmost characters within string A$, starting with the N%th character from the left.
SPACE$ (N%)	String of N% space characters.
TAB (N%)	String of spaces that would move from the current print position to column N% + 1. Preceded by Carriage Return and Line Feed if current print position is beyond column N% + 1.Not intended for the display.
VAL (A$)	Floating point value of numeric string A$.

3-7. The ASCII Function

Format: ASCII (string)

3-8. The ASCII function returns an integer equal to the ASCII (decimal) value of the first character in the specified string.

- ASCII has a string as an argument and yields an integer result.
- The string may be any length, subject only to string length limits.
- Refer to Appendix 1, ASCII/IEEE-488 Bus Codes, for a chart of ASCII characters and corresponding decimal numbers.
- Range of possible results is 0 to 255.
- ASCII characters generated by the 1720A give results from 0 to 127. The most significant bit (7) is reset to 0.
- Result is 128 higher when the most significant bit (7) in the character is set to 1. This allows string characters to be examined as 8-bit binary data bytes.
- A null (no characters) string input produces a 0 result.
- The ASCII function may be used in any expression or statement that allows the use of an integer variable.

3-9. The following examples illustrate the results of different uses of the ASCII function. A$ = "NOW" and B$ = "" (null string).

STATEMENT	RESULT
`PRINT ASCII (A$)`	Display 78, the ASCII value of "N", the first character of string A$.
`B% = ASCII (A$)`	Places 78, the ASCII value of "N", the first character of string A$, into integer variable B%.
`IF ASCII (A$) = 13% THEN 1200`	Branch to line 1200 if the first character of string A$ is a Carriage Return.
`PRINT ASCII (B$ + A$)`	Display 78. The null string has no ASCII value: B$ (null) + A$ (NOW) equals "NOW" ("N" is still the first character).
`PRINT ASCII ("N")`	Display 78. A string constant may also be used.

3-10. The LEFT Function

Format: LEFT (string, number of characters)

3-11. The LEFT function returns a substring of the specified string, starting from the left. The number of characters returned is, if possible, the number specified by the second argument.

- LEFT has as arguments a string and an integer, and yields a string.
- LEFT returns a null string when the number of characters is specified as 0.
- LEFT returns an identical string when the number of characters is specified equal to or longer than the string.

3-12. In the following examples, the string A$ contains the characters "THIS IS THE FIRST EXAMPLE STRING", and L% = 11%.

STATEMENT	RESULT
`X$ = LEFT (A$, L%)`	X$ = "THIS IS THE", the leftmost 11 characters of A$.
`PRINT X$; " LEFT PART"`	Displays "THIS IS THE LEFT PART".

3-13. The RIGHT Function

Format: RIGHT (string, starting character position)

3-14. The RIGHT function returns a substring of the specified string, starting from the specified character position, to the right end of the string.

- RIGHT has as arguments a string and an integer, and yields a string.
- RIGHT returns a null string when the starting character position specified is longer than the string.
- RIGHT returns an identical string when the starting character position is specified as 0 or 1.

3-15. In the following examples, the string A$ contains the characters "THIS IS THE FIRST EXAMPLE STRING".

STATEMENT	RESULT
`Y$ = RIGHT (A$, 27%)`	Y$ = "STRING", the characters of A$ starting 27 characters from the left.
`PRINT "RIGHT "; RIGHT (A$, 27%)`	Displays "RIGHT".

3-16. The MID Function

Format: MID (string, starting character position, number of characters)

3-17. The MID function returns a substring of the specified string, starting from the specified character position, and including the specified number of characters.

- MID has as arguments a string and two integers, and yields a string.
- MID (A$, S%, N%) is equivalent to LEFT (RIGHT (A$, S%),N%).
- MID returns a null string when the number of characters specified is zero.
- MID returns a null string when the starting character position specified is longer than the string.
- MID returns an identical string when the starting character position specified is as 0 or 1, and the number of characters specified is equal to or longer than the string.

3-18. In the following examples, the string A$ contains the characters "THIS IS THE FIRST EXAMPLE STRING". X$ and Y$ are taken from the previous examples.

STATEMENT	RESULT
`Z$ = MID (A$, 13%, 13%)`	Z$ = "FIRST EXAMPLE", 13 characters of A$, starting at position 13.
`PRINT Z$;" ";Y$;" ";X$;`	Displays "FIRST EXAMPLE STRING THIS IS THE".

3-19. The LEN Function

Format: LEN (string)

3-20. The LEN function returns the number of characters contained in a string.

- LEN has a string as an argument, and yields an integer.
- The string count includes leading and trailing blanks and null characters.
- LEN returns 0 from a null string (no characters) input.

3-21. The following examples illustrate the results of uses of the LEN function.

STATEMENT	RESULT
`X% = LEN (A$)`	Place the length of string A$ into the integer variable X%.
`PRINT LEN (A$)`	Display the length of A$. For example, if A$ contains "HELLO!", 6 is displayed.

3-22. The INSTR Function

Format: INSTR (starting search position, string, substring)

3-23. The INSTR function searches for a specified substring within a string and returns the starting location of the substring.

- INSTR has as arguments an integer and two strings, and yields an integer.
- The integer is the starting character position for the search.
- The first string is the string to be searched.
- The second string is the substring to be searched for.
- The substring must be exactly and wholly contained within the string for the search to be successful.
- INSTR returns 0 when the substring is not found.
- INSTR returns 0 when the starting character position is greater than the length of the string.
- INSTR returns, in integer form, the position in the string of the first character of the substring when the substring is found.
- INSTR returns the number of the starting character position when the substring is null, and the starting character position is less than the length of the string.

3-24. The following program generates the interaction printed below it. The NOTE printed below each example is commentary in this manual, and not part of the output from the program.

```
1000  REM -- Demonstrate INSTR Function
1020  A$ = "This string has 35 characters in it"
1030  PRINT "The given string is: "; A$
1040  PRINT "Type the substring to be found"
1050  INPUT B$ \ PRINT
1060  PRINT "Type the position at which to start the search";
1070  INPUT S% \ PRINT
1080  PRINT "The value of INSTR (S%, A$, B$) is the same as:"
1090  PRINT "INSTR ("; S%; ', "'; A$; '", "'; B$; '"):';
1100  PRINT INSTR (S%, A$, B$)
1110  PRINT \ PRINT
1120  GOTO 1020
```

Results of running the program:

```
The given string is: This string has 35 characters in it
Type the substring to be found? has
Type the position at which to start the search? 6
The value of INSTR (S%, A$, B$) is the same as:
INSTR (6, "This string has 35 characters in it", "has"): 13
```

NOTE: Character 13 of A$ is the "h" of has.

```
The given string is: This string has 35 characters in it
Type the substring to be found? has
Type the position at which to start the search? 14
The value of INSTR (S%, A$, B$) is the same as:
INSTR (14, "This string has 35 characters in it", "has"): 0
```

NOTE: There is no "has" from character 14 to the end of A$.

```
The given string is: This string has 35 characters in it
Type the substring to be found? strung
Type the position at which to start the search? 1
The value of INSTR (S%, A$, B$) is the same as:
INSTR (1, "This string has 35 characters in it", "strung"): 0
```

NOTE: There is no "strung" in A$.

```
The given string is: This string has 35 characters in it
Type the substring to be found? IN
Type the position at which to start the search? 1
The value of INSTR (S%, A$, B$) is the same as:
INSTR (1, "This string has 35 characters in it", "IN"): 0
```

NOTE: The search is sensitive to whether or not the letters are capitalized.

```
The given string is: This string has 35 characters in it
Type the substring to be found? in
Type the position at which to start the search? 1
The value of INSTR (S%, A$, B$) is the same as:
INSTR (1, "This string has 35 characters in it", "in"): 9
```

NOTE: "in" is embedded in "string". It begins at character 9 of A$.

```
The given string is: This string has 35 characters in it
Type the substring to be found? in
Type the position at which to start the search? 1
The value of INSTR (S%, A$, B$) is the same as:
INSTR (1, "This string has 35 characters in it", "in "): 31
```

NOTE: Search for i - n - (space) to find the word "in".

```
The given string is: This string has 35 characters in it
Type the substring to be found?
Type the position at which to start the search? 5
The value of INSTR (S%, A$, B$) is the same as:
INSTR (5, "This string has 35 characters in it", ""): 5
```

NOTE: The null string is found at location 5.

```
The given string is: This string has 35 characters in it
Type the substring to be found?
Type the position at which to start the search? 36
The value of INSTR (S%, A$, B$) is the same as:
INSTR (36, "This string has 35 characters in it", ""): 0
```

NOTE: A search for the null string beyond the end of A$ fails.

3-25. The CHR$ Function

Format: CHR$ (integer)

3-26. The CHR$ function returns an ASCII string character corresponding to an integer input.

- CHR$ has integer an as argument, and yields a single-character string (8-bit binary pattern).

- Refer to Appendix I, ASCII/IEEE-488 Bus Codes, for numbers from 0 to 127. Use the number in the decimal column to produce a desired symbol, display control, character, or binary pattern.

- To set the upper bit (7), add 128 to the decimal number corresponding to the desired bit pattern.

- The range of acceptable input is –32768 to 32767. The upper byte of the integer is ignored.

- Other references in this manual: Touch Sensitive Display

3-27. The following example illustrates the results of common uses of the CHR$ function.

```
1000  A = 64 \ B% = 50% \ C% = 70%
1010  PRINT CHR$ (A)                    ! Displays "@"
1020                                     ! (ASCII value of "@" is 64)
1030  B$ = CHR$ (B%)                     ! Assign "2" to B$
1040                                     ! (ASCII value of 2 is 50)
1050  PRINT CHR$ (55) + CHR$ (C%)        ! Displays "7F"
1060                                     ! (ASCII value of 7 is 55)
1070                                     ! (ASCII of F is 70)
1080  PRINT CHR$ (A + B% + 3)            ! Displays "u"
1090                                     ! (ASCII value of u is 117)
1100  PRINT CHR$ (2880)                  ! Displays "@"
1110                                     ! (ignores upper 7 bits)
1120  PRINT CHR$ (72.65)                 ! Displays "H"
1130                                     ! (Truncates fraction)
```

NOTE

Some integers used in the CHR$ function result in display commands when printed and do not appear on the screen. For example PRINT CHR$(7) activates the beeper.

3-28. The CPOS Function

Format: CPOS (row number, column number)

3-29. The CPOS function returns a string which directly positions the display cursor when printed.

- CPOS has as arguments, two integers, and yields an 8-character string.

- The range of each number acceptable to CPOS is 0 to 32767.

- The range of row numbers acceptable to the display is 0 to 16 in normal display mode, and 0 to 8 in double size display mode.

- The range of line numbers acceptable to the display is 0 to 80 in normal display mode, and 0 to 40 in double size display mode.

- In cursor positioning commands, the positions 0 and 1 are identical.

- Numeric strings in the result are two characters in length. Numbers between 0 and 9 are given a leading 0.

- This function is discussed in greater detail in the section Touch Sensitive Display.

3-30. The following examples illustrate the results of different uses of the CPOS function.

STATEMENT	RESULT
`PRINT CPOS (2, 4)`	Place the cursor at line 2 column 4.
`PRINT CHR$(27); "[02;04H"`	Place the cursor at line 2 column 4. This is identical to the previous example.
`A$ = CPOS(2, 4)`	Place the string "ESCape [02;04H" into A$.
`PRINT CPOS(A, B); "HELLO"`	Displays "HELLO" at line A, column B.

3-31. The TAB Function

Format: TAB (column number)

3-32. The TAB function returns a string of spaces that advance the current print position on an external printer to one column past the specified column number.

- TAB has an integer as an argument, and yields a string of spaces that may be preceded by Carriage Return and Line Feed.

- Any positive integer may be used (0 to 32767). TAB does not limit the length of a line.

- Because TAB is intended for an external printer, column position may be different than the display cursor.

- The current print position used by TAB to compute the number of spaces required is the total number of characters since the last Carriage Return and Line Feed.

- The display cursor column position will be different from the current print position when the current line contains a display control command.

- The count for current print position includes every character transmitted, and does not assume that the printer responds to any positioning commands.

- A Carriage Return and Line Feed sequence precedes the string of spaces when the current print position is more that one column beyond the specified column number.

- TAB returns a null string when the current print position is one greater that the specified column number.

- See the Input and Output section for a discussion of the use of the PRINT statement through an opened channel to an external serial printer port.

- See also the IEEE-488 Bus Input and Output section for a discussion of the use of the PRINT statement through the instrumentation ports.

- See the Touch Sensitive Display section for techniques of display cursor control.

3-33. The following examples illustrate the results of uses of the TAB function with the display. The display cursor position is initially at column 1.

STATEMENT	RESULT
`PRINT TAB(5), "HELLO"`	Advance 5 spaces (column 6) and display "HELLO".
`PRINT "HELLO"; TAB(3); "THERE"`	Display "HELLO", move the cursor down one line, and display "THERE" without leading spaces. Column 4 (TAB (3) + 1) is left of the cursor after displaying "HELLO".
`PRINT CPOS(6, 3); TAB(5); "HELLO"`	Display "HELLO" at the start of line 7. Column 6 (TAB (5) +1) is left of the current print position since CPOS (6<3) is an 8-character string, bringing the current print position to 9, even though the display cursor moves to row 6, column 3.
`PRINT CPOS(6, 3); TAB(13); "HELLO"`	Display "HELLO" at column 8 on line 6. Since CPOS (6,3) is an 8-character string, the current position is 9 when the TAB to column 14 (13 + 1) is calculated. TAB returns 5 spaces (14 - 9), moving the display cursor from its actual column 3 position to column 8.

3-34. The following example displays the numbers 100 through 154 in 10 columns that are each 8 spaces wide. The column number index is added to the bottom for reference. The extra space to the left of each column is for the sign (blank = positive). The PRINT statement in line 50 advances the cursor to the next line. Refer to the section General Purpose Fluke Basic Statements for a discussion of FOR-NEXT program loops.

```
10    FOR I = 100 TO 150 STEP 5
20    FOR K = 0 TO 4
30    PRINT TAB(8 * K); I+K;
40    NEXT K
50    PRINT
60    NEXT I
```

Results:

```
100     101     102     103     104
105     106     107     108     109
110     111     112     113     114
115     116     117     118     119
120     121     122     123     124
125     126     127     128     129
130     131     132     133     134
135     136     137     138     139
140     141     142     143     144
145     146     147     148     149
150     151     152     153     154
```

3-35. The SPACE$ Function

Format: SPACE$ (number of spaces)

3-36. The SPACE$ function returns a string of spaces as specified.

- SPACE$ has as argument an integer, and yields a string of spaces.
- SPACE$ is useful for formatting display and print outputs.

3-37. The following example illustrates results of using SPACE$ to format a display.

```
10    N% = 5%
100   Y$ = SPACE$ (N%)            ! Y$ equals "....." (5 spaces)
400   A$ = "This is the first example"
450   B$ = "STRING"
500   PRINT A$; SPACE$ (N%); B$
```

Display results:

```
This is the first example:  STRING
```

3-38. The NUM$ Function

Two Formats: NUM$ (integer or floating point number)
NUM$ (integer or floating point number, string)

3-39. The NUM$ function returns a string of characters in the format that a PRINT or PRINT USING statement would output the given number.

- NUM$ has as arguments an integer or a floating point number, and an optional character string, and yields a numeric character string with appropriate spaces and decimal.
- When input is limited to an integer or floating point number, NUM$ returns a character string in the format that PRINT would output the number. For example, if the input number is 1.00000, NUM$ returns " 1 ". (Note leading and trailing space.)
- When input includes a comma followed by a character string, NUM$ returns a character string in the same format that PRINT USING would output the number using the specified string. For example, if the input is 5.5, with the format string "#.## ", NUM$ returns "5.50E+01".

- Refer to the discussion of PRINT USING in the General Purpose Fluke BASIC Statements section for proper formatting of the input string, and the resulting output string.

3-40. The following examples illustrate the results of using the NUM$ function.

STATEMENT	RESULT
`Y$ = NUM$ (10.5)`	Y$ is assigned the string "10.5 ".
`Y$ = NUM$ (X * A)`	Same as above, except that an expression is used.
`Z$ = NUM$ (10.5, "##.##")`	Z$ is assigned the string "10.50".
`Z$ = NUM$ (X, S$)`	Same as above, except that numeric and string variables are used.

3-41. The VAL Function

Format: VAL (string)

3-42. The VAL function returns the floating point numeric value of a numeric string.

- VAL has a string as an argument and yields a floating point number.
- The string argument to VAL should be a legal floating point number.
- String input consisting only of spaces, or a null (empty) string, returns the value 0.
- Spaces, and Nulls may precede the input numeric string.
- Spaces, Nulls, Carriage Returns, and Line Feeds may follow the input numeric string.
- Error 803 results when the input string contains non-numeric elements.

3-43. The following examples illustrate results of common uses of VAL.

STATEMENT	RESULT
`Y = VAL (A$)`	The value of the numeric string A$ is assigned to the floating point variable Y. For example, if A$ contains "1000", Y is assigned the value 1000.
`Z = VAL (" ")`	Assign Z the value 0. (The string consists only of spaces).

3-44. MATHEMATICAL FUNCTIONS

3-45. Mathematical functions perform calculations on numeric quantities.

- Mathematical functions operate on integer or floating point numbers.

- Mathematical functions accept as input an expression that evaluates to a numeric quantity.

- Mathematical functions return an integer or floating point number.

- Conversion between numeric data types is automatic where required by the function or specified by the user. This conversion process requires additional processing time.

- The domain of acceptable input values for some functions is limited or not continuous.

- The range of resulting output values for some functions is not continuous or has points of underflow (too close to zero) or overflow (too large).

- The definition of each function includes limitations of domain and range.

3-46. The discussion that follows divides mathematical functions into two categories: general-purpose functions and trigonometric functions.

3-47. General Purpose Mathematical Functions

3-48. The general purpose mathematical functions supplied with Fluke BASIC include an assortment of tools to simplify computation tasks on collected data. Included are a square root function, natural and common logarithms, exponentiation of e, absolute value, sign, and greatest integer.

3-49. The SQR Function

Format: SQR (numeric expression)

3-50. The square root function returns a floating point number equal to the square root of the input value.

- SQR has a floating point number as an argument, and yields a floating point.

- The square root is the value which, if multiplied by itself, produces the given input value.

- The domain of input values is positive numbers and zero. Error 604 results when a negative number is used as input.

- The range of output values reaches underflow (error 602) for input values between 1E-154 and zero.

3-51. The LOG Function

Format: LOG (numeric expression)

3-52. The LOG function returns a floating point number equal to the common logarithm of the input value.

- LOG has a floating point number as an argument , and yields a floating point number.

- The common logarithm is the exponential power that, if applied to the number 10, produces the given input value.

- The domain of input values is positive numbers. Error 606 results when zero or a negative number is used as input.

- The range of output values is -307 to +308.

3-53. The LN Function

Format: LN (numeric expression)

3-55. The LN function returns a floating point number equal to the natural logarithm of the input value.

- LN has a floating point number as an argument, and yields a floating point number.

- The natural logarithm is the exponential power that, if applied to the number e, produces the given input value.

- The value used by the 1720A for e is 2.71828182845905

- The domain of input values is positive numbers. Error 606 results when zero or a negative number is used as input.

- The range of output values is between -707 and +710.

3-56. The EXP Function

Format: EXP (numeric expression)

3-57. The EXP function returns a floating point number equal to the result of raising the number e to an exponential power equal to the given input value.

- EXP has a floating point number as an argument, and yields a floating point number.

- The exponential function is the result of raising the number e to an exponential power equal to the given input value.

- The value used by the 1720A for e is 2.71828182845905

- The domain of input values is negative numbers, zero, and positive numbers up to and including 174. Error 605 results when an input of 175 or greater is used.

- Input values of -175 or less produce a result of 0.

- The range of output values is zero and the positive numbers between E-76 and E+76.

3-58. The ABS Function

Format: ABS (numeric expression)

3-59. The ABS function returns the absolute value of a floating point or integer number.

- ABS has a floating point number or integer, as an argument, and yields a positive floating point number or integer.

- ABS changes a negative value to positive, but has no effect on positive values or zero.

- The domain of input values is any positive or negative floating point or integer value, or zero.

- The range of output values is positive floating point or integer values, and zero.

3-60. The SGN Function

Format: SGN (numeric expression)

3-61. The SGN function returns the sign of a floating point or integer number. This is called the signum function, to distinguish it from SIN.

- SGN has a floating point or integer number as an argument, and yields one of three integer: 1, 0, or -1.

- A result of 1 indicates a positive number.

- A result of 0 indicates an input of 0.

- A result of -1 indicates a negative number.

- The domain of input values is any positive or negative floating point or integer value, or zero.

- The range of output values is the integers 1, 0, and -1.

3-62. The INT Function

Format: INT (numeric expression)

3-63. The INT function returns the greatest integer less than or equal to a given floating point number.

- INT has as input a floating point number, and yields a floating point number.

- The domain of input values is any positive or negative floating point value, or zero.

- The range of output values is positive and negative floating point values, and zero.

- If INT is given an integer argument, e.g., INT (A%), INT will return the integer argument unchanged.

3-64. Trigonometric Functions

3-65. Fluke BASIC also includes four trigonometric functions: sine, cosine, tangent, and arctangent. These functions have some discontinuities and limits that can produce unexpected errors when used improperly. The discussions that follow define these factors.

3-66. Radian Angular Measure

3-67. Trigonometric functions in Fluke BASIC use radian measure for angular quantities. This is a simple concept that fits trigonometric calculations better than the use of degrees.

- The value PI stored by Fluke BASIC (3.14159265358979) represents an approximation of the ratio between the circumference and the diameter of any circle.

- Radian measure defines the angular distance around a circle as 2*PI radians. This is the equivalent of 360 degrees.

- Angles are thus easily defined as fractional parts of PI. For example:

  PI radians = 180 degrees
  PI / 2 radians = 90 degrees

- To convert from degrees to radians, multiply by (PI / 180).

- To convert from radians to degrees, multiply by (180 / PI).

- The descriptions of trigonometric functions that follow use radian measure exclusively.

3-68. The SIN Function

Format: SIN (angle in radians)

3-69. The SIN function returns the sine of an angle that is expressed in radians.

- SIN has a floating point number as an argument, and yields a floating point number.

- The range of the input values is between and including the limits of ± 32767 radians. Error 607 results from input values outside these bounds.

- The range of output values is between and including the limits of ± 1.

- Input values within approximately 1E-16 of integer multiples of PI give a result of zero, rather than underflow error.

3-70. The COS Function

Format: COS (angle in radians)

3-71. The COS function returns the cosine of an angle that is expressed in radians.

- COS has a floating point number as an argument, and yields a floating point number.

- The range of input values is between and including the limits of ± 32767 radians. Error 607 results from input values outside these bounds.

- The range of output values is between and including the limits of ± 1.

- Input values within approximately 1E-16 of integer multiples of PI/2 give a result of zero, rather than underflow error.

3-72. The TAN Function

Format: TAN (angle in radians)

3-73. The TAN function returns the tangent of an angle that is expressed in radians.

- TAN has as arguments a floating point number, and yields a floating point number.

- The range of input values is between and including the limits of ± 32767 radians. Error 607 results from input values outside these bounds.

- Input values within approximately 1E-16 of integer multiples of PI/2 result in error 603. The tangent function has infinite discontinuities at these points.

3-74. The ATN Function

Format: ATN (numeric expression)

3-75. The ATN function returns the arctangent in radians of a floating point numeric input value.

- ATN has a floating point number as an argument, and yields a floating point number.

- The arctangent is the inverse of the tangent. ATN returns an angle, expressed in radians, whose tangent value is the given input.

- The range of input values is any floating point number.

- The range of output values between and including the limits of ± PI/2.

- Input values within approximately 1E-16 of 0, result in an output of 0. Underflow error does not occur.

Section 4
General-Purpose Fluke Basic Statements

4-1. INTRODUCTION

4-2. This section describes the General-Purpose Fluke BASIC statements needed to write a running BASIC program. Subsequent sections present expanded capabilities of these statements, as well as more specialized statements.

4-3. OVERVIEW

4-4. The ANSI standard BASIC programming language has been adapted for instrumentation control by Fluke. This enables the instrument system designer to customize a system without the need for extensive programming support. The Fluke BASIC statements presented in this section are the general purpose statements with which the user will be most familiar. They are arranged in alphabetical order. Each statement is presented with a syntax diagram, a complete definition, and examples. The following comments apply to all statements:

- A line number between 1 and 32767 must precede each statement if it is to be stored for later use in a running program.

- If line length, exclusive of the line number, is kept at 74 characters or less, any line can be renumbered to any acceptable number without exceeding the maximum 79 character length.

- In general, Fluke BASIC corresponds to ANSI standard minimal BASIC for the statements that are not enhancements for instrumentation control. Exceptions to this are given in the Introduction section.

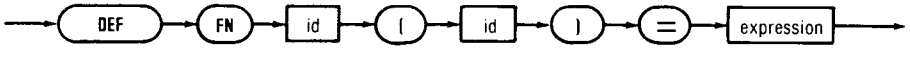
- Except where specifically stated otherwise, all Fluke BASIC statements can be executed in immediate mode without a line number.

- The \ character must be used to separate statements in multiple statement lines.

4-5. STATEMENT DEFINITIONS

4-6. The DEF FN Statement

DEFine



4-7. The DEF FN statement allows functions to be defined for subsequent use in expressions.

- The function name can be any variable name, such as A%, FA$, Z, including variables previously used in the program. (e.g., FNA% does not conflict with A% already existing in the program.)
- The argument name must be a simple variable name (i.e., not subscripted).
- The argument name is local to the definition. This means that the same variable name used elsewhere in the program does not affect and is not affected by the definition of the function or by the function's execution.
- An argument name must be included in the function definition even if it is not used.
- The argument name is enclosed in parentheses following the name of the function.
- Other references in this manual: None

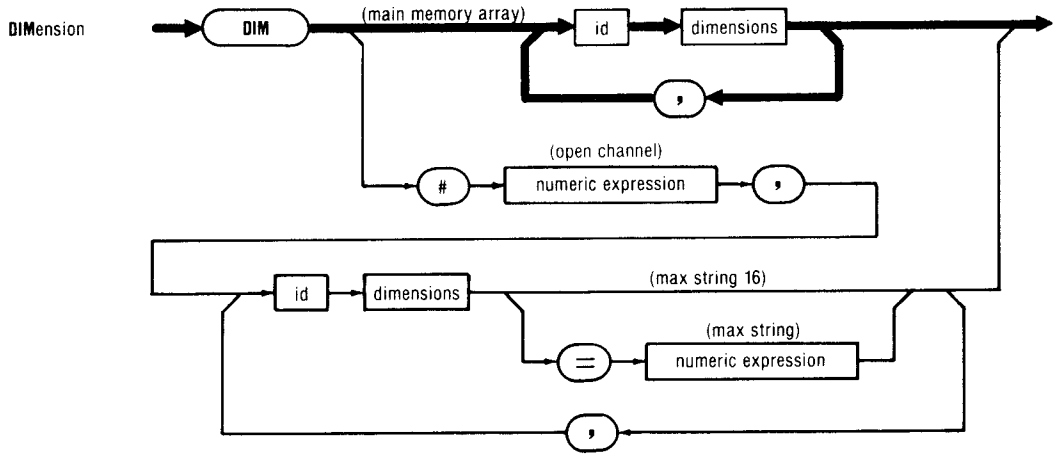
4-8. The following example defines user function A to be three times the argument passed to the function, designated as X. After the function has been defined it may be used anywhere in the program just as if it were a system function, i.e., as a part of an expression.

```
10 DEF FNA(X) = 3 * X          ! Defines function as 3 * X
20 X = 5                       ! Assigns 5 to X
30 PRINT FNA(7)                ! Displays 21 (3 times 7),
40 REM                         ! not 15 (3 times X)
```

4-9. The following example illustrates the use of a system function as part of the definition of a user function. In this example, user function B will return the arcsine of the input value (e.g., if the next statement were Y=FNB(.5) then Y would be given the value PI/6 or 0.5235988.)

```
100 DEF FNB(C) = ATN (C / SQR (1 - C * C))
```

4-10. The DIM Statement



4-11. The DIM statement reserves memory or file space for arrays.

- An array is a sequence of variables having the same name.
- Each variable is an "element" of the array.
- The maximum array index for each dimension may be specified as one less than the required number of elements, as element counting begins with zero.
- One or two dimensions may be specified.
- A two dimensional array may be considered as a matrix with the first dimension representing rows and the second dimension representing columns.
- A previously opened channel must be referenced to open a virtual array.
- A single DIM statement may dimension one or more arrays, separated by commas.
- Other references in this manual: Virtual Arrays.

4-12. The following examples illustrate some common uses of the DIM statement, with comments on the results of each statement.

STATEMENT	RESULT
`DIM A(5)`	Dimensions a six-element one-dimensional floating point array with the following elements: A(0) A(1) A(2) A(3) A(4) A(5)
`DIM A%(2, 3)`	Dimensions a twelve-element two-dimensional integer array with 3 rows (0-2) and 4 columns (0-3): A%(0,0) A%(0,1) A%(0,2) A%(0,3) A%(1,0) A%(1,1) A%(1,2) A%(1,3) A%(2,0) A%(2,1) A%(2,2) A%(2,3)
`DIM OP$(5)`	Dimensions a six-element string array that could be used to store operator messages.
`DIM A(5), A%(2,3), OP$(5)`	This statement accomplishes the tasks of the three previous examples in one statement.

4-13. The END Statement

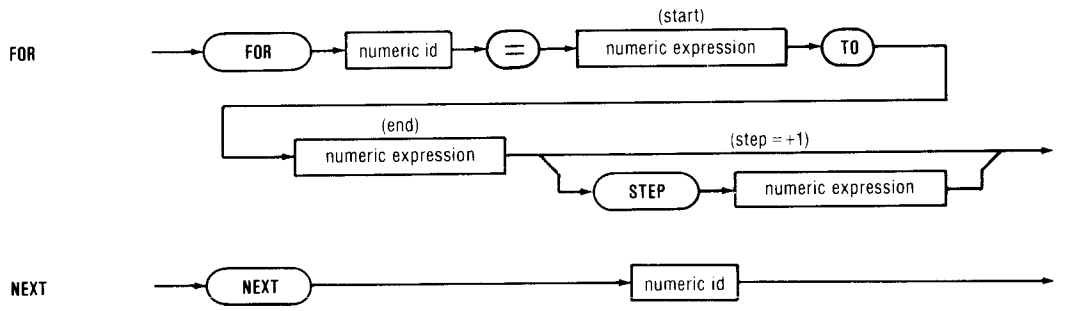
4-14. The END statement may be used to indicate the absolute end, or the end of the main body of a program.

- Subroutine code may follow the END statement.
- Code following END will not be executed unless it has been accessed by a GOSUB or GOTO (or implied GOTO via IF...THEN with a line number).
- END is optional.
- Other references in this manual: None

4-15. The suggested use of this statement is to designate the physical end of your program:

```
32767 END
```

4-16. The FOR and NEXT Statements



4-17. The FOR statement sets up a loop which repeatedly executes the statements contained between the FOR statement and the NEXT statement.

- There must not be a FOR without a NEXT, or a NEXT without a FOR.
- The FOR statement specifies the number of times the loop is to be repeated by specifying limit points and step increment of the index.
- If no step increment is specified, step +1 is assumed.
- The index must be numeric.
- The index must not be an array element.
- When a FOR statement is initially encountered, the index value is compared to the limit. If the index is past the limit, the FOR-NEXT loop is not executed.
- The NEXT statement compares the index with the limit after incrementing it.
- Other references in this manual: None

4-18. The following examples compare two different ways of constructing program loops, using the FOR - NEXT, and using IF - GOTO. Note the different comparisons at line 100, depending on the sign of the STEP.

FOR - NEXT

```
10  FOR I = 1 TO -5 STEP -2

20  ! Other statements
30  !
100 NEXT I
110 ! Other statements
```

IF - GOTO

```
10  I = 1
15  IF I < (-5) GOTO 110
20  ! Other statements
30  !
100 I=I + (-2) \ GOTO 15
110 ! Other statements
```

```
 10  FOR J = 10 TO 100 STEP 20

 20  ! Other statements
 30  !
100 NEXT J
110 ! Other statements
```

```
 10  J = 10
 15  IF J > 100 GOTO 110
 20  ! Other statements
 30  !
100 J=J + 20 \ GOTO 15
110 ! Other statements
```

4-19. The following example loops five times, and then prints the index value. Note that the index has incremented to six.

```
10  FOR I%=1% TO 5%                    ! Increments by 1 through loop
20     PRINT I%                        ! Display value of I%
30  NEXT I%                            ! Repeat loop fill F=5%
40  PRINT "INDEX VALUE IS";I%
```

The display will show:

```
1
2
3
4
5
Index value is 6
```

4-20. The following example illustrates what happens when the index never equals the final value.

```
10  FOR A% = C% TO D% STEP 2%          ! C% = 5 and D% = 10
20    PRINT A%                         ! Display value of A%
30  NEXT A%                            ! Increments A% by 2
40  REM                                ! Loops until A% = 10
```

The display will show:

```
5
7
9
```

NOTE

If line 10 used C and D rather than C% and D%, and if C = .6 and D = 10.6, the display would show:

```
1
3
5
7
9
```

because real values are rounded before assignment to an integer variable.

4-21. The following example illustrates the nesting of two FOR - NEXT loops. Note that one blank line occurs between first and second display lines. This happens because the last value displayed in the line above did not have 16 columns available, causing a carriage return-line feed in the display (See PRINT command).

```
10  FOR I% = 1% TO 4%
20      FOR J% = 1% TO 5% STEP I%
30        PRINT I%; J%,                ! Print values and then tab
40        REM                          ! to next column
50      NEXT J%
60      PRINT                          ! Move to the next line
70  NEXT I%
```

The display will show:

```
1  1          1  2          1  3          1  4          1  5

2  1          2  3          2  5
3  1          3  4
4  1          4  5
```

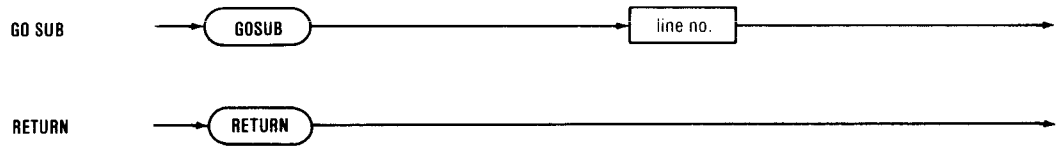
4-22. The following example decrements the index by -.1.

```
10  FOR I = 1 TO 0.5 STEP -.1
20        PRINT I;
30  NEXT I
```

The display will show:

```
1 0.9   0.8   0.7   0.6   0.5
```

4-23. The GOSUB and RETURN Statements



4-24. The GOSUB statement transfers control to the beginning of a subroutine. The RETURN statement terminates the subroutine and returns control to the statement following the GOSUB.

- BASIC allows nested subroutines.
- Within a subroutine, a GOSUB may call another or the same subroutine.
- A GOTO within a subroutine should not pass control permanently to a program segment outside the context of the subroutine.
- Any valid BASIC statements may be used in the body of a subroutine. Available memory is the only limit to the length of subroutines.
- Other references in this manual: None

4-25. In the following example the subroutine displays the operator's options and obtains his or her choices. The subroutine is called at line 180. The display subroutine calls another subroutine (line 1300) to obtain identification of the selected choice (line 1080). When the RETURN statement at line 1390 is executed, control is transferred to line 1090. Three subroutine options are illustrated, each with different interpretations for GOSUB and RETURN.

4-26. The following points should be noted about this example:

- GOTO 31000 causes termination of the test without returning from the subroutine. This normally does not represent good program structure.
- ELSE 1500 causes control to be transferred to lines 1500 - 1590 which can be considered an extended part of subroutine 1000. The RETURN at line 1590 will cause line 190 to be executed next.

- RETURN at line 1090 will cause immediate termination of the subroutine 1000. Line 190 is executed next.

```
100    ! Other statements
180    GOSUB 1000                          ! Get operator choice
190    ! Other statements
999    STOP
1000   REM -- SUBROUTINE -- Display and Execute Operator Options
1010   PRINT "Enter choice by pressing display"
1020   PRINT\PRINT                         ! Skip 2 lines
1030   PRINT "  [Continue test]"           ! KR% = 1 or 2
1040   PRINT\PRINT\PRINT                   ! Skip 3 lines
1050   PRINT "  [Adjust Instrument]"       ! KR% = 3 or 4
1060   PRINT\PRINT\PRINT                   ! Skip 3 lines
1070   PRINT "  [Terminate Test]"          ! KR% = 5 or 6
1080   GOSUB 1300                          ! Get choice, KR%
1090   IF KR%>2% THEN IF KR%>4% GOTO 31000 ELSE 1500 ELSE RETURN
1300   REM -- SUBROUTINE -- Get response from Keyboard
1310   ! On exit KR% will be the row touched (from 1 to 6)
1320   ! KC% will be the column touched (from 1 to 10)
1330   ! Other statements
1390   RETURN
1500   REM -- SUBROUTINE -- Adjust Instruments
1510   ! Other statements
1590   RETURN
31000  REM -- Termination Routine
31010  ! Place instruments in standby state and save test results
31020  ! Other statements
32767  END
```

4-27. The GOTO Statement

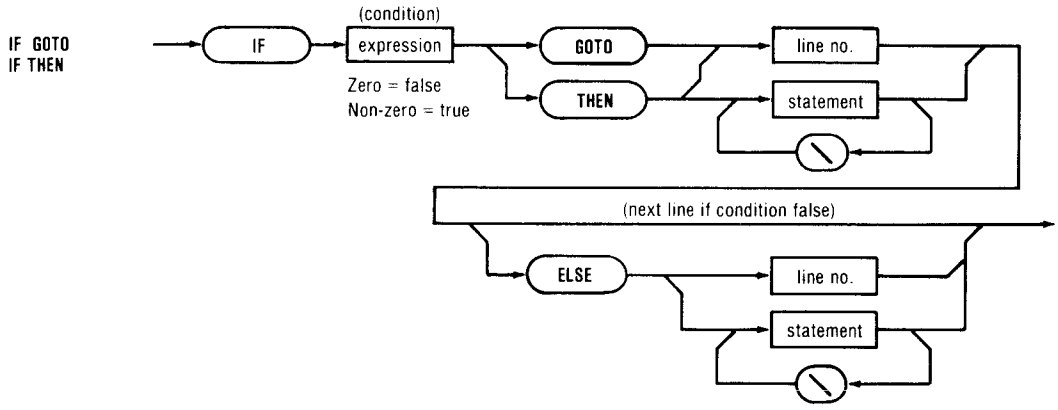
4-28. The GOTO statement causes a program to unconditionally branch to the specified line number.

- GOTO must not be used to branch out of a FOR-NEXT loop.
- Other references in this manual: None

4-29. The following program example illustrates one use of the GOTO statement. The GOTO statement at line 80 causes line 30 to be executed next:

```
10   REM -- Main Program Routine
20   REM
30   GOSUB 4000                    ! Set-up Sequences
40   GOSUB 5000                    ! Test #1
50   GOSUB 6000                    ! Test #2
60   GOSUB 7000                    ! Test #3
70   GOSUB 8000                    ! Test #4
80   GOTO 30                       ! Start Again
```

4-30. The IF-GOTO, IF-THEN and IF-THEN-ELSE Statements



4-31. The IF statement evaluates a condition represented by an expression. The resulting course of action depends on the result of that evaluation, and the structure of the statement.

NOTE

Relational expressions such as A=B, A%<3%, A$>=B$ evaluate as -1 if true and 0 if false.

- The conditional expression must result in a value representable as an integer.
- A non-zero result is TRUE. A zero result is FALSE.
- Control passes to the line number following GOTO, or to the statement or line number following THEN, if the result is TRUE.
- Control passes to the next program line if the result is FALSE and ELSE is not specified.
- Control passes to the statement or line number following ELSE if the result is FALSE and ELSE is specified.

- A line number must follow GOTO, if it is used.
- A line number or a valid BASIC statement must follow THEN, if it is used.
- Multiple statements, separated by the \ character, may be used after THEN, and after ELSE. Each statement will be done in sequence only if control is passed to that portion of the IF statement as defined above.
- Other references in this manual: None

4-32. The following examples illustrate the results of various uses of the IF statement:

STATEMENT	RESULTS
`IF A THEN 100`	If A is non-zero, go to line 100. If A is zero go to the next line.
`IF A>B THEN A=B`	If A is greater than B, set A equal to B. Then go to the next line.
`IF NOT A% GOTO 500`	If A% equals -1 (logical true), ignore this statement and go to the next line. Otherwise, go to line 500.
`IF A%+B% GOTO 500`	If A%+B% is non-zero (A% not equal to -B%), go to line 500. Otherwise go to the next line.
`IF A%+B% THEN A=B*5\B=10`	If A%+B% is non-zero (A% not equal to -B%), assign B 5 to A, and assign 10 to B. Then go to the next line.
`IF A% OR B% GOTO 500`	If either A% or B% is non-zero, go to line 500. Otherwise go to the next line.
`IF A+B=C THEN PRINT C`	If A+B equals C, display value of C. Then go to the next line.
`IF (1<A) AND (A<6) THEN 450`	If the value of A lies within the open interval (1,6) transfer control to line 450.

4-33. The following program example shows some common uses of the IF statement. The relational expression in line 140 uses the AND operator to ensure that both conditions are true before transferring control to line 5000.

```
100  REM -- Determine Test to Run
110  PRINT "ENTER UNIT SERIAL NUMBER"; !Print prompt
120  INPUT SN%
130  IF SN% > 12563% THEN 110          !Equivalent to IF...GOTO 110
140  IF (11000% <= SN%) AND (SN% < 11500%) GOTO 5000
150  REM -- Run Standard Text
160  ! Other statements
5000 REM -- Run Test for 'Special'
5010 ! Other statements
```

4-34. The following example shows how the IF statement acts on various values. Examining the results will aid in understanding the IF statement. Note that line 90 transfers control back to line 30 except when A% is 0.

```
10   REM -- Illustration of IF with integers
20   PRINT "INTEGER VALUE","ONE'S COMPL.","ODD","DIVISIBLE BY FOUR"
30   INPUT A%
40   PRINT A%, NOT A%,                    !Print values and tab
50   IF A% AND 1% THEN PRINT 'YES;        !Is A% odd?
60   PRINT,                               !Go to next tab on display
70   IF A% AND 3% THEN PRINT; \ GOTO 90!Is A% divisible
80   PRINT 'YES';                         !by four?
90   PRINT                                !Line Feed
100  IF A% THEN 30
110  END
```

Results:

```
INTEGER VALUE  ONE'S COMPL.    ODD            DIVISIBLE BY FOUR

1              -2              YES

4              -5                             YES

1568           -1569                          YES

9              -10             YES

-568            567                           YES

-1              0              YES

-4              3                             YES

0              -1                             YES
```

4-35. The following example includes the ELSE option. When A is less than 5, B will be set to FNS(A) and the statements following line 200 will be executed until the STOP statement on line 290 is encountered. When A is greater than or equal to 5, B will be set to FNT(A) and the statements following Line 300 will be executed.

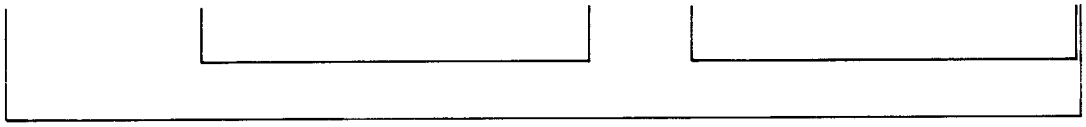
```
10   ! Other statements

100  IF A < 5 THEN 200 ELSE 300

200  B = FNS(A)                        ! Use 'S' function

210  ! Other statements

290  STOP

300  B = FNT(A)                        ! Use 'T' function

310  ! Other statements
```

4-36. The following example shows the ELSE option followed directly by statements. When A is less than B, J is set to B raised to the power A, and T is set to 0. When A is greater than or equal to B, J is set to the value of A raised to the power B, and T is set to 1.

```
IF A < B THEN J = B^A \ T = 0 ELSE J = A^B \ T = 1
```

4-37. The following example illustrates nested IF-THEN-ELSE statements. The ELSE is always associated with the closest IF and THEN statement to the left that is not already associated with another ELSE, as indicated by the diagram below the statement. (Note: This statement computes M = minimum of A, B, or C.

```
IF A<B THEN IF A<C THEN M=A ELSE M=C ELSE IF B<C THEN M=B ELSE M=C
```



The logic is as follows:

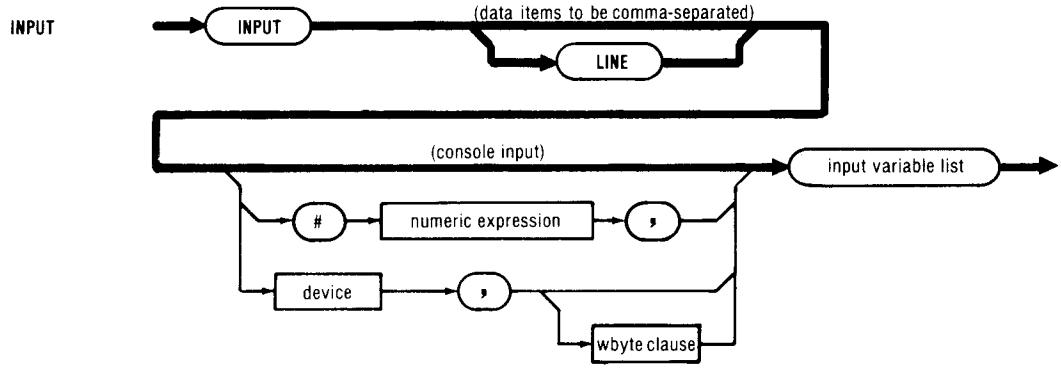
If A is less than B, and then if A is less than C, A is the minimum (M).

If A is less than B, and then if A is greater than or equal to C, C is the minimum (M).

If A is greater than or equal to B, and then if B is less than C, B is the minimum (M).

If A is greater than or equal to B, and then if B is greater than or equal to C, C is the minimum (M).

4-38. The INPUT Statement



4-39. The INPUT statement assigns inputs from external devices to variables. This discussion describes keyboard inputs only.

- One or more variables separated by commas must follow INPUT.

- INPUT assigns keyboard entries, separated by commas, to those variables.

- The variable type must match the entry type. String or floating point data cannot be assigned (input) to an integer variable.

- Assignment of values to corresponding variable names is made sequentially.

- When an entry matches the variable type, it is then assigned.

- If, after the comma, there is another variable to be assigned, another check is made of variable type.

- This sequence is repeated in a multiple input list until the end of the variable list has been reached. This should correspond with the termination of the input by the carriage return-line feed sequence.

- If an error occurs during assignment, a warning message will be displayed on the screen and a new prompt character, "? ", will be given. All variables must then be entered again.

- Strings may be entered in quoted or unquoted form.

- An unquoted string may not contain a comma or begin with a quote.

- Leading spaces in an unquoted string will be ignored.

- A string must be enclosed with quotes to contain commas or leading spaces.

- Internal quotes must be different from enclosing quotes.

- INPUT displays a ? character, followed by a space, and halts the program until RETURN is pressed. Keyboard entries must precede the RETURN.

- If RETURN is pressed without entering anything else, numeric variables are assigned 0 and strings are assigned the null string.

- Other references in this manual: Input and Output, IEEE-488 BUS Input and Output.

4-40. The following examples illustrate common uses of INPUT.

`INPUT A$`	Displays ?. Assigns the entry to A$.
`INPUT B%, A$`	Displays ?. Assigns the entries (e.g., 1776, GEORGE) as follows: B% = 1776 and A$ = "GEORGE".

4-41. In the following example the program will halt until the RETURN key is pressed. It then will display your entry and request another.

```
10   REM -- Demonstrate characteristics of string input

20   ! Error 801 is 'Too much data entered'

30   ! Error 803 is 'Illegal character in input'

40   !

100  PRINT 'INPUT A STRING?';          ! Get the string

110  INPUT A$

120  PRINT 'THE STRING IS: '; A$

130  PRINT \ PRINT                     !Skip two lines

140  GOTO 100
```

Running this program gives results as follows:

```
INPUT A STRING? EXAMPLE STRING ONE
THE STRING IS: EXAMPLE STRING ONE


INPUT A STRING? Inserting a, gives an error


?Input error 801 at line 110
? Placing the ',' in quotes won't help


?Input error 801 at line 110
? 'So place the entire string with a , in quotes'
THE STRING IS: So place the entire string with a , in quotes



INPUT A STRING? " Will also give an error


?Input error 803 at line 110
? 'The quotes must match'
THE STRING IS: The quotes must match


INPUT A STRING?
```

4-42. The example program below requires three inputs, integer, floating point, and string, separated by commas.

```
500  REM -- Enter test parameters
```

```
510  PRINT "ENTER TEST NUMBER, NOMINAL VALUE, UNITS";

520  INPUT T%, NV, U$

530  IF T% GOTO 510                ! Equivalent to IF T%<>0 GOTO 510
```

Running this program gives results as follows:

```
ENTER TEST NUMBER, NOMINAL VALUE, UNITS? 101, 2.222, mV

ENTER TEST NUMBER, NOMINAL VALUE, UNITS? 104, 8.888, mV

ENTER TEST NUMBER, NOMINAL VALUE, UNITS? 201, 1,V

ENTER TEST NUMBER, NOMINAL VALUE, UNITS? 203, 100,V

ENTER TEST NUMBER, NOMINAL VALUE, UNITS? 301, 100, mA

ENTER TEST NUMBER, NOMINAL VALUE, UNITS? 302, 1,A
```

4-43. The KILL Statement

4-44. The KILL statement deletes a specified file.

- A file name must be specified, enclosed in single or double quotes.

- The default file name extension is ". " (3 spaces). Any other file name extension must be specifically stated.

- Error 305 results if an attempt is made to kill a file that is not found.

- The file is deleted from the default System Device if the file name is not preceded by MF0: for the floppy disk, or ED0: for the optional electronic disk. Refer to the Input and Output Statements section for a discussion of the default System Device concept.

- Other references in this manual: None.

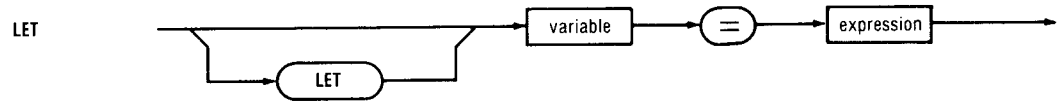
4-45. The following example illustrates the KILL statement used to delete FILE.DAT from the directory of the System Device.

```
KILL "FILE.DAT"
```

4-46. This next example has a file of four blocks (2048 characters) to save intermediate status information during the program. This "check pointing" file may be used to recover from power failures, etc. When the program terminates, the file is no longer needed so it is deleted at line 32766.

```
10        REM -- Meter Calibration Program
20        OPEN "CHECK.PT" AS NEW FILE 2 SIZE 4! Check pointing file
          .
          .
          .
32766     KILL "CHECK.PT"                  ! Delete check pt file
32767     END
```

4-47. The LET Statement



4-48. The LET statement evaluates an expression and assigns the results to a specified variable.

- The data types of the expression and the variable must be either numeric or string, not mixed.
- A numeric expression will be converted if necessary to integer or floating point, as defined by the variable.
- The word LET is optional.
- Other references in this manual: None

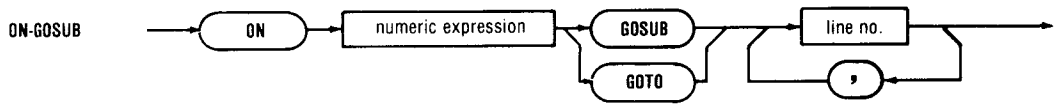
4-49. The following examples show the results of assigning values of expressions of one type to variables of another type:

STATEMENT	RESULT
`LET A%=-3.2`	A% has value -3
`B%=-12.6`	B% has value -13
`LET A1%=4.8`	A1% has value 5
`X=12%+A1%`	X has value 17
`Y="12E3"`	Error 600, not change in Y
`S$="HELLO "+"THERE"`	S$ contains HELLO THERE
`LET SX$=12%`	Error 600, no change in SX$

NOTE

The absolute value of a floating point value is rounded to an integer, the proper sign is assigned, and the value is then assigned to the integer variable.

4-50. The ON-GOTO and ON-GOSUB Statements



4-51. The ON- statement is used in two contexts. The first is as a multiple branch statement. The second is to designate the starting line of an event driven interrupt routine. This section discusses multiple branch statements.

ON (expression) GOTO (list)
ON (expression) GOSUB (list)

- The expression must be numeric.
- The expression is evaluated and rounded to obtain an integer.
- The integer is used as an index to select a line number from the list contained in the statement.
- The range of the integer must be between 1 and the number of line numbers contained in the list.
- Other references in this manual: Interrupt Processing

4-52. The following example illustrates a common use of ON - GOTO:

STATEMENT	MEANING
`ON A GOTO 400, 500, 600`	Transfers control to line 400 if A = 1, to line 500 if A = 2, or to line 600 if A = 3.

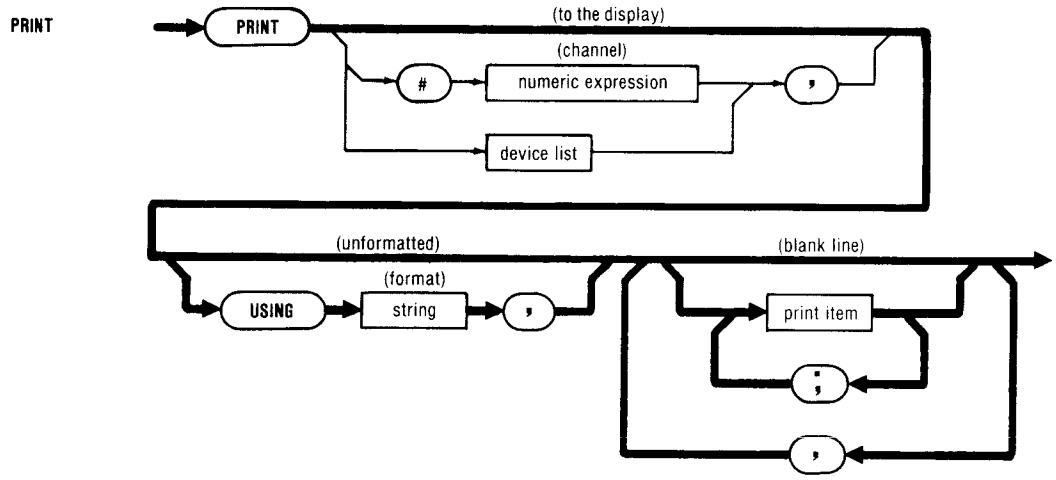
4-53. The following example illustrates one use of ON...GOTO at line 1030. Since the expression is normally rounded, INT(LOG(R)) is used instead of LOG(R) to ensure that the voltage divider is used for all values less than 1 volt (1000 millivolts).

```
1000 REM -- Connect Instrument to Test Station
1010 ! R on input is the value of the voltage to be applied
1020 ! R is in millivolts and is between 10 and 10^6
1030 ON INT(LOG (R)) GOTO 1200,1200,1300,1300,1400,1400
1200 REM -- Setup External Voltage Divider
1210 ! Other statements
1290 GOTO 1500
1300 REM -- Connect Instrument Directly
1310 ! Other statements
1390 GOTO 1500
1400 REM -- Give High Voltage Warning and Then Connect Instrument
1410 ! Other statements
1490 GOTO 1500
1500 REM -- Take Readings
1510 ! Other statements
```

4-54. The following example is equivalent to the previous example, except for the use of ON...GOSUB. Note that the code at line 1500 in the previous example follows line 1030 in this example. Lines 1290, 1390, 1490 are changed to RETURN statements.

```
1000 REM -- Connect Instrument to Test Station
1010 ! R on input is the value of the voltage to be applied
1020 ! R is in millivolts and is between 10 and 10^6
1030 ON INT(LOG (R)) GOSUB 1200, 1200, 1300, 1300, 1400, 1400
1040 REM -- Take Readings
1050 ! Other statements
1190 STOP                                  ! End of Program
1200 REM -- Setup External Voltage Divider
1210 ! Other statements
1290 RETURN
1300 REM -- Connect Instrument Directly
1310 ! Other statements
1390 RETURN
1400 REM -- Give High Voltage Warning and Then Connect Instrument
1410 ! Other statements
1490 RETURN
```

4-55. The PRINT Statement



4-56. The PRINT statement sends string, integer, or floating point data to an external device.

- PRINT sends data to the display unless otherwise specified in the statement.

- A single PRINT statement can send the results of more than one expression, including single or multiple constants or variables.

- Successive data items following PRINT must be separated by a comma or semicolon, to format the display.

- Unless otherwise specified, the data sent by a PRINT statement will be followed by a pair of ASCII control characters (carriage return and line feed).

- A comma or semicolon may be used to terminate a PRINT statement and suppress the Carriage Return. This will cause a succeeding PRINT statement to display data on the same line.

- Data items separated by commas are displayed in 16-character print fields with up to five print fields per line.

- If a data item is longer than one print field the next data item falls into the next empty print field even if the next print field is a succeeding 80 character line.

- A numeric data item is displayed with a leading space or sign, and atrailing space. It is printed out to seven significant digits. A value from .1 to 9999999 inclusive is printed out directly. A number less than .1 is printed out without E notation if all of its significant digits can be printed. All other values are printed in E notation (+0.dxxxxxxE+yyy), where d is a non-zero digit, x is any digit, and trailing zeros are dropped.

- Data items separated by semicolons are displayed side-by-side with no added spaces (other than those generated during numeric to decimal ASCII conversions).

- PRINT returns the cursor to the left-hand end of the next display line if the last data item is not followed by a comma or semicolon, or if no data items are specified.

- If the cursor is on the bottom line, PRINT scrolls the display upwards one line, unless the PRINT command is a cursor positioning command, or the display is in Page Mode.

- Other references in this manual: Input and Output, IEEE Input and Output.

4.58 The following examples illustrate some common uses of the PRINT statement:

STATEMENT	RESULTING DISPLAY	(^ = cursor)
`PRINT,,,`	(48 spaces)	^
`PRINT 4,5`	4 (15 spaces)	5 (1 spaces)
	^	
`PRINT 5;-6;`	5 -6 ^	
`A$="Hz"` `PRINT "K";A$;`	KHz^	

4-59. The following program example illustrates typical usage of PRINT statements to display readings taken from a voltmeter. The "," is used to format the displays in columns, and the ";" to print out the units together with the values. The comma at the end of line 1090 ensures that "** FAILED **" will be printed on the same line as test results that were found out of limits. Line 1120 ensures that when the test does not fail, any further data will not be displayed on the same line. The ";" at the end of lines 1100 and 1110 suppresses the carriage return and line feed after "** FAILED **" is displayed. Therefore, line 1120 has the same effect whether the test passes or fails.

```
1000 REM -- Display Readings In Volts and Display Limits
1010 ! R is value read by voltmeter
1020 ! LL is lower limit
1030 ! UL is upper limit
1040 ! Print ** FAILED ** if reading is outside tolerances
1050 !
1060 ! Heading:
1070 PRINT "LOWER LIMIT", "READING", "UPPER LIMIT"
1080 PRINT                              !Leave blank line
1090 PRINT LL; 'V', R; 'V', UL; 'V',
1100 IF LL > R THEN PRINT '** FAILED **';!Below tolerance
1110 IF UL < R THEN PRINT '** FAILED **';!Above tolerance
1120 PRINT                              !Next line
```

4-60. Assume LL has the value 2.9 and UL has the value 3.1. Then for various values of R, the display would appear as follows:

```
LOWER LIMIT     READING          UPPER LIMIT

2.9V            2.87V            3.1V            ** FAILED **

LOWER LIMIT     READING          UPPER LIMIT

2.9V            3.04V            3.1V

LOWER LIMIT     READING          UPPER LIMIT

2.9V            3.103V           3.1V            ** FAILED **
```

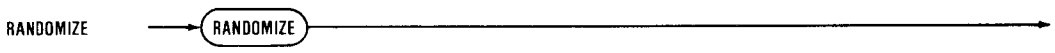
4-61. The following example shows PRINT as an Immediate Mode command. The result of the command is a single line of values as follows:

```
PRINT 2; EXP(2), 4; EXP(4), 6; EXP(6), 8; EXP(8)

2 7.389056 4 54.59815 6 403.4288 8 2980.958
```

NOTE

EXP(n) is the result of raising e (the natural log base) to the power n.

4-62. The RANDOMIZE Statement



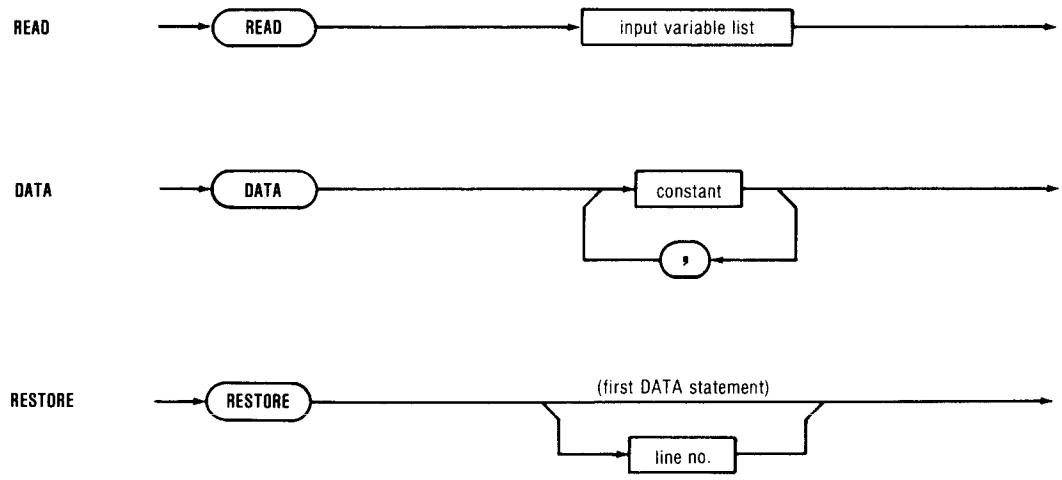
4-63. RANDOMIZE generates a random seed number for use with the RND function. The seed is generated from the system clock and has a value between 0 and 1.

- RANDOMIZE initializes a pseudo-random number for use by the RND function.
- Other references in this manual: None

4-64. Following is an example of the RANDOMIZE statement in a program:

```
10    RANDOMIZE
```

4-65. The READ, DATA, and RESTORE Statements



4-66. The READ, DATA, and RESTORE statements work together as follows.

- The DATA statement defines a sequence of data items to be processed by the READ statement. The data items within a single DATA statement are separated by commas.

- The READ statement assigns data values to a series of one or more variables.

- The READ statement assigns the next available data items in sequence, to the variables referenced.

- The DATA data types must match the corresponding READ variable types (strings for string variables, etc.).

- The DATA statement may occur before or after the READ statement.

- The DATA statement must be the last or only statement in the program line. The line may contain no trailing remarks. Everything following DATA is considered to be data.

- Legal data items are: quoted or unquoted strings or numeric constants.

- An unquoted string must not begin with a quote and must not contain commas.

- Leading spaces are ignored unless within a quoted string field.

- Numeric constants may not be quoted.

- A data pointer tracks which data items have been read.

- A DATA statement may contain more items than a subsequent READ statement contains variables.

- A second READ statement may continue reading data (assigning data items to its variables) at the point the first READ statement stopped reading.

- The RESTORE statement resets the pointer to the first data item of the first DATA statement in the program so that the items may be read again.

- If RESTORE specifies a line number, the pointer resets to the first data item of the first DATA statement in or after that program line.

- The RESTORE statement may be executed before all data items have been read from a DATA statement.

- It is not necessary to RESTORE the DATA items if they will be read only once in the program.

- Other references in this manual: None

4-67. This example illustrates the use of a FOR - NEXT loop to read a list of data items. Each item is a floating point number, so only one READ statement is necessary. To save each of the data items, use arrays and change line 110 to: READ A(I%). The RESTORE statement in this example would be useful only if the data values were to be used again in the program.

```
10   DATA 1.1, 2.2, 3.3, 4.4, 5.5
20   ! Other statements
100  FOR I%=1% TO 5%
110  READ A
120  PRINT A
130  NEXT I%
140  RESTORE
```

Running this program gives results as follows:

```
1.1
2.2
3.3
4.4
5.5
```

4-68. The following example illustrates multiple READ statements and a selective RESTORE statement. Note the double quotes used in the first DATA statement. The double quote is used to allow commas to be inserted in the string data. Note that if the quotes were omitted from line 10, A$ would be TEST FOR HIGH, and line 120 would give an error since the next data element would be LOW, which is not a number. Also note line 410 which resets the data pointer to allow reading the numeric values of line 20. The string data on line 10 is used only once.

```
10   DATA "TEST FOR HIGH, LOW, AND MEAN VALUES."
20   DATA 10, 7.5, 9
30   READ A$ \ PRINT A$                  ! Print out heading
100  REM -- Continuously Make Checks
110  ! Check for readings higher than upper limit
120  READ UL                             ! Get the upper limit
130  ! Other statements
210  ! Check for readings lower than lower limit
220  READ LL                             ! Get the lower limit
230  ! Other statements
```

```
310  ! Compute mean and compare to expected value
320  READ EV                              ! Get the expected value
330  ! Other statements
400  REM -- Reset instruments and Prepare for Next Test
410  RESTORE 20                           ! Reset data pointer to UL
420  ! Other statements
500  GOTO 110
```

4-69. The program segment that follows causes error 804 (bad DATA format) when statement 110 is executed, since the "!" character is not a legal part of a floating point number.

```
100  DATA 1,3,4                          ! Configuration data

110  READ A,B,C
```

4-70. The REM Statement

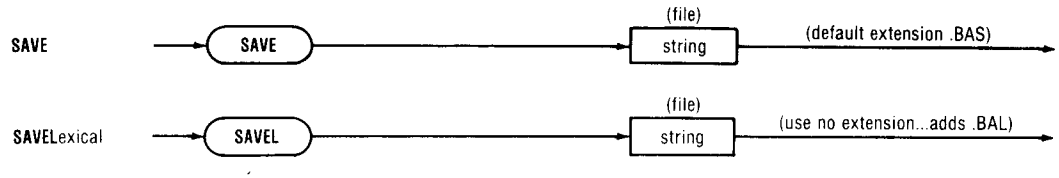
4-71. The REM statement, or the ! character, allows remarks to be inserted into a program for documentation purposes.

- Either REM or ! may immediately follow the line number.
- Remarks may follow any program statement except DATA.
- When the REM form follows a program statement, it must be separated from the statement by the character.
- Anything following REM or the ! character is considered a remark, and is ignored by BASIC.
- Other references in this manual: None

4-72. The following example illustrates some common uses of ! and the REM statement:

```
10   REM -- RESISTOR NETWORK VERIFICATION
20               !R1276 - R1299
30   !
40   REM -- Program History
50   !            Rev. No.  Date        Author
60   !              1.0     1/21/79     B. Hansen
70   !              2.0     11/13/79    N. Kelly
499  REM -- end header
500  PRINT 'Mount Resistor On Fixture' \REM Prepare for Test
510  PRINT 'Then Press RETURN';        !Display prompt
520  INPUT A$                          !Accept keyboard entry
```

4-73. The SAVE and SAVEL Statements



4-74. SAVE and SAVEL are used to store the user program currently residing in memory as a file. SAVE stores the file in ASCII text form. SAVEL stores the file in lexically analyzed form. A discussion of these forms follows.

- A file name must be specified, enclosed in single or double quotes. The file name may also be specified by a string variable.

- The file is stored on the default System Device if the file name is not preceded by MF0: for the floppy disk, or ED0: for the optional electronic disk. Refer to the Input and Output Statements section for a discussion of the default System Device concept.

- SAVE adds the file name extension .BAS when an extension is not specified.

- BASIC will request a confirming YES from the keyboard if an attempt is made to store a file under an existing file name. To avoid this interruption in a running program, first delete the file using a KILL statement.

- SAVE stores the ASCII form of the program in the largest available file space.

- SAVEL stores the lexical form of the program in the first available file space large enough to hold it. Note that this is different from SAVE.

- SAVE may also be used to obtain a printed listing of a program. To do so, specify the device name as either KB1: (RS-232-C Port 1) or KB2: (RS-232-C Port 2). The file name and extension are not required. See the User Manual for details on setting serial port baud rates.

- Other references in this manual: None.

4-75. In lexically analyzed form, a user program has binary numbers in place of ASCII character strings to represent line numbers, keywords, and operators. This form occupies less space and eliminates a processing step. Fluke BASIC programs in main memory are always in lexically analyzed form., even during editing. BASIC changes them to ASCII character form when needed for display, or for storage by a SAVE statement.

4-76. Working copies of user programs should be saved in lexically analyzed form, using SAVEL. In this form, programs will occupy less file storage space and will load into memory quicker.

NOTE

A program saved via SAVEL may not be executable if the version of Fluke BASIC under which it was saved differs from the version under which it is to be executed.

4-77. The lexically analyzed form of a program cannot be displayed directly, using the File Utility Program (FUP, discussed in the 1720A User Manual), or sent to an external printer. Consequently, back-up copies of user programs should be saved in ASCII character form, using SAVE. In this form, different versions of Fluke BASIC will be able to load and interpret the program.

4-78. Examples of SAVE and SAVEL used in immediate mode follow:

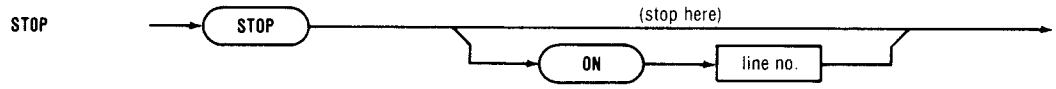
STATEMENT	RESULT
`SAVE "TEST1"`	Save the program currently in memory, in ASCII character form, on the System Device, under the name TEST1.BAS.
`SAVEL 'MF0:TEST2'`	Save the program currently in memory, in lexically analyzed form, on the floppy disk (MF0:), under the name TEST2.BAL.
`SAVE 'ED0:DATA.T71'`	Save the program currently in memory, in ASCII character form, on the optional electronic disk (ED0:), under the name DATA.T71.
`SAVE "KB1:"`	Send the program currently in memory, in ASCII character form, to RS-232-C Serial Port 1.

4-79. The following example generates a Command File on the floppy disk (MF0:) under the name COMMND.SYS. After completing this task, the program then saves itself on the floppy disk in ASCII character form under the name COMMND.BAS. An attempt is first made to delete an existing file by that name to avoid the requirement for keyboard input. However, the first time this program is run, the file COMMND.BAS does not yet exist. The attempt to delete it will result in error 305.

```
1000      ON ERROR GOTO 10060
1010      DATA " (Command input) "
1020      DATA " (Command input) "
            .
            .
(More DATA statements as needed)
            .
            .
10000     DATA "END"
10010     OPEN "MF0:COMMND.SYS" AS NEW FILE 1
10020     READ A$ \ IF A$ = "END" GOTO 10040
10030     PRINT #1, A$ \ GOTO 10020
10040     CLOSE 1
10050     KILL "MF0:COMMND.BAS" \ SAVE "MF0:COMMND" \ GOTO 10070
10060     IF ERL= 10050 AND ERR=305 THEN 'SAVE MF0: COMMAND' ELSE OFF ERROR
10070     END
```

4-80. Line 1000 enables the error interrupt (discussed in the Interrupt Processing section). The result is that an error 305 from the KILL statement in line 10050 simply transfers control to line 10060 without stopping the program for the error. Note that the KILL statement uses a complete file name, which is not required for the SAVE statement. Refer to the Input and Output Statements section for a discussion of the open channel used in lines 10010 through 10040.

4-81. The STOP Statement



4-82. The STOP statement halts execution of the BASIC program and displays the line number where the STOP occurred.

- STOP terminates program execution.
- STOP can be used to indicate "dead end" code branches, either because of errors or because of logical structuring.
- Other references in this manual: Program Debugging

4-83. The following example illustrates a common use of STOP:

```
10   REM -- TEST PROGRAM
20   ! Other statements
30   !
999  STOP                                ! End of main procedure
1000 REM -- SUBROUTINES
1010 ! Other statements
5000 END
```

Section 5
Input And Output Statements

5-1. INTRODUCTION

5-2. This section describes input and output statements that transfer data to and from devices that are not connected through the IEEE-488 instrument bus. The keyboard is the default device for input and the display is the default device for output. Other devices may be selected by a statement modifier, or through a previously opened data channel.

5-3. OVERVIEW

5-4. This section introduces two important concepts: system level devices, and data transfer channels. System level devices are resources available within the controller such as the file structured floppy and electronic disks, the keyboard and display, and the serial ports. A data transfer channel designates one of these system level devices as a source or destination for data transferred to or from a running program. File structured devices require a file name to be specified. Other devices, such as a serial port, require only the device designator.

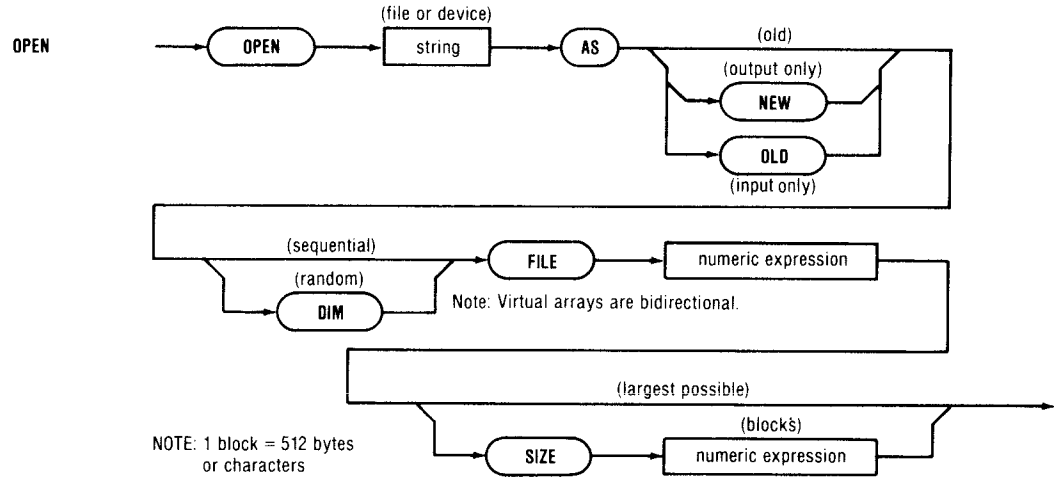
5-5. The devices that may be accessed through the techniques described in this section are listed in Table 5-1. The System Device, an important concept that can simplify programming, is the default location for files referenced by file name only. The floppy disk will be the System Device unless the optional electronic disk is designated as such.

Table 5-1. System-Level Devices

DEVICE NAME	DESIGNATION
System Device	SY0:
Floppy Disk	MF0:
Electronic Disk	ED0:
Display and Programmer Keyboard	KB0:
RS-232-C Serial Port 1	KB1:
RS-232-C Serial Port 2	KB2:

5-6. The System Device must be assigned with a File Utility Program (FUP) Command, either directly or through a start-up Command File. Refer to the 1720A User Manual for further information.

5-7. STATEMENT DEFINITIONS
5-8. The OPEN Statement



5-9. The OPEN statement assigns a data communication channel numbered between 1 and 6 to a file or device. The following points apply to any use of the OPEN statement:

- All subsequent input from and output to the file or device is made by reference to the channel number.

- The channel is sequential access, unless it is a virtual array channel. Data is sent to, or retrieved from, sequential channels in serial order.

- A virtual array channel, requiring the DIM specification, is random access. Data is sent to, or retrieved from, virtual arrays in any order.

- The string following OPEN indicates the name of the file or device. The default extension for file names is ". " (3 spaces).

- Other references in this manual: Virtual Arrays.

5-10. The AS NEW, AS OLD, and DIM optional constructions of the OPEN statement indicate specific and different actions for sequential and random channels. The following points discuss these differences:

- AS must be specified. If it is not followed by NEW or OLD, OLD is assumed.

- DIM specifies a random access virtual array file.

- For a sequential channel, NEW indicates an output channel. OLD, or no specification, indicates an input channel.

- For a random access virtual array channel, NEW indicates that an existing file by the specified name is to be deleted, if there is one.

- For a random access virtual array channel, OLD (or no specification) indicates that an existing file by the specified name is to be accessed. If the file is not found, error 305 results.

- Random access virtual array channels are bidirectional.

5-11. The numeric expression following FILE indicates the channel number to be assigned. Following are some points to be considered in selecting a channel number:

- The value of the numeric expression must be between 1 and 6.
- Each channel number can only be used for one operation at a time.
- A channel that was previously opened for a different purpose, and is no longer in use, must first be closed before being re-assigned.
- The CLOSE statement is discussed later in this section.

5-12. SIZE has significance only for NEW files. It is ignored for OLD files. The following points discuss some of the implications of the SIZE specification:

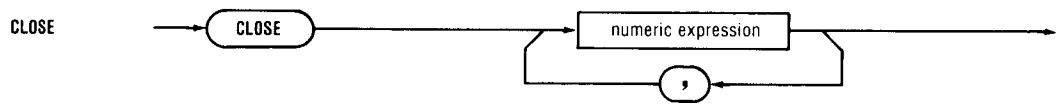
- The numeric expression following SIZE specifies the number of 512 character blocks to be reserved for a new file.
- A new file opened without a size specification will be assigned the largest available contiguous area.
- The first new file opened on the floppy or electronic disk is assigned all available free space if there is only one free area available. An attempt to open another new file on the same device, before the first new file is closed, results in error 306 (no room on device).
- SIZE may be used to limit the amount of space reserved so that a second new file can be opened on the same device.
- SIZE may also be used to verify that a new file will have required space available. For example, if a file requires 40 blocks, the statement:

```
OPEN "ED0"TEST.ARC" AS NEW FILE 1% SIZE 40%
```

will cause error 306 if 40 contiguous blocks are not available on the Electronic Disk (device name ED0:).

- The SIZE of a sequential file is the number of 512 character blocks needed to hold the file.

5-13. The CLOSE Statement



5-14. The CLOSE statement frees a previously opened channel forother use. As part of this process, some specific actions are taken:

- The input or output of data in memory to or from the specified channel is first completed.

- An End-of-File mark (CTRL/Z character) is then sent, if the channel was opened for sequential output.

- Extra space that had been reserved for a NEW file is once again made available.

- All opened channels can be closed by one CLOSE statement. Channel numbers are separated by commas.

5-15. The following example illustrates the effect of reserving more space for a new file than is actually used when the file is closed. If only 25 blocks are actually written to the file when line 510 is executed, the extra five blocks assigned to the file become available as free space on MF0: (i.e., usable by another OPEN command). Only the 25 blocks containing the information written by the program would remain as part of the file RESIST.DAT.

```
20  A$ = "MF0:" \ B$ = "RESIST.DAT"
30  OPEN A$ + B$ AS NEW FILE 5% SIZE 30%
40  ! Other statements
510 CLOSE 5%
```

5-16. The FLEN Variable

5-17. FLEN is a system variable set to the length, in blocks, of the last file opened.

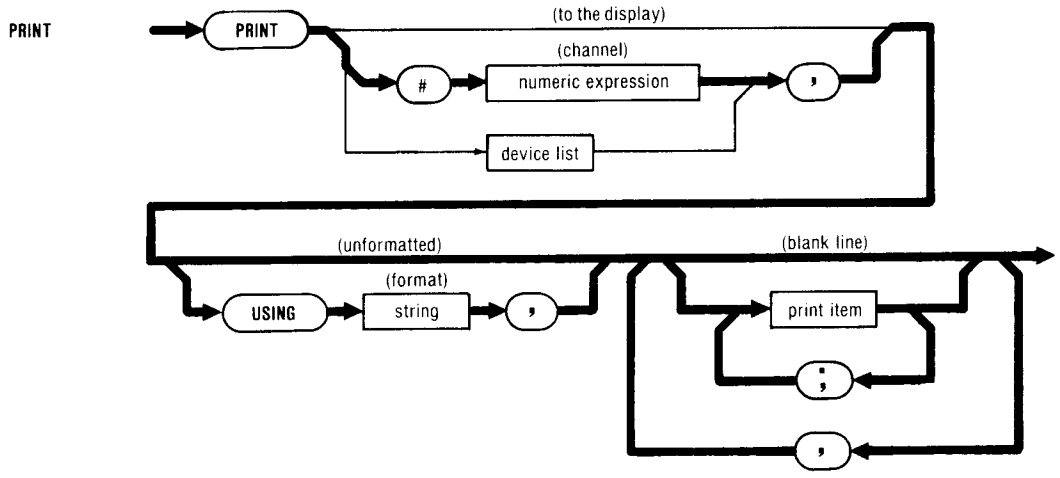
- FLEN may be used with OPEN to check the number of blocks in an open file before attempting to open a new file.

- One or more open files may need to be closed and perhaps deleted (see KILL in General-Purpose Fluke BASIC Statements) before opening a new file if the disk memory available is not sufficient to hold all opened files in contiguous areas.

5-18. In the following example, channel 1 is closed only if the last file opened has a length greater than five blocks. This may be done to limit open files and conserve disk space.

```
IF FLEN>5% THEN CLOSE 1%
```

5-19. The PRINT Statement



5-20. PRINT is described here for output to a system level device through a previously opened channel. PRINT may also be used for direct display output, or for output to instruments on the IEEE-488-1978 bus. These applications of PRINT are described in other sections in this manual.

- The numeric expression following PRINT selects a previously opened channel. See the OPEN statement described in this section.

- The items to be printed are listed, separated by either a comma or a semicolon.

- The items to be printed may include integer, floating point, and string expressions, as well as subranges of arrays and virtual arrays. Refer to the section on Data, Data Types, and Expressions, for a discussion of array subranges.

- The USING option may be specified for formatted output. Refer to the section on General-Purpose Fluke BASIC Statements for details.

- Other references in this manual: General-Purpose Fluke Basic Statements, IEEE-488 Input and Output.

5-21. The following examples illustrate some of the implications of specifying an array subrange in a PRINT statement. For a one-dimensional array, the designation A(m..n) is equivalent to the list of elements A(m) through A(n). Formatting may be as follows.

1. With semicolons:

```
PRINT A(m..n);
```

is equivalent to:

```
PRINT A(m); A(m+1); . . . ; A(n)
```

2. With commas:

```
PRINT A(m..n),
```

is equivalent to:

```
PRINT A(m), A(m+1), . . . , A(n),
```

3. With neither commas nor semicolons:

```
PRINT A(m..n)
```

is equivalent to:

```
PRINT A(m)
PRINT A(m+1)
.
.
.
PRINT A(n)
```

5-22. The following example illustrates the sequence of output of a two-dimensional array subrange. As with one-dimensional output, the formatting may be with semicolons, commas, or neither. This sequence is often described as "row-major", i.e., the first subscript (the row) changes slowest.

```
PRINT A(m..n, r..s)
```

is equivalent to:

```
PRINT A(m, r)
PRINT A(m, r+1)
.
.
.
PRINT A(m, s)
PRINT A(m+1, r)
PRINT A(m+1, r+1)
.
.
.
PRINT A(m+1, s)
.
.
.
PRINT A(n, r)
PRINT A(n, r+1)
.
.
.
PRINT A(n, s)
```

5-23. In the following three examples, NV(I) is a 11 element array used to store 10 expected nominal values for readings to be taken by an instrument. (Element zero is not used.) The 10 actual readings are in the 11 element array R(I). The examples illustrate different options for the printout format.

5-24. This first example produces three columns of data. Each element of data is printed out in default format. Note that this results in some items that are difficult to read, especially in the column marked %ERROR. The comma between the elements of the PRINT statement (line 1050) produces the 16-character columns.

```
10      OPEN "KB1:" AS NEW FILE 1
20      ! Other statements
30      !
200     GOSUB 1000
210     ! Other statements
220     !
250     CLOSE 1 \ STOP
1000    REM -- Subroutine: Output Readings To Printer
1010    PRINT #1                                        ! Blank line
1020    PRINT #1, 'NOMINAL VALUE', READING', '% ERROR'! Heading
```

```
1030    PRINT #1,                                              ! Blank line
1040    FOR I = 1 TO 10                                        ! Print results:
1050    PRINT #1, NV(I), R(I), (R(I) - NV(I)) / NV(I) * 100
1060    NEXT I
1070    RETURN
```

Results:

```
NOMINAL VALUE                READING                    %ERROR

 2.222                        2.19527                   -1.20298
 4.444                        4.361702                  -1.851896
 6.666                        6.758841                   1.392759
 8.888                        8.891673                   0.4133037E-01
-2.222                       -2.246359                  -1.096246
-4.444                       -4.50368                    1.342932
-6.666                       -6.545657                  -1.805325
-8.888                       -8.715492                  -1.940907
 100                          98.70009                  -1.299913
-100                         -99.38668                   0.6133201
```

5-25. This second example is an improvement on the previous one. The USING option of the PRINT statement formats the output as described in the section on General-Purpose Fluke BASIC Statements. The semicolons (;) in line 1050 separate numeric data elements from units designators (in this case, V for voltage). This results in a more readable format. Note question mark (?) in the last line of the % ERROR column. This results when a numeric value does not fit within the designated mask. The actual value in default format follows the question mark.

```
10   OPEN "KB1:" AS NEW FILE 1
20   ! Other statements
30   !
250  CLOSE 1 \ STOP
1000 REM -- Subroutine: Output Readings To Printer
1010 PRINT #1                                               ! Blank line
1020 PRINT #1, 'NOMINAL VALUE', 'READING', '%ERROR'         ! Print Heading
1030 PRINT #1                                               ! Blank line
1040 FOR I = 1 TO 10
1045 ! Nominal value and reading:
1050 PRINT #1, USING 'S###.###', NV(I); ' V', R(I); ' V'
1055 ! Error in percent:
1060 PRINT #1, USING 'S##.#', (R(I) - NV(I)) / NV(I) * 100; '%'
1070 NEXT I
1080 RETURN
```

Results:

```
NOMINAL VALUE                READING                    % ERROR

   2.222V                     2.195V                    -1.2%
   4.444V                     4.362V                    -1.9%
   6.666V                     6.759V                     1.4%
   8.888V                     8.892V                     0.0%
  -2.222V                    -2.246V                     1.1%
  -4.444V                    -4.504V                     1.3%
  -6.666V                    -6.546V                    -1.8%
  -8.888V                    -8.715V                    -1.9%
 100.000V                   98.700V                    -1.3%
-100.000V                    0.000V                    ? 100%
```

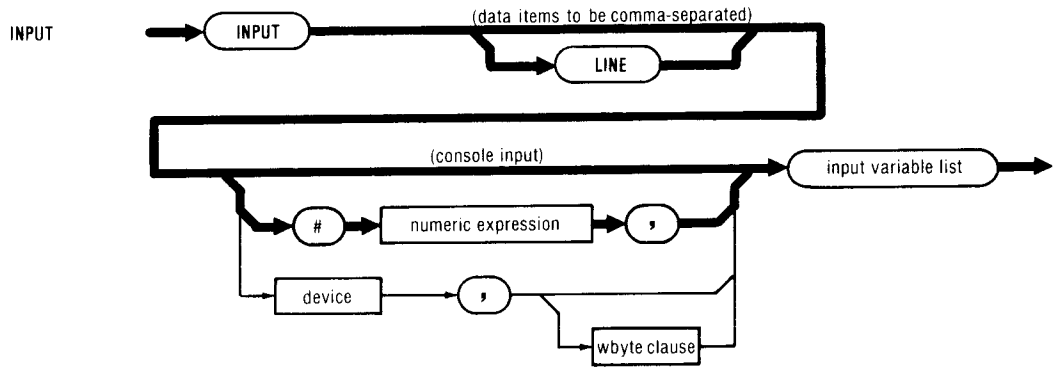
5-26. This third example illustrates the use of a subrange specification to print selected elements of an array. Elements of arrays NV and R, the nominal values and readings, are each printed one after the other, using a subrange specification. The string mask following USING is organized to space the elements. The semicolons used in the PRINT statement allow many elements to be printed on one line. Note the use of the PRINT statement at lines 1015, 1030, 1040, 1055, and 1065 to force a Carriage Return and Line Feed.

```
10   OPEN "KB1:" AS NEW FILE 1
     .
     .
250  CLOSE 1 \ STOP
1000 REM -- Subroutine: Output Readings To Printer
1005 ! Linearity Values:
1010 PRINT #1, USING "S##.###", "LIN:   "; NV(1..4);
1015 PRINT #1                                          ! Blank line
1020 PRINT #1, USING "S##.###", "RDGS:  "; R(1..4);    ! Print readings
1030 PRINT #1                                          ! Blank line
1040 PRINT #1                                          ! Blank line
1045 ! Negative Linearity:
1050 PRINT #1, USING "S##.###", "NEG LIN: "; NV(5..8);
1055 PRINT #1                                          ! Blank line
1057 ! Negative readings:
1060 PRINT #1, USING "S##.###", "NEG RDGS: "; R(5..8);
1065 PRINT #1                                          ! Blank line
1070 RETURN
```

Results:

```
LIN:         2.222  4.444  6.666  8.888
RDGS:        2.195  4.362  6.759  8.892
NEG LIN:    -2.222 -4.444 -6.666 -8.888
NEG RDGS:   -2.246 -4.504 -6.546 -8.715
```

5-27. The INPUT Statement



5-28. INPUT is described here for data input from a system level device through a previously opened channel. INPUT may also be used for direct keyboard input, or for input from instruments on the IEEE-488-1978 buses. These applications of INPUT are described in other sections in this manual.

- The optional LINE specification is discussed in the section General-Purpose Fluke BASIC Statements.

- The numeric expression following INPUT or INPUT LINE selects a previously opened channel. See the OPEN statement described in this section.

- The variables that will store the data input are listed, separated by a comma.

- The input data may include integer, floating point, and string expressions, as well as subranges of arrays and virtual arrays. Refer to the section on Data, Data Types, and Expressions, for a discussion of array subranges.

- Input to a two dimensional array subrange is assigned in the "row-major" manner described in the PRINT discussion in this section. Columns (second dimension) increment most often, and rows (first dimension) increment least often.

- Input to a two dimensional array subrange must be on a single line separated by commas.

- Input data must be specified in a line of not more than 80 characters when using this method to input values to an array.

- This 80 character restriction is removed when LINE is specified, since Carriage Return may be used to separate individual data entries. Refer to the discussion of the optional LINE specification in the section General-Purpose Fluke BASIC Statements.

- BASIC does not send a prompt character, "? ", when input is from a channel.

- Other references in this manual: General-Purpose Fluke Basic Statements, IEEE-488 Input and Output.

5-29. In the following example a sequential input channel from the keyboard is opened. When lines 110 and 120 are executed, the message displayed is the string at line 110. This technique allows an INPUT statement, such as line 120, to be used without the usual "?" prompt.

```
10   OPEN "KB0:" AS OLD FILE 2
20   ! Other statements
30   !
110  PRINT "ENTER THE SERIAL NO:  ";
120  INPUT #2, SN$
130  ! Other statements
```

5-30. In the following example, assume a device is attached to RS-232-C Port 1 which can print data sent to it and send data to the 1720A. Lines 20 and 30 assign KB1: simultaneously for input and output, using separate channels. This program can send prompts on the output channel (line 100) and receive data on the input channel (line 120). An example of such a device is a printing computer terminal. Note that a "?" is not sent at

line 100. This simultaneous assignment for input and output is not possible for a sequential channel to a file.

```
10   REM -- Demonstrate Input and Output From Same Device
20   OPEN 'KB1:' AS OLD FILE 1     ! Input channel 1
30   OPEN 'KB1:' AS NEW FILE 2     ! Output channel 2
40   ! Other statements
50   !
100  PRINT #2, A$                  ! Give prompt
110  INPUT #1, A(0..5)             ! Get 6 values
120  ! Other statements
```

Section 6
Virtual Arrays

6-1. INTRODUCTION

6-2. Virtual arrays are array variables stored on a mass storage device (floppy or electronic disk) but accessible as if they are in memory. The program has access to and may manipulate a virtual array as it would an ordinary array variable. The virtual array, however, is not limited to the space available in main memory. Other differences between virtual arrays and ordinary arrays are:

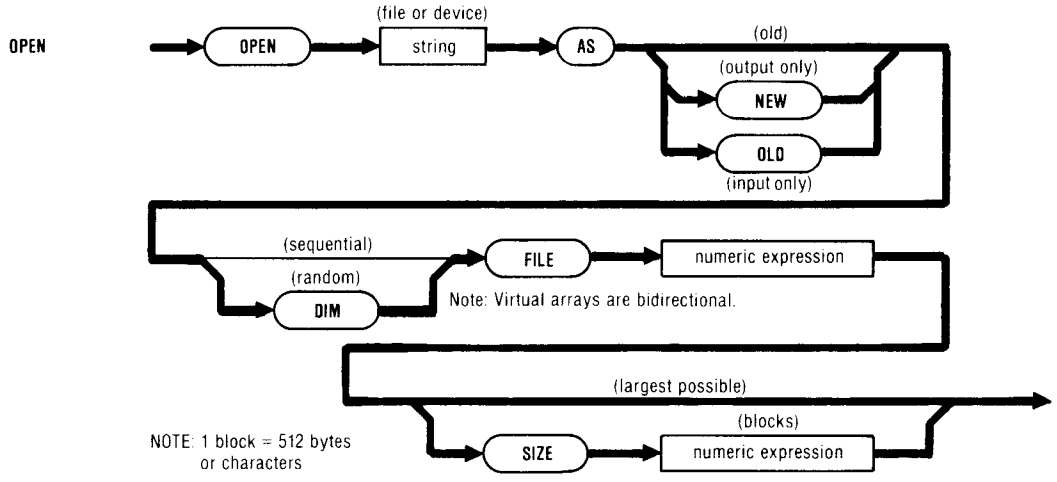
- Access of virtual array elements may require more time than the elements of main memory arrays.
- Virtual array strings are of fixed length.

6-3. Each virtual array file may contain a maximum of 65,536 (64K) bytes of data (128 blocks). Data is stored in virtual arrays in 1720A internal format so that, unlike ASCII sequential files, no data conversion is required during input or output.

6-4. OVERVIEW

6-5. Two methods of file organization are available: sequential files and virtual array or random access files. The programming techniques in this section deal with virtual array files. Sequential files are discussed in the section on Input and Output.

6-6. OPENING A VIRTUAL ARRAY FILE
6-7. The OPEN Statement



6-7. OPEN assigns a channel number to one or more virtual arrays.

- A file name specification in quotes must follow OPEN.

- The file will be on the default System Device (see the Input and Output section) unless MF0: or ED0: is included with the file name.

- A virtual array may only exist on a file-structured device (floppy disk or electronic disk) due to the random access requirement.

- Unlike ordinary channels described in the Input and Output section, virtual array channels are bidirectional.

- NEW specifies that the file will replace any existing file found with the specified name.

- OLD specifies that the channel will be associated with an existing file. Error 305 results if the file cannot be found.

- OLD is assumed when neither NEW nor OLD is specified.

- DIM must be included to specify that the channel opened will be a random access file.

- The numeric expression following FILE designates the channel number.

- Channels are integer numbers 1 through 6.

- Error 303 results if the channel is already in use.

- The numeric expression following SIZE specifies the number of 512-byte blocks to be allocated for a NEW file.

- The largest available contiguous space is allocated for a NEW file when SIZE is not specified.

- SIZE is ignored for OLD files.

- SIZE may need to be specified to open two or more NEW files simultaneously on the same device. If the device is packed, or has not had any file deletions, there is only one contiguous space available.

- Each floating point (real) element occupies 8 bytes (64 bits).

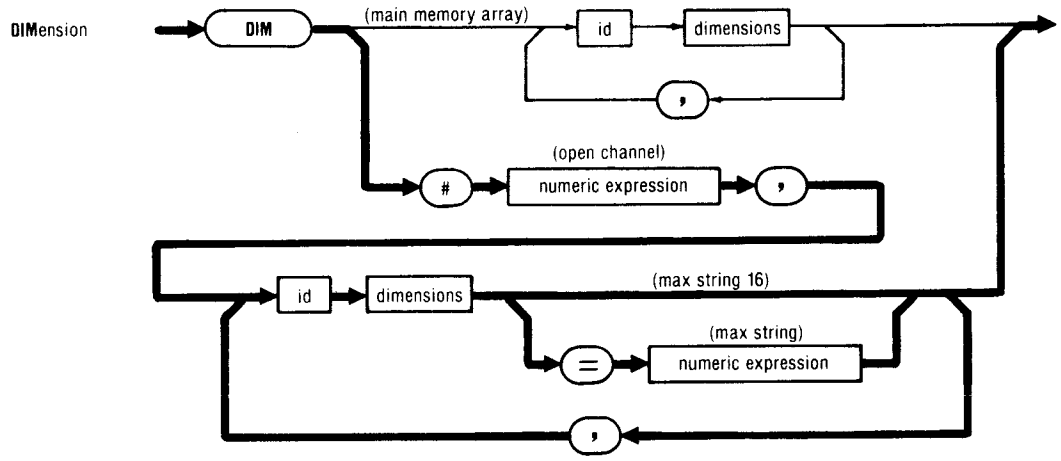
- Each integer element occupies 2 bytes (16 bits)

- Each string element occupies 16 characters, unless specified otherwise by the DIM statement (see below).

- When the channel is closed, information supplied by the DIM statement (see below) is used to reduce the space allocated to the virtual array to that actually required.

- Other references in this manual: Input and Output.

6-8. The DIM Statement



6-9. DIM (DIMension) assigns a previously opened channel to a virtual array and informs BASIC about data organization within the file.

- The # character and numeric expression following DIM specify an open channel from 1 to 6.

- String array declarations may specify the maximum length, in characters of each element string.

- This length specification follows the array identifier and array size specification. For example, DIM #4, Q$(63%, 63%) = 8% declares Q$ to be a virtual array, through channel 4, containing 64 X 64 string elements of 8 characters each.

- String element lengths in virtual arrays should be any power of 2 between 2 and 512 (2, 4, 8, 16, 32, 64, 128, 256 or 512).

- 16 characters per string is assumed when no length is specified.

- The virtual array DIM statement does not initialize string or numeric variables to nulls or zeros as does the ordinary DIM statement. For this reason, a value must be assigned to virtual array elements before they can be used as source variables. After being dimensioned they contain whatever bit patterns are in their respective disk storage areas.

- Other references in this manual: Input and Output.

6-10. In the following example, line 10 specifies that the virtual array file "RESULT.VRT" will be 20 blocks long and accessible on channel 1. Line 20 assigns four virtual arrays to the open channel 1.

```
10  OPEN "RESULT.VRT" AS NEW DIM FILE 1 SIZE 20

20  DIM #1, A$(63%) = 128%, C$(31%), B(40%), A%(500%)
```

6-11. Note that the data will fit in 20 blocks, but not in 19:

ARRAY	# OF ELEMENTS	SIZE	#OF CHARACTERS
A$	64	128	8192
C$	32	16 (default)	512
B	41	8	328
A%	501	2	1002
		TOTAL:	10034

19 blocks = (19 * 512) = 9728 characters

20 blocks = (20 * 512) = 10240 characters

6-12. For further information on size of arrays, refer to the discussion on Virtual Array File Organization in this section.

6-13. USING VIRTUAL ARRAYS

6-14. Once a virtual array file has been opened and a DIM statement for the channel has been executed, the virtual array elements may be used just as ordinary variables.

6-15. Using Virtual Arrays as Ordinary Variables

6-16. In the following example, elements of A% may be used wherever integer array elements may be used, except in RBYTE or WBYTE statements (see the section on IEEE-488 Bus Input and Output Statements).

```
10  OPEN "INTEGR.BIN" AS DIM FILE 1%
20  DIM #1%, A%(255%, 127%)
```

6-17. The following example shows that data may be read from or written to the file simply by writing the array name in an expression or assigning a value to an array element.

```
305 IF A%(I%, J%) > 0% THEN 350
430 LET A%(K%, 0%) = ABS (A% (K%, 1%)) + 2%
```

6-18. Using Virtual Array Strings

6-19. Strings in virtual arrays are considered by BASIC to be of fixed length. The default length is 16 characters, or as declared in the DIM statement (see the DIM discussion in this section).

6-20. The following example specifies a virtual string array with 32-character elements. Line 390 will display the number 32 regardless of the value assigned to that particular element of A$.

```
380 DIM #2%, A$(15%) = 32%
390 PRINT LEN(A$(I%))
```

6-21. When characters are assigned to a virtual array string element, BASIC will add null characters to the right end of the string until it equals the declared string element length. This can be the source of subtle errors. Consider the virtual array A$ of the previous example. The following program section attempts to add an * character to all of the elements of A$. This example will not work, and results in error 904.

```
570     FOR I% = 0% TO 15%
580     A$(I%) = A$(I%) + "*"
590     NEXT I%
```

6-22. Each element of A$ is allocated 32 characters. The expression A$(I%) + "*" results in a 33 character string. When this string is assigned to A$(I%), error 904 (string assigned longer than declared string length) results. It is necessary to strip trailing null bytes from the virtual array string value before appending the '*' character. One way to accomplish this is:

First, define two subfunctions:

```
DEF FN A%(V$) = SGN(INSTR (1%, V$, CHR$(0%) ))
DEF FN B%(V$) = INSTR(1%, V$, CHR$(0%)) - 1%
```

Then use these in this defined function:

```
DEF FN S$(V$)=LEFT(V$,(1% - FN A%(V$)) * LEN(V$) + FN B%(V$) *  FN A%(V$))
```

6-23. The defined function FN S$ will strip trailing NULL characters from the string elements of A$ prior to adding a character. With these functions previously defined, the following rewrite of the previous example will work without error, providing all elements of A$ have room for another character.

```
570     FOR I% = 0% TO 15%
580     A$(I%) = FN S$ (A$(I%)) + "*"
590     NEXT I%
```

6-24. Virtual Array File Organization

6-25. The following discussion describes how BASIC organizes virtual array files. Most of this is invisible to the programmer. However, an understanding of it will enable the user to optimize the use of virtual arrays.

6-26. When a virtual array file is opened, BASIC creates a 512-byte (1 block) buffer in main memory to hold the block of the file currently being used (one buffer is created for each virtual array file). Each file is considered to consist of a sequence of bytes, numbered 0 (first block) to n (last block). The description of each virtual array contains the channel number to which the file containing the array is attached, and the address within the file at which the array starts (the array's base address).

6-27. The base address for a virtual array is determined when the DIM statement declaring the array is processed. The base address assigned is the next available (higher) address which will not cause an array element to cross a block boundary. Each array element must be wholly contained within a 512-byte block. This restriction may be defined as: the base address of an array must be an integral multiple of the array element length and no array element may be longer than 512 bytes. This works since all virtual array elements have a length which is an integral power of 2.

6-28. Since virtual arrays are assigned addresses in the file in the order in which the arrays are declared in the DIM statement, the restrictions noted in the paragraph above suggest that it is possible to allocate file space efficiently or inefficiently when arrays having differing element lengths are assigned to the same file. This depends on the order in which array declarations appear in the DIM statement. To eliminate wasted file space, the simplest rule is that virtual array declarations should appear in the DIM statement (as read from left to right) in decreasing order of array element lengths. This rule ensures that if an element overlaps a block boundary, a minimum of space is left unused in the previous block.

NOTE

The unused space at the end of a virtual array file is available for use if a subsequent DIM statement enlarges the file. See the discussion that follows on equivalencing of virtual arrays.

6-29. In the following DIM statement, the arrays are allocated space as shown below the statement.

```
DIM #1%, A$ (10%) = 64%, B (10%, 9%), C% (1%, 4%)
```

A$	11 elements of 64 bytes each	= 704 bytes
B	110 elements of 8 bytes each	= 880 bytes
C%	10 elements of 2 bytes each	= 20 bytes
	TOTAL:	1604

6-30. The total space needed is 68 bytes greater than 3 blocks (1604 - (3 * 512)). The blocks will be allocated as follows. Note that the first three blocks are completely used, leaving only the extra 68 bytes for the fourth block. 444 bytes remaining in the fourth block are unused.

Block	Variable	Elements	Bytes
1	A$	8	512
2	A$	3	192
2	B	40	320
3	B	64	512
4	B	6	48
4	C%	10	20

6-31. Suppose the DIM statement is changed to read:

```
DIM #1%, C% (1%, 4%), B (10%, 9%), A$ (10%) = 64%
```

6-32. The total space needed remains 1604 bytes. The variables however are allocated to blocks as follows.

Block	Variable	Elements	Bytes
1	C%	10	20
1	unused	-	4
1	B	61	488
2	B	49	392
2	unused	-	56
2	A$	1	64
3	A$	8	512
4	A$	2	128

6-33. Only 508 bytes of bolck 1 and 456 bytes of block 2 are used. The unused portions, totalling 60 bytes, could not entirely contain one more data element in the sequence assigned. As a result, block 4 has only 384 bytes available, instead of the possible 444.

6-34. Programming Techniques

6-35. The key to the efficient use of virtual arrays is to minimize the number of data transfers to or from the virtual array file. Whenever a virtual array element is accessed, either to read its value or to assign to it a new value, BASIC determines the block number and the file in which the element exists. If the block required is not in the memory buffer, the required block is moved from the file into the memory buffer. The block previously held in the buffer is written to the file only if achange in its contents occurred.

6-36. Array Element Access

6-37. Array elements in a file are stored in row-major order which means that access to the elements in the storage order when the rightmost subscript varies most quickly, as in the following example:

```
4650  FOR I% = 0% TO 63%
4660    FOR J% = 0% TO 63%
4670      A% (J%, I%) = 0%
4680    NEXT J%
4690  NEXT I%
```

6-38. The form just discussed describes the most efficient access method for initializing the array A%. When the array A% is stored on a floppy disk, its initialization required 26.7 seconds. The following example is the least efficient access method:

```
4650  FOR I% = 0% TO 63%
4660    FOR J% = 0% TO 63%
4670      A% (I%, J%) = 0%
4680    NEXT J%
4690  NEXT I%
```

6-39. This second example requires 410 seconds, 15.4 times as long as the first example. The first example requires a new block to be read for every 256 elements of A% that are written. This second example, however, requires that a new block be read for every four elements of A% that are written.

6-40. Splitting Arrays Among Files

6-41. If information stored in different virtual arrays will often be required at the same time, placing the arrays in separate files will speed processing.

6-42. In some programs it may not be possible to use the null stripping function described previously in this section. This may be because the null characters are part of the string data required. However, the true string length can be determined without stripping the null characters. If there is no character which may be used to specify the end of a string in the virtual array, this information may be retained by placing an integer array parallel to the string array. Thus, for each string element, an integer element contains the length of the string.

6-43. In the following example, L%(I%) contains the length of string A$(I%). The significant characters of element A$(I%) can be recovered by a statement such as line 6370.

```
6360   DIM #1%, A$(1023%) = 16%, L%(1023%)
6370   B$ = LEFT (A$(I%), L%(I%))
```

6-44. A drawback to this method is that each time an element of A$ is read from the beginning of the file, another block of the file must be read to retrieve the corresponding element of L%.

6-45. A more efficient organization is to assign A$ to one virtual array file and L% to another file. This will cause the BASIC Interpreter to assign two buffers, one for the strings of A$ and one for the integers of L%.

```
4770    DIM #1%, A$(1023%) = 16%
4780    DIM #2%, L%(1023%)
```

6-46. In an actual test, accessing the elements of A$, one by one in increasing order, the first example (with A$ and L% in one file) required 386 seconds. With A$ and L% in separate files, only 15.8 seconds were required to perform that same processing, a 24:1 difference.

6-47. Re-using Virtual Array Declarations

6-48. When a virtual array DIM statement has been executed, the variables defined as virtual arrays remain defined even after the virtual array file has been closed. However, an attempt to access a virtual array when the channel has been closed will result in an error 308 (channel not open). Since the variables remain defined, it is possible to close a virtual array file and to later re-open it. Since the variables have already been defined, a new DIM statement is not necessary to re-open the file. Note, however, that the file must be re-opened using the same channel number as that used in the DIM statement defining the arrays.

6-49. If a number of separate virtual array files, all with the same data organization, must be processed by a program, it is possible (if no two files are to be processed simultaneously) to reuse the variable definitions. Consider the following processing sequence:

```
OPEN        first file
DIM         virtual arrays
            process first file
CLOSE       first file
OPEN        second file (on same channel as 1st file)
            process 2nd file
CLOSE       second file
.
.
.
```

6-50. In each processing loop, the same virtual array variables may be used to process the file data.

6-51. Equivalencing Virtual Arrays

6-52. It is possible to execute multiple DIM statements for a single channel. The second (and subsequent) DIM statements for a channel simply redefine the file organization as shown below. The first DIM statement allocates a 441-element array, A%, organized as a

21x21 matrix. The second DIM statement does not allocate array B% following A% but redefines the 441 integer elements of the file as a vector with the name B% (similar to a FORTRAN equivalence).

```
110   DIM #1%, A% (20%, 20%)
120   DIM #1%, B% (440%)
```

NOTE

Fluke BASIC makes no check for the consistency of virtual array equivalences. This is the responsibility of the programmer.

Section 7
IEEE-488 Bus Input and Output Statements

7-1. INTRODUCTION

7-2. This section describes the statements which are designed for communication with instruments on the two IEEE-488 Buses. These statements allow the 1720A to control any type of bus-compatible device.

7-3. OVERVIEW

7-4. Detailed information on instrument bus communication concepts and messages, control lines and data lines, and on timing, is available in the standard "IEEE-488-1978 Standard Digital Interface for Programmable Instrumentation". Copies of this standard are available from The Institute of Electrical and Electronic Engineers, 345 East 47th Street, New York, New York 10017. Appendix D gives the correspondence between BASIC commands and the sequence of bus actions which actually take place.

7-5. Communications on the bus occur on a detailed signal and code level as defined by the standard. However, for the most part, bus communication can be viewed on a functional level. On this level the user is concerned more about what is happening on the bus than how it is done. Bus communications are categorized as functional messages exchanged between devices. Each bus device in a system may be designed to implement only the messages which are important to its system purpose.

7-6. DEVICE AND PORT ADDRESSING

7-7. The following paragraphs describe in general terms how BASIC uses the IEEE-488 Bus ports. They also describe how BASIC performs primary and secondary addressing of instruments on those ports.

- The 1720A has two independent IEEE-488 instrumentation bus interfaces. These are addressed by BASIC with the statement modifier:

```
PORT  port expression
```

- The value of the port expression used in a statement must be either 0 or 1. Refer to the 1720A User Manual for identification of bus ports and information on cable connections.

- Each device on a bus connected to a port has a primary address in the range of 0 through 30. The address is normally determined by switch settings on or in the instrument. A BASIC command indicates a primary device address with the @ character:

```
@  device number
```

- A device number is any numeric expression with a value in the range 0 through 30 for an instrument connected to PORT 0, or in the range 100 through 130 for an instrument connected to PORT 1. The last two digits are the device number.

- The device number must correspond with the address switch settings in the instrument.

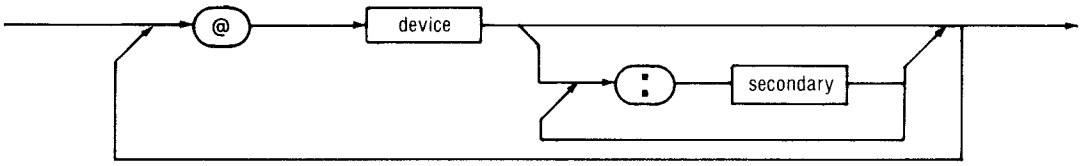
- The hundreds digit of the device number determines the port of the instrument being addressed: 0 for PORT 0, 1 for PORT 1 (default is 0).

- The IEEE-488 Bus standard also allows instruments to have one or more secondary addresses. The function associated with a secondary address depends on the instrument design. A BASIC command identifies a secondary address as part of the device number:

@ device number : secondary number : secondary number : . . .

- The secondary number is any numeric expression with a value in the range 0 through 31.

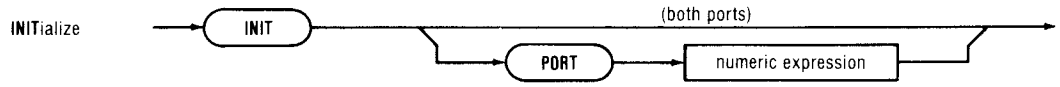
- More than one device may be addressed at once by listing device numbers in the command. The @ character is the only separator in a multiple device list (commas are not used). Secondary address numbers are optional. The device list syntax is as follows:



7-8. INITALIZATION AND CONTROL STATEMENTS

7-9. The following paragraphs describe Fluke BASIC statements which initialize, set up, or otherwise manipulate instruments either before or after data transfers.

7-10. The INIT Statement



7-11. INIT (INITialize) sends an interface clear message followed by remote enable and parallel poll unconfigure to the specified port, or to both ports if not specified.

- INIT places the bus in an idle state.

- INIT sends REN (remote enable), IFC (interface clear), UNL (unlisten), UNT (untalk), and PPU (parallel poll unconfigure) commands to the bus.

- REN enables remote programming of instruments.

- IFC unaddresses all listeners and talkers (places all talker and listener functions in an idle state) and terminates all handshakes (places source and acceptor handshakes in either an idle or wait state).

- PPU places devices with remote parallel poll programming capability into idle state. Such devices may be reconfigured for parallel poll with the CONFIG statement described in this section.

- INIT, without a port specified, initializes PORT 0 and PORT 1.

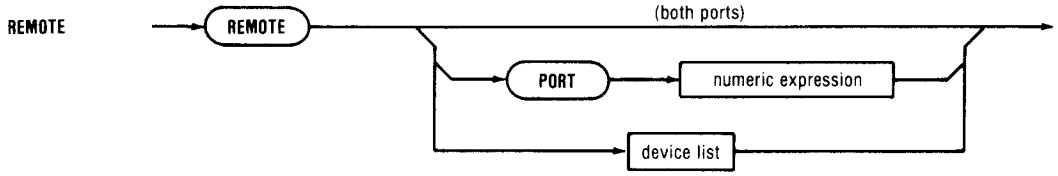
- To initialize only one port, the word PORT must be used, followed by an expression or number that evaluates to 0 or 1.

- Other references in this manual: None.

7-12. The following examples illustrate uses of the INIT statement:

STATEMENT	MEANING
`INIT`	Initialize both IEEE-488 interfaces.
`INIT PORT 1`	Initialize all instruments on IEEE-488 port 1.
`INIT PORT A%`	Initialize all instruments on the IEEE-488 port identified by the value of A% (must be 0 or 1).

7-13. The REMOTE Statement



7-14. REMOTE sets the REN (Remote Enable) line on the IEEE-488 Bus to true.

- REN is set true on both ports if no port number is given.

- REN is set true only on the specified port when a port number is specified.

- REMOTE may be followed by a device list.

- When REMOTE is followed by a device list, REN is set true on the port or ports represented by instruments in the device list.

- After this, the listed devices are addressed to listen (sent an MLA message).

- Normally this places the instruments into remote, depending on the instrument design.

- Other references in this manual: None.

7-15. In the following example, instrument port 0 is initialized at line 110. This sets REN true. However, the instruments on the Bus are not in remote state until they have received a MLA message. Line 120 sends the addresses (2 and 4).

```
110    INIT PORT 0
120    REMOTE @ 2 @ 4
```

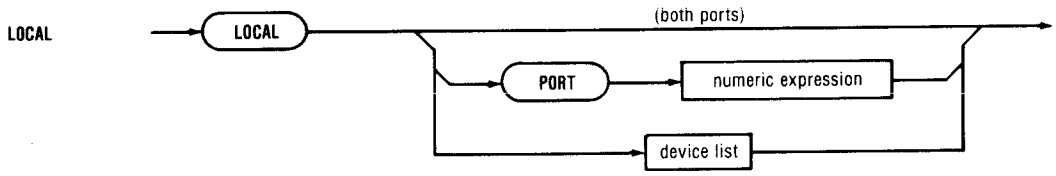
7-16. The following example is similar to the previous one. Note the use of the variable A to designate the port number and B to designate the address of the device by adding A*100 to B. This program correctly addresses the instrument without regard to which port it is on.

```
110    INIT PORT A
120    REMOTE @ A * 100 + B
```

7-17. The following statement sets REN true on both port 0 and 1.

```
510    REMOTE
```

7-18. The LOCAL Statement



7-19. LOCAL resets instruments to a local state. Typically, this means that front panel controls are activated.

7-20. Without a device list, LOCAL is the reverse of REMOTE.

- REN is set false on both ports if a port number is not specified.
- REN is set false only on the designated instrument port when a port is specified.
- LOCAL, without a device list, reverses all effects of the LOCKOUT statement.
- Other references in this manual: None.

7-21. With a device list, LOCAL sends a GTL (Go To Local) message on the ports identified in the device list.

- The instruments specified are first addressed as listeners.
- The GTL message is then sent.
- If LOCKOUT was previously sent, the GTL message will not activate the front panel of the instrument.
- Other references in this manual: None.

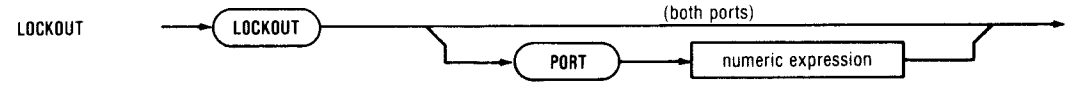
7-22. The following examples illustrate common usage of the LOCAL statement:

STATEMENT	RESULT
`LOCAL`	Set REN false on both ports.
`LOCAL PORT 1`	Set REN false on port 1.
`LOCAL @ 103:4`	Send GTL message to the instrument on port 1 at address 3, secondary address 4.
`LOCAL @ 7 @ 113`	Send GTL messages to the instrument on port 0 at address 7, and to the instrument on port 1 at address 13.

7-23. The following example illustrates the use of LOCAL to return an instrument temporarily to local control. Line 1180 returns the instrument to REMOTE control.

```
110   INIT                          ! Initialize both ports
120   REMOTE @ 2 @ 104              ! Place instruments in remote
130   ! other statements
140   !
1100  REM -- Subroutine: Special Instrument Usage
1110  LOCAL @ 104                   ! Set one instrument to local
1120  PRINT "SET INSTRUMENT TO PERFORM TEST"
1130  ! other statements
1140  !
1180  REMOTE @ 104                  ! Place instrument in remote
```

7-24. The LOCKOUT Statement



7-25. LOCKOUT disables not only front panel controls (as with REMOTE) but also any "return to local" function button that may be on an instrument. As defined in the IEEE-488-1978 Standard, any instrument addressed to listen after receiving a local lockout command will immediately be placed in the "Remote With Lockout State".

- LOCKOUT implements the LLO (Local Lockout) capability on the IEEE-488 Bus.
- LOCKOUT sets REN, and then sends LLO.
- This sequence is sent on both ports if no port number is given.
- This sequence is sent on only one port if a port number is specified.
- Other references in this manual: None.

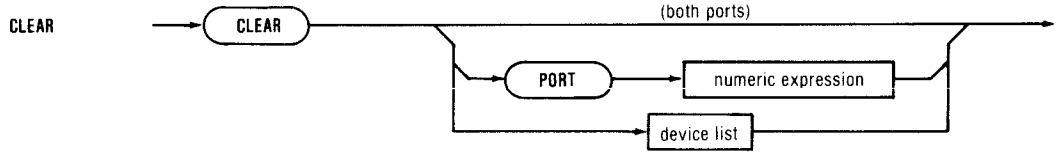
7-26. The following example uses the REMOTE statement with a device list after the LOCKOUT statement to immediately disable all local instrument control. The device addresses sent at line 160 place the instruments in "Remote With Lockout State" since LOCKOUT was previously sent at line 150.

```
100   REM--Initialize Ports, Disable Local Instrument Control
110   ! Instruments are at Port 0: address 2, 7, 9
120   ! And at Port 1: address 1
130   !
140   INIT
150   LOCKOUT
160   REMOTE @ 2 @ 7 @ 9 @ 101
170   ! other statements
```

7-27. The following example uses LOCKOUT to place instruments on port 0 into a state such that whenever they are used, they enter the "Remote With Lockout State". The CLEAR statement at line 100 insures that no previously addressed instrument is placed in "Remote With Lockout State". At line 140, when the instrument at address 2 of port 0 is addressed as a listener and sent the message "F2R0", it is placed in "Remote With Lockout State".

```
30    ! other statements
40    !
100   CLEAR                          ! Unlisten, Untalk all devices
110   LOCKOUT PORT 0
120   ! other statements
130   !
140   PRINT @ 2, "F2R0"
```

7-28. The CLEAR Statement



7-29. CLEAR sends a device clear or selected device clear message to the specified device(s), to all devices on the specified instrument port, or to all devices on both instrument ports.

7-30. CLEAR without a device list sends a device clear (DCL) message.

- CLEAR, without a port specified, sends DCL to both IEEE-488 Bus ports.
- To send DCL to only one port, the word PORT must be used, followed by an expression or number that evaluates to 0 or 1.

7-31. CLEAR with a device list sends a selected device clear (SDC) message.

- Instruments listed are addressed to be listeners.
- SDC (selected device clear) is then sent to the designated port(s).
- Other references in this manual: None.

7-32. The following examples illustrate common uses of the CLEAR statement:

STATEMENT	RESULT
`CLEAR`	Clear all instruments on both ports.
`CLEAR @ 1 @ 102`	Clear device 1 on port 0, and device 2 on port 1.
`CLEAR PORT 1`	Clear all devices on port 1.

7-33. The TRIG Statement

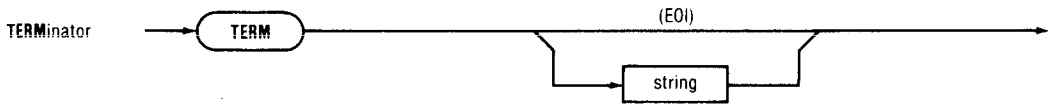
7-34. TRIG (TRIGger) addresses a set of instruments as listeners and then triggers them simultaneously. The effect of the trigger is dependent upon the instrument. For example, a digital multimeter may take a reading, or a source instrument may go from standby to operate.

- TRIG addresses the specified devices as listeners.
- The controller then sends a GET (Group Execute Trigger) message on the bus or buses represented in the device list.
- Other references in this manual: None.

7-35. The following examples illustrate common uses of TRIG.

STATEMENT	RESULT
`TRIG @ 22% @ 3%`	Trigger devices 22 and 3 on port 0.
`TRIG @ 1% @ 105%`	Trigger device 1 on port 0, and device 5 on port 1.

7-36. The TERM Statement



7-37. TERM (TERMinator) may be used to specify an 8-bit character which, when received from an IEEE-488 bus device in response to an INPUT statement, will act as a line terminator for the input data.

- The EOI (End Or Identify) line on the Bus will always terminate input, regardless of the use of the TERM statement.
- TERM allows the user to specify an arbitrary 8-bit byte that will also terminate input.
- The terminating character is Linefeed when TERM is not used.
- The terminating event is limited to the EOI bus control line when TERM is used without a character specified. In this mode, all 8-bit values are acceptable as input.
- Only one terminating character may be specified at a time.
- Other references in this manual: None.

7-38. The following examples illustrate common uses of TERM:

STATEMENT	MEANING
`TERM '?'`	Select the question mark character as an input terminator.
`TERM CHR$(255%)`	Select a character with all bits set to one (in an 8-bit byte) as the input terminator.
`TERM`	Limit input termination to the EOI line of the Bus.
`TERM ''`	Null character specification. Limit input termination to the EOI line of the Bus.

7-39. The TIMEOUT Statement

7-40. TIMEOUT sets a limit on the amount of time the 1720A Controller will wait for a response to an IEEE-488 Bus request. This prevents an instrument fault from halting the system.

- Error 408 results when the specified time is exceeded.
- The time allowed is 20 seconds when TIMEOUT is not used.
- The expression designates wait time in milliseconds.
- The range is 0 to 32767 milliseconds. Actual resolution is 10 milliseconds.
- Timeout protection is disabled by specifying zero milliseconds. This is especially useful during analysis of commands on the bus with a bus analyzer.
- Other references in this manual: None.

7-41. The following examples illustrate common uses of TIMEOUT.

STATEMENT	RESULT
`TIMEOUT 500`	Set timeout limit to 500 milliseconds.
`TIMEOUT 0%`	Disable timeout protection

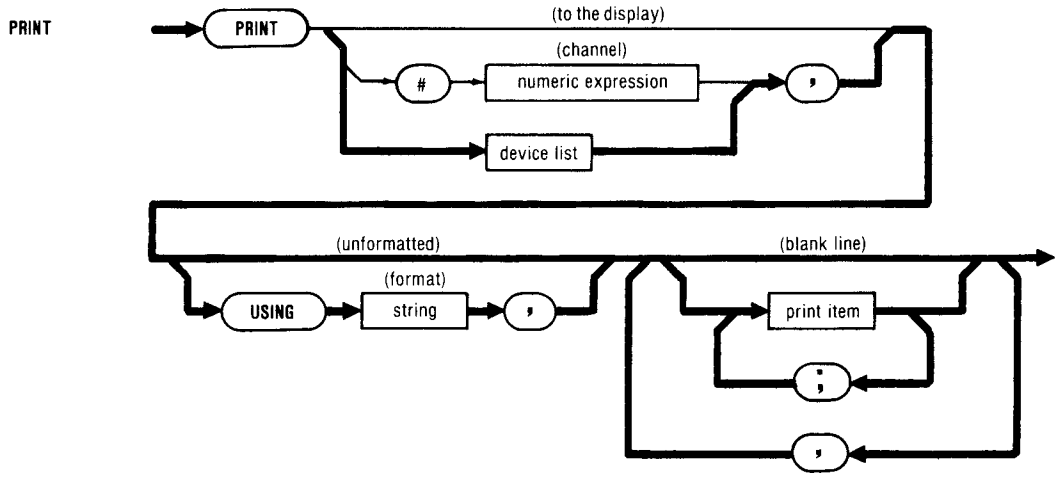
Table 7-1. Initialization and Control Statements Summary

STATEMENT	DESCRIPTION
CLEAR	Sends Device Clear (DCL) or Selected Device Clear (SDC) bus messages. Sets instruments to a ready state.
INIT	Halts port activity and prepares port for further messages.
LOCAL	Sets Remote Enable (REN) Bus line false or sends Go To Local (GTL) bus message to instruments in device list.
LOCKOUT	Disables local switch on all addressed instruments.
REMOTE	Sets Remote Enable (REN) Bus line true. Addresses any services specified as listeners.
TERM	Selects a terminator character for designating the end of an input stream from a device.
TIMEOUT	Sets a limit on the time the 1720A will wait for a response to a request.
TRIG	Addresses instruments in the device list and sends Group Execute Trigger (GET) Bus message.

7-42. INPUT AND OUTPUT STATEMENTS

7-43. This discussion describes INPUT and PRINT statements as they are used for instruments on the IEEE-488 Bus. These statements are also used for other types of input and output described elsewhere in this manual. Specific references are provided after each satatment definition. INPUT WBYTE is a specific IEEE-488 Bus statement which combines the characteristics of the INPUT statement with a WBYTE clause described later in this section.

7-44. The PRINT and PRINT USING Statements



7-45. PRINT and PRINT USING are used to output data to designated listeners. The instruments which are to receive output data are specified in a device list.

- A PRINT or PRINT USING command which is followed by a device list addresses the specified devices as listeners.
- All other devices are commanded to unlisten.
- When only an @ character follows PRINT or PRINT USING, with no device numbers, then no listen, unlisten, talk, or untalk commands are sent. Data is sent to the last bus addressed. This will increase the speed of data output.
- One or both instrument ports may be represented in the device list.
- Output data is sent character by character.
- Characters are sent exactly as a normal PRINT or PRINT USING statement, except for the use of the comma and semicolon to format the output.
- A comma following a data item in the output list indicates the EOI Bus line is to be set simultaneously with the last character of that item. It does not indicate tabulation to 16 character columns by sending extra spaces.
- A Carriage Return and Line Feed, with the EOI Bus line set simultaneously with the Line Feed, follows the last data item in a PRINT list when it is not terminated with either a comma or a semicolon.
- Array subranges may be included in the output list.
- A comma or semicolon following a subrange specification is equivalent to listing each element of the array subrange followed by the comma or semicolon.
- Other references in this manual: General Purpose Fluke BASIC Statements, Input and Output Statements.

7-46. The following example sends the characters "F0RA3" to the DVM if the value F1 is 0 and R1 is 3. Note the use of the mask "#" to eliminate the leading and trailing space that would normally be printed with F1 and R1. After the 3, a Carriage Return and a Line Feed (with EOI set true) is sent.

```
1700  REM -- Program DVM on Port 1, Address 4
1710  ! Send function and range:  F1 and R1
1720  PRINT @ 104%, USING "#", "F"; F1; "RA"; R1
```

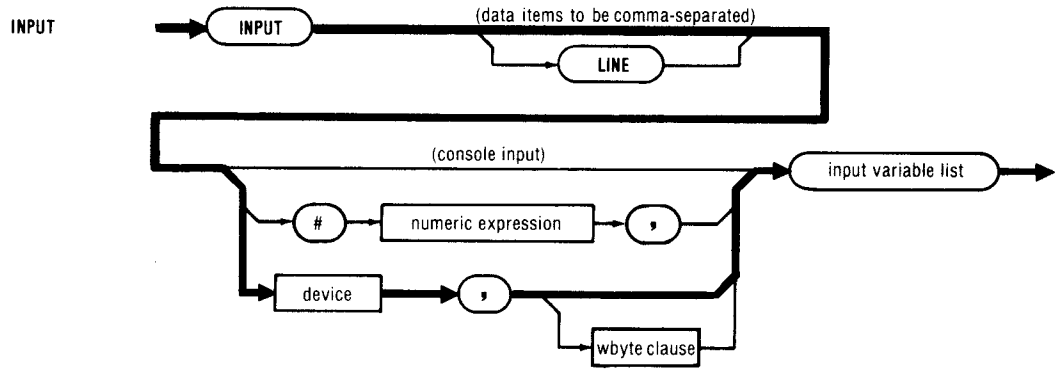
7-47. The following Immediate Mode PRINT statement addresses instruments on port 0 at device 2 and at device 4 with secondary address 6 as listeners. It then sends out the contents of strings A$(3) through A$(6) and sets EOI true with the last byte of each string.

```
PRINT @ 2 @ 4:6, A$(3%..6%),
```

7-48. The following example selects the instrument at address 2 as a listener at line 100. The indicated data is then sent and the instrument remains a listener. Since there is no other instrument bus activity before line 200, the characters "R1?" will be sent to the same instrument.

```
100   PRINT @ 2, "F153R0?"          ! Set up instrument
110   !
120   ! other non-instrument statements
130   !
200   PRINT @, "R1?"                ! Change range of instrument
210   ! other statements
220   !
```

7-49. The INPUT Statement



7-50. INPUT is used to receive data from instruments on the IEEE-488 Bus.

- The only syntax difference in the INPUT statement for IEEE-488 Bus instruments is the use of a device specification instead of a channel number.
- Only one device number may be specified.
- Input data items must be separated by commas.
- Input data is terminated by a Line Feed character or an alternate character specified by a previous TERM statement. The EOI Bus line may also be used by the transmitting instrument to terminate input.
- The instrument is addressed as a talker prior to reading data.
- The INPUT statement can be structured so that the 1720A will not address an instrument that has been previously addressed. This will increase the speed of data input. INPUT assumes there is a talker already on the bus when the @ character follows INPUT without a device specified. Incoming bus data is then simply assigned directly to the specified variables.
- Other references in this manual: General Purpose Fluke BASIC Statements, Input and Output Statements.

7-51. The following example addresses the instrument at device 4 on port 1 as a talker, and then reads four floating point (ASCII) values. The first three values should be terminated by commas, and the fourth (last) value should have a Line Feed as a terminator, unless a TERM statement has defined a different terminator character. The EOI line may also be used to terminate the last input.

```
2470 INPUT @ 102, A(1%..4%)
```

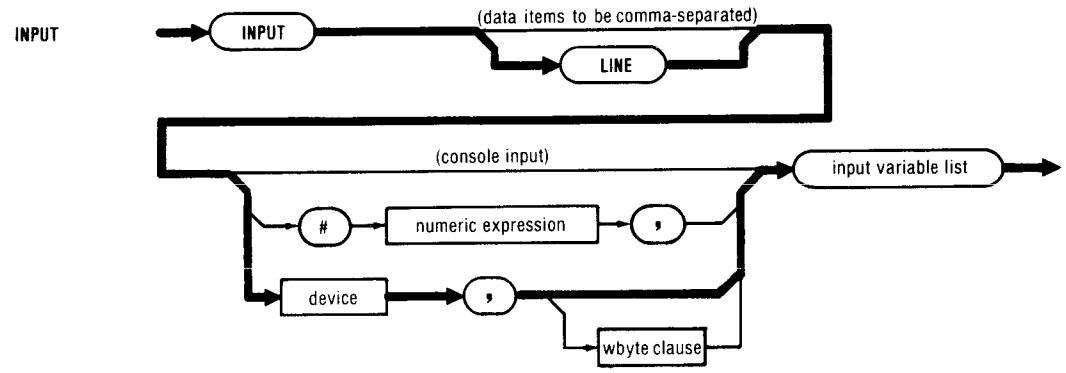
7-52. The following example uses a TERM statement to limit the terminator of the string A$ to the EOI Bus line. This will allow the instrument at device number 10 to transmit binary data or status information.

```
4100  REM -- Get Readings from Instrument
4110  TERM                        ! Limit terminator to EOI
4120  PRINT @ 10, "A3B?"          ! Set up instrument
4130  INPUT @ 10, A$              ! Get data
4140  ! other statements
```

7-53. The next example shows how INPUT can be used without a specific device designated.

```
5000  REM -- Get series of readings
5010  INPUT @ 4, A                  ! Get dummy readings
5020  FOR I=1 TO 100
5030  INPUT @, RC(I)                ! Get 100 readings
5040  NEXT I                        ! from same device
5050  ! other statements
```

7-54. The INPUT LINE Statement



7-55. The INPUT LINE construction of the INPUT statement allows binary data to be received from the bus and assigned to a string variable.

- The only syntax difference in the INPUT LINE statement for IEEE-488 Bus instruments is the use of a device specification instead of a channel number.
- Only one device number may be specified.
- Input data items may be separated by commas or by the terminating character.
- The terminating character is Line Feed, unless an alternate character is specified by a previous TERM statement.
- The EOI Bus line may also be used by the transmitting instrument to terminate input.
- Each input data item to a string variable may be up to 80 characters in length.
- When a device address is given, the instrument is addressed as a talker prior to reading data.
- Each variable or specified array location must receive an entry.
- The INPUT LINE statement can be structured so that the 1720A will not address an instrument that has been previously addressed. This will increase the speed of data input. INPUT LINE assumes there is a talker already on the bus when the @ character follows INPUT LINE without a device specified. Incoming bus data is then simply assigned directly to the specified variables.
- A string assigned a value by INPUT LINE will include the terminating character.
- Other references in this manual: General Purpose Fluke BASIC Statements, Input and Output Statements.

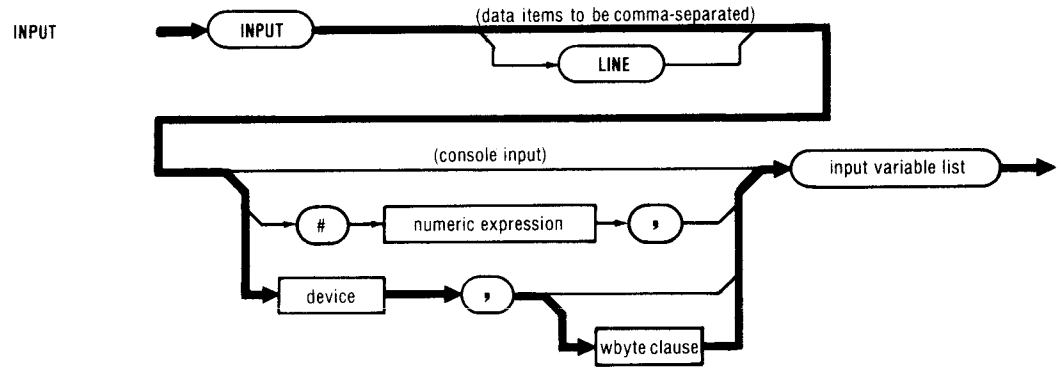
7-56. The following example allows a less restrictive data format than a simple INPUT statement. Any or all of the ASCII floating point values may be terminated by a Line Feed or the EOI line. Otherwise, this example is similar to the first INPUT statement example.

```
INPUT LINE @ 102, A(1%..4%)
```

7-57. INPUT LINE is commonly used to read strings which represent numeric data from instruments. Some instruments, however, transmit data with a left justified sign for ease of direct reading on a printer. The resulting string cannot be directly evaluated. The following subroutine removes the spaces between a sign and a numeric string. The string that was created by INPUT LINE is A$.

```
1000  S% = INSTR (1%,A$,"+")                  ! Search for a +
1010  IF S% = 0 THEN S% = INSTR (1%,A$,"-")   ! If none, search for a -
1020  IF S% THEN S$ = MID(A$, S%, 1%)         ! S$ is sign, if found
1030  S% = S% + 1%
1040  IF ASCII (RIGHT(A$, S%) = 32% THEN 1030! Skip spaces
1050  A$ = S$ + RIGHT (A$, S%)                ! A$ is now sign followed
1060  REM                                     ! by numerics.
```

7-58. The INPUT WBYTE Statement



7-59. INPUT WBYTE transmits a Bus message contained in a WBYTE clause prior to receiving each data item.

- Only one device number may be specified.
- Input data items must be separated by commas.
- The terminating character is Line Feed, unless an alternate character is specified by a previous TERM statement.
- The EOI Bus line may also be used by the transmitting instrument to terminate input.
- The instrument is addressed as a talker only once, prior to all data readings.
- INPUT WBYTE can be structured so that the 1720A will not address an instrument that has been previously addressed. This will increase the speed of data input. INPUT WBYTE assumes there is a talker already on the bus when the @ character follows INPUT without a device specified.

- The WBYTE clause selects a subrange of an integer array for output. This may contain addressing, bus messages and device dependent data. See the discussion of the WBYTE statement in this section.

- The WBYTE clause does not need to be sent to the same instrument port from which input is being taken.

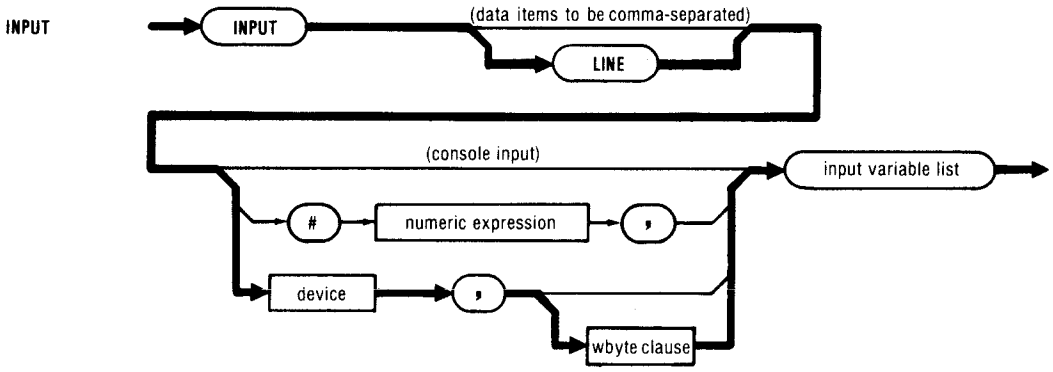
- Make sure that the WBYTE output does not unaddress the instrument as a talker if it is sent to the port from which input is read.

- The WBYTE clause is then sent again and the next input data item is read. This process is repeated until the input data list has been satisfied.

- After the WBYTE output, the specified instrument is addressed as a talker and one data item is read.

- Other references in this manual: General Purpose Fluke BASIC Statements, Input and Output Statements.

7-60. The INPUT LINE WBYTE Statement



7-61. The INPUT LINE WBYTE construction of the INPUT statement allows binary data to be received from the bus and assigned to a string variable, and transmits a Bus message contained in a WBYTE clause prior to receiving each data item.

- Only one device number may be specified.

- Input data items may be separated by commas or by the terminating character.

- The terminating character is Line Feed, unless an alternate character is specified by a previous TERM statement.

- The EOI Bus line may also be used by the transmitting instrument to terminate input.

- Each data item input to a string variable may be up to 80 characters in length.

- The instrument is addressed as a talker only once, prior to all data readings.

- INPUT LINE WBYTE can be structured so that the 1720A will not address an instrument that has been previously addressed. This will increase the speed of data input. INPUT LINE WBYTE assumes there is a talker already on the bus when the @ character follows INPUT LINE without a device specified.

- A string assigned a value by INPUT LINE WBYTE will include the terminating character.

- The WBYTE clause selects a subrange of an integer array for output. This may contain addressing, bus messages and device dependent data. See the discussion of the WBYTE statement in this section.

- The WBYTE clause does not need to be sent to the same instrument port from which input is being taken.

- Make sure that the WBYTE output does not unaddress the instrument as a talker if it is sent to the port from which input is read.

- After the WBYTE output, the specified instrument is addressed as a talker and one data line is read.

- The WBYTE clause is then sent again and the next line of input is read. This process is repeated until the input data list has been satisfied.

- Other references in this manual: General Purpose Fluke BASIC Statements, Input and Output Statements.

7-62. The following example sends a one-character trigger to an already addressed listener on port 1 via the WBYTE clause. It then addresses device 1 on port 0 as a talker. Each floating point ASCII value read from port 0 is placed in an element of A after the trigger command is sent.

```
3750  INPUT LINE @ 1%, {WBYTE PORT 1%, A%(0%)} A(0%..19%)
```

The sequence is:

1. Send A%(0%) on port 1
2. Read A(0%) on port 0
3. Send A%(0%) on port 1
4. Read A(1%) on port 0

. . .

. . .

39. Send A%(0%) on port 1
40. Read A(19%) on port 0

7-63. The following example initializes instrument port 0 and then sends a string of bus messages via the WBYTE clause. The array A% contains the following six commands: A%(0) = UNL, A%(1) = UNT, A%(2) = MLA 2, A%(3) = GET, A%(4) = UNL, and A%(5) = MTA 2. Instrument device 2 on port 0 is triggered fifty times, each time reading a string into the next element of array A$.

```
10    INIT PORT 0%
20    INPUT LINE @, {WBYTE A%(0%..5%)} A$(0%..49%)
```

The sequence is:

1. Send A%(0%..5%)
2. Read A$(0%)
3. Send A%(0%..5%)
4. Read A$(1%)
. . .
. . .
99. Send A%(0%..5%)
100. Read A$(49%)

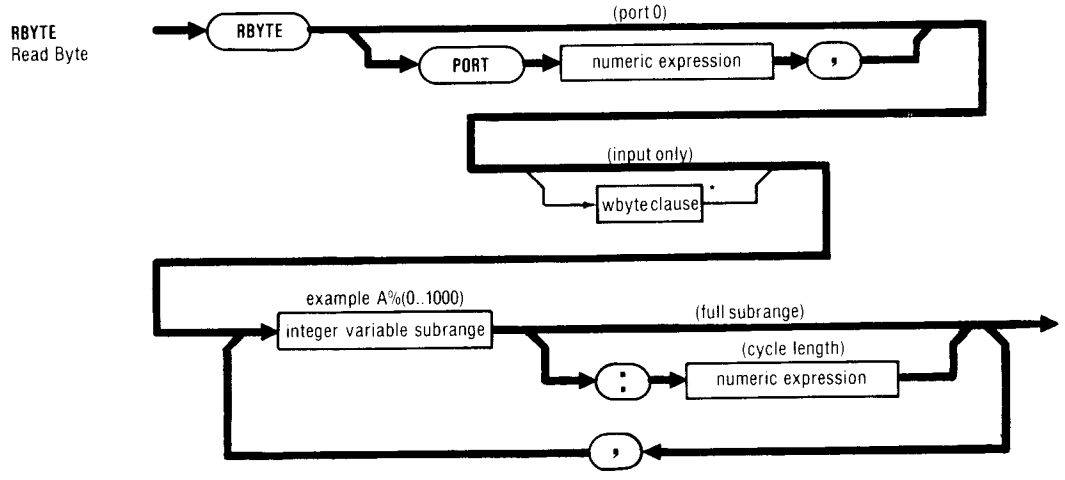
Table 7-2. IEEE-488 Bus Input and Output Statements

STATEMENT	DESCRIPTION
INPUT	Accepts data from an instrument.
INPUT LINE	Accepts a full line of data, including trailing carriage return and line feed.
INPUT WBYTE	Outputs a Bus message and receives data.
INPUT LINE WBYTE	Outputs a Bus message prior to receiving each line of data.
PRINT	Outputs a Bus message or data to instruments.
PRINT USING	Outputs data in the specified format to instruments.

7-64. IEEE-488 DATA TRANSFER STATEMENTS

7-65. The following discussion describes Fluke BASIC statements which provide direct access to the data lines and some of the control lines of the IEEE-488 Instrumentation Bus. These statements allow optimal handling of the binary data sent by some instruments in "high speed" mode. Two of the statements described below directly handle binary floating point information as described in the standard "IEEE Floating Point Arithmetic for Microprocessors". Copies of this standard are available from The Institute of Electrical and Electronic Engineers, 345 East 47th Street, New York, New York, 10017.

7-66. The RBYTE Statement



7-67. RBYTE (Read BYTE) reads a fixed-length block of arbitrary 8-bit binary data bytes from an instrument. The data from the instrument is placed in a specified integer variable array.

- The array must have only a single dimension (subscript).
- The array must be integer type.
- A virtual array may not be used.
- The ATN Bus line is first set false.
- Data is then read into the integer array elements as follows (bit 0 is low-order):

BITS	CONTENTS
0-7	8-bit data byte
8	Set to 1 if EOI was asserted by the talking device with this data byte
9-15	Set to zero

- A data byte sent with EOI asserted will then have the following characteristics:

1. Its numeric value will be greater than 255.
2. A bitwise AND with the 2^8 bit will produce a non-zero value. For example:

```
IF A%(5%) AND 256% THEN ...(EOI is true)
```

- RBYTE performs no device addressing. The WBYTE clause may be used to address an instrument as a talker.

7-68. The following example reads one data byte from port 0 into element 0 of the integer array A%. The instrument has already been addressed to talk.

```
2550  RBYTE PORT 0%, A%(0%)
```

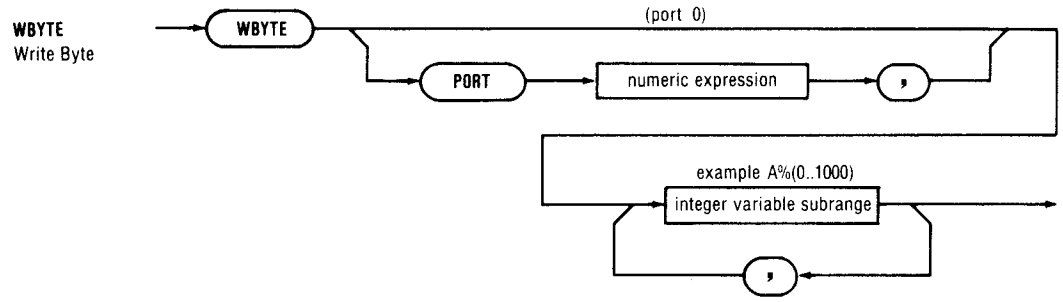
7-69. The following example reads fifteen data bytes from port 0 (the default port) into array A%, elements 0 through 14. Then seven data bytes are read, also from port 0, into elements 12 through 18 of the array C%.

```
5890  RBYTE A%(0%..14%), C%(12%..18%)
```

7-70. The following example uses RBYTE to take 100 readings, each three bytes long. To insure that the readings are correct, each third byte is tested to see that EOI was set.

```
1200  REM -- Subroutine: Get Instrument Reading
1210  ! other statements to set up instrument as talker, etc.
1220  !
1250  ! Now get readings
1260  RBYTE PORT P%, R(1%..300%)
1270  ! Insure that EOI is sent every third byte
1280  EF% = 0                          ! Error Flag = False
1290  FOR I% = 3% TO 300% STEP 3%      ! Error Flag = True?
1300  IF NOT (R(I%) AND 256%) THEN I% = 300% \ EF% = -1
1310  NEXT I%
1320  IF EF% GOTO 1400                 ! Error exit
1330  RETURN
```

7-71. The WBYTE Statement



7-72. WBYTE (Write BYTE) sends an arbitrary set of Bus commands or data bytes to a port. The ATN, EOI and data lines of the instrument bus may be set as desired with this command, with the restriction that ATN and EOI may not be set true simultaneously (since this invokes a parallel poll).

- The array must have only a single dimension (subscript).
- The array must be integer type.
- A virtual array may not be used.
- Data transmitted by WBYTE is taken directly from specified integer arrays (one byte per array element).
- Data within each array element is formatted as follows (bit 0 is low-order):

BITS	CONTENTS
0-7	8-bit data byte
8	Send EOI if set to one
9	Send ATN if set to one
10-15	Ignored

- WBYTE performs no automatic device addressing.
- The data sent from the array, however, may designate talker or listeners.
- WBYTE is used as a clause within some other IEEE-488 Bus control statements. When used as a clause, it is enclosed in curly brackets, as { WBYTE . . . } , and may only have one subrange designated.
- Other references are in this section.

7-73. The following example sends a binary data byte contained in array element 0 of array A% to port 1.

```
7440  WBYTE PORT 1%, A%(0%)
```

7-74. The following example sends seven data bytes from array A% (elements 1 through 7 in order) and two data bytes from array B% (elements 12 and 13) to port 0.

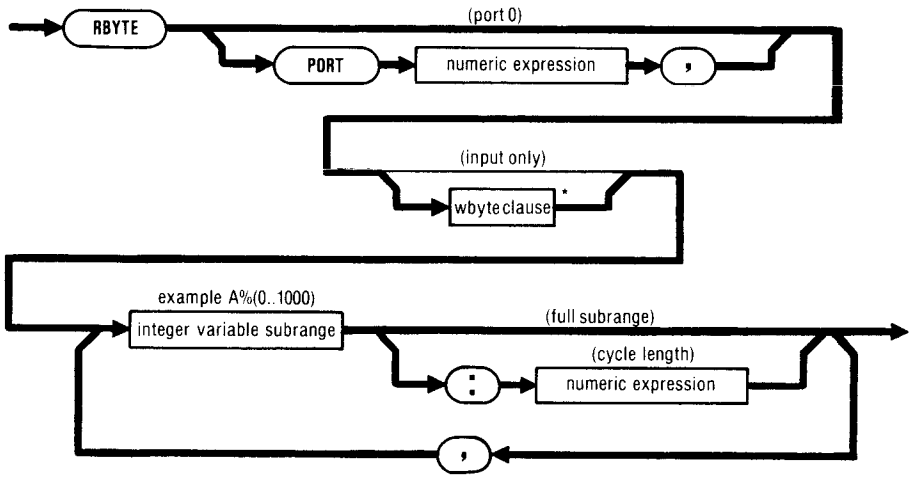
```
30080 WBYTE PORT 0%, A%(1%..7%), B%(12%..13%)
```

7-75. The following example illustrates one way the integer array may be defined and sent on port 0, the default port.

```
110   ! Assign array for programming instrument
120   D%(0) = 512% + 63%          ! ATN and UNL
121   D%(1) = 512% + 95%          ! ATN and UNT
122   D%(2) = 512% + 34%          ! ATN and listen address of 2
123   D%(3) = 63%                 ! "?" for trigger
124   D%(4) = 512% + 95%          ! ATN and UNL
125   D%(5) = 512% + 65%          ! ATN and talk address of 1
130   ! Send data on port 0
140   WBYTE D%(0%..5%)
```

7-76. The RBYTE WBYTE Statement

RBYTE
Read Byte



7-77. The WBYTE clause added to the RBYTE statement provides a means of sending commands or data to a port (via the WBYTE clause) prior to reading the data as specified by RBYTE.

- RBYTE WBYTE can be used for an instrument that requires an explicit trigger for each reading.
- The WBYTE data is sent prior to each RBYTE cycle.
- The cycle length is specified by the value of the expression following the ":" character.
- If no cycle length is specified, it is assumed to be the length of the array subrange received.
- Only one WBYTE subrange may be specified.
- Each array must have only a single dimension (subscript).
- Each array must be of type integer.
- No virtual arrays may be used.
- The total number of bytes read must be evenly divisible by the number of bytes per cycle. An RBYTE data specification A%(1..N):M reads a total of N bytes in N/M cycles of M bytes each. Two examples:

A%(0..20): 7 is a legal RBYTE cycle specification since the total number of bytes received (21) is evenly divisible by the number of bytes per cycle (7).

A%(0..20): 8 is not a legal RBYTE cycle specification since the total number of bytes received (21) is not evenly divisible by the number of bytes per cycle (8). This will cause an error 541.

- Other references in this manual: None.

7-78. The following example transmits the WBYTE data (elements 0 through 4 of array A%) on port 1 prior to reading elements 0 through 5 of array B% (from port 0) and prior to reading elements 0 through 11 of array C% (also from port 0). Note that each array subrange constitutes an RBYTE cycle.

```
6840  RBYTE {WBYTE PORT 1%, A%(0%..4%)} B%(0%..5%), C%(0%..11%)
```

7-79. The following example receives data read from port 0 (elements 0 through 59 of array A%) in 20 cycles of 3 bytes each. The WBYTE data is sent to port 0 prior to each RBYTE cycle. This statement will send an explicit trigger for each reading if the array Z%(0..5) is assigned as the D% array was in the third example of the WBYTE description.

```
5380  RBYTE {WBYTE Z%(0%..5%)} A%(0%..59%):3%
```

The sequence of this statement is:

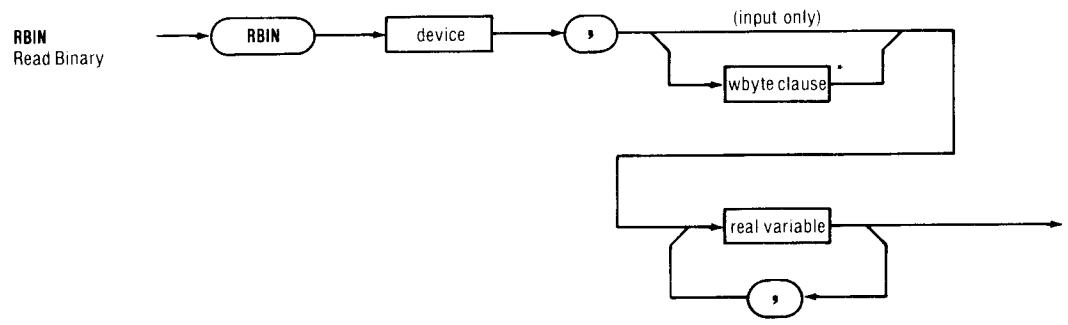
1. Send Z%(0..5)
2. Read A%(0..2) . . . (first 3 elements)
3. Send Z%(0..5)
4. Read A%(3..5) . . . (second 3 elements)
5. Send Z%(0..5)

. . .

. . .

39. Send Z%(0...5)
40. Read A%(57..59) . . . (last 3 elements)

7-80. The RBIN and RBIN WBYTE Statements



7-81. RBIN (Read BINary) receives single and double precision data in IEEE standard floating point format from IEEE-488 Bus instruments.

- The specified instrument device number is addressed as a talker.
- Instrument addressing is skipped when the @ character follows RBIN without a device specified.
- When the statement includes a WBYTE clause, data specified by the WBYTE clause is then sent to the specified port. See the discussion of WBYTE in this section.
- The WBYTE clause selects a subrange of an integer array for output. This may contain addressing, bus messages and device dependent data.
- The WBYTE clause does not need to be sent to the same instrument port from which input is being taken.
- Make sure that the WBYTE output does not unaddress the instrument as a talker if it is sent to the port from which input is read.
- A single floating point value is then received from the specified instrument. The value must be in the selected format.
- :4 specifies a four-byte single precision number.
- :8 specifies an eight-byte double precision number.
- The default format is eight-byte double precision.
- The variable or array subrange to receive the data must be floating point variable type.
- Single precision data is converted to double-precision (internal format) and assigned to the floating point variable. No conversion is necessary for data received in double precision form.
- If additional data is to be received, and a WBYTE clause isspecified, the data specified by the WBYTE clause is sent again prior to each reading.
- Other references in this manual: None.

7-82. The following example addresses device 5 on port 1 as a talker, and then reads six single-precision floating point values from port 1. The single-precision data will be converted to double-precision and assigned to elements 0 through 4 of array A and to variable C.

```
3650  RBIN @ 105%, A(0..4):4, C:4
```

7-83. The following example reads one double-precision value from the port on which IEEE-488 bus I/O was last performed. It assigns the value to variable B. Note that no device addressing is performed since no device address follows the @ character.

```
10940  RBIN @, B:8
```

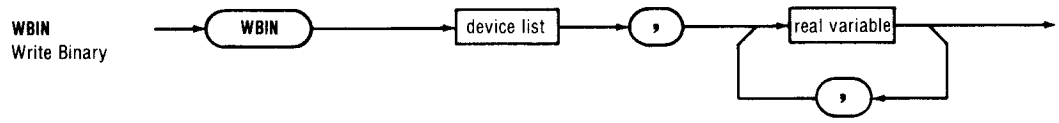
7-84. The following example reads 30 double-precision (8-byte) values from device 20 on port 0. Prior to reading each value from port 0, the WBYTE data (C%(0..5)) will be sent to port 0.

```
4720   RBIN @ 20%, {WBYTE PORT 0%, C%(0%..5%)} B(0%..4%, 0%..5%)
```

The values are assigned to matrix B in the following order:

B(0,0)
B(0,1)
.
.
B(0,5)
B(1,0)
.
.
B(4,4)
B(4,5)

7-85. The WBIN Statement



7-86. WBIN (Write BINary) sends numeric data to an IEEE-488 Bus instrument in single or double precision IEEE standard floating point format.

- The instrument with the specified device number is addressed as a listener.
- Instrument addressing is skipped when the @ character follows WBIN without a device specified.
- Data is then transmitted in the specified format.
- :4 specifies conversion to a four-byte single precision number.
- :8 specifies an eight-byte double precision number (no conversion required).
- The default format is eight-byte double precision.
- Other references in this manual: None.

7-87. The following example addresses device 4 with secondary address 2 on port 0 as a listener, and then transmits the values of B1 and Z (elements I% through J%) in single precision format.

```
4790  WBIN @ 4:2, B1:4, Z(I%..J%):4
```

7-88. The following example addresses device 1 on port 0 and device 20 on port 1 as listeners, and then transmits the specified values from array D in double precision format to both port 0 and port 1 (since both ports have been addressed). Values are transmitted from the array in the same order as those written in the third example for RBIN. Rows are output in column order.

```
WBIN @ 1% @ 120%, D(0%..1%, 5%..7%)
```

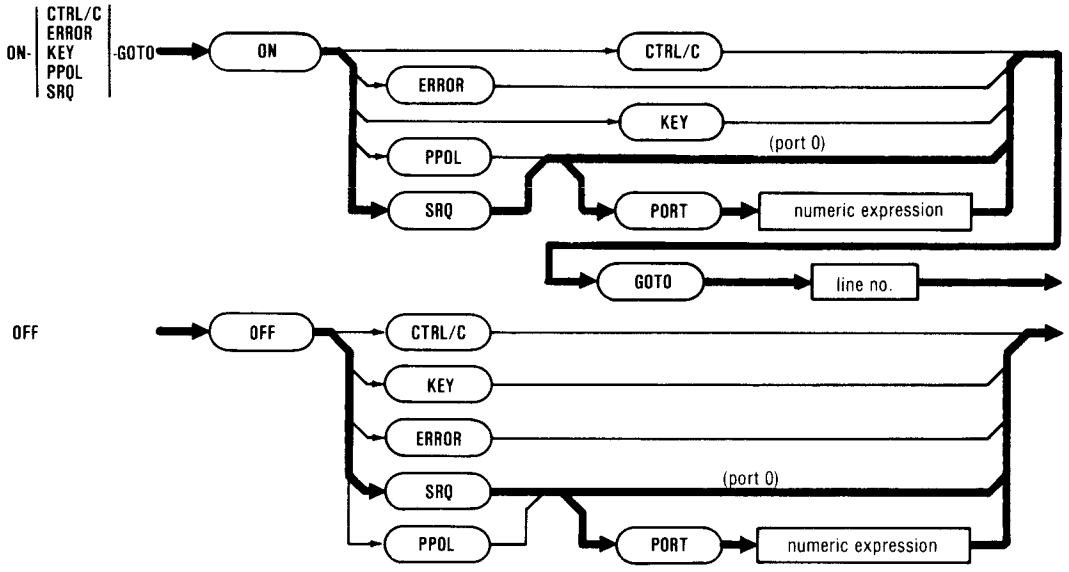
Table 7-3. Data Transfer Statements Summary

STATEMENT	DESCRIPTION
RBIN	Inputs double and single precision data in IEEE standard floating point format.
RBIN WBYTE	Outputs data designated by WBYTE prior to receiving double and single precision data in IEEE standard floating point format.
RBYTE	Inputs binary data bytes from the designated port.
RBYTE WBYTE	Outputs data designated by WBYTE prior to each designated input cycle of binary data bytes.
WBIN	Outputs double and single precision data in IEEE-standard floating point format.
WBYTE	Sends an integer variable array as Bus messages to the designated port.

7-89. IEEE-488 POLLING STATEMENTS

7-90. The statements discussed in the following paragraphs handle the polling, both serial and parallel, of instruments on the IEEE-488 bus.

7-91. The ON SRQ and OFF SRQ Statements



7-93. ON SRQ (ON Service ReQuest) allows a program to branch to a service request routine when an SRQ (Service Request) is received from an external device.

- ON SRQ enables service request interrupt processing.
- OFF SRQ disables service request interrupt processing.
- The specified line number indicates the service request handling routine's starting line number.
- Port 0 is assumed when the port specification is omitted.

- A RESUME statement branches back to the interrupted program.

- During the time interval between the occurrence of the SRQ interrupt and the execution of RESUME, there is no checking for the presence of SRQ.

- The interrupt to the service request routine will repeat immediately after the RESUME statement unless the service request routine polls the instrument and resets the service request.

- Other references in this manual: Interrupt Processing.

7-94. The SRQ handling routine typically performs a serial poll of instruments to locate the instrument requiring attention. The reasons for an instrument to set the SRQ (Service Request) Bus line depend on instrument design. It is often used to signal that the last command has been completed, data is ready, or an error has occurred.

7-95. In the following example, lines 10 and 20 indicate the starting line numbers of the SRQ handling routines (500 for port 0, 800 for port 1). Line 290 disables the ON SRQ response so that the SRQ routines will not be called after this point in the program. The RESUME statements on lines 750 and 890 indicate the end of each SRQ routine. When RESUME is encountered, execution continues at the statement following the one after which program execution was interrupted.

```
10    ON SRQ PORT 0% GOTO 500
20    ON SRQ PORT 1% GOTO 800
30    ! other statements
40    !
290   OFF SRQ \ OFF SRQ PORT 1%
300   ! other statements
310   !
400   STOP
500   ! Port 0 SRQ routine
510   ! other statements (e.g. serial polling)
750   RESUME
800   ! Port 1 SRQ routine
810   ! other statements (e.g. serial polling)
820   !
890   RESUME                  ! Return from interrupt
```

7-96. The SPL Function

Format: SPL (device number)

7-97. SPL (Serial PoLl) performs a serial poll of a specified instrument and returns an integer status byte result. By sequentially serial polling instruments and checking for SRQ in the status bytes, the SRQ routine can determine which instruments set SRQ. By examining the remaining bits of some instruments, it can determine why and take appropriate action.

- SPL is the only way to reset the interrupt status of ON SRQ.

- The result will be from 0 to 255.

- SPL may be performed at any time, although it is normally used in an SRQ handling routine.

- When an instrument is serial polled it returns a status byte. Bit 6 of the status byte is set to 1 if the polled instrument is the one that requested service.

- The remaining bits may indicate other status information of the instrument. Consult the manual of the instrument for their meanings.

- The instrument asserting SRQ true should deassert SRQ when it is serial polled.

- More than one instrument can hold SRQ true at the same time.

- Refer to Appendix I, ASCII/IEEE-488 Bus Codes, for a chart of binary bytes and decimal numbers. To use this chart, read the value returned by SPL in the decimal column, and read the status byte in the binary column.

- The Request Service bit of the serial poll response may be tested by:

```
IF SPL(device) AND 64% THEN ...
```

- Other references in this manual: Interrupt Processing.

7-98. The following example performs a serial poll on device 16 with secondary address 13 (on port 0). Then the statement(s) following THEN are executed if a non-zero result is returned by the instrument.

```
5390   IF SPL(16%:13%) THEN ...
```

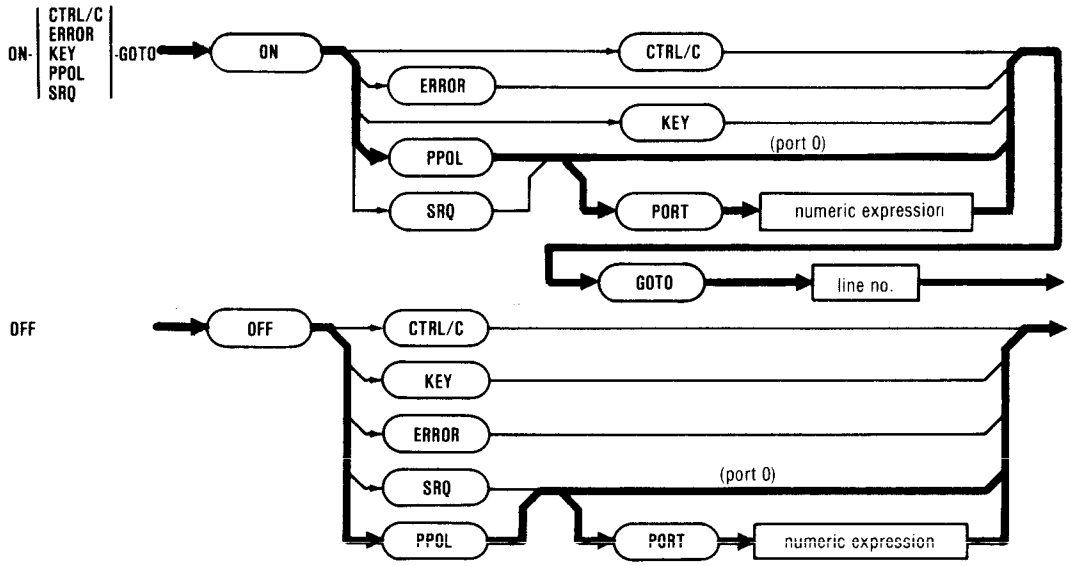
7-99. The following example assigns the result of a serial poll of device DV% on port P% to the variable Y%.

```
6820   Y% = SPL(P% * 100% + DV%)
```

7-100. As shown in Appendix I, 64 has the binary pattern 0100 0000 (bit 6 set). The following example uses AND 64% with the value of SPL to see which device caused the service request. The value of SPL is first saved in Y% and then manipulated several times in routine 5110 - 5190. This is necessary since the instrument asserting SRQ only gives its status once for this service request. Line 5130 checks for other status byte information.

```
5100  REM -- SRQ Handler - Port 0
5110  Y% = SPL(VM%)                    ! Voltmeter handling
5120  IF NOT (Y% AND 64%) THEN 5200
5130  Y1% = Y% AND (NOT 64%)           ! Get status apart from SRQ
5140  ! other statements to handle
5150  ! voltmeter status
5190  RESUME
5200  Y% = SPL(CN%)                    ! Counter handling
5220  IF NOT (Y% AND 64%) THEN 5300
5230  ! other statements to handle
5240  ! counter status
5290  RESUME
5300  ! statements for other instruments
```

7-101. The ON PPOL and OFF PPOL Statements



7-102. ON PPOL (ON Parallel POLl) causes periodic parallel polls to be performed on the specified port.

- ON PPOL enables parallel polling.
- OFF PPOL disables parallel polling.
- If no port is specified, port 0 is assumed.
- Parallel poll is done after each program statement.
- Parallel polling is halted by any ON GOTO interrupt. The next RESUME statement causes parallel polling to continue.
- If the result of a poll is not zero, control is passed to the specified line number.
- The line number indicates the beginning of a parallel poll handling routine.
- RESUME returns control to the next statement after the one completed when the interrupt occurred.

7-103. In the following example, line 10 indicates that the PPOL handling routine starts at line 1000. At line 550, the PPOL is disabled by OFF PPOL, and the program is halted by STOP.

```
10      ON PPOL GOTO 1000           ! Poll port 0
20      ! other statements
30      !
550     OFF PPOL \ STOP
1000    ! Port 0 PPOL handler
1010    ! other statements
1020    !
1070    RESUME                      ! Return from PPOL handler
```

7-104. The PPL Function

Format: PPL (port number)

7-105. PPL (Parallel PoLl) performs a parallel poll of a specified instrument bus port and returns an integer result.

- The result is an integer value between 0 and 255.

- Refer to Appendix I, ASCII/IEEE-488 Bus Codes, for a chart of binary patterns and decimal numbers. To use this chart, read the value returned by PPL in the decimal column, and read the binary pattern in the binary column.

- The correspondence between the IEEE-488 DIO lines and binary bit numbers is as follows:

DI0 Line	Bit Number	Numeric Weight
DI01	0	1
DI02	1	2
DI03	2	4
DI04	3	8
DI05	4	16
DI06	5	32
DI07	6	64
DI08	7	128

- Other references in this manual: CONFIG statement, this section.

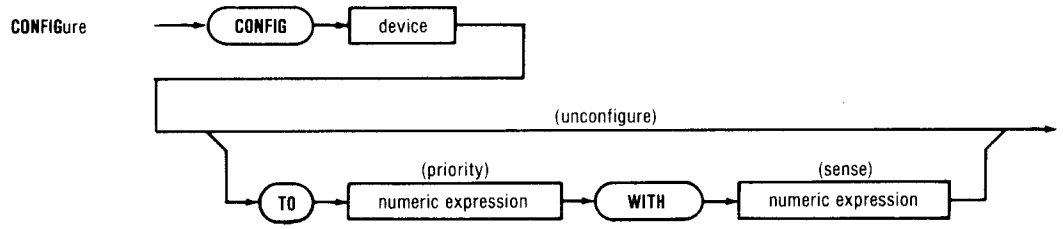
7-106. The following example assigns the results of a parallel poll of port 0 to the integer variable Y%.

```
4710  Y% = PPL(0%)
```

7-107. A bitwise AND may be used to test individual bits of the parallel poll result. The following example performs a parallel poll of port 1. If the DI08 line were asserted true, the statement(s) following THEN would be executed.

```
8460  IF PPL(1%) AND 128% THEN ...
```

7-108. The CONFIG Statement



7-109. CONFIG (CONFIGure) will either configure or unconfigure an instrument for parallel poll.

- TO specifies the DI0 line on which the instrument should respond to a parallel poll.

- WITH clause specifies the active sense (0 or 1) the instrument should use in responding to the poll.

- If the 'TO line WITH sense' clause is omitted, a PPD (Parallel Poll Disable) message will be sent to the instrument.

- Several instruments may be configured to respond with the same sense (1 or 0) on the same DI0 line.

- If the sense is 1, then their poll bits are logically ORed.

- If the sense is 0 then their poll bits are logically ANDed.

- Use the OR configuration to determine whether any instrument is busy or needing service.

- Use the AND configuration to determine if all instruments are busy or needing service.

- Other references in this manual: PPL Function, this section.

7-110. The following example configures device 4 on port 0 to respond on DI02 with a 1 as affirmative poll response.

```
14070  CONFIG @ 4 TO 2 WITH 1
```

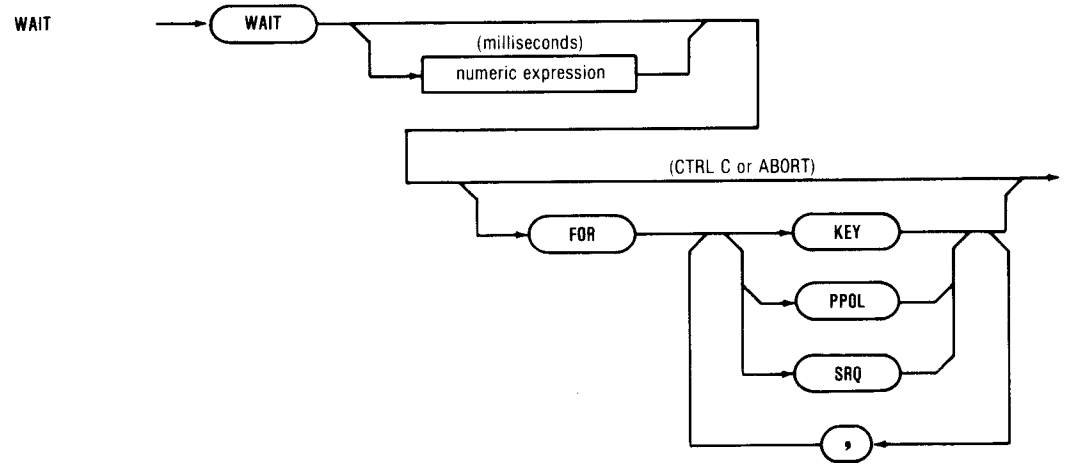
7-111. The following example configures device 21 on port 1 to respond on DI07 with a zero as affirmative poll response.

```
26480  CONFIG @ 121 TO 7 WITH 0
```

7-112. The following example disables device 4 on port 0 from responding to parallel polls.

```
580   CONFIG @ 4
```

7-113. The WAIT Statement



7-114. WAIT suspends program execution until either the specified time period elapses, or until an enabled interrupt occurs. By specifying an event to wait for (KEY, PPOL, or SRQ), an interrupt can be enabled for only the duration of the waiting period. (See the Touch Sensitive Display section for a discussion of KEY.)

- Wait time is indefinite if a time is not specified.

- The Wait time (expressed in milliseconds) should be a positive integer, or an expression that evaluates to a positive integer. The WAIT statement is ignored if the wait time is zero or negative.

- The minimum wait time is 10 milliseconds.

- Clock resolution is 10 milliseconds.

- Interrupt processing that has been previously activated by ON SRQ, ON PPOL, or ON KEY remains active, capable of terminating the wait.

- When an interrupt that has been enabled by an ON statement occurs, the program branches to the interrupt handling routine. The RESUME statement at the end of that routine branches back to the statement following WAIT.

- If one or more events are specified after FOR, these interrupts are enabled only for the duration of the wait period, unless already previously enabled. Occurrence of the specified event (KEY, PPOL, or SRQ) then terminates the wait.

- CTRL C or the ABORT switch will terminate the WAIT. If the program does not include a CTRL C handler, the program will be halted and control returned to Immediate Mode.

- Other references in this manual: Interrupt Processing, Touch Sensitive Display.

7-115. The following example requires the system to halt execution for 500 milliseconds or until an SRQ is received from any Bus port.

```
4490  WAIT 500 FOR SRQ
```

7-116. The following example waits an indefinite period for a Touch-Sensitive Display response from an operator, or for a non-zero parallel poll response on any port.

```
6970  WAIT FOR KEY, PPOL
```

7-117. The following example illustrates the use of WAIT in requiring program execution to halt for the period of time, in milliseconds, supplied by the arguments.

```
580    WAIT D%
```

Table 7-4 IEEE-488 Polling Statements Summary

STATEMENT	DESCRIPTION
CONFIG	Configures an instrument for a parallel poll.
ON PPOL	Enables automatic parallel polls.
OFF PPOL	Disables automatic parallel polls.
ON SRQ	Identifies a service request handling routine and enables the controller to respond to SRQ.
OFF SRQ	Disables service request response.
PPL Function	Returns parallel poll value.
SPL Function	Returns serial poll value.
WAIT	Halts execution for the specified time or until an enabled interrupt occurs.

Section 8
Interrupt Processing

8-1. INTRODUCTION

8-2. This section describes statements and functions which enable and process interrupts to a BASIC program. Interrupts are a response to events that may occur during normal program execution. For example, if the operator determines that an instrument test is not performing properly because of an outside influence (such as the IEEE-488 Bus cables not connected properly), the ABORT switch can be pressed, to terminate the test. Since an operator may not have access to the programming keyboard, the test program must have the ability to analyze conditions, make appropriate responses, and to restart. Event interrupt processing tasks are determined by the individual requirements of the system. Refer also to the IEEE-488 Bus Input and Output Statements section for further information on IEEE-488 Bus Polling statements.

8-3. OVERVIEW

8-4. Interrupt processing is discussed in three subject areas. First is a discussion of the technique of enabling the 1720A to respond with a predefined section of program code at any random time that an interrupting event occurs. Next is a discussion of the method of stopping program execution at a particular point to wait for an event to occur before proceeding. The final subject area is a discussion of the handling of errors. Program examples illustrate the concepts used.

8-5. Types of Interrupts

8-6. Fluke BASIC recognizes five types of interrupts: ERROR, CTRL C, KEY (the touch sensitive display), SRQ (service request), and PPOL (parallel poll).

8-7. The Error Interrupt

8-8. An error interrupt occurs when errors are detected during program execution. Errors are divided into three levels that differentiate the types of response possible.

- Fatal — Immediately terminates the program.
- Recoverable — Completes the statement or expression which caused it, then terminates the program unless acknowledged.
- Warning — Program continues running even if interrupt is not enabled.

8-9. The CTRL C Interrupt

8-10. A CTRL C interrupt may occur from either of two sources:

- Press the ABORT button on the front panel.
- Enter CTRL C on the programmer keyboard.

8-11. The KEY Interrupt

8-12. A KEY interrupt occurs whenever the Touch-Sensitive Display is touched.

8-13. The SRQ Interrupt

8-14. An SRQ interrupt occurs when an instrument on the selected IEEE-488 port issues a service request. See the IEEE-488 Bus Input and Output Statements section for additional information.

8-15. The PPOL Interrupt

8-16. A PPOL interrupt occurs when the parallel poll response from instruments on the selected IEEE-488 Bus port is a non-zero value. See the IEEE-488 Bus Input and Output Statements section for additional information.

8-17. Categories of Interrupts

8-18. There are two categories of interrupts. The category is determined by the method used to enable interrupts.

- On-event interrupts are explicitly acknowledged by user-written program code. These preempt the second category.
- Wait-for-event interrupts are implicitly acknowledged by their occurrence. Program flow then continues where it was suspended.

8-19. Interrupt Hierarchy

8-20. Interrupts have a hierarchial relationship that avoids conflicts when two or more interrupts become simultaneously active.

- ERROR and CTRL C share top priority. They preempt acknowledgement of each other and all other interrupts.
- Other interrupts that subsequently occur are not recognized until after the ERROR or CTRL C is acknowledged.
- There are two differences between ERROR and CTRL C.

  1. If a second ERROR occurs before the first is acknowledged, the program terminates immediately.
  2. If a second CTRL C occurs before the first is acknowledged, the second is ignored.

- Second priority is KEY.
- Third priority is SRQ.
- Fourth priority is PPOL.
- KEY, SRQ, and PPOL do not preempt each other. However, they may be preempted by either ERROR or CTRL C.

8-21. ON-EVENT INTERRUPTS

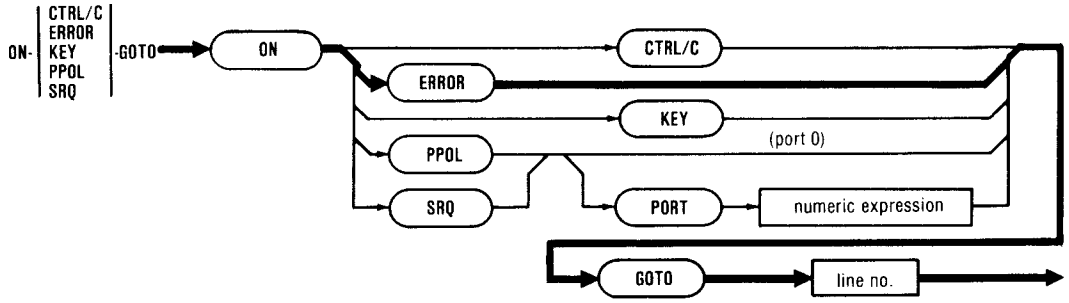
8-22. On-event interrupts enable the 1720A to respond to events that occur at random times that cannot be known when the program is written.

- There are three actions that occur with on-event interrupts:
- Interrupts are initially disabled.
- The interrupt must first be enabled (by ON event GOTO line number) to allow the interrupting event to redirect program sequence.
- After completing response to the interrupt that occurred, the interrupt must be acknowledged with a RESUME statement.
- If interrupt response will not be required any longer, it may then be disabled with an OFF statement.
- Any or all interrupting conditions may be activated simultaneously.
- Table 8-1 summarizes these actions.

Table 8-1. On-Event Interrupt Statements

Action	Statement
Enable	On interrupt name GOTO line number
Acknowledge	RESUME (line number)
Disable	OFF interrupt name

8-23. The ON ERROR GOTO Statement



8-24. The ON ERROR GOTO statement enables a program to respond to a random occurrence of an error condition by transferring control to a specified routine containing a user defined response.

- When an error is detected, control transfers to the specified line number immediately.
- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.
- Only level R and W errors can be processed by an error routine. Level F errors always terminate the program.

- Level R interrupts will terminate a program unless an ON ERROR GOTO statement has been executed so that the error condition can be treated.

- Without error processing, a level W error is ignored.

- When an error condition has been detected, further checking for interrupt conditions other than ERROR or CTRL/C is suspended until a RESUME is executed.

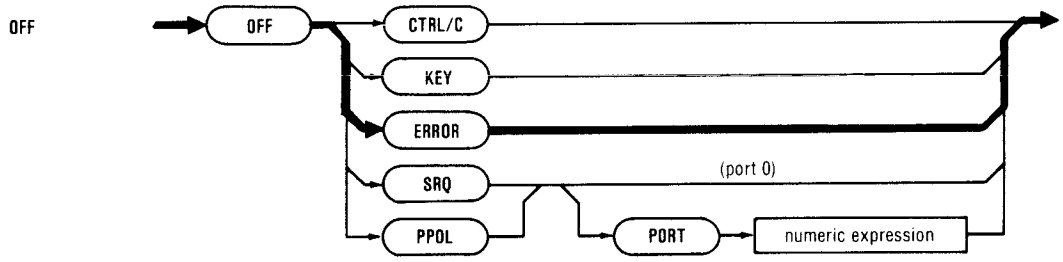
- When an error condition has been detected, the system variable ERL will contain the line number at which the error occurred, and ERR will contain the error number.

- If a second error is detected before encountering a RESUME statement, the program terminates immediately.

- If a CTRL/C interrupt occurs after error detection and before encountering a RESUME statement, error processing is suspended either temporarily or permanently. If the program includes CTRL/C interrupt processing with a RESUME statement, control will be returned to the error processing routine when CTRL/C processing is completed. See the ON CTRL/C GOTO statement in this section.

- Other references in this manual: None.

8-25. The OFF ERROR Statement



8-26. The OFF ERROR statement disables the action of a previous ON ERROR GOTO statement.

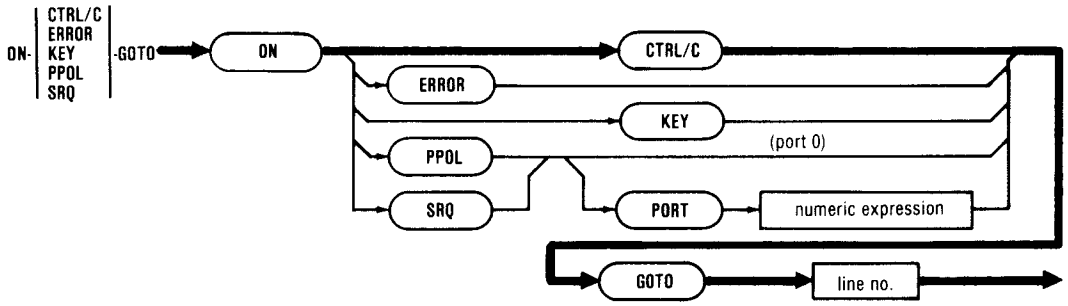
- An OFF ERROR statement in an error processing routine will terminate the program.

- After an OFF ERROR statement has been executed, a level R error will terminate the program.

- After an OFF ERROR statement, a level W error will be ignored.

- Other references in this manual: None.

8-27. The ON CTRL/C GOTO Statement



8-28. The ON CTRL/C GOTO statement enables a program to respond to a random occurrence of an ABORT switch or a CTRL C keyboard entry by transferring control to a specified program routine containing a user defined response.

- When an ABORT switch or a CTRL/C keyboard entry is detected, control transfers to the specified line number.

- The CTRL/C handling routine must explicitly acknowledge the interrupt with a RESUME statement.

- BASIC normally responds to the ABORT switch or a CTRL C keyboard input by terminating the program and returning to Immediate Mode. This statement alters the normal interpreter response.

- When a CTRL/C has been detected, further checking for interrupt conditions other than ERROR is suspended until RESUME is encountered.

- If a second CTRL/C is detected before encountering a RESUME statement, it is ignored.

- A RESUME statement will return control to execute the first statement not completed when the CTRL/C or ABORT key entry was detected.

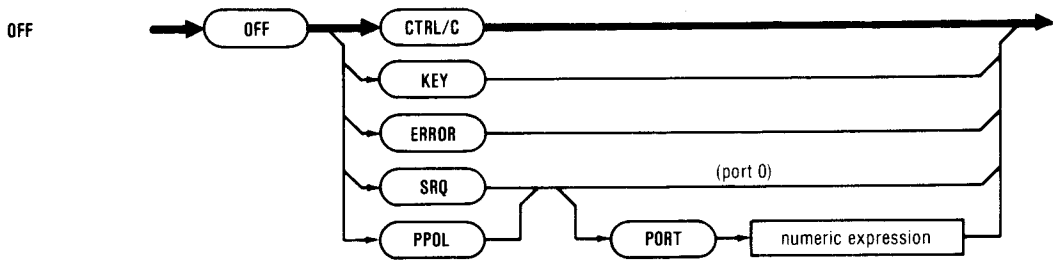
- If a level R or W error occurs after CTRL/C detection and before encountering a RESUME statement, CTRL/C processing is suspended either temporarily or permanently. If the program includes error processing, control will be returned to the CTRL/C processing routine when error processing is completed. Without error processing, a level W error is ignored. See the ON ERROR GOTO statement in this section.

- Other references in this manual: Introduction.

NOTE

Since CTRL/C is the only way to manually stop a BASIC program without deleting it, if not handled properly, a CTRL/C interrupt to an ON CTRL/C subroutine can lock a program into Run Mode.

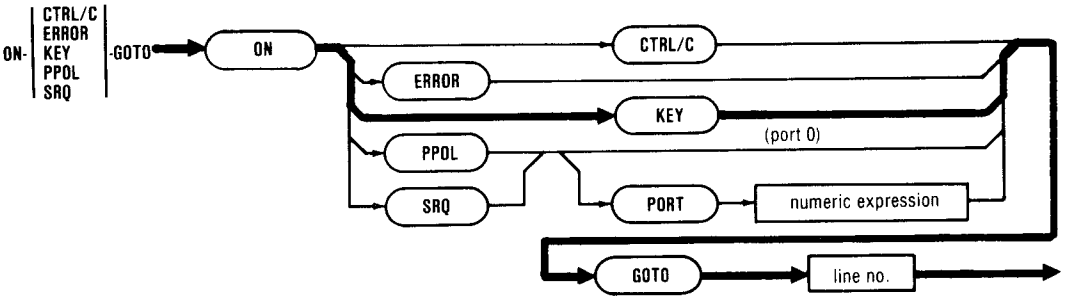
8-29. The OFF CTRL/C Statement



8-30. The OFF CTRL/C statement disables the action of a previous ON CTRL/C GOTO statement.

- An OFF CTRL/C statement in a CTRL/C interrupt processing routine will cause the 1720A to return to Immediate Mode.
- After an OFF CTRL/C statement, the ABORT switch or a CTRL C keyboard entry will return the controller to Immediate Mode.
- Other references in this manual: None.

8-31. The ON KEY GOTO Statement



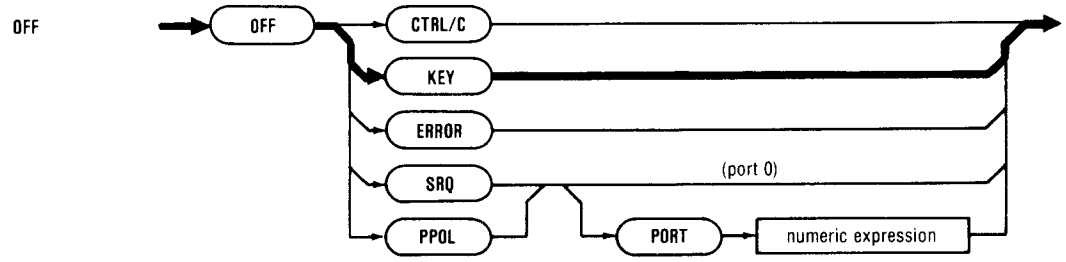
8-32. The ON KEY GOTO statement enables a program to respond to the occurrence of a key entry on the touch-sensitive display by transferring control to a specified program routine containing a user defined response.

- When a key entry is detected, control transfers to the specified line number after completion of the current statement.
- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.
- When a KEY entry has been detected, further checking for interrupt conditions other than ERROR or CTRL/C is suspended until RESUME is encountered.
- When a KEY entry has been detected, the system variable KEY will contain the number of the last touch key pressed.
- The system variable KEY is set whenever the touch sensitive display is pressed in an active area, regardless of whether ON KEY GOTO is used. It remains set until it is read by a program statement. (For example, K% = KEY)
- If the system variable KEY is non-zero when ON KEY GOTO is executed, control is immediately transferred to the specified line number.

- ON KEY does not reset the system KEY variable.

- Other references in this manual: The Touch Sensitive Display.

8-33. The OFF KEY Statement



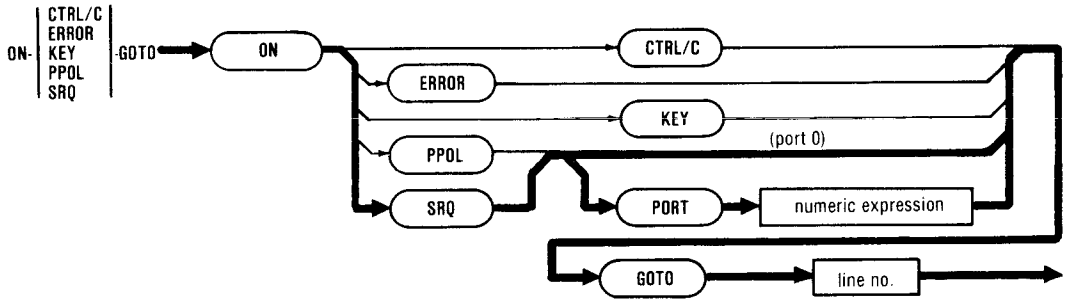
8-34. The OFF KEY statement disables the action of a previous ON KEY GOTO statement.

- An OFF KEY statement in a key interrupt processing routine will prevent the routine from being continuously re-entered if the KEY buffer is not reset in the routine.

- An OFF KEY statement in any interrupt processing routine will not have any additional effect.

- Other references in this manual: None.

8-35. The ON SRQ GOTO Statement



8-36. The ON SRQ GOTO statement enables a program to respond to the occurrence of a service request from an instrument by transferring control to a specified program routine containing a user defined response.

- The specified port is sampled after the completion of each statement. If a port is not specified, port 0 is sampled.

- When a service request is detected, control transfers to the specified line number after completion of the current statement.

- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.

- When a service request has been detected, further checking for interrupt conditions other than ERROR or CTRL/C is suspended until RESUME is encountered.

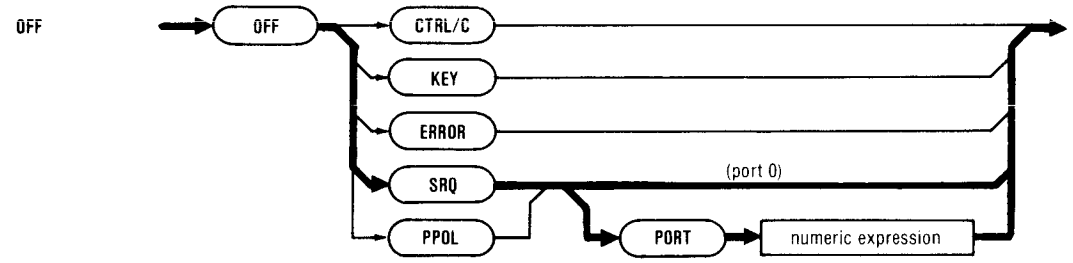
- An internal SRQ flag is set by a service request on the enabled port. It is reset in the controller by performing a serial poll on any instrument on the port requesting service (for example, Y% = SPL(10)). However, depending on the instrument, SRQ will probably be set again until the instrument requesting service is serial polled. This will cause the service request routine to be immediately re-entered after the RESUME statement.

- ON SRQ GOTO does not reset the internal SRQ flag.

- When SRQ's are present on both port 0 and port 1 simultaneously, the SRQ on post 0 will be responded to first.

- Other references in this manual: IEEE-488 Input and Output Statements.

8-37. The OFF SRQ Statement



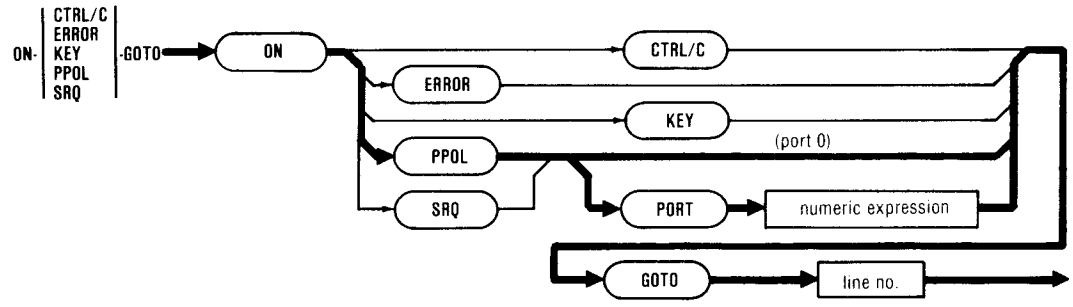
8-38. The OFF SRQ statement disables the action of a previous ON SRQ GOTO statement.

- An OFF SRQ statement in a service request processing routine will prevent the routine from being continuously re-entered if it does not reset the service request by performing a serial poll.

- An OFF SRQ statement in any interrupt processing routine will not have any additional effect.

- Other references in this manual: None.

8-39. The ON PPOL GOTO Statement



8-40. The ON PPOL GOTO statement enables a program to respond to a positive parallel poll response from a configured instrument by transferring control to a specified program routine containing a user defined response.

- The ON PPOL GOTO statement initiates parallel polling on the specified port, or on Port 0 if not specified. A poll will be performed following the completion of each BASIC statement.

- When positive response to a parallel poll is detected, control transfers to the specified line number after completion of the current statement.

- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.

- When a positive response to a parallel poll has been detected, further checking for interrupt conditions other than ERROR or CTRL/C is suspended until RESUME is encountered.

- If both Port 0 and Port 1 have RPOL interrupts enabled, port 0 will be checked for a parallel poll response prior to checking Port 1.

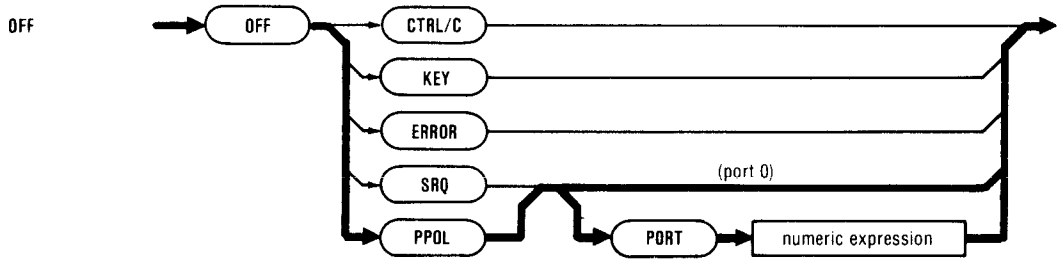
- Other references in this manual: IEEE-488 Input and Output Statements.

NOTE

Some instruments clear a parallel poll bit when the condition causing it disappears, or when the bus port is parallel polled.

When possible, the instrument responding to the parallel poll should be programmed within the processing routine to reset its poll response bit. If this bit remains set, the routine will be immediately re-entered after the RESUME statement.

8-41. The OFF PPOL Statement



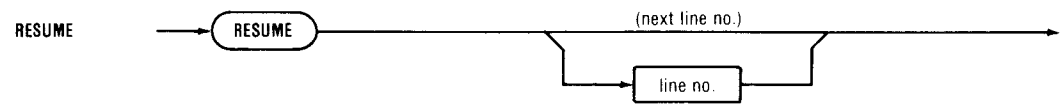
8-42. The OFF PPOL statement disables the action of a previous ON PPOL GOTO statement.

- An OFF PPOL statement in a parallel poll response routine will prevent the routine from being immediately re-entered after the RESUME statement if the routine does not clear the instrument poll response bit.

- An OFF PPOL statement in any interrupt processing routine will not have any additional effect.

- Other references in this manual: IEEE-488 Input and Output Statements.

8-43. The RESUME Statement



8-44. The RESUME statement acknowledges an interrupt and allows program operation to resume with the next statement after the one being completed when the interrupt occurred, or at another specified program location.

- RESUME (no line number) branches to the statement following the one being executed at the point the interrupt occurred.

- If the interrupt occurred in a multiple statement line, the program resumes with next statement on the line.

- There are two exceptions:

  1. Recoverable errors: The program resumes at the beginning of the statement that caused the error.

  2. Input Warning errors 801, 802, and 803: The INPUT statement which caused the error requests the value to be entered again. It did not accept the erroneous entry.

- RESUME (line number) branches to the specified line number.

- RESUME terminates the interrupt handler routine.

- Other references in this manual: None.

8-45. WAIT FOR EVENT INTERRUPTS

8-46. The WAIT (time) (FOR event) statement suspends program execution until the specified interrupt event occurs, or the specified time elapses.

- The interrupt is implicitly acknowledged by its occurrence.

- WAIT may be followed by a numeric expression specifying a period of time to wait for the interrupt.

- When the specified time elapses, interrupt checking stops and the program continues with the next statement.

- An interrupt previously enabled by an ON-GOTO statement remains enabled during the waiting period, whether or not the WAIT statement references it.

- WAIT interrupts have four forms as shown in Table 8-1, with their meanings.

- Each construction of the WAIT statement is separately discussed below.

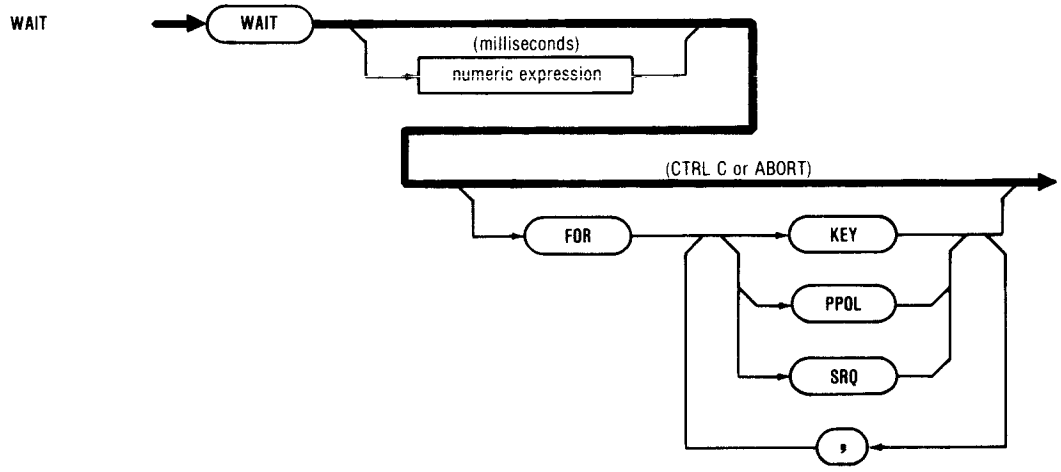
Table 8-1. Wait For Event Interrupt Forms

STATEMENT FORM	MEANING
WAIT	Suspend program execution until a CTRL/C or an interrupt enabled by ON-GOTO occurs.
WAIT numeric expression	Suspend program execution, up to the specified time limit until CTRL/C or an interrupt enabled by ON-GOTO occurs.

Table 8-1 Wait For Event Interrupt Forms (cont.)

STATEMENT FORM	MEANING
WAIT FOR (KEY)(,)(PPOL)(,)(SRQ)	Suspend program execution until CTRL/C, the specified interrupt, or an interrupt enabled by ON-GOTO occurs.
WAIT numeric expression FOR (KEY)(,)(PPOL)(,)(SRQ)	Suspend program execution, up to the specified time limit, until CTRL/C, the specified interrupt, or an interrupt enabled by ON-GOTO occurs.

8-47. The WAIT Statement



8-48. The WAIT statement suspends operation of the program indefinitely.

- The ABORT switch, or a CTRL/C keyboard entry, will always terminate the WAIT.

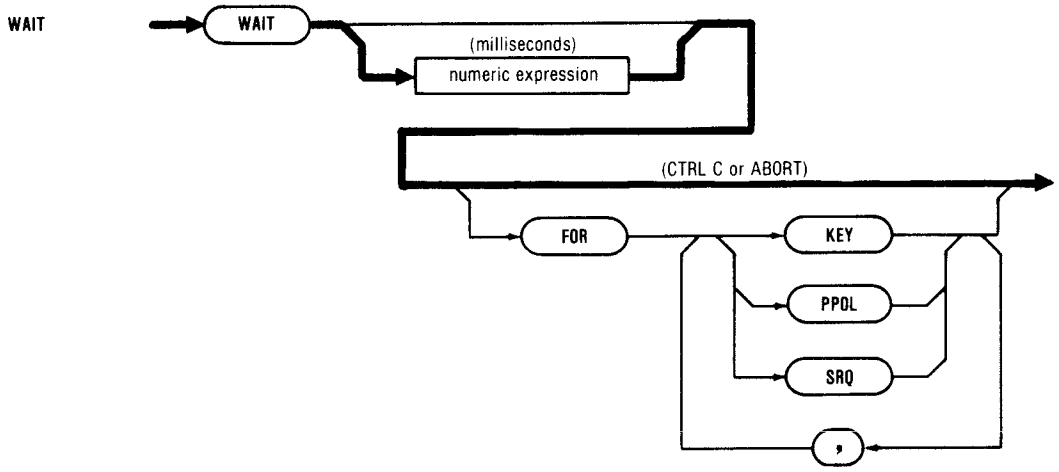
- If an ON CTRL/C GOTO statement has been executed, the ABORT switch, or a CTRL/C keyboard entry will terminate the WAIT and transfer control to the CTRL/C processing routine.

- The WAIT statement takes the place of the (not allowed) construct WAIT FOR CTRL/C, since this top priority interrupt always remains enabled.

- A KEY, SRQ, or PPOL interrupt will not terminate the WAIT unless previously enabled by ON-event GOTO.

- Other references in this manual: None.

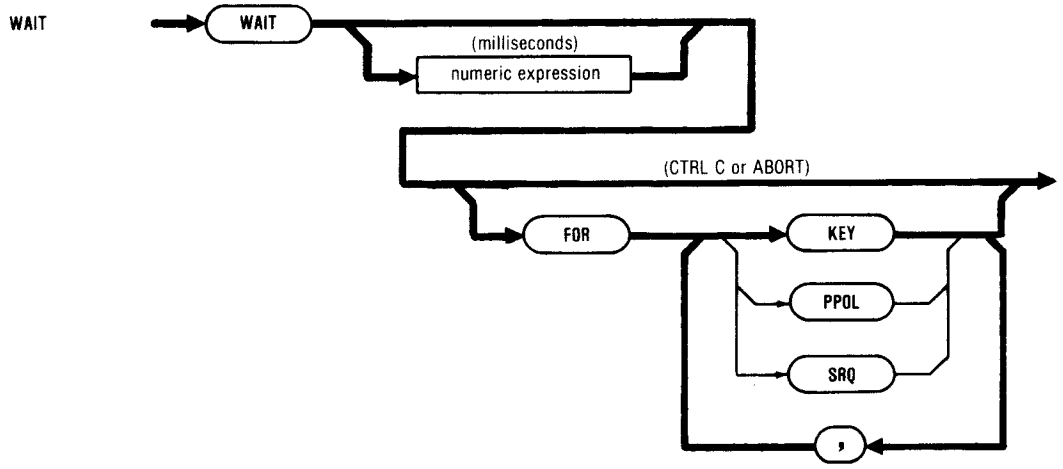
8-49. The WAIT Time Statement



8-50. The WAIT Time statement suspends operation of the program for the specified length of time.

- The length of time is specified in milliseconds as an integer number or numeric expression.
- Timer resolution is 10 milliseconds.
- A negative time value results in a waiting time of 0.
- Maximum wait time input is 32767 milliseconds. This results in a wait time of 32.77 seconds.
- Until the specified time period has elapsed, the WAIT can be terminated in any of the ways defined above under the WAIT statement.
- Other references in this manual: None.

8-51. The WAIT FOR KEY Statement



8-52. The WAIT FOR KEY statement suspends operation of the program indefinitely until the touch sensitive display is pressed in the active area.

- The WAIT can be terminated in any of the ways defined above under the WAIT statement.

- A time limit may be specified following the word WAIT as described above under the WAIT Time Statement.

- The WAIT is terminated whenever the touch sensitive display is pressed in the active area.

- A touch key input causes the program to continue with the next statement unless a previous ON KEY GOTO statement has been executed.

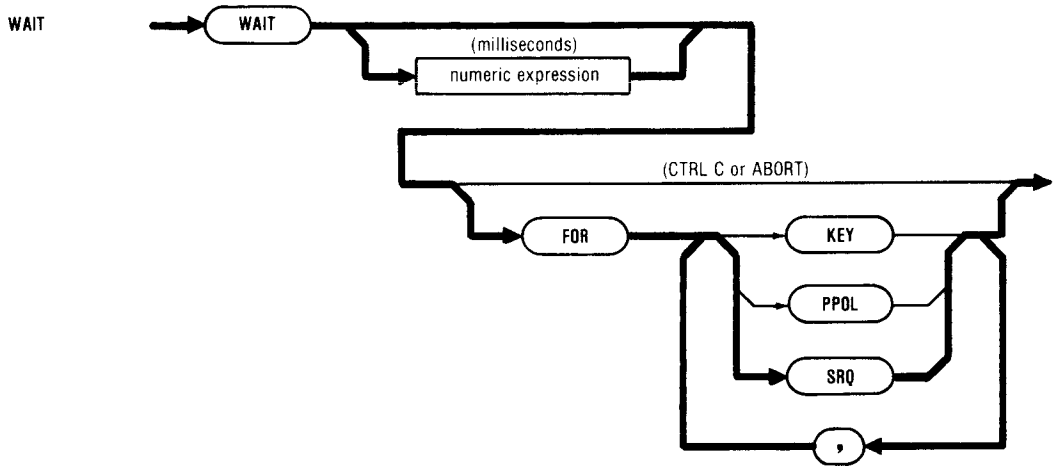
- When a previous ON KEY GOTO statement has been executed, a touch key input causes the program to transfer to the KEY processing routine.

- The system variable KEY is set whenever the touch sensitive display is pressed in an active area, regardless of whether WAIT FOR KEY is used. It remains set until it is read by a program statement. (For example, K% = KEY)

- Wait time is 0 if the system variable KEY is non-zero when WAIT FOR KEY is executed.

- Other references in this manual: None.

8-53. The WAIT FOR SRQ Statement



8-54. The WAIT FOR SRQ statement suspends operation of the program indefinitely until a service request is detected on either instrument port.

- The WAIT can be terminated in any of the ways defined above under the WAIT statement.

- A time limit may be specified following the word WAIT as described above under the WAIT Time Statement.

- The WAIT is terminated whenever a service request is detected on either instrument port.

- A service request causes the program to continue with the next statement unless a previous ON SRQ GOTO statement has been executed.

- When a previous ON SRQ GOTO statement has been executed, a service request causes the program to transfer to the service request processing routine.

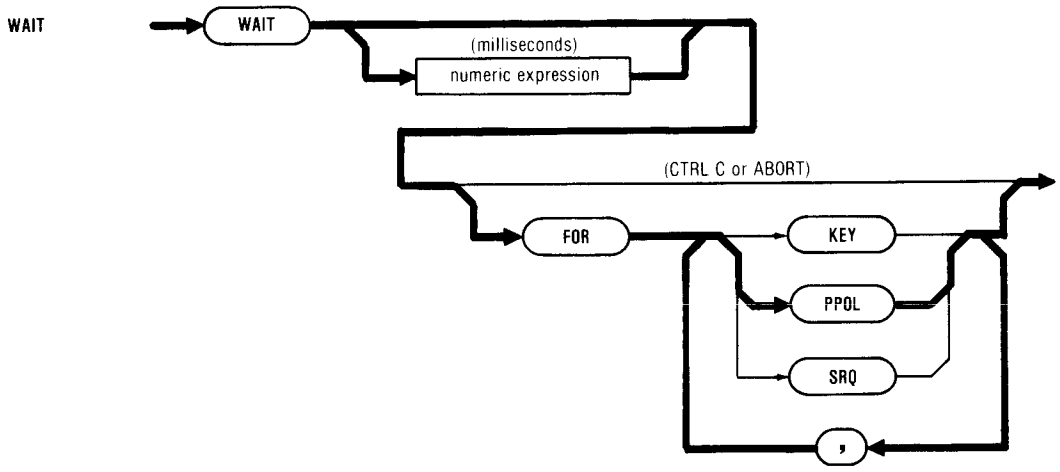
- An internal SRQ flag is set by a service request. It is reset in the controller by performing any serial poll (for example, Y% = SPL(10)). However, depending on the instrument, SRQ will probably be set again until the instrument requesting service is serial polled.

- WAIT time is 0 if the internal SRQ flag was not reset afterthe last service request.

- WAIT FOR SRQ does not reset the internal SRQ flag.

- Other references in this manual: IEEE-488 Input and Output Statements.

8-55. The WAIT FOR PPOL Statement



8-56. The WAIT FOR PPOL statement suspends operation of the program and initiates continuous parallel polling indefinitely until a positive parallel poll response is detected on either instrument port.

- The WAIT can be terminated in any of the ways defined above under the WAIT statement.

- A time limit may be specified following the word WAIT as described above under the WAIT Time Statement.

- The WAIT is terminated whenever a positive parallel poll response is detected on either instrument port.

- A positive parallel poll response causes the program to continue with the next statement unless a previous ON PPOL GOTO statement has been executed.

- When a previous ON PPOL GOTO statement has been executed, a positive parallel poll response causes the program to transfer to the parallel poll processing routine.

- The program statements following WAIT FOR PPOL should cause the parallel poll response bit of the responding instrument to be reset, if possible.

- WAIT time is 0 if either instrument port has a positive parallel poll response when WAIT FOR PPOL is executed.

- See the note in the description of ON PPOL GOTO in this section.

- Other references in this manual: IEEE-488 Input and Output Statements.

8-57. ERRORS AND ERROR HANDLING

8-58. Appendix B lists the errors which can occur in executing BASIC statements. There are three levels: Fatal, Recoverable, and Warning.

8-59. Fatal Errors

8-60. A fatal error (level F) terminates a program immediately, returns the 1720A to Immediate Mode, and displays an error message.

- There is no way to recover a program from a level F error.

8-61. Recoverable Errors

8-62. A recoverable error (level R) terminates a program immediately, returns the 1720A to Immediate Mode, and displays an error message.

- A level R error can be recovered from if:

  1. It occurs after execution of an ON ERROR GOTO statement.

  2. The program routine referenced by ON ERROR GOTO includes a RESUME statement.

  3. A second error does not occur before the RESUME.

  4. The program routine referenced by ON ERROR GOTO does not include an OFF ERROR statement.

- An OFF ERROR statement causes subsequent level R errors to terminate the program as described above.

8-63. Warning Errors

8-64. A warning error (level W) allows a program to continue running.

- An error message is normally displayed.

- An error message is not displayed if the the four conditions described above under level R errors are met.

- Input warning errors 801, 802 and 803, caused by keyboard entries, request that the entry be repeated ("?" is displayed).

- Error 803, caused by an illegal character in a VAL argument or a non-keyboard input, truncates the bad character and all characters following it. Characters up to the one causing the error are left intact.

- Error 904 truncates all excess characters, leaving the rest intact.

8-65. Error Variables

8-66. There are two system variables for errors that may be used as variables in a program to determine what action to take when level R or W errors occur:

- ERL - the line number of the statement which caused the most recent error.

- ERR - the error number of the most recent error.

NOTE

ERL and ERR are reset when the RUN statement is executed, but not when GOTO or CONT is executed in Immediate Mode.

8-68. INTERRUPT PROCESSING PROGRAM EXAMPLES

8-69. The program examples presented below illustrate the concepts developed in this section with portions of programs. These examples are not complete programs.

8-70. In the following example, the programmer anticipated that the variable I may be zero at some point and, rather than allow the program to halt, included an error handling routine that checks ERR for 603 (divide by zero error) and ERL for the line number of the possible divide by zero error (other divide by zero errors presumably should halt execution). If the error or line number that caused the interrupt are not the ones expected, then the OFF ERROR disables the error handler routine and displays the error number and line number of the error. If error 603 did occur at line 120 to generate the interrupt, then the message is printed by line 1020. Line 1030 causes a branch to line 130, and the "ON ERROR GOTO 1010" is still active.

```
20    ON ERROR GOTO 1010
      .
      .
      .
110   . . . .
120   A = 20 / I
130   . . . .
      .
      .
      .
1000  ! ERROR HANDLER
1010  IF ERR 603 OR ERL < > 120 THEN OFF ERROR
1020  PRINT "DIVIDE BY ZERO ERROR"
1030  RESUME 130
```

8-71. The following example illustrates an appropriate way to terminate a FOR-NEXT loop within a subroutine after resuming from an error handling routine. The FOR-NEXT loop was reset to its terminal value (-10) in the error handling routine so that the loop will not be repeated when control is resumed from the error handler. If control is resumed at line 4040 in the subroutine, error 521 (not a well structured statement) results. In this example, the FOR-NEXT loop is allowed to terminate naturally.

```
30      ON ERROR GOTO 1010
        .
        .
        .
100     DIM Y (20)
110     GOSUB 4000
120     . . . .
        .
        .
        .
1000    ! ERROR HANDLER
1010    IF ERR < > 606 THEN OFF ERROR \   ! Negative LOG Argument
1020    PRINT "LOG ARGUMENT <= 0"
1030    I = -10
1040    RESUME 4030
        .
        .
        .
4000    ! CALCULATE AND STORE LOG ROUTINE
4010    FOR I = 10 TO -10 STEP -1
4020    Y (10 + I) = LOG (I)
4030    NEXT I
4040    RETURN
```

8-72. The following example uses a bus timeout (error 408) to terminate an input from the IEEE-488 Bus. When the sending device has ceased output to the bus, the 100-millisecond timeout set in line 100 generates an error that is handled in the error handling routine at line 1000 by first checking to see if it is the proper type of error. The error handling routine then determines which reading was the last reading (line 1020), resets the FOR-NEXT loop, and resumes executing at the NEXT statement. The statement GOTO 1050 at line 1010 causes a RESUME to re-execute any other statement that causes an error. When it is re-executed, the program terminates unless it is a warning error, because the error will occur again.

```
10      ON ERROR GOTO 1010
        .
        .
        .
100     TIMEOUT (100)                     ! Set BUS timeout to 100 msecs.
110     FOR I% = 0% TO 999%
120     INPUT LINE @ C3%, R$(I%)
130     NEXT I%
140     PRINT R$ (0%..N%)
150     STOP
        .
        .
        .
1000    ! TIMEOUT ERROR HANDLER
1010    IF ERR < > 408 THEN OFF ERROR
1020    N% = I% - 1%                      ! N% = Number of readings
1030    I% = 999%                         ! Properly terminates FOR-NEXT
1040    RESUME 130
```

8-73. In the following example, RESUME returns control to the error causing statement rather than to a specified line number. The value of D iscorrected in line 1010 so that the statement on line 30 will be re-executed properly.

```
10      ON ERROR GOTO 1000
20      D = 0
30      I = 3 / D
40      PRINT "I="; I
        .
        .
        .
1000    IF D < > 0 GOTO 32767             ! To END statement
```

```
1010    D = 1E-76
1020    RESUME
        .
        .
        .
32767   END
```

8-74. In the following example, D is not adjusted and the RESUME statement branches to a PRINT statement.

```
10      ON ERROR GOTO 1010
20      D = 0
30      I = 3 / D
40      PRINT "I="; I
50      PRINT "D="; D
        .
        .
        .
1000    ! ERROR ROUTINE
1010    IF D < > 0 GOTO 32767              ! To END statement
1020    RESUME 50
        .
        .
        .
32767   END
```

8-75. The following error routine checks for a legal input for a tangent value. The error is produced on line 30 when BASIC tries to find the tangent of the entered value. The likely error would be a divide by zero error. RESUME branches to line 20 to allow re-entry of the A value.

```
10      ON ERROR GOTO 100
20      PRINT "Enter value for TANGENT"; INPUT A
30      X = TAN (A)
        .
        .
        .
100     IF ERR = 603 AND ERL = 30 THEN PRINT "ILLEGAL VALUE"
110     RESUME 20
```

8-76. The following example uses the ERR system variable to check for a number of possible error conditions and has the operator correct the problem by using the Touch-Sensitive Display.

```
10      INIT PORT 0                        ! Initialize Bus 0
20      ON ERROR GOTO 1010                 ! Enable error interrupt
30      CLOSE 2                            ! Ensure file 2 not open
40      OPEN "DATA" AS NEW FILE 2          ! Open data file
50      INPUT AD%                          ! Enter device address
60      !
70      ! Command instrument to take readings
80      !
90      INPUT LINE @ AD%, R$               ! Input the reading
100     !
110     ! Other statements
120     PRINT #2, R$                       ! Save the reading on file 2
130     !
140     ! Other statements
150     !
900     STOP
910     !
920     !
1000    ! ERROR HANDLER
```

```
1010  IF ERR = 801 OR ERR = 803 GOTO 1040 ! Check for illegal entry
1020  IF ERR < > 401 AND ERR < > 402 GOTO 1060 ! Check instrument
      address
1030  PRINT "Wrong Bus address: reenter" ! Display operator message
1040  RESUME 50                          ! Re-execute line 50
1050  !
1060  IF ERR < > 300 AND ERR < > 301 GOTO 1140 ! Check for good disk
1070  IF ERR = 300 THEN PRINT "Load disk"; \ GOTO 1090
1080  PRINT "Remove write protect label";
1090  PRINT "Then touch display"         ! Display message
1100  K = KEY                            ! Clear key buffer
1110  WAIT FOR KEY                       ! Enable and wait for key
1111  ! interrupt
1120  K = KEY                            ! Clear key buffer
1130  RESUME                             ! Re-execute line 40
1140  OFF ERROR                          ! Terminate program on other
                                         ! errors
```

8-77. In the following example, if a CTRL/C character is generated, the error routine beginning at line 5000 will take control. The operator is given the choice of stopping or continuing execution of the program. The operator enters either 1 or 2 to indicate continuation or halt. If neither 1 or 2 is selected, the screen redisplays the choices. If 1 is selected, the RESUME statement branches to the line that was being executed when the CTRL/C character was encountered. If 2 is selected, the OFF CTRL/C statement halts execution of the program and returns to Immediate Mode.

```
10    ON CTRL/C GOTO 5010
      .
      .
      .
5000  ! CTRL/C TERMINATION ROUTINE
5010  PRINT "SELECT ONE" PRINT
5020  PRINT SPACE$(21);"1 CONTINUE"
5030  PRINT SPACE$(21);"2 STOP"
5040  PRINT SPACE$(21);\INPUT C
5050  IF C < > 1 AND C < > 2 GOTO 5010
5060  IF C = 2 THEN OFF CTRL/C
5080  RESUME
```

8-78. In the following example, if ABORT or CTRL/C is pressed, the ABORT routine at line 5010 is executed. The routine allows the operator to confirm the ABORT or CTRL/C by selecting either 1 (continue) or 2 (halt readings). In this way, an inadvertent CTRL/C or ABORT need not necessarily halt readings.

```
10    ON CTRL/C GOTO 5010
      .
      .
      .
100   PRINT @ 3%, 'VR2?'           ! Take readings from instrument 3
110   INPUT @ 3%, V
120   PRINT V
130   GOTO 100                     ! Take another reading
      .
      .
      .
5000  ! ABORT BUS READING ROUTINE
5010  PRINT "SELECT ONE" \ PRINT "1 = CONTINUE" \ PRINT "2 = STOP"
5020  INPUT A
5030  IF A < > 1 AND A < > 2 GOTO 5010
5040  IF A = 1 THEN RESUME ELSE OFF CTRL/C
```

8-79. In this example, the WAIT statement requires that a 500 millisecond delay be performed before printing the words "1720A CONTROLLER". The statement on line 120 clears the screen, then the sequence is repeated until something, such as CTRL/C, interrupts.

```
100     WAIT 500
110     PRINT CHR$ (27%); "[2J"          !Erase screen
120     PRINT "1720A CONTROLLER"
130     GOTO 100
```

8-80. The WAIT statement on line 110 allows a two-second delay between source generation (2VDC) from the 5100A (Fluke Calibrator) and measurement on the 8500A (Fluke Digital Multimeter). This allows the 5100A time to generate the source voltage before attempting to make a measurement with the 8500A.

```
90      D% = 2000%
100     PRINT @ 1%, '2V,N'               ! 5100A Programmed for 2VDC
110     WAIT D%
120     PRINT @ 2%, '?'
130     INPUT @ 2%, V                    ! Reading from 8500A
140     PRINT V
```

8-81. The following example reads the system clock each time the display is touched. The 1720A computes the difference and displays the elapsed time between the first and second time the display is touched. Note that the key buffer is initially cleared (line 30) to ensure a WAIT at line 70 and cleared again at line 90 and 140 for the same purpose.

```
10      ! **** TIMER ****
20      !
30      ! Clear key buffer
40      E$ = CHR$(27)+'['                ! Escape sequence for display
50      PRINT E$; '2J'                   ! Clear display
60      PRINT CPOS (6, 26); "Touch to START"; ! Position cursor
61      K% = KEY                         ! and request start
70      WAIT FOR KEY                     ! Wait till display is touched
80      T1 = TIME                        ! Find time from system clock
90      K% = KEY                         ! Clear key buffer
100     PRINT E$; 'J'; " T1 = "; T1; ! Position cursor; display time 1
110     PRINT CPOS(8, 26); "Touch to STOP "; ! Position cursor
111     ! and display stop request
120     WAIT FOR KEY                     ! Wait until display is touched
130     T2 = TIME                        ! Find time from system clock
140     K% = KEY                         ! Clear key buffer
150     PRINT "   T2 = "; T2             ! Display time 2
160     PRINT CPOS(10,32); "ELAPSED TIME = "; (T2-T1) / 1000;" SECONDS"
170     GOTO 60                          ! Repeat program
```

Results from running this program:

```
Touch to START T1 =       2484420
Touch to STOP  T2 =       2489580

    ELAPSED TIME  =       5.16 SECONDS

Touch to START T1 =       2493550
Touch to STOP  T2 =       2493930

    ELAPSED TIME  =       0.38 SECONDS
```

Section 9
The Touch Sensitive Display

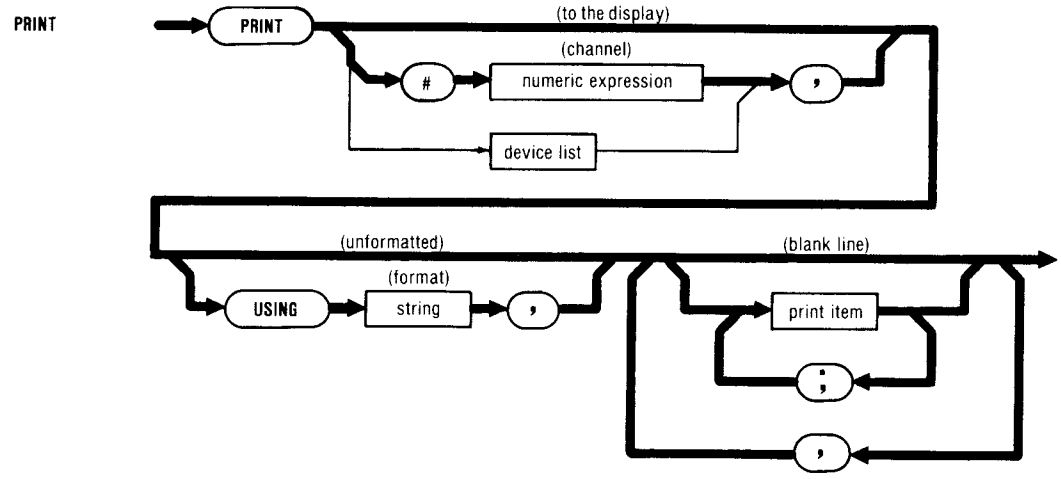
9-1. INTRODUCTION

9-2. This section discusses techniques for effectively using the Touch-Sensitive Display feature of the 1720A Instrument Controller. The discussion covers two general categories of display functions: output and input.

9-3. The display included with the 1720A offers several useful features that enable the user to customize operator message outputs. These features include character graphics, double size characters, reverse video, double intensity, and blinking. In addition, the cursor control functions and erasing capabilities give the programmer complete control over the display.

9-4. Input from the operator of an instrumentation system controlled by a 1720A is accomplished by means of the touch sensitive panel overlaying the display. This panel is capable of identifying one of sixty distinct blocks or touch areas when the operator applies moderate touch pressure. By integrating display messages with expected responses, an operator can be directed to supply needed numerical or decision inputs without interfering with the process at hand.

9-5. USING THE DISPLAY FOR OUTPUT
9-6. The PRINT Statement



9-7. The PRINT statement is used in Fluke BASIC to send code information or characters to the display.

- All code sequences described in this section that affect the display must be transmitted via PRINT to the display.

- Character and code sequences for display purposes may be conveniently stored in virtual arrays and called when needed by a PRINT statement.

- An output channel (NEW) may be opened to the display (KB0:), using an OPEN statement. The PRINT statement would then direct display output to that channel number.

- Other PRINT references in this manual: General Purpose BASIC Statements, Input and Output.

9-8. The following examples illustrate PRINT usage within the context of this definition:

STATEMENT	MEANING
`PRINT "Fluke 1720A"`	Display "Fluke 1720A".
`DIM A$(2)`	Dimension a 3 string array.
`A$(0) = "Fluke 1720A"`	Place message in the first string.
`PRINT A$(0)`	Display message from the array element.
`OPEN "KB0:" AS NEW FILE 3` `PRINT #3, "Fluke 1720A"`	Open channel 3 to the display. Send message to the display channel.

9-9. The ASCII Character Set

9-10. The 1720A uses ASCII characters for display as shown in column 1 of Appendix I. The first 32 characters (numbered 0 through 31) are defined as control codes, which include such functions as backspace (BS) or carriage return (CR). When generated by the keyboard, many of these control codes are intercepted, and a block (character number 127) is displayed in their place. However, the CHR$(n%) string function can be used within a program to generate any eight-bit code pattern for the display, including the control codes.

9-11. Many of the control codes reserved by the ASCII standard are not used by the 1720A. However they are used by the display system to provide some useful symbols. These symbols are stored in a ROM and are separate from the graphics characters described later in this section. Like other displayable ASCII characters from the ROM, these symbols may be enhanced with higher intensity, reverse video, blinking, and underlining.

NOTE

The John Fluke Mfg. Co., Inc. reserves the right to make changes to the symbols contained in character generator ROMS.

9-12. Figure 9-1 contains a short program that will display all of the displayable characters generated by the character ROM in double size display format. The block (character number 127) is displayed in place of all control characters that are not displayable. The space character is just left of the "!". Touch the screen to clear the display.

```
10      ES$ = CHR$(27) + "[" \ BL$ = CHR$(127)
20      PRINT ES$ + "1p"; CPOS(2,5); BL$;
30      FOR I = 1 TO 6 \ PRINT CHR$(I); \ NEXT I
40      FOR I = 7 TO 13 \ PRINT BL$; \ NEXT I
50      FOR I = 14 TO 26 \ PRINT CHR$(I); \ NEXT I
60      PRINT BL$;
70      FOR I = 28 TO 31 \ PRINT CHR$(I); \ NEXT I
80      PRINT CPOS(4,5); \ FOR I = 32 TO 63 \ PRINT CHR$(I); \ NEXT I
90      PRINT CPOS(6,5); \ FOR I = 64 TO 95 \ PRINT CHR$(I); \ NEXT I
100     PRINT CPOS(8,5); \ FOR I = 96 TO 127 \ PRINT CHR$(I); \ NEXT I
110     WAIT FOR KEY \ PRINT ES$ + "p";
```

Figure 9-1. Character Display Program

9-13. The CHR$ String Function

Format: CHR$ (numeric expression)

9-14. The CHR$ function creates an eight-bit ASCII-coded string character from the lower eight bits of the integer value of the numeric expression.

- The expression may be either integer or floating point.
- The allowable range of values is -32768 to 32767.
- If the expression has a floating point result, it is converted to an integer by truncation, i.e. the integer value will be the largest integer smaller than or equal to the floating point value.
- When sent to the display via PRINT, the character displayed or the control response corresponds to the integer represented by the lower seven bits of the value of the numeric expression. For example, PRINT CHR$(72) displays the letter H, just as PRINT "H" does.
- CHR$(x) will generate 7-bit ASCII codes corresponding to the decimal column in Appendix I, if the numbers 0 through 127 are used.
- To set the eighth bit to one, add 128 to the value in the decimal column corresponding to the desired character or code pattern.
- Other CHR$ references in this manual: None

9-15. The TAB String Function

Format: TAB (n)

9-16. The TAB function creates a string of space characters that would move the current print position forward to column n+1. TAB is intended for use with printers, but can also be used with the display.

- The value n is an integer value between 0 and 80.

- The current print position is the total number of characters transmitted since the last Carriage Return character, including all non-printable or non-displayable characters.

- The current print position may be ahead of the display cursor if a display control character sequence was included since the last carriage return character. These character sequences are discussed later in this section.

- A Carriage Return and Line Feed character sequence will precede the string of spaces if the current print position is already at or beyond column n.

- Other TAB references in this manual: None.

9-17. The following example illustrates an alternate method tab function for the display, using a defined function. The function works by returning the cursor to the beginning of the line, and then moving right to the column specified by the function argument (column 33 in line 40). The string functions in line 30 change the function argument to string form and then remove space characters from either side of it.

```
10   ES$ = CHR$(27) + "["          ! Control sequence identifier
20   CR$ = CHR$(13)                ! Carriage Return
30   DEF FN TB$(C) = CR$ + ES$ + MID (NUM$(C),2,LEN (NUM$(C))-2) + "C"
40   PRINT FN TB$(33); "HERE"
```

9-18. Special Display Control Characters

9-19. Eight of the ASCII control characters are used by the 1720A display module. These characters are listed in Table 9-1. The ASCII mnemonic is presented with the corresponding CHR$(n) for each of these characters.

Table 9-1. Special Display Control Characters

CHR$(N)	MNEMONIC	FUNCTION	RESULT
CHR$(7)	BEL	BELL	Activates the 1720A Beeper.
CHR$(8)	BS	BACKSPACE	Moves the cursor to the left one column, if not already positioned at the leftmost column.
CHR$(9)	HT	HORIZONTAL TAB	Moves the cursor to the next tab stop, located every 8 columns.
CHR$(10) CHR$(11) CHR$(12)	LF VT FF	LINE FEED VERTICAL TAB FORM FEED	These three commands all move the cursor to the next lower line, in the same column. The display scrolls upward if the cursor is on the bottom line.
CHR$(13)	CR	CARRIAGE RETURN	Moves the cursor to the beginning of the current line.
CHR$(24) CHR$(26)	CAN SUB	CANCEL SUBSTITUTE	Cancels a display control character sequence if sent as part of the sequence.
CHR$(27)	ESC	ESCAPE	Starts a display control character sequence, discussed later in this section.

9-20. Display Control Character Sequences

9-21. The 1720A display uses a subset of ANSI Standard X3.64-1977 to define sequences of ASCII characters for added control of the display. These character sequences enable the user to customize operator message outputs. Among the capabilities provided are character graphics, double size characters, reverse video, double intensity, and blinking. The cursor control functions and erasing capabilities give the programmer complete control over the display.

9-22. The generalized command format for display control sequences is:

Display Control Identifier	Task Number	Seperator	Task Number	Task Letter
ESC [	n	;	n	T

- Display control character sequences are sent to the display via a PRINT statement.
- CHR$(27) is used to generate the ESCape code.
- The [character is not used by the scrolling cursor controls discussed below.
- Use quotes around the characters that follow CHR$(27) so that PRINT will handle them as strings.
- The semicolon separator is used to separate multiple task numbers that have the same task letter.
- Use either + or ; to link strings together. For example, PRINT CHR$(27) + "[1;5;7m" selects high intensity, blinking, and reverse image for all characters that follow.
- Other characters for display may immediately follow a display control sequence. For example, PRINT CHR$(27) + "[1;5;7m Fluke 1720A".
- Other generalized command format references in this manual: None

9-23. Cursor Controls

9-24. Escape control character sequences can move the cursor to any position on the screen, either relative to the current position or absolutely, by designating line and column. Scrolling commands allow movement of the entire display up or down when movement is beyond top or bottom limits. Table 9-2 presents cursor controls, with the PRINT statement required to produce them for the display. The semicolon shown at the end of each PRINT statement inhibits the Carriage Return and Line Feed codes that normally follow a print statement. Note that the string function CPOS(l%,C%) produces an equivalent character string, and is used in an identical manner. The CPOS string function is discussed in further detail later.

Table 9-2. Cursor Controls

ACTION	PRINT STATEMENT	EXAMPLE
Up n lines	PRINT ES$ + "nA";	PRINT ES$ + "4A";
Down n lines	PRINT ES$ + "nB";	PRINT ES$ + "3B";
Right n columns	PRINT ES$ + "nC";	PRINT ES$ + "21C";
Left n columns	PRINT ES$ + "nD";	PRINT ES$ + "17D";
Direct to line, column	PRINT ES$ + "1,cH";	PRINT ES$ + "5,23H";
Direct to line, column	PRINT CPOS (1%,c%);	PRINT CPOS (5%,23%);
Scroll down one*	PRINT EX$ + "D";	PRINT EX$ + "D";
Scroll up one*	PRINT EX$ + "M";	PRINT EX$ + "M";
Next line scroll*	PRINT EX$ + "E";	PRINT EX$ + "E";

NOTE 1.

This table assumes these previous assignments, EX$=CHR$(27)
ES$=EX$ + "["

NOTE 2.

** No scrolling takes place if cursor is not at screen top or bottom.*

9-25. The CPOS String Function

Format: CPOS (line, column)

9-26. CPOS creates a string which, when sent to the display by a PRINT statement, positions the cursor at the specified line and column.

- The string is always eight characters long, in the form: ESCape [ll ; cc H. For example, CPOS(3,20) is equivalent to CHR$(27)+"[03;20H".
- The line and column are numeric expressions.
- If the expression for line or column is floating point, it must be within the range of integers, and will be truncated to an integer.
- If either the line or column is less than zero, a value of zero is assigned.
- If the line or column is greater than 99, a value of 99 is assigned.
- Additional limits are imposed by the video display module:

  1. A line or column number of zero is interpreted as one.
  2. A line number greater than 16 is interpreted as 16.
  3. A column number greater than 80 is interpreted as 80.
  4. In double-size display mode, these limits are respectively line 1, line 8 and column 40.

- Other CPOS references in this manual: None

9-27. The CPOS function may be assigned to a string variable or added tostrings for display formatting. It may take various forms as shown in the following examples which display "Fluke 1720A" at line 10, column 30.

Example 1:

```
10   A$ = CPOS(10,30)
20   B$ = A$ + "Fluke 1720A"
30   PRINT B$
```

Example 2:

```
10   PRINT CPOS(10,30); "Fluke 1720A"
```

Example 3:

```
10   PRINT CPOS(10,30) + "Fluke 1720A"
```

9-28. The CPOS string function may also be used for more creative displays. The following example displays a scaled SIN function, using CPOS:

```
10   ! *** Display Sine Function ***
20   !
30   FOR X = 1 to 80                !Setup loop for display length
40   Y = 6 * SIN (2 * PI * (X/40)) + 9 !Compute scaled sine function
50   PRINT CPOS(Y,X); '*';          !Position cursor and display *
60   NEXT X                         !Loop
```

9-29. The following example illustrates a method of causing the cursor to disappear during a double size message display. The technique is to move the cursor alternately to the lower left and lower right corners. Normally, the ON KEY interrupt would be enabled (discussed later in this section), allowing operator input to break the loop:

```
1000 PRINT CPOS(8,1)
1010 PRINT CPOS(8,40)
1030 GOTO 1000
```

9-30. Character Enhancements

9-31. Character display enhancements are used to draw attention to a word or portion of a displayed message.

- Four different enhancements are available: high intensity, underlining, blinking, and reverse image (dark characters on light background).

- Enhancement commands may be used in multiple combinations.

- A single or multiple enhancement command occupies one character position on the display, regardless of the length of the character string required.

- Enhancement commands define the characteristics of all displayed characters to follow, until another enhancement command is encountered.

- Overwriting a display location containing an enhancement command will delete the enhancement.

- Each enhancement command cancels all previous enhancements.

9-32. Table 9-3 presents the enhancement commands, with the PRINT statement required to produce them for the display. The semicolon shown at the end of each PRINT statement inhibits the Carriage Return and Line Feed codes that normally follow a print statement.

Table 9-3. Character Enhancement Commands

COMMAND	PRINT STATEMENT
Enhancements OFF	PRINT ES$ + "m";
Enhancements OFF	PRINT ES$ + "0m";
High Intensity	PRINT ES$ + "1m";
Underline	PRINT ES$ + "4m";
Blinking	PRINT ES$ + "5m";
Reverse Image	PRINT ES$ + "7m";

NOTE 1.

*This table assumes this previous assignment, EX$=CHR$(27) + "[" *

9-33. The underline and reverse image enhancements are subject to some limitations imposed by the video module. The refresh scanning rate of the display exceeds the rate at which characters are written into the display memory. As a result, these character enhancements will momentarily cause the entire remaining display to have underlines or light background until the enhancements OFF command is written into the display. To avoid this, place the enhancements OFF command at the correct display location, then back up to one location ahead of the start of the message and print the enhancement selections along with the message. The use of the Display Worksheet (Fluke Part Number 533547, pad of 50) will make this task easier. Remember that each enhancement command occupies one display location, and that the cursor moves forward one location after a command is placed.

9-34. The following example shows the correct sequence of statements to display "Fluke 1720A" in reverse image on line 8, column 14.

```
10   E$ = CHR$(27) + '['                ! Set up escape sequence
20   PRINT E$; '2J'                     ! Erase display
30   PRINT CPOS(4,25);                  ! Move to end of message
40   PRINT E$; 'm'                      ! Disable all enhancements
50   PRINT CPOS(4,13);                  ! Move to beginning of message
60   PRINT E$; '7m';                    ! Enable reverse image
70   PRINT 'Fluke 1720A'                ! Display message
```

9-35. The following example displays "Fluke 1720A" as in the last example. Line 60 enables all enhancements in a single command that will occupy one display location.

```
10   E$ = CHR$(27) + '['                ! Set up escape sequence
20   PRINT E$; '2J'                     ! Erase display
30   PRINT CPOS(4,25);                  ! Move to end of message
40   PRINT E$; 'm';                     ! Disable all enhancements
50   PRINT CPOS(4,13);                  ! Move to beginning of message
60   PRINT E$; '1;4;5;7m';              ! Enable all enhancements
70   PRINT 'Fluke 1720A'                ! Display message
```

9-36. Mode Commands Introduction

9-37. Mode commands affect three different areas: character size, the alternate graphics characters, and the disabling of keyboard inputs.

- Mode commands do not occupy a display character position.
- Each of these three areas is discussed separately below.

9-38. Character Size

Normal size: PRINT CHR$(27) + "[p";
PRINT CHR$(27) + "[0p";

Double size: PRINT CHR$(27) + "[1p";

9-39. Character size commands affect the entire display, by changing the basic timing of the display scan. Double size characters are doubled in both height and width, occupying the area of four normal size characters. Such characters are more easily recognized from a distance.

- Normal size allows 16 lines of up to 80 characters each.
- Double size allows 8 lines of up to 40 characters each.
- Character size commands do not occupy a display character position.
- Character size commands clear the screen and return the cursor to the home (upper left) position. If the command is not terminated with a semicolon the cursor will then move down one line.
- The video display module will remain in normal or double size display mode, according to the most recent character size command, even if the program is completed.
- The video display module starts up in normal size display mode until commanded to double size.
- Other references in this manual: None.

9-40. Graphics Characters

Enable graphics characters: PRINT CHR$(27) + "[3p";
Disable graphics characters: PRINT CHR$(27) + "[2p";

9-41. The alternate graphics character set is presented in Figure 9-2. When graphics mode is enabled, these characters are displayed from alternate pattern generation circuitry in place of the numbers 0 through 9, and the : character.

- Graphics enable and disable commands do not occupy a character display location.
- Graphics characters cannot be given character enhancements.
- Graphics characters may be enabled and disabled as many times as desired within a single display. It is possible, for example, to display a box around a number.

• The names given to the graphics characters in Figure 9-2 are significant. If the characters are used as their name indicates, a display can be set up that changes in size only (not shape) as the display mode is changed between normal and double size. Program code written in this manner can be used in other application programs.

CHARACTER	NORMAL SIZE	DOUBLE SIZE	CHARACTER NAME
0			Top Right Corner
1			Top Left Corner
2			Bottom Right Corner
3			Bottom Left Corner
4			Top Intersect
5			Right Intersect
6			Left Intersect
7			Bottom Intersect
8			Horizontal Line
9			Vertical Line
:			Crossed Line

NOTES:
1. To enable Graphics Mode, send the display ESC [3p
   In BASIC, PRINT CHR$(27);"[3p";
2. To disable Graphics Mode, send the display ESC [2p
   In BASIC, PRINT CHR$(27);"2p";
3. In Graphics Mode, characters in the left column are displayed as shown.
4. Use the character names as defined to construct illustrations that do not change form between normal and double size.

Figure 9-2. Graphic Mode Characters

- Graphics commands select a mode of the video display module. It will remain in the mode most recently selected even if the program is completed.

- The video display module starts up in normal (graphics disabled) mode until commanded to enable graphics.

- Other Graphics Character references in this manual: None.

9-42. The program presented in Figure 9-3 will display the graphics characters, first in normal size, and then, when the screen is touched, in double size. Touching the screen a second time will clear the display and reset it to normal size with graphics disabled.

```
10   ES$ = CHR$(27) + "[" \ CL$ = ES$ + "2J"
20   PRINT CL$; ES$ + ";3p"; CPOS(8,24); \ GOSUB 50
30   PRINT CL$; ES$ + "1;3p"; CPOS(4,5); \ GOSUB 50
40   PRINT ES$ + ";2p"; CL$
45   STOP
50   PRINT "0 1 2 3 4 5 6 7 8 9 :"
60   WAIT FOR KEY \ K% = KEY \ RETURN
```

Figure 9-3. Graphics Character Display Program

9-43. The following example program illustrates:

1. Use of the graphics characters to draw a box.

2. Character size commands.

3. Display character enhancements.

9-44. This program will draw a box, and then display "FLUKE 1720A" underlined and in reverse image within the box. Every two seconds the entire display will change from normal to double size and then back. Touching the screen will reset all modes to normal and clear the screen.

9-45. In this example, the display is created by a subroutine. The same code is used for both normal and double size. This is due to two separate techniques:

1. Graphics codes are used as defined by their names, without reference to how they look.

2. Cursor movement is relative to the center point, established before entering the subroutine.

```
10   ! Display Demonstration Program
20   !
30   ON KEY GOTO 800              !Program exit path
40   !
50   ! String variables:
60   !
70   ES$ = CHR$(27) + "["         !Control sequence identifier
80   NS$ = ES$ + "p"              !Normal size
90   DS$ = ES$ + "1p"             !Double size
100  NC$ = CPOS(8,40)             !Normal size center
110  DC$ = CPOS(4,20)             !Double size center
120  GR$ = ES$ + "3p"             !Graphics mode
130  CG$ = ES$ + "2p"             !Clear graphics mode
140  D1$ = ES$ + "B" + ES$ + "D"  !Down one
150  U1$ = ES$ + "A" + ES$ + "D"  !Up one
160  L1$ = ES$ + "2D"             !Left one
```

```
170  HM$ = CPOS(1,1)                !Home position
180  !
190  ! First do it normal size:
200  !
210  PRINT NS$; NC$;                !Normal size, center.
220  GOSUB 390                      !Display
230  WAIT 2000                      !Wait 2 seconds
240  !
250  ! Then do it double size:
260  !
270  PRINT DS$; DC$;                !Double size, center
280  GOSUB 390                      !Display
290  WAIT 2000                      !Wait 2 seconds
300  !
310  ! Then repeat the sequence:
320  !
330  GOTO 210
340  !
350  ! The display subroutine:
360  !
370  ! First draw a box:
380  !
390  PRINT GR$                      !Graphics mode
400  PRINT ES$ + "14D"; ES$ + "2A"; !Left 14, up 2
410  !
420  PRINT "l"                      !Upper left corner
430  !
440  FOR I = 1 TO 28
450  PRINT "8";                     !Top line
460  NEXT I
470  !
480  PRINT "0"                      !Upper right corner
490  !
500  FOR I = 1 TO 5
510  PRINT Dl$; "9";                !Right side
520  NEXT I
530  !
540  PRINT Dl$; "2";                !Lower right corner
550  !
560  FOR I = 1 TO 28
570  PRINT Ll$; "8";                !Bottom line
580  NEXT I
590  !
600  PRINT Ll$; "3";                !Lower left corner
610  !
620  FOR I = 1 TO 5
630  PRINT Ul$; "9";                !Left side
640  NEXT I
650  !
660  PRINT CG$                      !Clear graphics mode
670  !
680  ! Place the message in the box:
690  !
700  PRINT ES$ + "2B"; ES$ + "25C"; !Down 2, right past message
710  PRINT ES$ + "m";               !Display enhancements off
720  PRINT ES$ + "23D";             !Left, just before message
730  PRINT ES$ + "4;7m";            !Select character enhancements
740  PRINT "F L U K E   1 7 2 0 A"; !Display message
750  PRINT HM$                      !Cursor to home position
760  RETURN
770  !
780  ! Clean up the display before leaving:
790  !
800  K% = KEY                       !Empty the KEY buffer
810  PRINT CG$; NS$;                !Reset graphics, clear screen,
820      !normal size, home position
830  END
```

9-46. Keyboard Disable and Enable

Disable keyboard: PRINT CHR$(27) + "[5p";
Enable keyboard: PRINT CHR$(27) + "[4p";

9-47. When disabled, all programmer keyboard inputs except control codes are ignored.

- The touch sensitive display remains active for inputs.
- Disable and enable commands do not occupy a character display position.
- A semicolon (;) placed at the end of a keyboard enable or disable command will inhibit the transmission of Carriage Return and Line Feed to the display.
- Keyboard disable/enable status is a mode of the display module. This status will remain as defined in the most recent disable or enable command, even after a program ends.
- The 1720A starts up with the keyboard enabled.
- Refer to the Introduction section for a discussion of enabling the keyboard with CTRL T.
- Other Keyboard Disable And Enable references in this manual: None.

9-48. Erasing

9-49. Commands are provided to allow a program to erase all or part of a line or of an entire display. Table 9-4 summarizes the erasing commands.

Table 9-4. Erase Commands

ACTION	PRINT STATEMENT
Erase to end of line	PRINT CHR$(27) + "[K";
Erase to end of line	PRINT CHR$(27) + "[0K";
Erase from start of line	PRINT CHR$(27) + "[1K";
Erase all of line	PRINT CHR$(27) + "[2K";
Erase to end of screen	PRINT CHR$(27) + "[J";
Erase to end of screen	PRINT CHR$(27) + "[0J";
Erase from start of screen	PRINT CHR$(27) + "[1J";
Erase all of screen	PRINT CHR$(27) + "[2J";

- Erase commands do not occupy a character display position.
- Partial erase commands are relative to the current cursor position.
- A semicolon placed at the end of an erase command will cause the cursor position to remain unchanged.
- Other Erase references in this manual: None.

9-50. USING THE DISPLAY FOR INPUT

9-51. The Touch Sensitive Display included with the 1720A Instrument Controller enables operation without a programmer keyboard. This feature allows the user to design operator input prompts in application programs that relate directly to the task at hand. The operator is free to maintain his or her focus upon the assigned task without the distraction of a complex keyboard or non-relevant decisions.

9-52. Figure 9-4 1720A Controller Programming Worksheet illustrates the relationship between the touch sensitive panel overlaying the display, and the locations of displayed characters. Note that the touch sensitive area is divided into 60 blocks numbered sequentially from the upper left. Each block covers 3 double size characters, or 12 normal size characters. Columns and rows of both normal and double size characters are numbered as the CPOS string function would address them. This grid pattern worksheet is a useful planning tool available from Fluke in pads of 50 sheets. Order Fluke part number 533547.

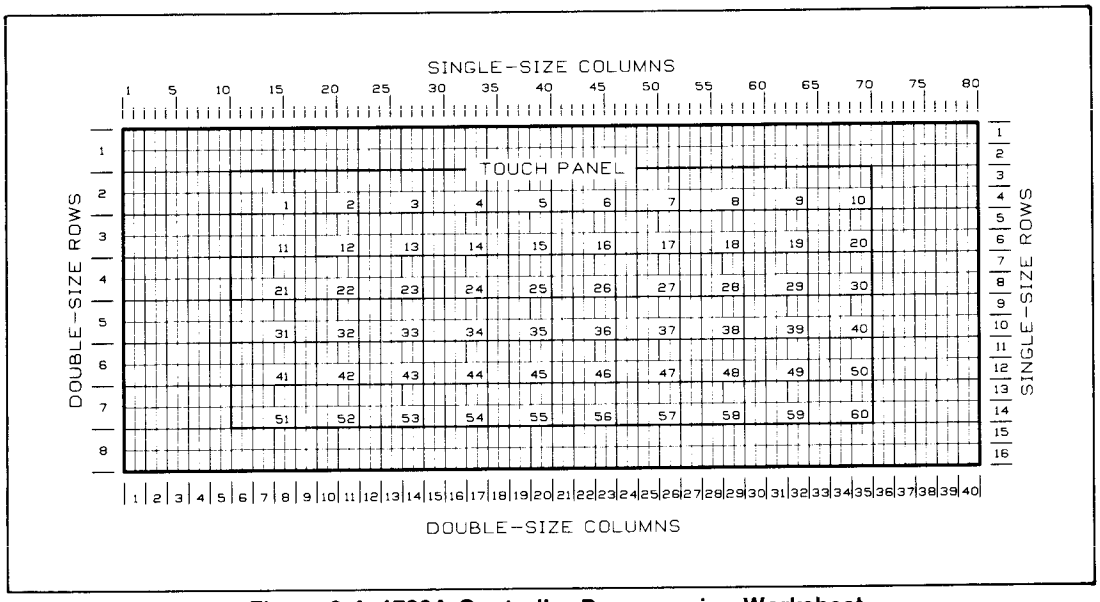


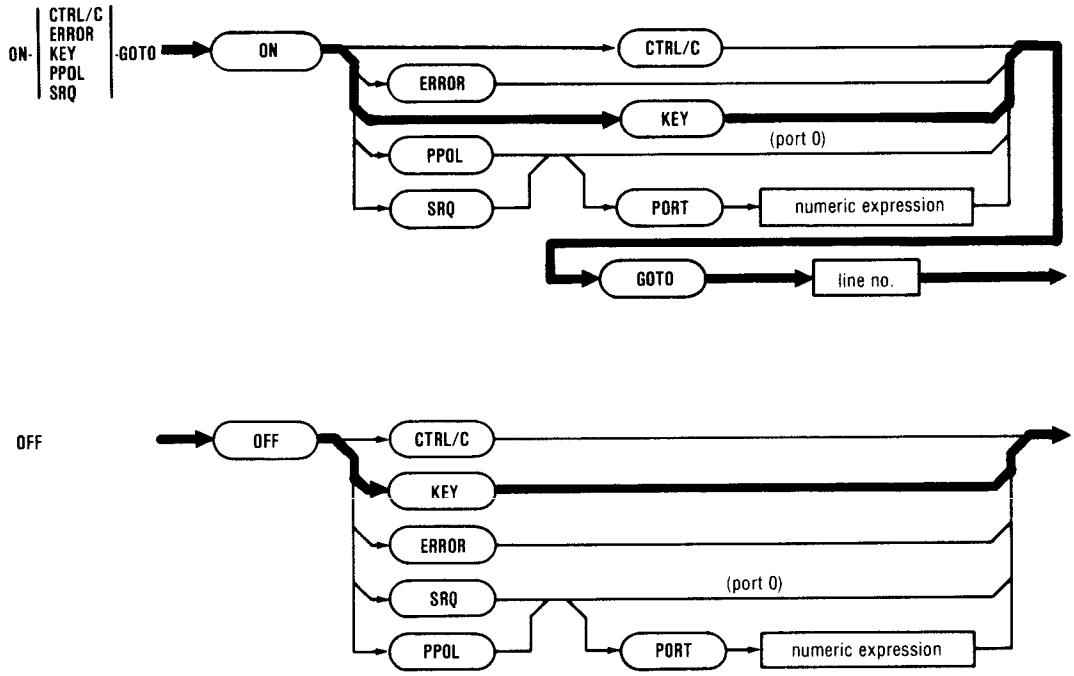
Figure 9-4. 1720A Controller Programming Worksheet

9-53. The KEY Variable

9-54. The system variable KEY contains the number of the last touch panel block pressed. The KEY variable has the following characteristics:

- KEY is an integer ranging from 0 to 60.
- A KEY value of zero means that no touch panel block has been pressed since the last time the value of KEY was used by a BASIC statement or since the last time a RUN command was executed.
- When KEY is accessed by a program (e.g., "K% = KEY" or "IF KEY = 0 THEN..."), it is set to zero.
- KEY may be used in any context that requires an integer variable.
- KEY cannot be assigned a value, except by pressing the touch sensitive display.

9-55. The ON KEY and OFF KEY Statements



9-56. ON KEY GOTO (line number) is one form of the ON-event interrupt enabling statement described in the Interrupt Processing section.

- When the ON KEY statement is encountered, the number stored in KEY is repeatedly checked for non-zero.

- A zero value for KEY will cause the GOTO to be ignored until KEY becomes non-zero.

- Control will transfer to the line number following GOTO if a non-zero value for KEY is detected any time after the ON KEY Statement has been executed.

- The section of program referenced by GOTO may be an interrupt processing routine, with a RESUME statement, or it may be a program exit.

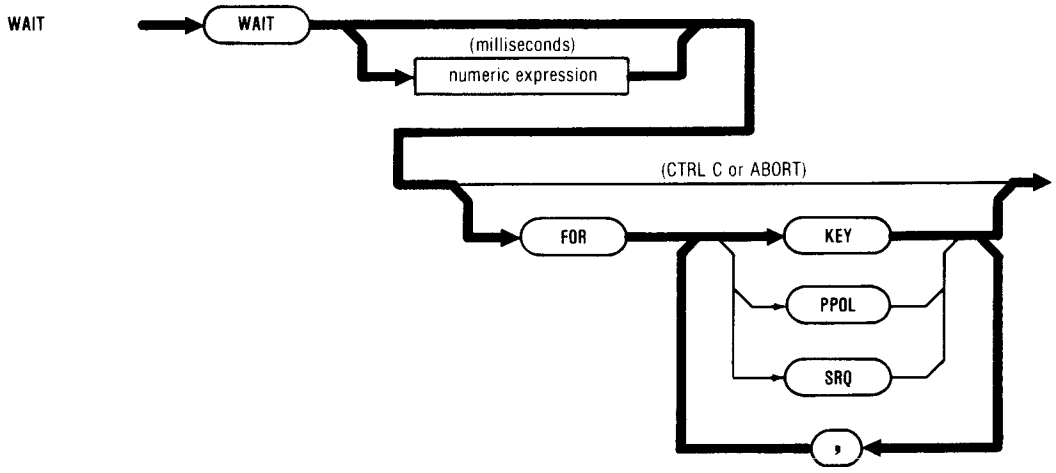
- After control has been transferred, the next RESUME statement will transfer control back to the program line that would have been executed next if the non-zero KEY had not been detected.

- After control has been transferred, further KEY entries on the touch sensitive display are ignored, until a RESUME statement is encountered.

- OFF KEY disables further checking of the value of KEY.

- Other references in this manual: Interrupt Processing.

9-57. The WAIT FOR KEY Statement



9-58. WAIT (time) FOR KEY is one form of the WAIT FOR (event) statement described in the Interrupt Processing section.

- When the WAIT FOR KEY statement is encountered, the number stored in KEY is checked for non-zero.
- The maximum time to wait may be specified by a number or numeric expression. It must evaluate to an integer between -32768 and 32767.
- The statement will be ignored if the specified time is zero or negative.
- The wait time will be indefinite (until the touch sensitive display is pressed) if no time is specified.
- A zero value for KEY will cause program execution to stop until KEY becomes non-zero, or until the specified time has elapsed.
- A non-zero value for KEY will immediately terminate the wait condition, passing control to the next program statement.
- Other references in this manual: Interrupt Processing.

9-59. The following example illustrates the KEY variable and the WAIT FOR KEY interrupt. Line 10 clears the KEY buffer. This resets any value it contained from touching the display before the program started. Line 20 halts the program until the touch sensitive display is touched. Line 30 prints the KEY number. Line 30 also clears the key buffer. If the buffer were not cleared, line 20 would detect it again, allowing line 30 to display the same key value repeatedly.

```
10   K% = KEY                          ! Clear key buffer
20   WAIT FOR KEY                      ! Enable key interrupt
30   PRINT 'KEY = '; KEY               ! Display key value
40   GOTO 20
```

Sample displayed results:

```
KEY = 41
KEY = 1
KEY = 23
KEY = 57
KEY = 18
```

9-60. The following example program displays the number of each key in double size directly under the spot that was touched.

- Flag KF% in line 80 enables the interrupt routine to clear the "Touch the Display" message from the display.

- After displaying a prompt message, line 120 halts the program until the display is touched. Then line 130 enables the KEY interrupt.

- Since the KEY buffer has a number in it from touching the display, line 130 immediately branches to the interrupt routine, disabling the KEY interrupt.

- Since the KEY flag KF% is initially zero, line 220 clears the "Touch the Display" message.

- Line 230 sets the KF% flag to one, so that subsequent passes through the routine will not clear the display.

- Lines 240 through 290 compute the position of the spot that was touched and display the KEY number at that spot.

- Line 300 reenables the KEY interrupt and branches back to the main routine. Since the ON KEY GOTO statement is still active, line 140 waits for another touch on the display.

- If a touch occurs within 10 seconds, the interrupt routine is reentered.

- If a touch does not occur within 10 seconds, line 150 disables the KEY interrupt and branches to line 80 starting the sequence over again.

```
10   ! *** Display Keys ***
20   !
30   ! Displays the KEY number of the spot touched. clears
40   ! the display to start over if not touched within 10 seconds.
50   !
60   K% = KEY                    !Clear KEY buffer
70   ES$ = CHR$(27) + "["        !Display escape sequence
80   KF% = 0%                    !KEY flag
90   PRINT ES$; "2J"; ES$; "1p" !Clear display, double size
100  PRINT CPOS(4,10);           !Position cursor
110  PRINT "Touch the Display!" !Display prompt message
120  WAIT FOR KEY                !Wait until display is touched
130  ON KEY GOTO 200             !Enable KEY interrupt
140  WAIT 10000                  !Wait 10 seconds
150  OFF KEY                     !Disable KEY interrupt
160  GOTO 80                     !Loop
170  !
180  ! *** Key Interrupt Routine ***
190  !
200  K% = KEY                    !Get KEY number
210  IF KF% = 1% THEN 240        !First time through?
220  PRINT ES$; "2J"             !Clear screen
230  KF% = 1%                    !Set KEY flag
240  KI% = K% - 1%               !Compute KEY index
250  TR% = INT (KI% / 10%)       !Compute touch panel row
260  DR% = TR% + 2%              !Compute display row
270  DC% = 6% + (KI% - 10% * TR%) * 3% !Compute display column
280  PRINT CPOS(DR%,DC%);        !Position cursor to spot touched
290  PRINT USING "##", K%;       !Display key on spot touched
300  RESUME 140                  !Wait for another key
```

9-61. AN INTERACTIVE DISPLAY PROGRAM

9-62. The following program combines graphics and the KEYfunction to create an interactive display without a keyboard. The display is a matrix over the touch-sensitive area containing the characters 0-9, A-Z +, -, *, /, SPACE, DELETE, and ENTER. It requests the operator to enter information. As the operator touches the characters, they are entered into string OE$ and displayed. Previous characters are deleted when DELETE is touched. The operator entry subroutine returns to the main program (operator entry loop) when ENTER is touched. A typical user program could then use the operator entries in string OE$. This example program simply requests another entry.

9-63. The first section of the program (up to line 1500) sets up the display. Lines 1010 through 1350 create the display matrix by printing concatenated strings of graphic characters. Lines 1360 through 1500 print the characters within the matrix.

9-64. The second section of the program requests operator entries. It compares the KEY value against limits and adds the corresponding ASCII character to string OE$.

```
1000  ! *** Interactive Display Program ***
1001  !
1002  ! Combines graphics and KEY function for an operator keyboard
1003  !
1004  ! Set Up the Display:
1005  !
1010  DIM S$(11)              !Dimension graphics variable
1020  ES$ = CHR$(27) + "["    !Display control identifier
1030  PRINT ES$ + "3p"; ES$ + "2J" !Enable graphics, clear display
1040  T$ = CPOS(2,10) + "188888" !First line
1050  X = 9
1060  A$ = "488888"
1070  GOSUB 12100
1080  S$(0) = T$ + "0"
1090  T$ = "588888"           !Part of 3rd, 5th, 7th lines
1100  A$ = ":88888"
1110  GOSUB 12100
1120  T$ = T$ + "6"
1130  FOR I = 2 TO 6 STEP 2  !Add CPOS to 3rd, 5th, 7th lines
1140  S$(I) = CPOS(I+2,10) + T$
1150  NEXT I
1160  T$ = "9"               !Part of 2nd, 4th, 6th, 8th lines
1170  X = 10
1180  A$ = "     9"          !(5 spaces, then 9)
1190  GOSUB 12100
1200  S$ = T$
1210  FOR I = 1 TO 7 STEP 2  !Add CPOS to 2nd, 4th, 6th, 8th lines
1220  S$(I) = CPOS(I+2,10) + T$
1230  NEXT I
1240  T$ = CPOS(10,10) + "388888" !9th line
1250  X = 9
1260  A$ = "788888"
1270  GOSUB 12100
1280  S$(8) = T$ + "2"        !(8's are 16 long, spaces are 6 long:)
1290  T$ = CPOS(14,10) + "188888888888888880      188888888888888880 "
1300  S$(9) = T$ + "188888888880" !10th line (there are 10 8's)
1310  T$ = CPOS(13,10) + "9                 9      9                 9 "
1320  S$(10) = T$ + "9          9" !11th line (there are 10 spaces)
1330  T$ = CPOS(14,10) + "388888888888888882      388888888888888882 "
1340  S$(11) = T$ + "388888888882" !12th line (there are 10 8's)
1350  PRINT S$(0..11); ES$ + "2p"; !Display matrix, disable graphics
1360  A = 1                   !Display 1st row characters
1370  X = 48
1380  Y = 57
1390  GOSUB 11100
1400  X = 65                  !Display 2nd row characters
1410  Y = 74
```

```
1420  GOSUB 11100
1430  X = 75                    !Display 3rd row characters
1440  Y = 84
1450  GOSUB 11100
1460  X = 85                    !Display 4th row characters
1470  Y = 90
1480  GOSUB 11100
1490  PRINT CPOS(9,49)+"+";CPOS(9,55)+"-";CPOS(9,61)+"*";CPOS(9,67)+"/";
1500  PRINT CPOS(13,16)+"ENTER";CPOS(13,41)+"SPACE";CPOS(13,62)+"DELETE"
1510  !
1520  ! ** Operator Entry Loop **
1530  !
1540  OP$ = "ENTER SOMETHING" !Operator prompt
1550  GOSUB 10100             !Call operator entry routine
1560  !
1570  ! User program to utilize operator inputs should go here.
1580  !
1590  GOTO 1540               !Loop
1600  STOP                    !End of program
1610  !
10000 ! ** Subroutine to Accept Operator Entries **
10010 !
10100 OE$ = ""                !Null entry, display prompt & entry:
10110 PRINT CPOS(1,1); OP$; " - "; ES$ + "K"; OE$; CPOS(8,1)
10120 K = KEY                 !Empty KEY buffer
10130 WAIT FOR KEY            !Wait for display touch
10140 K% = KEY                !Get KEY number
10150 PRINT CHR$(7%);         !Sound beeper
10160 IF K% < 11% THEN OE$ = OE$ + CHR$(47% + K%) ! Check for 0 - 9
10170 IF K% > 10% AND K% <37% THEN OE$ = OE$+CHR$(54% + K%)!Check A-Z
10180 IF K% = 37% THEN OE$ = OE$ + "+" !Check for +
10190 IF K% = 38% THEN OE$ = OE$ + "-" !Check for -
10200 IF K% = 39% THEN OE$ = OE$ + "*" !Check for *
10210 IF K% = 40% THEN OE$ = OE$ + "/" !Check for /
10220 IF K% = 59% OR K% = 60% THEN OE$=LEFT(OE$,LEN(OE$)- 1%) !Delete?
10230 IF K% > 54% AND K% < 58% THEN OE$ = OE$ + " " !Check for Space
10240 IF K% > 50% AND K% < 54% THEN 10250 ELSE 10110 !Check for Enter
10250 PRINT CPOS(16,1)+"Your last entry:"; ES$+"K";OE$;!Display message
10260 RETURN
10270 !
11000 ! ** Subroutine to Display Characters **
11010 !
11100 A = A + 2               !Increment line number
11110 B = 13                  !Set column number
11120 FOR I = X TO Y          !Loop on range of characters
11130 PRINT CPOS(A,B) + CHR$(I); !Display character
11140 B = B + 6               !Increment column
11150 NEXT I                  !Next character
11160 RETURN
11170 !
12000 ! ** Subroutine to Create Graphic Character Strings **
12010 !
12100 FOR I = 1 TO X          !Loop on set of graphic parts
12110 T$ = T$ + A$            !Add set to temporary string
12120 NEXT I                  !Next set
12130 RETURN
12140 !
32767 END
```

Section 10
Program Chaining

10-1. INTRODUCTION

10-2. Specific tasks to be executed sequentially are handled by physically separate programs. Rather than use a GOSUB routine call, a RUN statement is used to call a separate program into main memory to be executed. This section describes the conventions and programming techniques for program chaining.

10-3. Program chaining is useful when the size of a program becomes larger than conveniently fits into user memory. Dividing a large program into a number of smaller logical segments allows more variable storage and processing capability in each program segment.

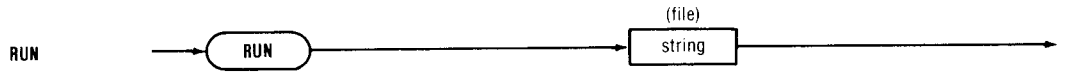
10-4. OVERVIEW

10-5. This section describes the RUN statement, the COM statement, virtual array files, and sequential data files in a chained program structure. Cross references are given for additional information available in this manual on the RUN statement, virtual arrays, and the OPEN and CLOSE statements.

10-6. STATEMENT DEFINITIONS

10-7. Program statements described below are used for chaining multiple programs that are stored on the floppy or electronic disk. The Immediate Mode use of RUN is described in the Introduction section.

10-8. The RUN Program Statement



10-9. The RUN statement is used to branch from one program to another.

- The file name of the program follows RUN. The file name may be any legal string expression.

- The program will be searched for on the default System Device if the file name is not prefixed with MF0: (floppy disk) or ED0: (electronic disk). The default System Device is discussed in the Input And Output statements section.

- Error 305 results if the program file is not found.

- When the program file is located, it is loaded into main memory over the previous program, and control is transferred to it.

- Data stored in variables in the previous program is lost unless it was reserved in a common area, or stored on a file device as explained in this section.

- Other references in this manual: Introduction.

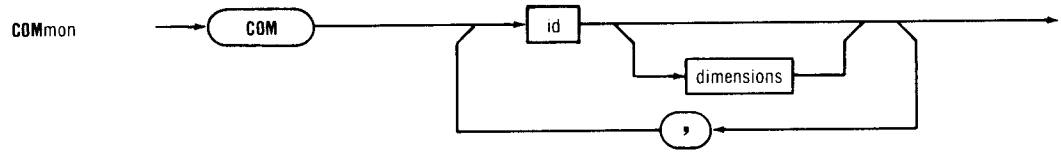
10-10. The following example searches for a program file named TEST2 on the System Device. If it is located, TEST2 is loaded into main memory and executed.

```
1050    RUN "TEST2"                          ! Chain to program TEST 2
1060    END
```

10-11. The following example searches for a program file under a name stored in string B$. If the file is located, it is loaded into main memory and executed.

```
1050    RUN B$                               ! Chain to program named by B$
1060    END
```

10-12. The COM Statement



10-13. COM reserves variables and arrays in a common area for reference by chained programs.

- Only floating point and integer variables may be stored in the common area.

- String variables may not be stored in the common area.

- String variables may be stored in virtual arrays for access by chained programs. This technique is discussed in this section.

- All programs accessing a common area must use COM statements that are identical in order, type, and array sizes; the actual variable names, however, may be different.

10-14. For example, assume that a chained program requires the use of three floating point simple variables, an integer simple variable, a floating point array, and an integer array defined in a previous program. The first program could use a COM statement such as:

```
10      ! Program A
20      COM  A, B, C, F%, D(24%), T%(100%)
        .
        .
        .
1050    RUN "B"
1060    END                                  ! End of program A
```

10-15. The second program could then use:

```
10      ! Program B
20      COM L1, L2, L3, Q%, K(24%), P%(100%)
        .
        .
        .
```

10-16. Note that while the names of the variables stored in the common area have changed between programs, the order and type of the variables are exactly the same.

10-17. VIRTUAL ARRAYS IN CHAINED PROGRAMS

10-18. Properly used, virtual arrays become a unique tool for controlling and keeping track of multiple task sequences. It is possible, for example, to structure a floppy disk that will continue with the next task to complete in a task list, even if it is carried to another 1720A Controller and given a RESTART. The paragraphs that follow discuss these techniques.

10-19. Introduction to Virtual Array Chaining Techniques

10-20. An advantage of using virtual arrays to pass chaining information is that even if a power failure occurs, the status of the processing performed by the set of programs is preserved in the virtual array. When the capability to survive power failures is not required, chaining information may be kept in a COM variable or array.

10-21. Virtual arrays can be used to control the execution of chained programs. When writing a set of chained programs, open a channel for a virtual array file with the number of elements matching the number of chained programs called from the main program (to chain three programs, dimension the virtual array with 3 elements -- 0, 1, and 2). Initialize this array to indicate no programs have been executed.

10-22. Each element of the array will take either an OFF or ON value which indicates whether or not its associated program has been executed yet. Each program sets an element of the array to the value which indicates that the program has been run. To begin, initialize the elements to zero then set the elements to one as each program is executed. If the programs are halted for any reason, the array values will show which program was interrupted.

10-23. In this way, a rerun of the main program would execute only those programs which have not yet been executed. Programs which were executed before the interruption will not be rerun, or will be rerun only under the circumstances indicated by the programmer.

10-24. Virtual arrays may also be used as a common storage area for a set of chained programs without using the COM statement. Once a virtual array has been initialized, any program can use the array by using the proper open, dimension, and closing sequences as described in the section on Virtual Arrays.

10-25. Example of Virtual Array Chaining Techniques

10-26. One program (named EXEC in this example) executes a set of n programs in sequence so that if a program does not execute successfully, it may be re-executed. The virtual array file in this example will be named CHAIN.BIN.

NOTE

Each of the programs (PROG1 through PROGn) can have a set of subordinate chained programs.

10-27. Assume there are five chained programs, with names PROG1, PROG2, PROG3, PROG4, and PROG5. An initialization program (called EXECIN in this example) is executed to create the file CHAIN.BIN, and to initialize the chaining information required to indicate that none of the chained programs has been executed.

```
10      !
20      ! EXECIN - Initialize chaining information for exec
30      !
40      OPEN "CHAIN.BIN" AS NEW DIM FILE 1% SIZE 1%
50      DIM #1%, A%(5%)
60      FOR I% = 1% TO 5%
70      A%(I%) = 0%                         ! Set to not-run
80      NEXT I%
90      CLOSE 1%
100     RUN "EXEC"                          ! Start program execution
110     END                                 ! Of EXECIN
```

10-28. The EXEC program below checks the elements of the array in the file CHAIN.BIN until a zero element is found:

```
10      !
20      ! EXEC - CHAIN TO THE NEXT INCOMPLETE PROGRAM
30      !
40      !
50      OPEN "CHAIN.BIN" AS DIM FILE 1%
60      DIM #1%, A%(5%)
70      FOR I% = 1% TO 5%
80      IF A%(I%) < > 0% THEN 120
90      ! FOUND A PROGRAM THAT NEEDS TO BE RUN
100     CLOSE 1%                            ! Release Channel 1
110     RUN "PROG" + NUM$(I%, "#")          ! Chain to PROG 1
120     NEXT I%
130     !
140     ! If control reaches this point,
150     ! all programs have been completed.
160     !
170     CLOSE 1%
180     KILL "CHAIN.BIN"                    ! Release file space
190     PRINT "ALL PROGRAMS COMPLETED."
200     END                                 ! Of EXEC
```

10-29. Each of the programs, PROG1 through PROG5, performs the processing required of them, but each should follow this format (PROG1 is illustrated).

```
10      !
20      ! PROG 1 - Explanation of total program function
30      !
        .
        .
        .
10000   OPEN "CHAIN.BIN" AS DIM FILE 1%
10010   DIM #1%, CH%(5%)
10020   CH%(1%) = 1%                        ! Signal that PROG1 is complete
10030   CLOSE 1%                            ! Rewrite status to file
10040   RUN "EXEC"                          ! Chain through "EXEC"
10050   END                                 ! Of PROG1
```

10-30. Using Sequential Data Files

10-31. Sequential data files may be used as common data storage areas for chained programs. The major difference between using the sequential file as opposed to the virtual array file is the difference in access methods. Sequential files are non-random and must be read from or written to sequentially. They are not array oriented and, therefore, cannot

be referenced via a dimensioned array name and element. If it is convenient to use all data in a file in the order it has been stored, a sequential data file is easier to use than a virtual data file.

10-32. For example, an instrument taking multiple readings produces data (e.g., voltages) that should be stored for processing by a separate program module. That data may be written into a sequential file by the data collection segment of the program system (assuming use of chained program techniques) in the order the data was read by the instrument. A report generating program segment may then process the data.

10-33. The principal use of sequential data files is the storage of variable length ASCII character strings. Sequential files should not be used for the storage of arbitrary binary data since some characters (CTRL/Z, Carriage Return, Line Feed) have a special meaning to the BASIC system's input and output routines.

10-34. ASCII data may be written to a sequential file or to an RS-232-C port by means of the PRINT statement. For example:

```
1030 PRINT #5%, "VALUE"; I%; ","; J%
```

10-35. It may be read by an INPUT statement:

```
4010 INPUT #3%, A%(0%..5%)
```

10-36. Note also that the contents of a sequential file may be examined or sent to a printer by the File Utility Program (FUP). This is not true of virtual array files.

Section 11
Program Debugging

11-1. INTRODUCTION

11-2. A program is seldom perfect the first time it is typed in. Errors are expected and Fluke BASIC provides several ways to detect and correct them. Programming errors are called bugs. The procedures used to find and correct errors are called debugging methods. This section describes these methods.

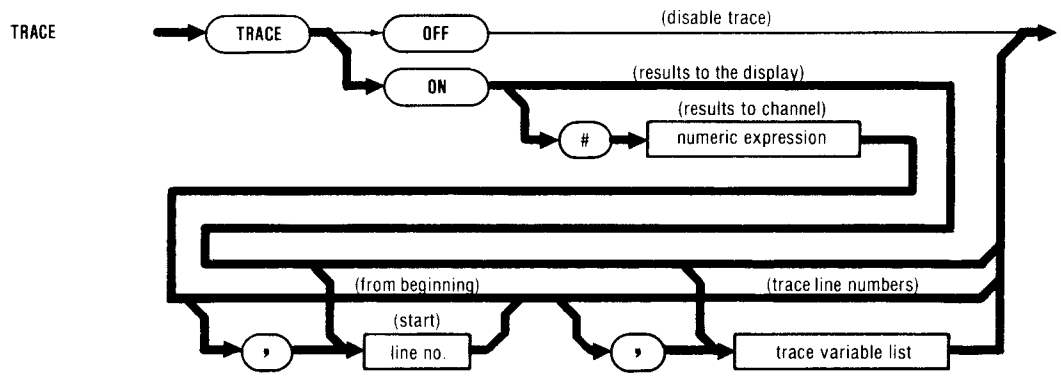
11-3. Errors fall into two categories: errors in syntax and errors in logic. Syntax errors are format or typographical errors, such as mismatched parentheses or an unrecognizable keyword. The BASIC interpreter will notify the programmer of a few of these at the time the statement is first typed in and notify him of any remaining syntax errors the first time the program is run. The interpreter does not, however, notify the programmer of logic errors which show up as incorrect program results. Without additional tools these logic errors can be difficult to locate. This section presents the tools provided for locating logic errors.

11-4. OVERVIEW

11-5. This section describes debugging tools that are primarily useful for locating logic errors. These tools allow the programmer to observe program flow and variable assignment while the program executes statements. These statements do not check for syntax errors. They are designed to simplify the task of locating logic errors in the program.

11-6. DEBUGGING TOOLS

11-7. The TRACE ON Statement



11-8. TRACE prints a record of line numbers encountered or changes in variable values.

- If a previously opened channel is specified, the results of the trace are sent to the channel. Otherwise the results are sent to the display.
- A starting line number for tracing may be specified. If it is not specified, tracing starts with the first line following the execution of the TRACE statement.
- Tracing is activated when the specified start line, or the first line, is encountered.
- TRACE may be used in either Immediate or Run Mode.
- Other references in this manual: None.

11-9. Line Number Tracing

11-10. A line number trace has the following forms:

STATEMENT	MEANING
`TRACE ON`	Trace line numbers from the first line and send the results to the display.
`TRACE ON` line number	Trace line numbers from the specified line and send the results to the display.
`TRACE ON # channel`	Trace line numbers from the first line and send the results to the open channel.
`TRACE ON # channel, line number`	Trace line numbers from the specified line and send the results to the display.

- A line number trace and a variable trace will not execute concurrently.
- A line number trace occurring after a variable trace specifies a new line number after which variable tracing will resume, provided no TRACE OFF occurred in the interim.

11-11. The following examples illustrate the results of different forms of line number trace statements.

STATEMENT		RESULT
`30`	`TRACE ON`	Start a line number trace at the next line following line 30.
`500`	`TRACE ON 1275`	Start a line number trace when line 1275 is reached.
`750`	`TRACE ON # 3%, 400`	Start a line number trace when line 400 is reached. Send the trace output to channel 3.

11-12. The line number trace displays a series of numbers representing the line numbers or the statements executed. The following example illustrates typical results.

Program:

```
10      TRACE ON
20      I% = 0%
30      I% = I% + 1%
40      IF I% < 3% THEN 30
50      TRACE OFF
60      END
```

Results:

```
20
30
40
30
40
30
40
50

Ready
```

11-13. Variable Tracing

11-14. A variable trace has the following forms:

STATEMENT	MEANING
`TRACE ON` variable list	Trace changes in value of selected variables from the first line and send the resluts to the display.
`TRACE ON #` channel, variable list	Trace changes in value of selected variables from the first line and send the results to the open channel.
`TRACE ON` line number, variable list	Trace changes in value of selected variables from the specified line and send the results to the display.
`TRACE ON #` channel, line number, variable list	Trace changes in value of selected variables from the specified line and send the results to the open channel.

- If a list of variables is specified, the trace is of changes in values of those variables. Otherwise the trace is of line numbers encountered.
- A variable trace may specify one or more variables of any type: string, integer, and floating point.
- A variable trace of an array may use the form A() as the variable. A() means "TRACE ON all elements of array A".
- An array must be previously dimensioned before tracing.
- A variable trace and a line number trace will not execute simultaneously.
- Two or more variable traces will execute simultaneously. For example, TRACE ON A followed by TRACE ON B is equivalent to TRACE ON A,B.
- A variable trace occurring after a line number trace turns off the line number trace.

11-15. A variable trace statement resembles the line number trace statement except that a list of variable names is included. The following example specifies trace output to channel 2, tracing to start at line 340, and tracing of changes in values of A%, elements 3 and 4 of array B, and all of array A$:

```
30     TRACE ON #2%, 340, A%, B(3%, 4%), A$()
```

NOTE

A variable trace of an array cannot be done without first dimensioning the array with a DIM statement.

11-16. Variable trace display output takes the following form:

line number identifier type(indices) = new value

Where:

1. Line number is the number of the line in which the variable was assigned a new value.

2. Identifier is the name of the variable.

3. Type is % for integers, $ for strings.

4. Indices identify which element of the array is being traced on and displayed (for array elements only).

5. New value is the new value assigned.

11-17. This example shows the result of a trace of an array variable:

```
TRACE ON A (1, 2)
```

Displays the value of A(1,2) when it is assigned. For example,

```
220 A(1,2) = 47.3386
```

11-18. This example program illustrates the display resulting from a trace of an integer array program:

```
10      DIM A% (2%, 2%)
20      TRACE ON A%()
30      FOR I% = 0% TO 2%
40      FOR J% = 0% TO 2%
50      A% (I%, J%) = I% * J%
60      NEXT J%
70      NEXT I%
80      TRACE OFF
90      END
```

Results:

```
50  A%(0,0) = 0
50  A%(0,1) = 0
50  A%(0,2) = 0
50  A%(1,0) = 0
50  A%(1,1) = 1
50  A%(1,2) = 2
50  A%(2,0) = 0
50  A%(2,1) = 2
50  A%(2,2) = 4
```

11-19. Other Trace Options

11-20. TRACE ON line number can be used to define a trace region within a program. The example below traces the array A$ only in the subroutine starting at line 110. Until TRACE OFF is executed, TRACE ON continues to trace all variables for which a TRACE ON was executed, and continues to send trace output to the specified channel or the display.

```
10      DIM A$ (5%, 5%)
20      TRACE ON 110, A$()                ! Start tracing array A$ at
                                          line 110
30      !
40      FOR I% = 0% to 5%
50      A$ (I%, 0%) = CHR$ (ASCII (' ') + I%)
60      GOSUB 110
70      NEXT I%
80      TRACE OFF
90      STOP
100     !
110     FOR J% = 1% TO 5%
120     A$ (I%, J%) = A$ (I%, J% - 1%) + CHR$ (ASCII(' ') + I% + J%)
130     NEXT J%
140     TRACE ON 110
150     RETURN
160     END
```

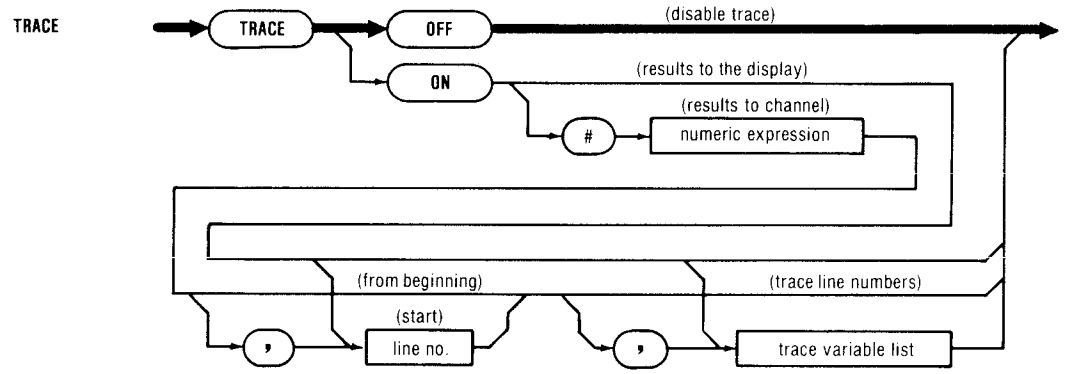
11-21. The following trace display output results from running this program. Refer to Appendix I, ASCII/IEEE-1978 Bus Codes, and note the display characters that follow SPACE, character number 32, for clarification of these results.

```
120   A$(0,1) = !
120   A$(0,2) = !"
120   A$(0,3) = !"#
120   A$(0,4) = !"#$
120   A$(0,5) = !"#$%
120   A$(1,1) = !"
120   A$(1,2) = !"#
120   A$(1,3) = !"#$
120   A$(1,4) = !"#$%
120   A$(1,5) = !"#$%&
120   A$(2,1) = "#
120   A$(2,2) = "#$
120   A$(2,3) = "#$%
120   A$(2,4) = "#$%&
120   A$(2,5) = "#$%&'
120   A$(3,1) = #$
120   A$(3,2) = #$%
120   A$(3,3) = #$%&
120   A$(3,4) = #$%&'
120   A$(3,5) = #$%&'(
120   A$(4,1) = $%
120   A$(4,2) = $%&
120   A$(4,3) = $%&'
120   A$(4,4) = $%&'(
120   A$(4,5) = $%&'()
120   A$(5,1) = %&
120   A$(5,2) = %&'
120   A$(5,3) = %&'(
120   A$(5,4) = %&'()
120   A$(5,5) = %&'()*
```

11-22. It is also possible to send trace output to different channels. Output is sent to one channel at a time. The following example illustrates this:

```
10      OPEN "TRACE1.DAT" AS NEW FILE 1%      ! First trace channel
20      OPEN "TRACE2.DAT" AS NEW FILE 2%      ! Second trace channel
100     TRACE ON A%, B$(), C()                ! Send output to console
250     TRACE ON #1%                          ! Send output to channel 1
370     TRACE ON #2%                          ! Send output to channel 2
1010    TRACE OFF                             ! Discontinue all tracing
1020    END
```

11-23. The TRACE OFF Statement



11-24. TRACE OFF disables any pending or active trace assigned in the program and destroys the variable list.

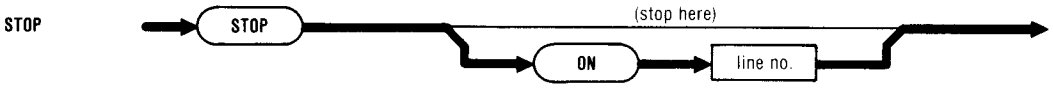
11-25. The following example illustrates that TRACE ON starts only a line number trace after TRACE OFF.

```
10    TRACE ON A, B                         ! Trace variables A and B
50    TRACE OFF                             ! Halt the variable trace
100   TRACE ON                              ! Start a line number trace
```

11-26. The following example illustrates a way to suspend tracing until a later point in a program.

```
10      TRACE ON A, B                       ! Trace variables A and B
50      TRACE ON 100                        ! Stop trace until line 100
100     ! Resume tracing variables A and B
```

11-27. The STOP ON Statement

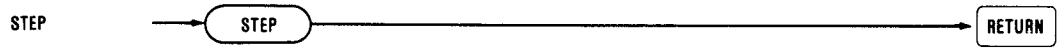


11-28. STOP ON line number, stops execution of a program.

- STOP ON line number, allows a program to be run in sections during logic debugging.
- The program stops at the line number of the STOP when ON line number is not included.
- The program stops at the line number following ON, without executing it, when ON line number is included.
- STOP ON may be executed in either Immediate or Run Mode.

- STOP ON line number enables the STEP command (see below).
- Other references in this manual: General Purpose Fluke BASIC Statements.

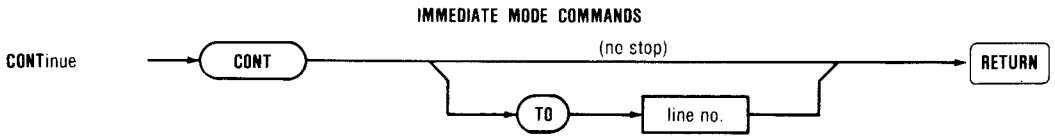
11-29. The STEP Command



11-30. The Immediate Mode STEP command sets a mode in which each statement within a program is executed individually by pressing RETURN.

- STEP must first be enabled by a breakpoint stop in a running program, caused by STOP ON or CONT TO.
- After a breakpoint stop, type STEP to select Step Mode.
- From Step Mode, type CTRL C or any Immediate Mode command to return to Immediate Mode.
- Any BASIC command or statement that is available in Immediate Mode can also be used to exit Step Mode.
- Step Mode, one statement is executed each time RETURN is pressed.
- After executing each statement, the display reads: STOP ON LINE n, where n is the next line to be executed.
- When used with a variable TRACE ON, the display will also show changes in selected variables whenever a statement assigns a new variable value.
- Other references in this manual: None.

11-31. The CONT TO Command



11-32. The Immediate Mode CONT TO line number command causes program execution to continue from a breakpoint stop caused by STOP, STOP ON, CONT TO, or CTRL C.

- The CONT TO line number command is available only in Immediate Mode.
- CONT TO line number must first be enabled by a breakpoint stop in a running program, caused by STOP, STOP ON, CONT TO, or CTRL C.
- Any subsequent action other than entering the CONT TO line number command disables the command. The program must then be rerun to the breakpoint.
- Program execution continues at the statement following the last statement executed.

- When TO line number is included, program execution again stops if the specified line number is encountered, without executing the statement.

- CONT TO can be used instead of or in addition to STOP ON and CONT, to move quickly through a subroutine or loop that has already been confirmed, during Step Mode logic debugging.

11-33. The following example program is in main memory during the interaction that follows.

```
10      FOR I% = 1% TO 2%
20      PRINT I% + I%
30      NEXT I%
40      PRINT "Done!"
50      END
```

11-34. With the above program in main memory, the following sequence of commands and RETURN entries would produce the responses shown. Programmer entry is shown to the left, and 1720A response is shown to the right.

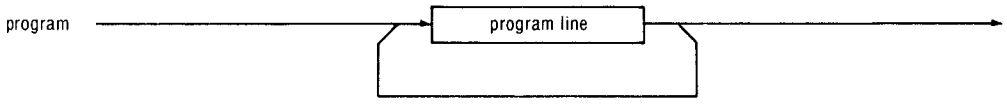
PROGRAMMER ENTRIES	1720A RESPONSES
	Ready
STOP ON 10	
	Ready
RUN	
	Stop at line 10
	Ready
STEP	
	Stop at line 10
	Ready
(RETURN)	
	Stop at line 20
	Ready
(RETURN)	
	2
	Stop at line 30
	Ready
(RETURN)	
	Stop at line 20
	Ready
(RETURN)	
	4
	Stop at line 30
	Ready
(RETURN)	
	Stop at line 40
	Ready
(RETURN)	
	Done!
	Stop at line 50
	Ready
STOP ON 10	
	Ready

PROGRAMMER ENTRIES	1720A RESPONSES
RUN	
	Stop at line 10
	Ready
CONT TO 40	
	2
	4
	Stop at line 40
	Ready
CONT	
	Done!
	Ready

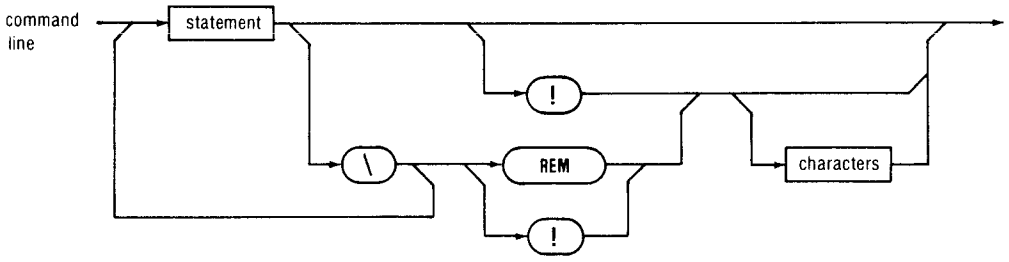
APPENDIX A
Fluke Enhanced Basic Supplementary Syntax Terminology Diagrams

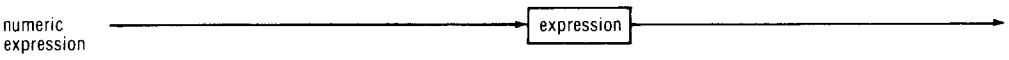
These diagrams supplement those used throughout this manual. They are in three categories: Program and Command Line Syntax, Expression Syntax, and Miscellaneous Syntax. The last category lists the aphabetically the syntactic element which complete the definitions of commands and statements to an elementary ievel.

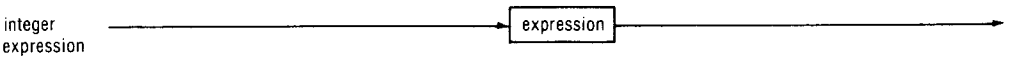
Program and Command Line Syntax

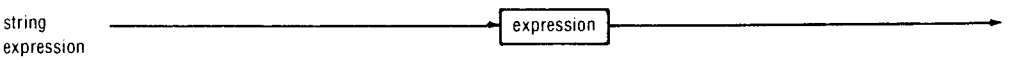


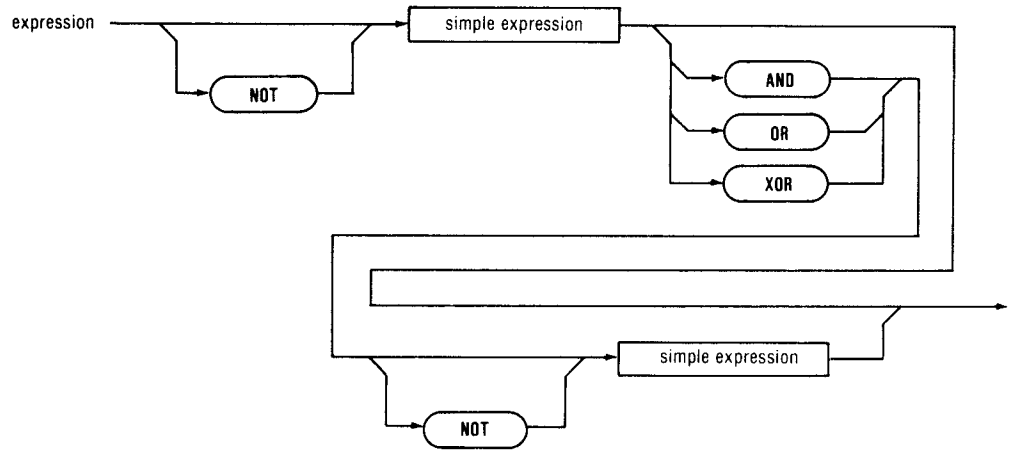
Statement: given in main sections of manual and in Reference Guide.

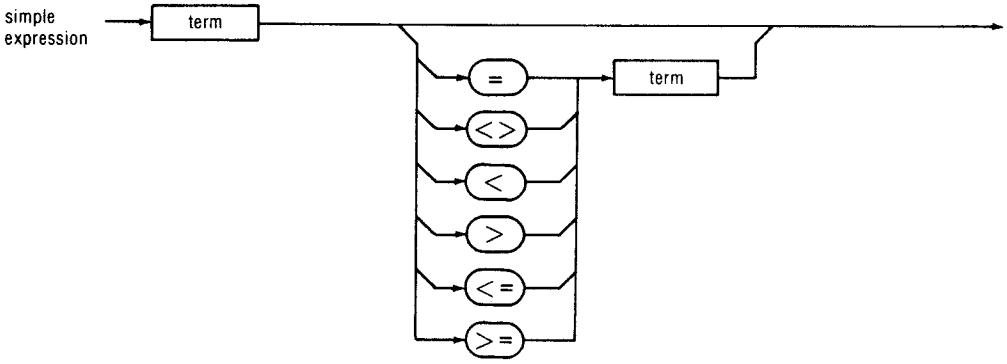


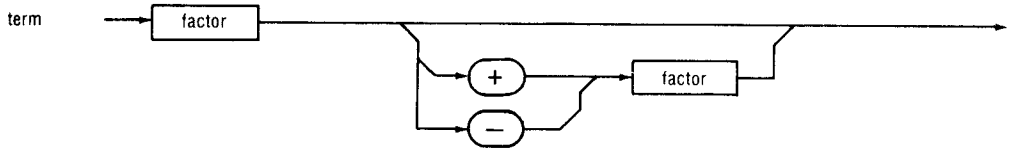


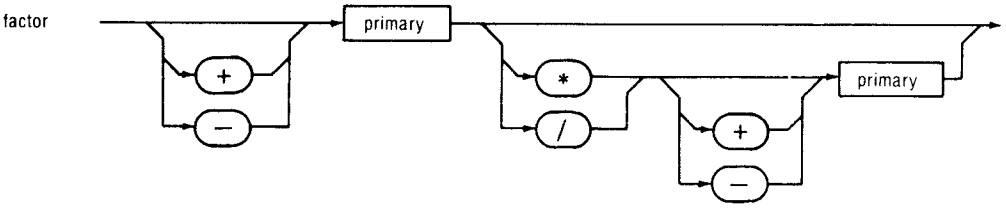


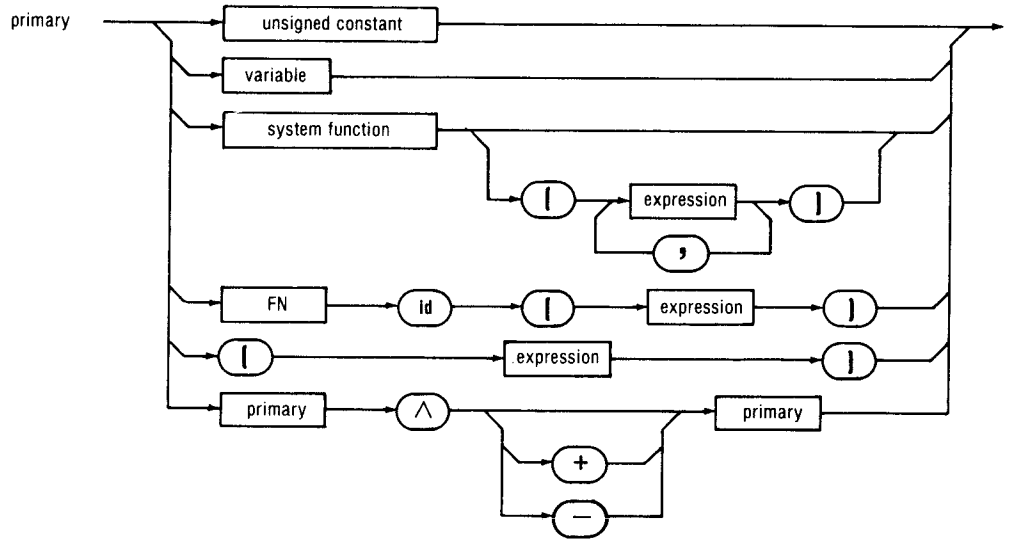



expression
NOT
simple expression
AND
OR
XOR
simple expression
NOT

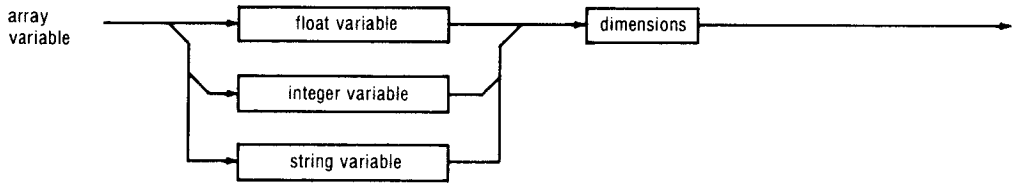

simple expression
term
=
<>
<
>
<=
>=
term

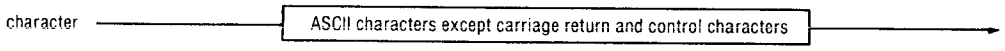

term
factor
+
−
factor

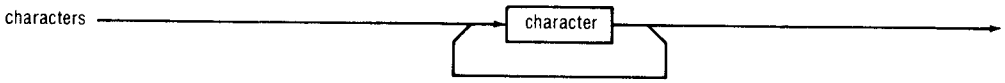

factor
+
−
primary
*
/
+
−
primary

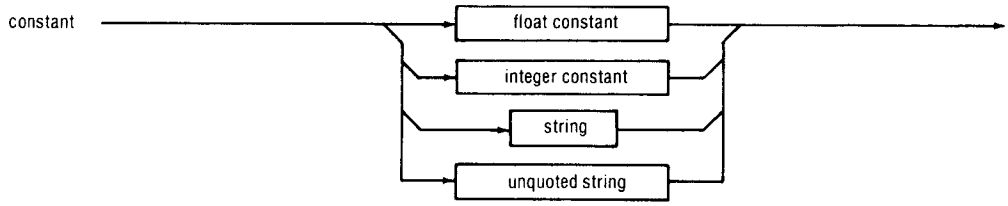


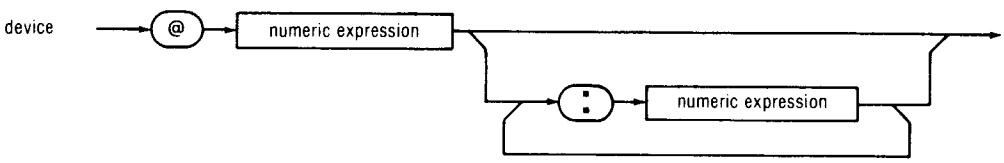
Miscellaneous Syntax

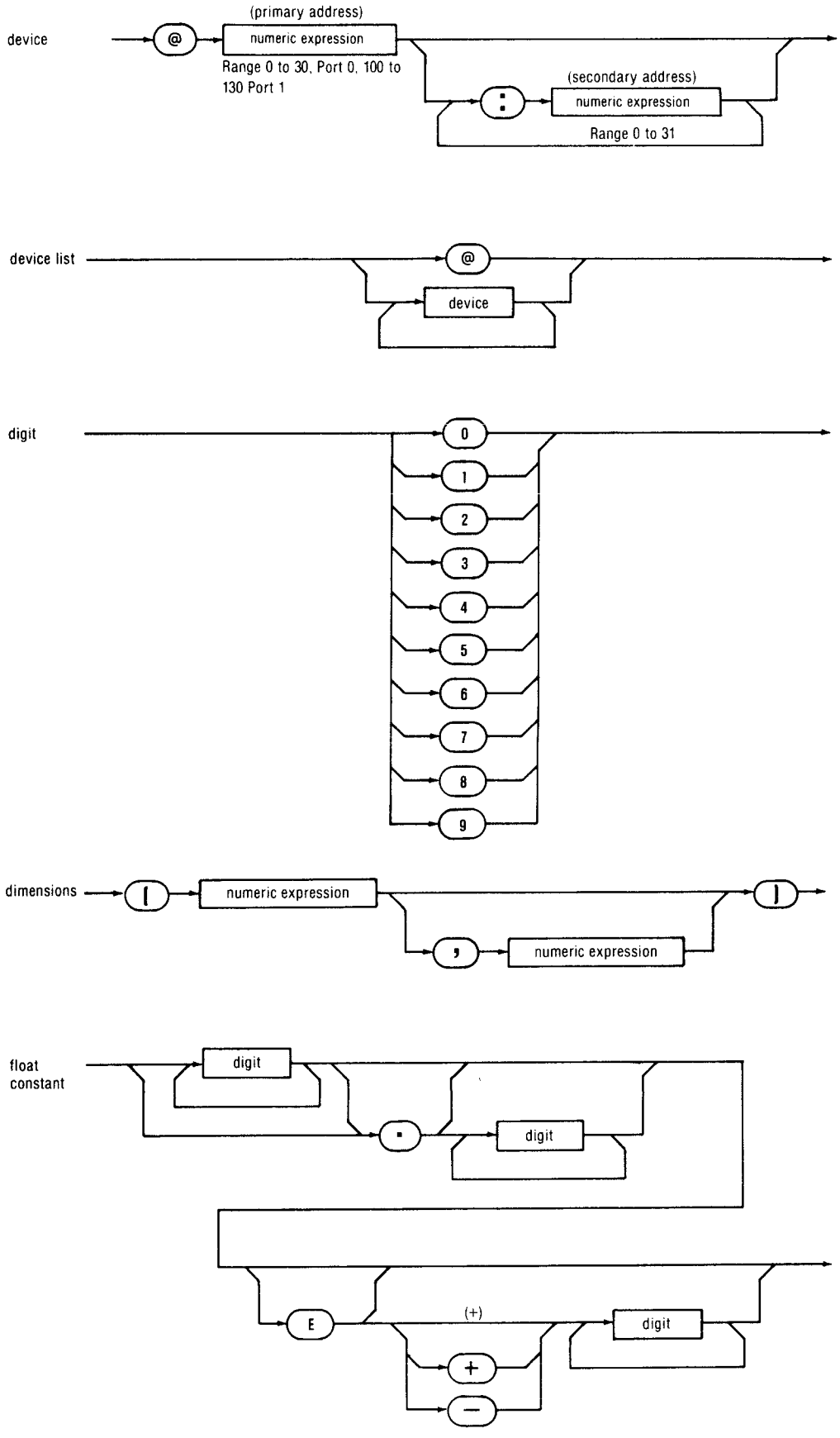


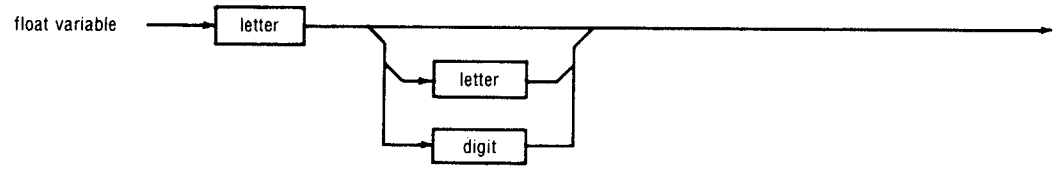


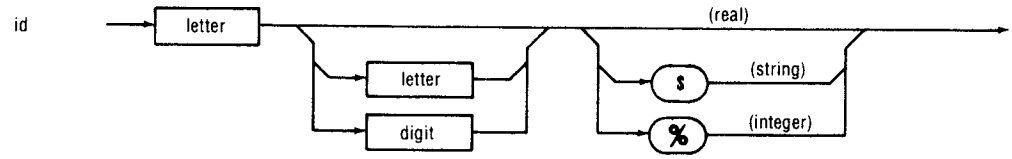


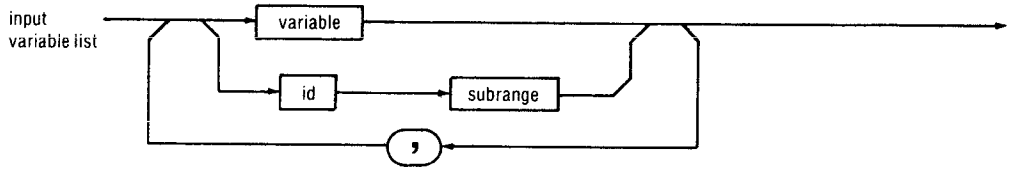


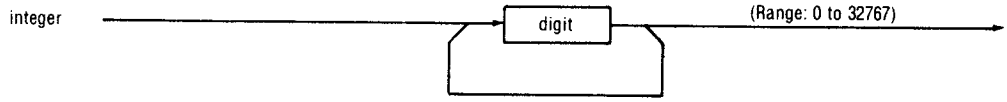


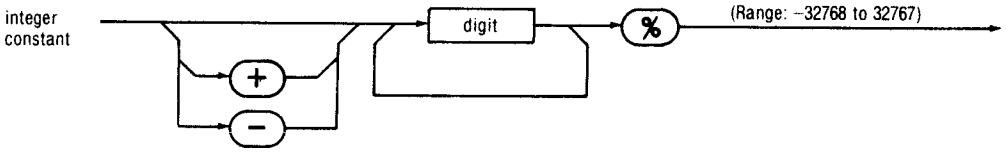

device
@
(primary address)
numeric expression
Range 0 to 30, Port 0, 100 to 130 Port 1
:
(secondary address)
numeric expression
Range 0 to 31
device list
@
device
digit
0
1
2
3
4
5
6
7
8
9
dimensions
(
numeric expression
,
numeric expression
)
float constant
digit
.
digit
E
(+)
+
−
digit

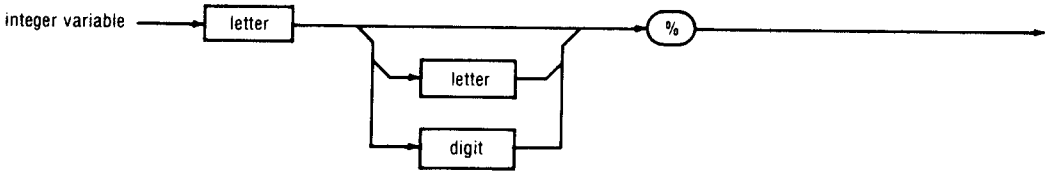

float variable
letter
letter
digit

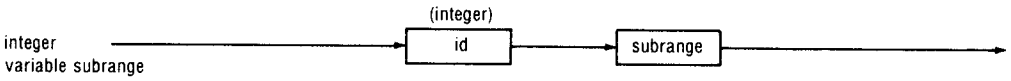

id
letter
letter
digit
(real)
$
(string)
%
(integer)

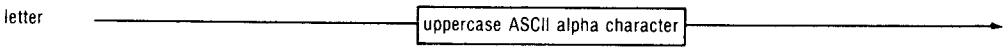

input
variable list
variable
id
subrange
,

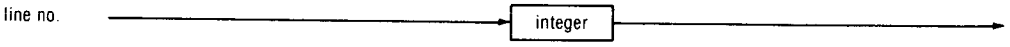

integer
digit
(Range: 0 to 32767)

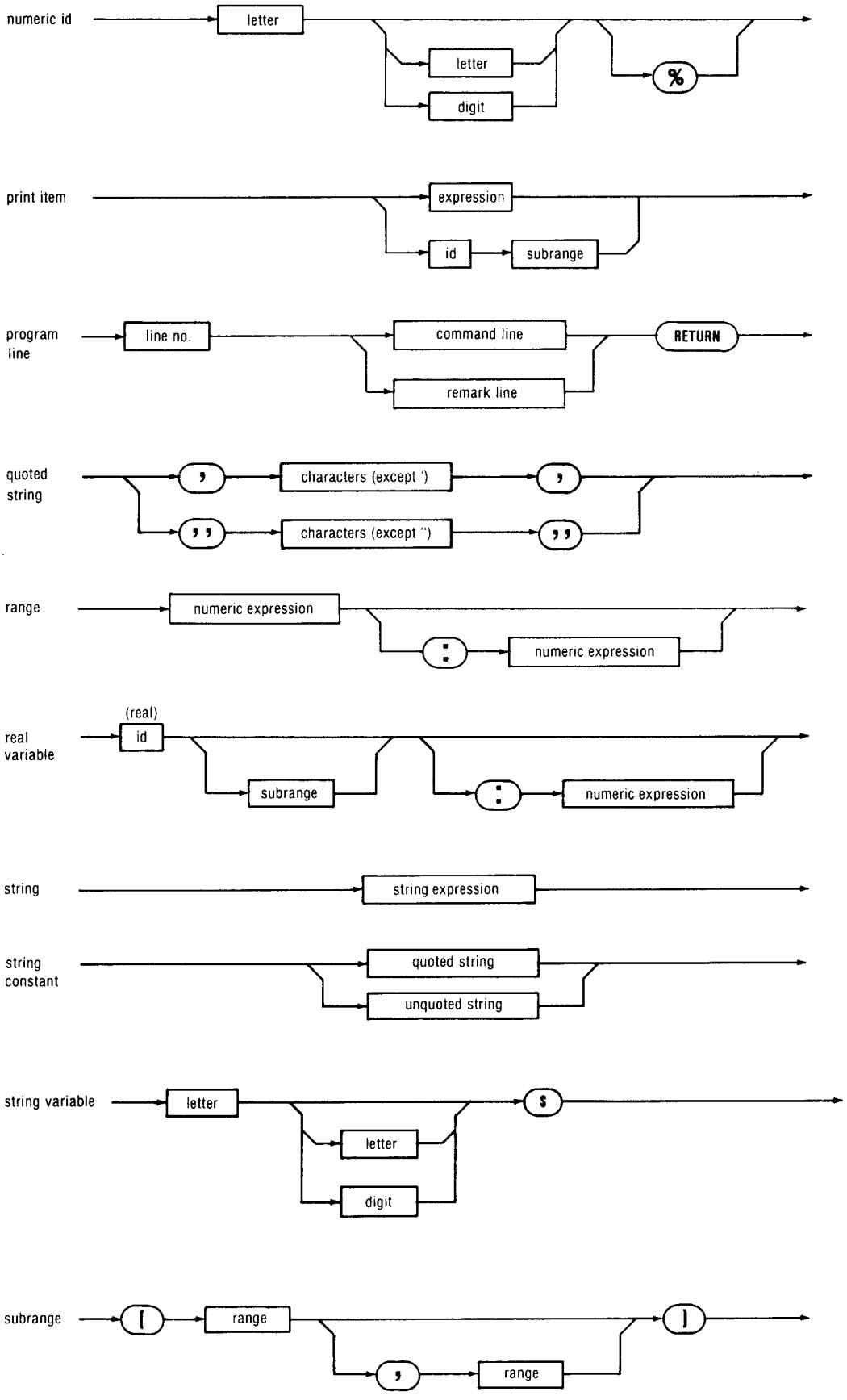

integer
constant
+
−
digit
%
(Range: −32768 to 32767)

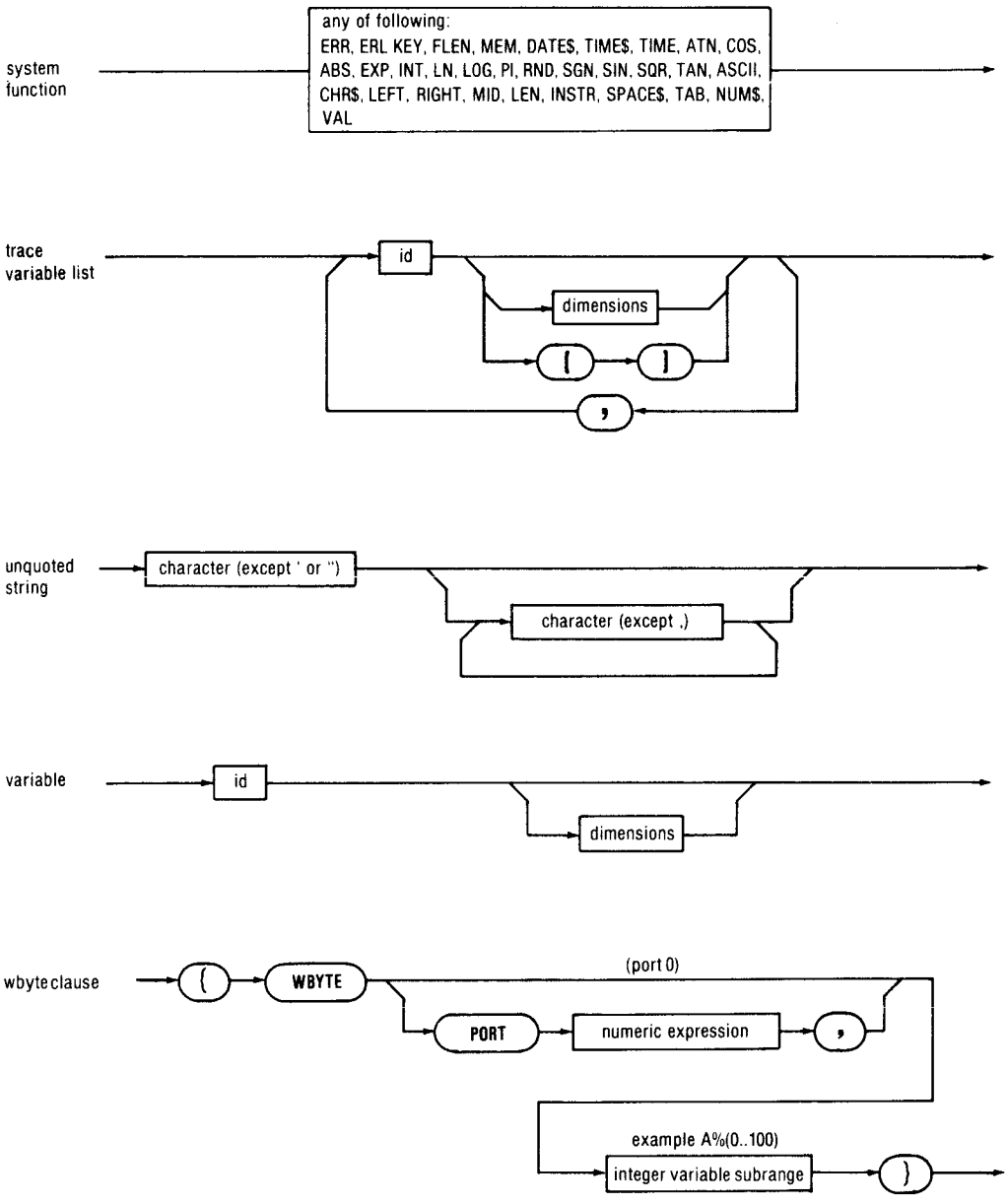

integer variable
letter
letter
digit
%


integer
variable subrange
(integer)
id
subrange


letter
uppercase ASCII alpha character


line no.
integer


numeric id
letter
letter
digit
%
print item
expression
id
subrange
program line
line no.
command line
remark line
RETURN
quoted string
'
characters (except ')
'
''
characters (except ")
''
range
numeric expression
:
numeric expression
(real)
real variable
id
subrange
:
numeric expression
string
string expression
string constant
quoted string
unquoted string
string variable
letter
letter
digit
$
subrange
(
range
,
range
)


system function
any of following:
ERR, ERL KEY, FLEN, MEM, DATE$, TIME$, TIME, ATN, COS, ABS, EXP, INT, LN, LOG, PI, RND, SGN, SIN, SQR, TAN, ASCII, CHR$, LEFT, RIGHT, MID, LEN, INSTR, SPACE$, TAB, NUM$, VAL
trace variable list
id
dimensions
[
]
,
unquoted string
character (except ' or ")
character (except ,)
variable
id
dimensions
wbyteclause
(
WBYTE
(port 0)
PORT
numeric expression
,
example A%(0..100)
integer variable subrange
)

APPENDIX B
Error Codes

ERROR CODES

TYPE	NUMBER	LEVEL	ERROR
Overflow	0	F	User storage overflow
	1	F	Virtual array declared larger than 65536 bytes
	2	F	Virtual array file smaller than delared arrays
System	100	F	BASIC interpreter error
	101	F	Incompatible lexical file
Command	200	F	Immediate mode command in RUN
	201	F	Cannot CONTinue
	202	F	STEP outside break mode
I/O	300	R	Disk not loaded
	301	R	Write protected
	302	R	Illegal channel specified
	303	R	Channel in use
	304	R	Not a valid device
	305	R	File not found
	306	R	No room on device
	307	R	Read or write past end of file
	308	R	Channel not open
	309	R	Serial port data lost
	310	R	Input line too long
	311	R	Disk or E-Disk error
	312	R	Illegal filename
	313	F	Channel not open as random file
	314	F	Sequential read/write to random file
	315	F	Virtual array assigned to non-file-structured device
	316	F	Illegal lexical filename
	317	R	Illegal directory on device
	318	R	Read from new file, or write to old file

ERROR CODES (cont)

TYPE	NUMBER	LEVEL	ERROR
Instrument Bus Control	400	R	Illegal port number
	401	R	Illegal device address
	402	R	Illegal secondary address
	403	R	Bus handshake incomplete
	404	R	Too many ports specified for function
	405	R	No devices attached to port
	406	R	No ports configured
	407	R	Specified port unavailable for function
	408	R	Timeout during Bus I/O transfer
	409	R	Illegal WBYTE data
	410	R	Parallel poll bit number out of range
	411	R	Parallel poll bit sense not zero or one
	412	R	Bus timeout value out of range
	413	R	TERM string longer than 1 character
Syntax	500	F	Unrecognizable
	501	F	Illegal character
	502	F	Illegal subscript
	503	F	Mismatched parentheses
	504	F	LET
	505	F	IF
	506	F	Line number
	507	F	PRINT
	508	F	PRINT USING
	509	F	INPUT
	510	F	DIMension
	511	F	DEFine
	512	F	FOR
	513	F	FOR without NEXT
	514	F	NEXT without FOR
	515	F	Mismatched quotes
	516	F	Illegal expression
	517	F	OPEN
	518	F	CLOSE
	519	F	Instrument bus command
	520	F	COM
	521	F	Illegal statement structure
	522	F	Illegal variable name
	523	F	ON
	524	F	OFF
	525	F	TRACE
	526	F	Illegal file size
	527	F	RENumber parameter
	528	F	RENumber
	529	F	ELSE without IF
	530	F	NEXT
	531	F	INPUT WBYTE needs a bus address
	532	F	Illegal subrange specification
	533	F	Illegal RBYTE/WBYTE data type
	534	F	Cannot specify column for RBYTE/WBYTE

ERROR CODES (cont)

TYPE	NUMBER	LEVEL	ERROR
Syntax (cont.)	535	F	Scalar illegal for RBYTE/WBYTE
	536	F	Virtual array illegal with RBYTE/WBYTE
	537	F	2-dimensional array illegal with RBYTE/WBYTE
	538	F	CONFIG
	539	F	RBYTE
	540	F	RBYTE increment <=0
	541	F	Illegal RBYTE cycle length
	542	F	WBYTE clause
	543	F	Illegal data type for RBIN or WBIN
	544	F	WAIT
Math	600	F	Mode mixing
	601	R	Arithmetic overflow
	602	R	Arithmetic underflow
	603	R	Divide by zero
	604	R	Square root of negative
	605	R	Exponent too large
	606	R	Log argument zero or negative
	607	R	Trig function argument too large
	608	R	Improper argument(s) in power
Transfer	700	F	GOTO or GOSUB
	701	F	RETURN without GOSUB
	702	F	RESUME without interrupt
	703	F	Undefined function call
	704	R	On...GOTO range
Input	800	R	Out of data in READ
	801	W	Too much data typed
	802	W	Not enough data typed
	803	W	Illegal character in input or VAL argument
	804	F	DATA syntax
Variable	900	F	Variable does not exist
	901	W	DIMension of variable already in use
	902	R	Subscript out of range
	903	F	COMmon of variable already in use
	904	W	String longer than virtual array field
	9905	F	Incompatible COM declaration

F = Fatal R = Recoverable W = Warning

APPENDIX C
Internal Structure Of Variables

C-1. INTRODUCTION

C-2. The purpose of this appendix is to provide some information about the means of data storage used by the BASIC processor which may be helpful in applications programming.

C-3. VARIABLE STORAGE MEMORY REQUIREMENTS

C-4. Each variable used in a BASIC program is entered in one of several symbol tables (depending upon its type) so that, when a reference to the variable occurs, the BASIC interpreter has available the information needed to find the variable's value or address in memory. The amount of space taken by each variable's symbol table entry depends upon the variable type, organization (simple or dimensioned), and where in memory the variable is stored (e.g., COMmon or virtual array). We will call the symbol table information "overhead" in the following paragraphs; the overhead for each type of variable is detailed in Table C-1.

Table C-1. Variable Storage Memory Requirements

Simple variable	4 bytes
Simple variable in COMmon	6 bytes
Dimensioned variable	8 bytes
Dimensioned variable in COMmon	10 bytes
Virtual array	14 bytes

C-5. In addition to the symbol table information, additional memory is used to hold the value of the variable itself depending upon the variable type. Table C-2 details the memory requirements for each variable type for all variables EXCEPT those stored in virtual arrays.

Table C-2. Additional Memory Requirements

Integer	2 bytes
Floating-point	8 bytes
String	4 bytes + 18 bytes for every 16 characters in string

C-6. The amount of memory required to represent a variable in main memory (i.e., NOT in a virtual array) may be calculated using the following formula:

m = s + n * l

NOTE

m = total memory required (bytes)

s = symbol table overhead (bytes)

n = number of values represented (1 for a simple variable, or the number of elements in an array.

l = length of a variable of that type (bytes).

C-7. The following examples illustrate the process of calculating memory requirements.

1. A simple integer variable requires 6 = 4 + (1 * 2) bytes.

2. A floating-point array dimensioned to (19,39), which has a total of 800 = (19 + 1) * (39 + 1) elements, would require a total of 6408 = 8 + (800 * 8) bytes.

3. A simple string (having no characters assigned to it) would require 8 = 4 + (1 * 4) bytes. If the string were asssigned 50 characters, an additional 72 (i.e., 4 * 18) bytes would be used to hold the value, since for every 16 characters (or for any fragment of the string shorter than 16 characters) a total of 18 bytes is required.

4. A string array dimensioned to (24) contains a total of 25 = 1 + 24 elements. If all elements of the array were set to the null string (having no characters assigned), the memory required would be 108 = 8 + (25 * 4) bytes. As the elements of the array were assigned non-null values, EACH array element's requirements would have to be calculated using the rule that every 16 characters requires 18 bytes of additional storage.

C-8. The memory requirements for a virtual array file may be calculated by the formula:

m = n * v + 512

NOTE

m = total memory required (bytes)

n = number of variables declared in this virtual array file

v = virtual array variable's symbol table overhead (14 bytes).

C-9. The requirements for a virtual array file are basically the size of the I/O buffer (512 bytes) which must be created when the file is OPENed (and which disappears when the file is CLOSEd) plus 14 bytes (the symbol table overhead for each array declared within the file) for every array in the file. In the case of

multiple DIMension statements referring to the same channel (see section 6 regarding the "equivalencing" of virtual arrays), only one 512 byte buffer is created, but 14 bytes is still required for every array definition.

C-10. I/O BUFFER MEMORY REQUIREMENTS

C-11. Every time an OPEN statement is executed, some of the available memory is allocated to provide a temporary data holding area called a buffer. The size of the buffer created depends upon whether or not the device is file-structured. The memory requirements are:

File-structured device (e.g., disk or E-disk) 512 bytes
Other devices (e.g., console or RS-232 port) 82 bytes

C-12. If insufficient memory is available, the BASIC interpreter will report a memory overflow error (error number 0). Whenever a CLOSE statement is executed, the memory used by the buffer is released for other uses.

C-13. COMMON MEMORY REQUIREMENTS

C-14. The COMmon area is a region of memory holding only the values of variables, i.e., no symbol table data is present. During program chaining, for example, the symbol table is destroyed, but the COMmon area remains intact, so that another program has access to COMmon data via the COM statement. The memory required by the COMmon area may be calculated as:

m = (nr * r) + (ni * i)

NOTE

m = total memory required by COMmon (bytes)

nr = the total number of floating-point values contained in COMmon

r = the length of a floating-point value (8 bytes)

ni = the total number of integer values contained in COMmon

i = the length of an integer value (2 bytes).

C-15. The COMmon area will remain intact until the BASIC interpreter ceases execution (via the EXIT statement, or when the CTRL/P key is pressed) or until a DELETE ALL statement is executed, which re-initializes the BASIC interpreter.

APPENDIX D
IEEE-488 Bus Messages

D-1. INTRODUCTION

D-2. The IEEE-488 standard interface is a bus-structured interconnection method. The bus has sixteen signal lines divided as follows: eight data lines, five bus management lines, and three handshake lines. In addition to these messages lines there are eight ground lines.

D-3. DATA LINES

D-4. The DATA lines carry raw data, Universal Commands, Address Commands, Addressed Commands, and Device-dependent Commands. Table D-1 presents the various commands which can be sent on the data lines in command mode (determined by the state of the ATN Bus Management Line line). Refer to Appendix I for the actual codes involved.

Table D-1. Command Messages

Universal Commands (for all devices)

- LLO-Local Lockout. Disables device LOCAL switches.
- DCL-Device Clear. Clears each device to manufacturer's default status.
- PPU-Parallel Poll Unconfigure. Unconfigures all devices with programmable remote configuration capability.
- SPE-Serial Poll Enable. Causes a device to enter Serial Poll mode.
- SPD-Serial Poll Disable. Disables data I/O lines from Serial Poll status.

Address Commands (or Addresses for each device comparing data I/O lines)

- MLA-My Listen Address. Causes a Device to become a listener.
- UNL-Unlisten. Disables a device from listener status.
- MTA-My Talk Address. Causes a device to become a talker.
- OTA-Other Talk Address. Disables a device from talker status if any other device has received an MTA on data I/O lines 1-5.
- UNT-Untalk. Disables a device from talker status whether or not another talker has been assigned.

Table D-1. Command Messages (cont.)

Addressed Commands (For Addressed-to-Listen devices)

- GTL-Go To Local. Returns device from REMOTE mode.
- SDC-Selected Device Clear. Clears a device to manufacturer's default status.
- PPC-Parallel Poll Configure. Sets up device to be configured with the PPE or PPD messages.
- GET-Group Execute Trigger. Initiates a device function to a selected, addressed group of devices.
- TCT-Take Control. A talker device becomes a Controller In Charge.

Secondary Commands (for all devices)

- MSA-My Secondary Address. Follows an MTA or MLA to allow the device extended listener or talker capability.
- OSA-Other Secondary Address. Like an MSA except will not unaddress if the first MTA equalled its address switch settings.
- PPD-Parallel Poll Disable. Unconfigures device.
- PPE-Parallel Poll Enable Configures device.

Device-Dependent Commands

- DAB-Data Byte. Any byte on the data I/O lines transferred between a talker and one or more listeners.
- EOS-End of String. Line feed or carriage return written across the data I/O lines to indicate an end of a logical string.
- NUL-Null. Zeros across the data I/O lines to initialize the bus.
- PPR-Parallel Poll Response. Response to a request for a Parallel Poll.
- STB-Status Byte. Response to an SPE (Serial Poll Enable). Concurrent with RQS.
- SRQ-Service Request. Response to an SPE if Service is requested.

D-5. BUS MANAGEMENT LINES

D-6. The bus management lines each have specific management or data transfer control functions. Table D-2 presents the five specific commands which call for some immediate action or flag a condition existing on the interface. Each command corresponds to the bus wire of the same name.

Table D-2. Bus Management Lines

- ATN-Attention. Specifies how data is to be interpreted.
- IFC-Interface Clear. Resets interface of all devices to a known state.
- REN-Remote Enable. Prepares all devices for accepting remote (from the Controller) commands.
- EOI-End or Identify. Indicates that a talker device has ended a multi-byte transfer or that a controller is in a polling sequence.
- SRQ-Service Request. Indicates that a device wants to interrupt the current sequence of events for attention.

D-7. HANDSHAKE LINES

D-8. The three handshake lines are used to synchronise data transfers. Table D-3 describes the three handshake lines.

Table D-3. Handshake Lines

- DAV-Data Valid. Indicates that a device has available and valid data on the DATA line.
- NRFD-Not Ready for Data. Indicates that a listener device is not ready to accept data.
- NDAC-Not Data Accepted. Indicates that a listener device has not yet accepted data.

NOTE

Any of these messages may also be given in the 'not' form to indicate the reverse message meaning. For instance, DAC would indicate that data has been accepted.

D-9. COMMAND MESSAGE SEQUENCES

D-10. The various BASIC statements described in the IEEE-488 Bus Input and Output Statements section initiate certain message sequences on the bus. Table D-4 presents these message sequences.

Table D-4. BASIC IEEE-488 Bus Command Message Sequences

BASIC COMMAND	MESSAGE SEQUENCE
CLEAR (by port)	$\overline{\text{EOI}}$ ATN UNL UNT DCL

Table D-4. BASIC IEEE-488 Bus Command Message Sequences (cont.)

BASIC COMMAND	MESSAGE SEQUENCE
CLEAR (by device)	$\overline{\text{EOI}}$ ATN UNL UNT MLA (for each device) SDC
CONFIG (to line with sense)	$\overline{\text{EOI}}$ ATN UNL UNT MLA PPC PPE UNL
CONFIG (by device for unconfigure)	$\overline{\text{EOI}}$ ATN UNL UNT MLA PPD UNL
INIT (by port)	REN $\overline{\text{ATN}}$ IFC $\overline{\text{IFC}}$ (after 100 usec wait) $\overline{\text{EOI}}$ ATN UNL UNT PPU
LOCAL (by port)	$\overline{\text{REN}}$
LOCAL (by device)	$\overline{\text{EOI}}$ ATN UNL UNT MLA (for each device) GTL
LOCKOUT (by port)	REN $\overline{\text{EOI}}$ ATN LLO
REMOTE (by port)	REN

Table D-4. BASIC IEEE-488 Bus Command Message Sequences (cont.)

BASIC COMMAND	MESSAGE SEQUENCE
REMOTE (by device)	REN $\overline{\text{EOI}}$ ATN UNL UNT MLA (for each device
TRIG	$\overline{\text{EOI}}$ ATN UNL UNT MLA (for each device) GET
SPL	$\overline{\text{EOI}}$ ATN UNL UNT MTA SPE $\overline{\text{ATN}}$ (accept status byte) ATN UNT SPD
PPL	ATN EOI (Accept poll status) $\overline{\text{EOI}}$
PRINT USING	$\overline{\text{EOI}}$ ATN UNL
NOTE *Issued only if device(s) specified in PRINT statement.*	
	MLA (for each device) $\overline{\text{ATN}}$ (output data)
INPUT LINE WYTE	(output WBYTE data) $\overline{\text{EOI}}$
NOTE *Issued only if device specified with INPUT statements.*	

Table D-4. BASIC IEEE-488 Bus Command Message Sequences (cont.)

BASIC COMMAND	MESSAGE SEQUENCE
	ATN UNL UNT MTA $\overline{\text{ATN}}$ (accept data byte) (output WBYTE data) $\overline{\text{EOI}}$ $\overline{\text{ATN}}$ (accept data byte) . . . (output WYBTE data) $\overline{\text{EOI}}$ $\overline{\text{ATN}}$ (accept data)
RBYTE	$\overline{\text{EOI}}$ $\overline{\text{ATN}}$ (accept data)
WBYTE	EOI (set as required) ATN (set as required) (output data byte) EOI (set as required) ATN (set as required) (output data byte) . . .

APPENDIX E
Wbyte Decimal Equivalents

NOTE

Refer to the WBYTE discussion in the IEEE-488 Bus Input and Output Statements. Use these decimal values when building up an array of data bytes to be output to the IEEE-488 Bus.

Table E-1. Decimal Equivalents

BUS MESSAGE	DECIMAL EQUIVALENT
EOI	256
ATN	512
GTL	513
GET	520
DCL	532
UNL	575
UNT	607

Table E-2. Address Messages

DEVICE ADDRESS	MLA	MTA	MSA	DEVICE ADDRESS	MLA	MTA	MSA
0	544	576	608	16	560	592	624
1	545	577	609	17	561	593	625
2	546	578	610	18	562	594	626
3	547	579	611	19	563	595	627
4	548	580	612	20	564	596	628
5	549	581	613	21	565	597	629
6	550	582	614	22	566	598	630
7	551	583	615	23	567	599	631
8	552	584	616	24	568	600	632
9	553	585	617	25	569	601	633
10	554	586	618	26	570	602	634
11	555	587	619	27	571	603	635
12	556	588	620	28	572	604	636
13	557	589	621	29	573	605	637
14	558	590	622	30	574	606	638
15	559	591	623	31	575	607	639

APPENDIX F
Parallel Poll Enable Codes

NOTE

Refer to the CONFIG statements discussion before using Table F-1.

Table F-1. Parallel Poll Enable Codes

	RESPONSE LINE	DECIMAL
SENSE "0"	DI01	96
	DI02	97
	DI03	98
	DI04	99
	DI05	100
	DI06	101
	DI07	102
	DI08	103
SENSE "1"	DI01	104
	DI02	105
	DI03	106
	DI04	107
	DI05	108
	DI06	109
	DI07	110
	DI08	111

APPENDIX G
Display Controls

FLUKE ENHANCED BASIC DISPLAY CONTROLS

TYPE	ACTION	PRINT STATEMENT
CURSOR CONTROLS	Up n lines	PRINT CHR$(27);"[nA";
	Down n lines	PRINT CHR$(27);"[nB";
	Right n columns	PRINT CHR$(27);"[nC";
	Left n columns	PRINT CHR$(27);"[nD";
	Direct line, column	PRINT CHR$(27);"[l;cH"; or PRINT CPOS(l%,c%);
	Down 1 line (scroll)	PRINT CHR$(27);"D";
	Up 1 line (scroll)	PRINT CHR$(27);"M";
	Next line (scroll)	PRINT CHR$(27);"E";
ENHANCEMENTS	Enhancements off	PRINT CHR$(27);"[m";
	High intensity	PRINT CHR$(27);"[1m";
	Underline	PRINT CHR$(27);"[4m";
	Blinking	PRINT CHR$(27);"[5m";
	Reverse image	PRINT CHR$(27);"[7m";
MODES	Normal size	PRINT CHR$(27);"[p";
	Double size	PRINT CHR$(27);"[1p";
	Graphics OFF	PRINT CHR$(27);"[2p";
	Graphics ON	PRINT CHR$(27);"[3p";
	Enable Keyboard	PRINT CHR$(27);"[4p";
	Disable Keyboard	PRINT CHR$(27);"[5p";
ERASING	To line end	PRINT CHR$(27);"[K";
	From line start	PRINT CHR$(27);"[1K";
	All of line	PRINT CHR$(27);"[2K";
	To display end	PRINT CHR$(27);"[J";
	From display start	PRINT CHR$(27);"[1J";
	All of display	PRINT CHR$(27);"[2J";

NOTES:

1. Save typing. Pre-define ES$=CHR$(27)+"["

2. Multiple enhancement or mode commands are semicolon-separated. Example: PRINT E$;1;5;7;

3. Display controls are introduced by "ESCape [" (scrolling controls do not use "[".) Any method that sends the required character sequence to the

display will give the above results. For example, if a data file is created of "ESC [" character sequences, typing the name of the file in FUP mode will cause the display response.

APPENDIX H
Graphic Mode Characters

FLUKE ENHANCED BASIC GRAPHICS MODE CHARACTERS

CHARACTER	NORMAL SIZE	DOUBLE SIZE	CHARACTER NAME
0			Top Right Corner
1			Top Left Corner
2			Bottom Right Corner
3			Bottom Left Corner
4			Top Intersect
5			Right Intersect
6			Left Intersect
7			Bottom Intersect
8			Horizontal Line
9			Vertical Line
:			Crossed Line

NOTES:

1. To enable Graphics Mode, send the display ESC [3p
   In BASIC, PRINT CHR$(27);"[3p";

2. To disable Graphics Mode, send the display ESC [2p

   In BASIC, PRINT CHR$(27);"2p";

3. In Graphics Mode, characters in the left column are displayed as shown.

4. Use the character names as defined to construct illustrations that do not change form between normal and double size.

APPENDIX I
ASCII/IEEE-488-1978 Bus Codes

ASCII/IEEE-488-1978 BUS CODES

1720A DSPLY	DECIMAL	OCTAL	HEX	BINARY 7654 3210	MESSAGE (ATN = TRUE)		DEV. NO.
NUL	0	000	00	0000 0000	ADDRESSED COMMANDS	——	
β	1	001	01	0000 0001		GTL	
γ	2	002	02	0000 0010		——	
δ	3	003	03	0000 0011		——	
ε	4	004	04	0000 0100		SDC	
ζ	5	005	05	0000 0101		PPC	
ι	6	006	06	0000 0110		——	
BEL	7	007	07	0000 0111		——	
BS	8	010	08	0000 1000		GET	
HT	9	011	09	0000 1001		TCT	
LF	10	012	0A	0000 1010		——	
VT	11	013	0B	0000 1011		——	
FF	12	014	0C	0000 1100		——	
CR	13	015	0D	0000 1101		——	
ο	14	016	0E	0000 1110		——	
π	15	017	0F	0000 1111		——	
ρ	16	020	10	0001 0000	UNIVERSAL COMMANDS	——	
σ	17	021	11	0001 0001		LLO	
τ	18	022	12	0001 0010		——	
υ	19	023	13	0001 0011		——	
φ	20	024	14	0001 0100		DCL	
χ	21	025	15	0001 0101		PPU	
ψ	22	026	16	0001 0110		——	
ω	23	027	17	0001 0111		——	
Ω	24	030	18	0001 1000		SPE	
↓	25	031	19	0001 1001		SPD	
→	26	032	1A	0001 1010		——	
ESC	27	033	1B	0001 1011		——	

1720A DSPLY	DECIMAL	OCTAL	HEX	BINARY 7654 3210	MESSAGE (ATN=TRUE)		DEV. NO.
↑	28	034	1C	0001 1100		——	
←	29	035	1D	0001 1101		——	
	30	036	1E	0001 1110		——	
/	31	037	1F	0001 1111		——	
SPACE	32	040	20	0010 0000	LISTEN ADDRESSES	MLA	0
!	33	041	21	0010 0001		MLA	1
"	34	042	22	0010 0010		MLA	2
#	35	043	23	0010 0011		MLA	3
$	36	044	24	0010 0100		MLA	4
%	37	045	25	0010 0101		MLA	5
&	38	046	26	0010 0110		MLA	6
'	39	047	27	0010 0111		MLA	7
(	40	048	28	0010 1000		MLA	8
)	41	049	29	0010 1001		MLA	9
*	42	050	2A	0010 1010		MLA	10
+	43	051	2B	0010 1011		MLA	11
,	44	054	2C	0010 1100		MLA	12
-	45	055	2D	0010 1101		MLA	13
.	46	056	2E	0010 1110		MLA	14
/	47	057	2F	0010 1111		MLA	15
0	48	060	30	0011 0000	LISTEN ADDRESSES	MLA	16
1	49	061	31	0011 0001		MLA	17
2	50	062	32	0011 0010		MLA	18
3	51	063	33	0011 0011		MLA	19
4	52	064	34	0011 0100		MLA	20
5	53	065	35	0011 0101		MLA	21
6	54	066	36	0011 0110		MLA	22
7	55	067	37	0011 0111		MLA	23
8	56	070	38	0011 1000		MLA	24
9	57	071	39	0011 1001		MLA	25
:	58	072	3A	0011 1010		MLA	26
;	59	073	3B	0011 1011		MLA	27
<	60	074	3C	0011 1100		MLA	28
=	61	075	3D	0011 1101		MLA	29
>	62	076	3E	0011 1110		MLA	30
?	63	077	3F	0011 1111		UNL	
@	64	100	40	0100 0000		MTA	0
A	65	101	41	0100 0001		MTA	1
B	66	102	42	0100 0010		MTA	2
C	67	103	43	0100 0011		MTA	3

1720A DSPLY	DECIMAL	OCTAL	HEX	BINARY 7654 3210	MESSAGE (ATN=TRUE)		DEV. NO.
D	68	104	44	0100 0100	TALK ADDRESSES	MTA	4
E	69	105	45	0100 0101		MTA	5
F	70	106	46	0100 0110		MTA	6
G	71	107	47	0100 0111		MTA	7
H	72	110	48	0100 1000		MTA	8
I	73	111	49	0100 1001		MTA	9
J	74	112	4A	0100 1010		MTA	10
K	75	113	4B	0100 1011		MTA	11
L	76	114	4C	0100 1100		MTA	12
M	77	115	4D	0100 1101		MTA	13
N	78	116	4E	0100 1110		MTA	14
O	79	117	4F	0100 1111		MTA	15
P	80	120	50	0101 0000	TALK ADDRESSES	MTA	16
Q	81	121	51	0101 0001		MTA	17
R	82	122	52	0101 0010		MTA	18
S	83	123	53	0101 0011		MTA	19
T	84	124	54	0101 0100		MTA	20
U	85	125	55	0101 0101		MTA	21
V	86	126	56	0101 0110		MTA	22
W	87	127	57	0101 0111		MTA	23
X	88	130	58	0101 1000		MTA	24
Y	89	131	59	0101 1001		MTA	25
Z	90	132	5A	0101 1010		MTA	26
[	91	133	5B	0101 1011		MTA	27
\	92	134	5C	0101 1100		MTA	28
]	93	135	5D	0101 1101		MTA	29
^	94	136	5E	0101 1110		MTA	30
—	95	137	5F	0101 1111		UNT	
`	96	140	60	0110 0000	SECONDARY ADDRESSES	MSA	0
a	97	141	61	0110 0001		MSA	1
b	98	142	62	0110 0010		MSA	2
c	99	143	63	0110 0011		MSA	3
d	100	144	64	0110 0100		MSA	4
e	101	145	65	0110 0101		MSA	5
f	102	146	66	0110 0110		MSA	6
g	103	147	67	0110 0111		MSA	7
h	104	150	68			MSA	8
i	105	151	69	0110 1001		MSA	9
j	106	152	6A	0110 1010		MSA	10
k	107	153	6B	0110 1011		MSA	11

1720A DSPLY	DECIMAL	OCTAL	HEX	BINARY 7654 3210	MESSAGE (ATN=TRUE)		DEV. NO.
l	108	154	6C	0110 1100		MSA	12
m	109	155	6D	0110 1101		MSA	13
n	110	156	6E	0110 1110		MSA	14
o	111	157	6F	0110 1111		MSA	15
p	112	160	70	0111 0000	SECONDARY ADDRESSES	MSA	16
q	113	161	71	0111 0001		MSA	17
r	114	162	72	0111 0010		MSA	18
s	115	163	73	0111 0011		MSA	19
t	116	164	74	0111 0100		MSA	20
u	117	165	75	0111 0101		MSA	21
v	118	166	76	0111 0110		MSA	22
w	119	167	77	0111 0111		MSA	23
x	120	170	78	0111 1000		MSA	24
y	121	171	79	0111 1001		MSA	25
z	122	172	7A	0111 1010		MSA	26
{	123	173	7B	0111 1011		MSA	27
|	124	174	7C	0111 1100		MSA	28
}	125	175	7D	0111 1101		MSA	29
~	126	176	7E	0111 1110		MSA	30
▩	127	177	7F	0111 1111			

APPENDIX J
Glossary and List of Mnemonics

NOTE

The words in CAPITALS within the text are terms defined elsewhere in this glossary.

ADDRESS can refer to a BUS ADDRESS, DEVICE ADDRESS, PROGRAM ADDRESS, or DATA ADDRESS. In each case, an address is a coded number representing the location of an item.

ADDRESS COMMAND is a BUS command from a CONTROLLER commanding an instrument at a designated address to talk or listen.

ADDRESSED COMMAND is a BUS command from a CONTROLLER intended for all instruments that have been addressed as TALKER or LISTENER.

APPLICATION PROGRAM is a user written PROGRAM designed to perform specified functions in a working environment.

ASCII is the American Standard Code for Information Interchange. ASCII is a standardized code set of 128 characters, including full alphabetic (upper and lower case), numerics, and many useful control characters.

ASYNCHRONOUS DATA is transmitted at random times, normally one character at a time. See SYNCHRONOUS DATA.

BASIC (Beginner's All-Purpose Symbolic Instruction Code) is a general purpose, HIGH-LEVEL LANAGUAGE that has been widely accepted because of its versatility and the ease with which it can be learned. Fluke BASIC has added commands for instrument control.

BAUD RATE is serial transfer rate in bits per second including all framing bits used to identify the start and end of characters or messages.

BINARY is a number system based on zero (0) and one (1) representations. It is often used to represent DATA or instruction codes. Counting in binary looks like 0, 1, 10, 11, 100, 101, 110, 111, 1000, ...

BIT is a contraction of binary digit. A BIT is either a one or a zero and represents the smallest single unit of computer information. Bits are often used in groups of eight to represent ASCII characters.

BLOCK is 512 DATA or PROGRAM BYTES.

BOOTSTRAP is a short program usually permanently recorded in ROM whose only function is to read an OPERATING SYSTEM from BULK STORAGE into system memory and transfer control to it.

BUFFER is a temporary storage area in MAIN MEMORY used to store input and output DATA.

BULK STORAGE is a device attached to a computer that can store much more PROGRAM or DATA information than the computer's main memory can hold. The 1720A Controller incorporates two types of bulk storage: Electronic Disk and Floppy Disk.

BUS is the IEEE-488-1978 standardized interconnection system used for connecting instruments into a system. Also, BUS can refer to any set of parallel connections that have the same meaning for each unit connected to them.

BUS ADDRESS is a 7-bit code placed on the BUS in COMMAND MODE to designate an instrument as a TALKER or LISTENER.

BYTE is a grouping of eight BITS of information into a coded representation of all or part of a number or instruction. Often a 7-bit ASCII character is referred to as a byte, with the eighth bit available for PARITY if needed. BYTES are commonly considered as 8-bit storage areas to represent ASCII characters.

CHAINING is a method of operating a PROGRAM that is larger than available MAIN MEMORY. The technique is to break the program into smaller elements, and call in the next element from BULK STORAGE as each succeeding element is completed. Requires highly modular programming to be effective.

CHANNEL is a communication path opened between an APPLICATION PROGRAM and a FILE or a SYSTEM DEVICE.

CHARACTER STRING is a grouping of ASCII characters.

COMMAND FILE is a FILE that, when designated active by FDOS, is used as a substitute for keyboard inputs. In the 1720A, the Command File has the name COMMND.SYS, and is processed only once each time the 1720A is initialized by RESTART or Power-On.

COMMAND MODE is selected when a CONTROLLER sets the ATN (attention) line. In this mode, instruments on the BUS are addressed or unaddressed as TALKERS and LISTENERS.

CONSOLE MONITOR is the system manager for the 1720A Controller. The CONSOLE MONITOR allows the user access to available system SOFTWARE, such as FUP, or BASIC.

CONSTANTS are fixed values which may be FLOATING POINT, INTEGER or STRING DATA types.

CONTROL CHARACTERS are used to produce specific actions such as terminating program execution, exiting from the Editor, halting and restarting scrolling.

CONTROLLER is a device connected to a BUS capable of designating instruments as TALKERS or LISTENERS by using BUS message sequences. A DEVICE does not need to be programmable to act as a CONTROLLER. However, a CONTROLLER needs to examine the DATA or status of instruments to determine appropriate conditions for designation changes. There can be only one active CONTROLLER on a BUS at one time.

CPU (Central Processor Unit) is the controlling instruction and data processor in any computer system. In the 1720A Controller, the CPU is the microprocessor and its supporting components located on the CPU circuit module.

CRT (Cathode Ray Tube) is the display screen on the 1720A front panel.

CURSOR is the visible pointer on the CRT display that allows the user to recognize the position being pointed to by the system SOFTWARE.

DATA is numerical information that has been collected for interpretation by a PROGRAM.

DATA BASE is a stored and defined collection of DATA that is made available for report generation or further calculations by a PROGRAM.

DATA BASE MANAGEMENT SYSTEM (DBMS) is any systematic approach to storing, updating, and retrieving information as a common DATA BASE for many users.

DATA FILE is a FILE holding either RANDOM or SEQUENTIAL ACCESS information.

DATA LINES are eight of the sixteen BUS lines which carry either data or multiline BUS messages (UNIVERSAL, ADDRESSED and ADDRESS COMMANDS).

DATA MODE is the normal mode of the BUS where the CONTROLLER, if there is one, has left the ATN (attention) line false. All transfers of DATA or instructions are between instruments.

DATA PROCESSING is the capability of performing calculations upon collected DATA and formatting it into readable reports.

DEBUGGING includes all methods used to detect and correct SYNTAX and structure errors in a PROGRAM.

DEFAULT is the option that the SYSTEM SOFTWARE selects when the user does not specify an option.

DEVICE is a hardware resource that can act as a source or destination of DATA. In this manual, DEVICE is used in two different ways. The first way is identified by two letters, a number, and a colon. This represents internal devices recognized by FDOS. The second way is identified by "@" followed by a number from 0 to 30. This represents external devices, such as instruments connected to the BUS.

DEVICE ADDRESS is a number used by a PROGRAM to designate an external device for data transfer.

DISPLAY CONTROLS are ANSI-standard character sequences of ASCII characters which produce a desired display effect such as cursor position or reverse image.

EDITOR is a SYSTEM SOFTWARE program that enables a user to generate and update an APPLICATION PROGRAM.

EIA is the Electronic Industries Association, publishers of Standard RS-232-C for SERIAL DATA PORTS.

ELECTRONIC DISK on the 1720A Controller is a BULK STORAGE device that uses dynamic RAM memory in a design that functionally emulates a FLOPPY DISK. The electronic disk is at least 10 times as fast as the floppy disk and has no moving parts to wear or cause noise.

EPROM is a ROM that can be erased and reprogrammed by an equipment manufacturer using specialized equipment. The letters mean Erasable Programmable Read Only Memory.

ESCAPE SEQUENCES are strings of characters including an ESC character, a numeric parameter and a function code which are recognized as DISPLAY CONTROLS.

EXPRESSIONS are combinations of data-names, numeric literals, and named constants, joined by one or more arithmetic operators in such a way that the expression as a whole can be reduced to a single numeric value.

EXTENDED LISTENER is a LISTENER instrument that requires a two-BYTE address. See SECONDARY COMMANDS.

EXTENDED TALKER is a TALKER instrument that requires a two-BYTE address. See SECONDARY COMMANDS.

FDOS is the Floppy Disk OPERATING SYSTEM program. FDOS operates in MAIN MEMORY and is stored in BULK STORAGE.

FILE is a collection of information designated by name as a unit.

FILE-STRUCTURED DEVICE is any BULK MEMORY device where PROGRAMS and DATA may be stored and retrieved via a SYSTEM DIRECTORY.

FILE UTILITY PROGRAM (FUP) is a file management program provided with the standard 1720A Controller software package.

FIRMWARE refers to computer programs and data that are recorded in permanent memory. See ROM.

FLAG is a condition indicator. System flags can be used to indicate the presence of COMMAND FILES or to indicate a state of system readiness.

FLOATING POINT is a means of representing numeric values in a manner which is useable for most calculations. They are characterized by wide range (up to 308 places from decimal) and high resolution (up to 15 places). Note that the inexactness of floating point representation occasionally must be considered. For example, IF 7*(1/7)=1 will evaluate false. See INTEGER.

FLOPPY DISK is a BULK STORAGE recording device that uses a flexible mylar disk similar to recording tape to record PROGRAMS and DATA. The location of information on the disk is identified by track (distance from center) and sector (pie-shaped radial subdivision).

HANDSHAKE refers to the 3-wire HARDWARE PROTOCOL used to exchange DATA on the BUS.

HANDSHAKE LINES are three of the sixteen BUS lines which are given the mnemonics DAV, NRFD, and NDAC to indicate a remote instrument's readiness to send or receive data.

HEXADECIMAL is a number system based on 16 rather than 10 digits. Sometimes called hex, the system uses A, B, C, D, E, and F, to represent the additional numbers, e.g., A hex = 10 decimal. Counting in hexadecimal looks like: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B... Hexadecimal is useful because binary numbers can be broken into 4-BIT groups and read directly, e.g., 1011 binary = B hex.

HIGH LEVEL LANGUAGE is any programming language that requires conversion through a COMPILER or INTERPRETER into MACHINE CODE instructions. Examples of high-level languages are BASIC or Pascal.

IEEE is the Institute of Electrical and Electronic Engineers, publishers of standard 488-1978 used for interconnecting instruments to the 1720A Controller through the BUS.

IEEE-488-1978 is a BUS standard agreed upon by participating instrument manufacturers for the interconnections of instruments into a functional system. The standard is maintained and published by the Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY, 10017, USA. Also known as GPIB (General Purpose Interface Bus).

IMMEDIATE MODE is a method to use BASIC directly as each line is typed in rather than storing a sequence of lines as a PROGRAM for later execution. In Immediate Mode, line numbers are not used and each line is executed as soon as the RETURN key is pressed.

INTEGER variables are used to represent integral quantities. The are characterized by limited range (-32768 to 32767). Integers are normally used for event counting, or for comparisons where exactness is required. See also FLOATING POINT.

INTERFACE is a hardware and software technique to interconnect a device to a system. For example, in the 1720A, the DMA/Floppy Interface is necessary to allow the system to gain access to the Floppy Disk.

INTERPRETER is a system software program that interprets the statements of a HIGH LEVEL LANGUAGE program (such as BASIC).

LEXICAL FILE is an intermediate form of an APPLICATION PROGRAM that occupies less space and eliminates some processing steps for the Fluke BASIC INTERPRETER.

LISTENER is a BUS device designated by a CONTROLLER to receive DATA or instruct a designated TALKER or CONTROLLER. There can be more than one LISTENER on a BUS at the same time.

LOADER is a program which places another program into MAIN MEMORY for execution.

LOGICAL EXPRESSION is an EXPRESSION containing variables, constants, function references, etc. separated by LOGICAL OPERATORS and parentheses.

LOGIC OPERATORS are functions that perform comparisons, selections, matching, etc. In BASIC, the logical operators are AND, OR, NOT, and XOR. These are used for either Boolean operation or for bit-manipulation.

MACHINE CODE is the coded BIT-patterns of directly executable MACHINE DEPENDENT computer instructions, represented by numbers or zero-one (0, 1) patterns (BINARY).

MACHINE-DEPENDENT programs will operate only on a particular model of computer.

MACHINE-INDEPENDENT programs will operate on any computer system that has the necessary hardware and supporting SOFTWARE.

MAIN MEMORY is the RAM memory from which the microcomputer directly executes all instructions and which is used for fast, intermediate storage of DATA or PROGRAM mode.

MANAGEMENT LINES are five of the sixteen lines on a BUS. These lines are given the mnemonics ATN, IFC, REN, EOI, and SRQ, and call for an immediate and specific action or flag a condition existing on the BUS.

OPERATING SYSTEM is a computer program that manages the resources of computer through task scheduling, I/O handling and file management. See FDOS.

OPERATOR is a term for symbols within an APPLICATION PROGRAM such as + or < that identify operations to be performed.

OPERATOR'S KEYBOARD on the 1720A is the TOUCH-SENSITIVE DISPLAY.

PARALLEL POLL is a method of simultaneously checking the status of up to 8 instruments on a BUS by assigning each instrument a DATA LINE to transmit a service request.

PARITY is a scheme of error detection that used one extra BIT for each unit of information (such as a BYTE). The extra (parity) bit is set to one (zero) if the total number of one-bits is even (odd).

PORT is an external connector on any device that is used for transfer of data.

PRIMARY COMMANDS are the ASCII characters typically used as BUS commands.

PROGRAM is any meaningful sequence of computer instructions that cause a SYSTEM to accomplish a desired task.

PROM (Programmable Read Only Memory) is a ROM that can be programmed by an equipment manufacturer using specialized equipment.

PROTOCOL is a set of rules for exchange of information between a SYSTEM and a device or between a system and a system.

RAM stands for RANDOM ACCESS MEMORY, but by common usage it has come to mean the high-speed VOLATILE semiconductor memory that is normally used for SYSTEM and USER MEMORY.

RANDOM ACCESS is a method where each word of a file can be accessed via its own discrete ADDRESS. See also SEQUENTIAL ACCESS.

RASTER is the scanning pattern of an electron beam on a CRT display. A raster display uses the same scan pattern all the time, forming images by turning the beam on and off at appropriate times.

ROM stands for Read Only Memory, used for permanently recorded computer programs and data.

RS-232-C is an interconnection standard agreed upon by participating manufacturers of data communication equipment for the transfer of serial digital data between Data Communication Equipment (DCE) and Data Terminal Equipment (DTE).

SCIENTIFIC NOTATION is a system for describing real or integer numbers via a shorthand form of FLOATING POINT notation.

SECONDARY COMMANDS are BUS commands used to increase the ADDRESS length of EXTENDED TALKERS and LISTENERS to 2 BYTES.

SERIAL DATA is transmitted one BIT at a time over a single wire at a predefined BAUD RATE.

SEQUENTIAL ACCESS is a method of accessing DATA in a file by looking at each piece of DATA, in order, until a match is found. See also RANDOM ACCESS.

SEQUENTIAL FILE is a SEQUENTIAL ACCESS file.

SERIAL POLL is a method of sequentially determining which instrument on a BUS has requested service. One instrument at a time is checked via the eight DATA LINES.

SERIAL PORT is an external PORT that conforms to the industry standard RS-232-C. Normally, ASYNCHRONOUS ASCII codes are used unless otherwise desired.

SIMPLE VARIABLE is a Fluke BASIC program variable that is either an integer or floating point value (not a character string) and contains only one value (not dimensional).

SOFT-SECTORED is a disk format where the beginning of every sector on a disk is determined by checking certain data patterns. Hard-sectored disks have predetermined sector beginnings designated by a physical marker, such as a hole.

SOFTWARE refers to computer PROGRAMS and DATA that are recorded and used on a medium that can be erased and rewritten by program command.

SOURCE is used in this manual in two different ways: the first way is the right side of a FUP command designating the input portion of a communication channel. The second way is an instrument connected to the BUS and transmitting either COMMAND MODE or DATA MODE information.

STRING variables represent collections of characters that may or may not be numeric.

STRUCTURED PROGRAMMING is a method of programming which required an initial design process that lays out the program structure in a modular form. Structured programming eliminates the 'spaghetti code' program by keeping GOTO's to a minimum and by using SUBROUTINES to structure the program into discrete, easily readable modules.

SUBROUTINES are sections of a program that perform a specific function upon request of the main program (or another subroutine). Used in BASIC via the GOSUB statement.

SYNTAX is the proper 'grammar' required for an INTERPRETER to recognize and execute a PROGRAM statement.

SYNCHRONOUS DATA is data transmitted in predetermined message block sizes with a clock signal to synchronize the receiver. See ASYNCHRONOUS DATA.

SYSTEM is any interconnection of instruments or other devices that cooperate to accomplish a task. A CONTROLLER is not an essential part of a system unless the designations of TALKERS and LISTENERS needs to be changed during the task. Other CONTROLLER capabilities such as DATA PROCESSING or acting as a centralized control point often make a CONTROLLER necessary.

SYSTEM DEVICE is the designated file-structured DEVICE on the CONTROLLER that acts as the primary file storage module. The floppy disk or Electronic Disk may be designated as the system device via the FUP Assignment option.

SYSTEM DIRECTORY is the listing of PROGRAM and DATA files on a BULK STORAGE, FILE STRUCTURED DEVICE.

SYSTEM MEMORY is RAM memory allocated for use by the OPERATING SYSTEM and UTILITIES or BASIC INTERPRETER.

SYSTEM SOFTWARE is the collection of PROGRAMS used to handle file management procedures on a SYSTEM.

TALKER is a DEVICE that has been designated by the CONTROLLER on the BUS to send data to LISTENERS.

TOUCH-SENSITIVE DISPLAY is the term used for the combination of the display screen and the touch-sensitive panel which acts as the OPERATOR'S KEYBOARD.

UNIVERSAL COMMANDS sent across the DATA LINES of a BUS affect all devices whether or not they are designated as LISTENERS.

USER MEMORY is the area reserved in MAIN MEMORY for storage and execution of user written APPLICATION PROGRAMS and DATA.

VARIABLES are representations of a quantity, or the quantity itself, which can assume any of a given set of values. They may be integer, string, or floating point value designators.

VIRTUAL ARRAY is an array stored on a FILE-STRUCTURED storage media as a RANDOM ACCESS file. VIRTUAL ARRAYS can be integer, string, or floating point array with one or two dimensions. Once a VIRTUAL ARRAY file has been opened and the virtual array has been dimensioned the array elements are handled by the programmer exactly as they are in main memory array.

List of Abbreviations and Mnemonics

ACK	Acknowledge
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
ATN	Attention (IEEE-488 Bus management line)
BASIC	Beginner's All-purpose Symbolic Instruction Code
BEL	Bell (ASCII)
BS	Backspace (ASCII)
CAN	Cancel (ASCII)
CD	Carrier Detect (RS-232-C line)
CFP	Command File Processor
CIC	Controller In Charge (IEEE-488)
CLEAR	Clear (BASIC)
CLOSE	Close (BASIC)
COM	Common (BASIC)
CONFIG	Configure (BASIC)
CONMON	Console Monitor
CONT	Continue (BASIC)
CPU	Central Processing Unit
CR	Carriage Return
CR/LF	Carriage Return/Line Feed Sequence
CRT	Cathode Ray Tube
CTRL	Control Key
CTS	Clear To Send (RS-232-C line)
DAB	Data Byte
DAC	Data Accepted
DATA	Data (BASIC)
DAV	Data Valid (IEEE-488 Bus handshake line)
DBMS	Data Base Management System
DCE	Data Communications Equipment (RS-232-C)
DCL	Device Clear (IEEE-488)
DCn	Device control 1, 2, 3, or 4 (ASCII)
DEF	Define (BASIC)
DEL	Delete (ASCII)
DELETE	Delete (BASIC)
DIM	Dimension (BASIC)
DIOn	Data Input/Output 1 through 8 (IEEE-488)
DLE	Data Link Escape (ASCII)
DMA	Direct Memory Access
DSR	Data Set Ready (RS-232-C line)
DTE	Data Terminal Equipment (RS-232-C)
DTR	Data Terminal Ready (RS-232-C line)
EDIT	Edit (BASIC)
EIA	Electronic Industries Association
ENQ	Enquiry (ASCII)
EOF	End of File
EOI	End Or Identify (IEEE-488 management line)

EOT	End of Transmission
EPROM	Eraseable Programmable Read-Only Memory
ESC	Escape
ETB	End of Transmission Block (ASCII)
ETX	End of Text
EXIT	Exit (BASIC)
FDOS	Floppy Disk Operating System
FF	Form Feed
FOR	For (BASIC)
FUP	File Utility Program
GET	Group Execute Trigger (IEEE-488)
GND	Ground
GOSUB	Go to Subroutine (BASIC)
GOTO	Go To (BASIC)
GPIB	General Purpose Interface Bus
GTL	Go To Local (IEEE-488)
HT	Horizontal Tab
IC	Integrated Circuit
IDY	Identify (IEEE-488)
IEEE	Institute of Electrical and Electronic Engineers
IF	Interface
IFC	Interface Clear (IEEE-488 Bus management line)
INIT	Initialize (BASIC)
INPUT	Input (BASIC)
I/O	Input/Output
KILL	Kill (BASIC)
LED	Light Emitting Diode
LET	Let (BASIC)
LF	Line Feed
LIST	List (BASIC)
LLO	Local Lockout (IEEE-488)
LOCAL	Local (BASIC)
LOCKOUT	Lockout (BASIC)
MLA	My Listen Address (IEEE-488)
MSA	My Secondary Address (IEEE-488)
MTA	My Talk Address (IEEE-488)
NAK	Negative Acknowledge
NDAC	Not Data Accepted (IEEE-488 Bus handshake line)
NEXT	Next (BASIC)
NRFD	Not Ready For Data (IEEE-488 Bus handshake line)
NUL	Null (ASCII)
OFF	Off (BASIC)
OLD	Old (BASIC)
ON	On (BASIC)
OPEN	Open (BASIC)
OSA	Other Secondary Address (IEEE-488)
OTA	Other Talk Address (IEEE-488)
PCG	Primary Command Group (IEEE-488)
PPC	Parallel Poll Configure (IEEE-488)
PPD	Parallel Poll Disable (IEEE-488)
PPE	Parallel Poll Enable (IEEE-488)
PPRn	Parallel Poll Response 1 through 8 (IEEE-488)
PPU	Parallel Poll Unconfigure (IEEE-488)
PRINT	Print (BASIC)

PROM	Programmable Read Only Memory
RAM	Random Access Memory
RANDOMIZE	Randomize (BASIC)
RBIN	Read Binary (BASIC)
RBYTE	Read Byte (BASIC)
READ	Read (BASIC)
REC	Received Data (RS-232-C)
REM	Remark (BASIC)
REMOTE	Remote (BASIC)
REN	Remote Enable (IEEE-488 Bus management line)
REN	Renumber (BASIC)
RESTORE	Restore (BASIC)
RESUME	Resume (BASIC)
RET	Return (RS-232-C signal reference line)
RETURN	Return (BASIC)
RFD	Ready For Data (IEEE-488)
RI	Ring Indicator (RS-232-C line)
ROM	Ready-Only Memory
RS	Record Separator (ASCII)
RTS	Request To Send (RS-232-C)
RUN	Run (BASIC)
SAVE	Save (BASIC)
SAVEL	Save Lexical (BASIC)
SCTS	Secondary Clear To Send (RS-232-C line)
SDC	Selected Device Clear (IEEE-488)
SI	Shift In (ASCII)
SO	Shift Out (ASCII)
SOH	Start Of Heading (ASCII)
SP	Space
SPD	Serial Poll Disable (IEEE-488)
SPE	Serial Poll Enable (IEEE-488)
SREC	Secondary Received Data (RS-232-C line)
SRTS	Secondary Request To Send (RS-232-C line)
SRQ	Service Request (IEEE-488)
STB	Status Byte (IEEE-488)
STEP	Step (BASIC)
STOP	Stop (BASIC)
STX	Start of Text (ASCII)
SXMT	Secondary Transmitted Data (RS-232-C line)
TCT	Take Control (IEEE-488)
TERM	Terminator (BASIC)
TRACE	Trace (BASIC)
TRIG	Trigger (BASIC)
TSD	Touch-Sensitive Display
UART	Universal Asynchronous Receiver-Transmitter
UNL	Unlisten (IEEE-488)
UNT	Untalk (IEEE-488)
US	Unit Separator (ASCII)
USEC	Microsecond
UUT	Unit Under Test
VT	Vertical Tab
WAIT	Wait (BASIC)
WBIN	Write Binary (BASIC)
WBYTE	Write Byte (BASIC)