HP Instrument BASIC Users Handbook

HEWLETT
PACKARD

HP Part No. E2083-90000
Printed in USA

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard Company (HP) shall not be liable for any errors contained in this document. HP MAKES NO WARRANTIES OF ANY KIND WITH REGARD TO THIS DOCUMENT, WHETHER EXPRESS OR IMPLIED. HP SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Warranty Information

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013.

Use of this manual and magnetic media supplied for this product are restricted. Additional copies of the software can be made for security and backup purposes only. Resale of the software in its present form or with alterations is expressly prohibited.

MS-DOS is a U.S. registered trademark of Microsoft Corporation.

Printing History

This is the second printing of the *HP Instrument BASIC Users Handbook.* Changes in this manual include support for MS-DOS^R file systems.

February 1990 - First Edition
August 1990 - Second Edition

Handbook Organization

Welcome

This manual will introduce you to the HP Instrument BASIC programming language, provide some helpful hints on getting the most use from it, and provide a general programming reference. It is divided into three books, *HP Instrument BASIC Programming Techniques, HP Instrument BASIC Interfacing Techniques,* and *HP Instrument BASIC Language Reference.* The first two books provide some introductory material on programming and interfacing. However, if you have no programming knowledge, you might find it helpful to study a beginning level programming book.

This manual assumes that you are familiar with the operation of HP Instrument BASIC's front-panel interface or keyboard and have read or reviewed the manual which describes operation of HP Instrument BASIC with your specific instrument.

HP Instrument BASIC is implemented as an "embedded controller"—that is, a computer residing inside an instrument. Hence, all references in this manual to the "computer" also refer to HP Instrument BASIC installed in an instrument.

What's In This Handbook?

HP Instrument BASIC Programming Techniques contains explanations and programming hints organized by concepts and topics. It is not a complete keyword reference. Instead it covers programming concepts, showing how to use the HP Instrument BASIC language.

For explanations and hints regarding interfacing, see the *HP Instrument BASIC Interfacing Techniques* book.

HP Instrument BASIC Language Reference contains a detailed keyword reference.

For HP BASIC Programmers

Many programmers already familiar with HP Series 200/300 BASIC will want to use the HP Instrument BASIC manual set to look up keywords and find necessary specifics about the way HP Instrument BASIC is implemented. If this is your situation, you may want to refer to the following manuals and sections as needed:

- "Graphics and Display Techniques", in your instrument-specific manual for information on using the display for graphics and text program output.
- "The HP-IB Model", in your instrument-specific manual to learn how HP Instrument BASIC interfaces with the host device, (if using an embedded controller) and its external HP-IB port.
- "Interfacing with an External Controller", in your instrument-specific manual for a description of how to transfer data between external and internal programs, how to upload and download programs and how to control HP Instrument BASIC programs from an external controller.
- "Keyword Guide to Porting", at the end of *HP Instrument BASIC Programming Techniques,* for a quick determination of what commands are implemented and how they relate to recent versions of the corresponding HP Series 200/300 BASIC command.

Most importantly, you will find a complete command reference and a list of error messages in the *HP Instrument BASIC Language Reference.* If you need to refresh your memory on any other topics, you can consult the manuals on Programming and Interfacing techniques as necessary.

Contents

1. Manual Organization

2. Program Structure and Flow

3. Numeric Computation

4. Numeric Arrays

5. String Manipulation

1

Manual Organization

Welcome

This manual is intended to introduce you to the HP Instrument BASIC programming language and to provide some helpful hints on getting the most use from it. This manual assumes that you are familiar with the operation of HP Instrument BASIC's front-panel interface or keyboard and have read or reviewed the manual which came with your instrument which describes operation of HP Instrument BASIC with your specific instrument. Most topics concerning running, recording, loading, saving and debugging programs are covered there. This manual serves as a general language reference and programming tutorial for those with some rudimentary knowledge of programming in BASIC or another language. If you have no programming knowledge, you might find it helpful to study a beginning level programming book. However, some beginners may find that they are able to start in this manual by concentrating on the fundamentals presented in the first few chapters. If you are a programming expert or are already familiar with the BASIC language of other HP computers, you may start faster by going directly to the *HP Instrument BASIC Language Reference* and checking the keywords you normally use.

HP Instrument BASIC is implemented as an "embedded controller"—that is, a computer residing inside an instrument. Hence, all references in this manual to the "computer" also refer to HP Instrument BASIC installed in an instrument.

What's In This Manual?

This manual contains explanations and programming hints organized by concepts and topics. It is not a complete keyword reference. Instead it covers programming concepts, showing how to use the HP Instrument BASIC language. *HP Instrument BASIC Language Reference* contains a detailed keyword reference. For explanations and hints regarding interfacing, see the *HP Instrument BASIC Interfacing Techniques* book.

The following section gives an overview of the chapters in this manual.

Overview of Chapters

Chapter	Topics
Chapter 2: Program Structure and Flow	This chapter describes how the execution order of programs and how to direct and control it.
Chapter 3: Numeric Computation	This chapter covers mathematical operations and the use of numeric variables.
Chapter 4: Numeric Arrays	This chapter covers numeric array operations.

Chapter 5: String Manipulation
This chapter explains the tools used for the processing of characters, words, and text in your program.

Chapter 6: Subprograms and User-Defined Functions
This chapter describes using alternate contexts (or environments), available as user-defined functions or subprograms.

Chapter 7: Data Storage and Retrieval
This chapter shows many of the alternatives available for storing the data that is intended as program input or created as program output.

Chapter 8: Using a Printer
This chapter tells how to use an external printer, and how to use formatted printing for both printer and CRT output.

Chapter 9: Handling Errors
This chapter discusses techniques for intercepting errors that might occur while a program is running.

Chapter 10: Keyword Guide to Porting
This chapter summarizes the HP Instrument BASIC keywords by categories, with differences between HP Instrument BASIC and HP Series 200/300 BASIC.

What's Not in this Manual

This is a manual of programming techniques, helpful hints, and explanations of capabilities. It is not a rigorous derivation of the HP Instrument BASIC language. Any statements appropriate to the topic being discussed are included in each chapter, whether they have been previously introduced or not. Since most users will not read this manual from cover to cover anyway, the approach chosen should not present any significant problems. In those cases when you have difficulty getting the meaning of certain items from context, consult the Index to find additional information.

2

Program Structure and Flow

There are four general categories of program flow. These are:

- Sequence
- Selection (conditional execution)
- Repetition
- Event-Initiated Branching

This chapter tells you how to use these types of program flow.

Sequence

The simplest form of sequence is linear flow. Linear flow allows many program lines to be grouped together to perform a specific task in a predictable manner. Keep these characteristics of linear flow in mind:

- Linear flow involves *no* decision making. Unless there is an error condition, the program lines will always be executed in exactly the same order.
- Linear flow is the default mode of program execution. Unless your include a statement that stops or alters program flow, the computer will always execute the next higher-numbered line after finishing the line it is on.

Halting Program Execution

There are three statements that can halt program flow.

The END Statement

The primary purpose of the END statement is to mark the end of the main program. When an END statement is executed, program flow stops and the program moves into the stopped (non-continuable) state.

The STOP Statement

The STOP statement acts like an END statement in that it stops program flow. You can use a STOP statement to halt program flow at some point other than the end of the program. When a STOP statement is executed, program flow stops and the program moves into the stopped (non-continuable) state.

The PAUSE Statement

You use the PAUSE statement to *temporarily* halt program execution, leaving the program variables intact. Execution halts until instructed to continue by the operator.

Here is an example of the use of PAUSE:

```
100  Radius=5
110  Circum=PI*2*Radius
120  PRINT INT(Circum)
130  PAUSE
140  Area=PI*Radius^2
150  PRINT INT(Area)
160  END
```

When the program runs, and the computer prints 31 on the CRT. Then when you continue, the computer prints 78 on the CRT. One common use for the PAUSE statement is in program troubleshooting and debugging. Another use for PAUSE is to allow time for the computer user to read messages or follow instructions.

Simple Branching

An alternative to linear flow is branching. Branching is simply a redirection of sequential flow. The simplest form of branching uses the statements GOTO and GOSUB. Both statements cause an unconditional branch to a specified location in a program.

Using GOTO

The GOTO statement causes the program to branch to either a line number or the line label. Following are examples of the GOTO statement:

```
30  REM GOTO branches here
     .
     .
100 GOTO 30
     .
     .
150 GOTO Label_xyz
     .
     .
300 Label_xyz:  ...
```

Using GOSUB

The GOSUB statement transfers program execution to a subroutine. A **subroutine** is simply a segment of a program that is entered with a GOSUB and exited with a RETURN. There are no parameters passed and no local variables are allowed in the subroutine.

The GOSUB is very useful in structuring and controlling programs. It is similar to a procedure call in that program flow automatically returns to the line following the GOSUB statement. The GOSUB statement can specify either the line label or the line number of the desired subroutine entry point. The following are examples of GOSUB statements:

```
100  GOSUB 1000
200  GOSUB Label_abc
       .
       .
1000 REM subroutine begins here
1010 Label_abc:
       .
       .
1500 RETURN
```

Remember that each time a subroutine is called by a GOSUB, control returns to the line immediately following the GOSUB when the RETURN is encountered in the subroutine. Note that if you omit the RETURN statement in a subroutine the program will continue executing beyond the point at which you expected it to return, until it encounters another RETURN or one of the halting statements (PAUSE, STOP, or END).

Selection

The heart of a computer's decision-making power is the category of program flow called **selection,** or **conditional execution.** As the name implies, a certain segment of the program either is or is not executed according to the results of a test or condition. This section presents the conditional-execution statements according to various applications. The following is a summary of these groupings.

- Conditional execution of one segment.
- Conditionally choosing one of two segments.
- Conditionally choosing one of many segments.

Conditional Execution of One Segment

The basic decision to execute or not execute a program segment is made by the IF ... THEN statement. This statement includes a numeric expression that is evaluated as being either true or false. If true (non-zero), the conditional segment is executed. If false (zero), the conditional segment is bypassed. Note that any valid numeric expression is allowed for the test expression.

The conditional segment can be either a single HP Instrument BASIC statement or a program segment containing any number of statements. The first example shows conditional execution of a single statement.

```
100 IF Ph>7.7 THEN PRINT "Ph Value has been exceeded!
```

Notice the test (Ph>7.7) and the conditional statement (Print "Ph Value ... ") which appear on either side of the keyword THEN. When the computer executes this program line, it evaluates the expression Ph>7.7. If the value contained in the variable Ph is 7.7 or less, the expression evaluates to 0 (false), and the line is exited. If the value contained in the variable Ph is greater than 7.7, the expression evaluates as 1 (true), and the PRINT statement is executed.

Prohibited Statements

Certain statements are not allowed as the conditional statement in a single-line IF ... THEN. The following statements are not allowed in a single-line IF ... THEN.

Keywords used in the declaration of variables:

COM DIM INTEGER REAL

Keywords that define context boundaries:

DEF FN FNEND SUB SUBEND END

Keywords that define program structures:

CASE
CASE ELSE
ELSE
END IF
END LOOP
END SELECT
END WHILE
EXIT IF
FOR
IF
LOOP
NEXT
REPEAT
SELECT
UNTIL
WHILE

Keywords used to identify lines that are literals:

DATA REM

Conditional Branching

Powerful control structures can be developed by using branching statements in an IF ... THEN. For example:

```
110  IF Free_space<100 THEN GOSUB Expand_file
120  !  The line after is always executed
```

This statement checks the value of a variable called Free_space, and executes a file-expansion subroutine if the value tested is not large enough. One important feature of this structure is that the program flow is essentially linear, except for the conditional "side trip" to a subroutine and back.

The conditional GOTO is such a commonly used technique that the computer allows a special case of syntax to specify it. Assuming that line number 200 is labeled "Start", the following statements will all cause a branch to line 200 if X is equal to 3.

```
IF X=3 THEN GOTO 200
IF X=3 THEN GOTO Start
IF X=3 THEN 200
IF X=3 THEN Start
```

When a line number or line label is specified immediately after THEN, the computer assumes a GOTO statement for that line. This improves the readability of programs.

Multiple-Line Conditional Segments

If the conditional program segment requires more than one statement, a slightly different structure is used. For example:

```
100  IF Ph>7.7 THEN
110    PRINT "The value of Ph has been exceeded!"
120    PRINT "Final Ph =";Ph
130    GOSUB Next_tube
140  END IF
150  !  Program continues here
```

If Ph is less than or equal to 7.7 the program skips all of the statements between the IF..THEN and the END IF statements and continues with the line following the END IF statement. If Ph is greater than 7.7, the computer executes the three statements between the IF ... THEN and END IF statements. Program flow then continues at line 150. Any number of program lines can be placed between a THEN and an END IF statement including other IF..END IF statements. Including other IF..END IF statements is called **nesting** or **nested constructs.** For example:

```
1000  IF Flag THEN
1010    IF End_of_page THEN
1020      FOR I=1 TO Skip_length
1030        PRINT
1040        Lines=Lines+1
1050      NEXT I
1060    END IF
1070  END IF
```

Choosing One of Two Segments

Often you want a program flow that passes through only one of two paths depending upon a condition. This type of decision is shown in the following diagram:

```
Flag = 1                          Flag = 0
   |                                 |
   |    400   IF Flag THEN          ---
   |    410     R=R+2                  |
   |    420     Area=PI*R^2            |
  ---   430   ELSE                  <--
  |     440     Width=Width+1        |
  |     450     Length=Length+1      |
  |     460     Area=WIdth*Length    |
  |     470   END IF                 |
  -->   480   Print "Area =";Area    |
   |    490   ! Program continues    |
   v                                 v
```

HP Instrument BASIC has an IF ... THEN ... ELSE structure which makes the one-of-two choice easy and readable.

Choosing One of Many Segments

The SELECT ... END SELECT is similar to the IF ... THEN ... ELSE ... END IF construct, but allows the definition of several conditional program segments. Only one segment executes each time the construct is entered. Each segment starts after a CASE or CASE ELSE statement, and ends when the next program line is a CASE, CASE ELSE, or SELECT statement.

Consider for example, the processing of readings from a voltmeter. Readings have been entered that contain a function code. These function codes identify the type of reading and are shown in the following table:

Function Code	Type of Reading
DV	DC Volts
AV	AC Volts
DI	DC Current
AI	AC Current
OM	Ohms

This example shows the use of the SELECT construct. The function code is contained in the variable Funct$. The rules about illegal statements and proper nesting are the same as those for the IF ... THEN statement.

```
2000  SELECT Funct$
2010  CASE "DV"
2020    !
2030    ! Processing for DC Volts
2040    !
2050  CASE "AV"
2060    !
2070    ! Processing for AC Volts
2080    !
2090  CASE "DI"
2100    !
2110    ! Processing for DC Amps
2120    !
2130  CASE "AI"
2140    !
2150    ! Processing for AC Amps
2160    !
2170  CASE "OM"
2180    !
2190    ! Processing for Ohms
2200    !
2210  CASE ELSE
2220    BEEP
2230    PRINT "INVALID READING"
2240  END SELECT
2250  ! Program execution continues here
```

Notice that the SELECT construct starts with a SELECT statement specifying the variable to be tested and ends with an END SELECT statement. The anticipated values are placed in CASE statements. Although this example shows a string tested against simple literals, the SELECT statement works for numeric or string variables or expressions. The CASE statements can contain constants, variables, expressions, comparison operators, or a range specification. The anticipated values, or **match items,** must be of the same type (numeric or string) as the tested variable.

The CASE ELSE statement is optional. It defines a program segment that is executed if the tested variable does not match any of the cases. If CASE ELSE is not included and no match is found, program execution simply continues with the line following END SELECT.

A CASE statement can also specify multiple matches by separating them with commas, as shown below.

```
CASE -1,1,3 TO 7,>15
```

If an error occurs when the computer tries to evaluate an expression in a CASE statement, the error is reported for the line containing the SELECT statement. An error message pointing to a SELECT statement actually means that there was an error in that line *or* in one of the CASE statements following it.

Repetition

There are four structures available for creating repetition. The FOR ... NEXT structure repeats a program segment a predetermined number of times. Two other structures, REPEAT ... UNTIL and WHILE ... END WHILE, repeat a program segment indefinitely, waiting for a specified condition to occur. The LOOP ... EXIT IF structure is used to create an iterative structure that allows multiple exit points at arbitrary locations.

Fixed Number of Iterations

The general concept of repetitive program flow can be shown with the FOR ... NEXT structure. The FOR statement marks the beginning of the repeated segment and establishes the number of repetitions. The NEXT statement marks the end of the repeated segment. This structure uses a numeric variable as a **loop counter.** This variable is available for use within the loop, if desired. The following example shows the basic elements of a FOR ... NEXT loop.

```
10   FOR X=10 TO 0 STEP -1
20    BEEP
30    PRINT X
40    WAIT 1
50   NEXT X
60   END
```

In this example, X is the loop counter, 10 is the starting value, 0 is the final value, -1 is the step size and the repeated segment is composed of lines 20 through 50. Note that if the step counter is not specified, a default value of 1 is assumed.

When all variables involved are integers, the number of iterations can be predicted using the following formula:

```
INT((Step_Size + Final_Value - Starting_Value)/(Step_Size))
```

Thus, the number of iterations in the example above is 11.

Conditional Number of Iterations

Some applications need a loop that is executed until a certain condition is true, without specifying the number of iterations involved.

For example, suppose you want to print the value of successive powers of two, but only until the value is greater than 1000. The REPEAT ... UNTIL is more flexible than the FOR ... NEXT in this case. Consider the following example program:

```
10   X=2
20   I=1
30   PRINT X;
40   REPEAT
50     X=2^(I+1)
60     I=I+1
70     PRINT X;
80   UNTIL X>1000
90   END
```

This program will calculate the value of each power of 2 until the value is greater than 1000. If you ran this program, the results would be:

```
2  4  8  16  32  64  256  512  1024
```

The WHILE loop is used for the same purpose as the REPEAT loop. The only difference between the two is the location of the test for exiting the loop. The REPEAT loop has its test at the bottom. This means that the loop is always executed at least once, regardless of the value of the condition. The WHILE loop has its test at the top. Therefore, it is possible for the loop to be skipped entirely. The following shows this.

```
10   X=2
20   I=1
30   PRINT X;
40   WHILE X<1000
50     X=2^(I+1)
60     I=I+1
70     PRINT X;
80   END WHILE
90   END
```

Arbitrary Exit Points

The looping structures discussed so far allow only one exit point. There are times when this is not the desired program flow. The LOOP..EXIT IF construct allows you to have any number of conditional exits points. Also, the EXIT IF statement can be at the top or bottom of the loop. This means that the LOOP structure can serve the same purposes as REPEAT ... UNTIL and WHILE ... END WHILE.

The EXIT IF statement must appear at the same nesting level as the LOOP statement for a given loop. This is best shown with an example. In the "WRONG" example, the EXIT IF statement has been nested one level deeper than the LOOP statement because it was placed in an IF ... THEN structure.

WRONG:

```
600  LOOP
610    Test=RND-.5
620    IF Test<0 THEN
630      GOSUB Negative
640    ELSE
650      EXIT IF Test>.4
660      GOSUB Positive
670    END IF
680  END LOOP
```

RIGHT:

Here is the proper structure to use.

```
600  LOOP
610    Test=RND-.5
620  EXIT IF Test>.4
630    IF Test<0 THEN
640      GOSUB Negative
650    ELSE
660      GOSUB Positive
670    END IF
680  END LOOP
```

Event-Initiated Branching

HP Instrument BASIC provides a tool called **event-initiated branching,** which uses interrupts to redirect program flow. Each time the program finishes a line, the computer executes an "event-checking" routine. If an enabled event has occurred, then this "event-checking" routine causes the program to branch to a specified statement).

Types of Events

Event-initiated branching is established by the ON..event statements. Here is a list of the statements:

ON ERROR an interrupt generated by a run-time error

ON INTR an interrupt generated by an an interface

ON KEY an interrupt generated by pressing a softkey

ON TIMEOUT an interrupt generated when an interface or device has taken longer than a specified time to respond to a data-transfer handshake

The following example demonstrates an event-initiated branch using the ON KEY statement.

```
100   ON KEY 1 LABEL "Inc" GOSUB Plus
110   ON KEY 5 LABEL "Dec" GOSUB Minus
120   ON KEY 8 LABEL "Abort" GOTO Bye
130   !
140   Spin:  DISP X
150     GOTO Spin
160   !
170   Plus:  X=X+1
180     RETURN
190   !
200   Minus: X=X-1
210     RETURN
220   Bye: END
```

The ON KEY statements are executed only once at the start of the program. Once defined, these event-initiated branches remain in effect for the rest of the program.

The program segment labeled "Spin" is an infinite loop. If it weren't for interrupts, this program couldn't do anything except display a zero. However, there is an implied "IF ... THEN" at the end of each program line due to the ON KEY action. As a result of softkey presses, either the "Plus" or the "Minus" subroutines are selected or the program branches to the END statement and terminates. If no softkey is pressed, the computer continues to display the value of X. The following section of "pseudo-code" shows what the program flow of the "Spin" segment actually looks like to the computer.

```
Spin: display X
 if Key1 then gosub Plus
 if Key5 then gosub Minus
 if Key9 then goto Bye
goto Spin
```

The labels are arranged to correspond to the layout of the softkeys. The labels are displayed when the softkeys are active and are not displayed when the softkeys are not active. Any label which your program has not defined is blank. The label areas are defined in the ON KEY statement by using the keyword LABEL followed by a string.

Deactivating Events

All the "ON-event" statements have a corresponding "OFF-event" statement. This is one way to deactivate an interrupt source. For example OFF KEY deactivates interrupts from the softkeys. Pressing a softkey while deactivated does nothing.

Disabling Events

It is also possible to temporarily disable an event-initiated branch. This is done when an active event is desired in a process, but there is a special section of the program that you don't want to be interrupted. Since it is impossible to predict when an external event will occur, the special section of code can be "protected" with a DISABLE statement.

```
100   ON KEY 9 LABEL " ABORT" GOTO Leave
110   !
120 Print_line:  !
130   DISABLE
140   FOR I=1 TO 10
150     PRINT I;
160     WAIT .3
170   NEXT I
180   PRINT
190   ENABLE
200   GOTO Print_line
210   !
220 Leave: END
```

This example shows a DISABLE and ENABLE statement used to "frame" the `Print_line` segment of the program. The "ABORT" key is active during the entire program, but the branch to exit the routine will not be taken until an entire line is printed. The operator can press the "ABORT" key at any time. The key press will be **logged,** or remembered, by the computer. Then when the ENABLE statement is executed, the event-initiated branch is taken.

Chaining Programs

With HP Instrument BASIC, it is also possible to "chain" programs together; that is, one program may be executed, which in turn loads and runs another. This method is often used when you have several large program segments that will not all fit into memory at the same time. This section describes program chaining methods.

Using GET

The GET command is brings in programs or program segments from an ASCII file, with the options of appending them to an existing program and/or beginning program execution at a specified line.

The following statement:

```
GET "George",100
```

first deletes all program lines from 100 to the end of the program, and then appends the lines in the file named "George" to the lines that remained at the beginning of the program. The program lines in file "George" would be renumbered to start with line 100.

GET can also specify where program execution begins. This is done by specifying two line identifiers. For example:

```
100 GET "RATES",Append_line,Run_line
```

specifies that:

1. Existing program lines from the line label "Append_line" to the end of the program are to be deleted.
2. Program lines in the file named "RATES" are to be appended to the current program, beginning at the line labeled "Append_line"; lines of "RATES" are renumbered if necessary.
3. Program execution is to resume at the line labeled "Run_line".

Although any combination of line identifiers is allowed, the line specified as the start of execution must be in the main program segment (not in a SUB or user-defined function). Execution will not begin if there was an error during the GET operation.

Example of Chaining with GET

A large program can be divided into smaller segments that are run separately by using GET. The following example shows a technique for implementing this method.

First Program Segment:

```
10  COM Ohms,Amps,Volts
20  Ohms=120
30  Volts=240
40  Amps=Volts/Ohms
50  GET "Wattage"
60  END
```

Program Segment in File Named "Wattage":

```
10  COM Ohms,Amps,Volts
20  Watts=Amps*Volts
30  PRINT "Resistance (in ohms)   = ";Ohms
40  PRINT "Power usage (in watts) = ";Watts
50  END
```

Lines 10 through 40 of the first program are executed in normal, serial fashion. Upon reaching line 50, the system deletes all program lines of the program and then GETs the lines of the "Wattage" program. Note that since they have similar COM declarations, the COM variables are preserved (and used by the second program). This feature is very handy to have while chaining programs.

Program-to-Program Communications

As shown in the preceding example, if chained programs are to communicate with one another, you can place values to be communicated in COM variables. The only restriction is that these COM declarations must *match exactly*, or the existing COM will be cleared when the chained program is loaded. For a description of using COM declarations, see the "Subprograms" chapter of this manual.

One important point to note is the use of the COM statement. The COM statement places variables in a section of memory that is preserved during the GET operation. Since the program saved in the file named "Wattage" also has a COM statement that contains three scalar REAL variables, the COM is preserved (it matches the COM declaration of the "Wattage" program being appended with GET).

If the program segments did not contain matching COM declarations, all variables in the mis-matched COM statements would be destroyed by the "pre-run" that the system performs after appending the new lines but before running the first program line.

3

Numeric Computation

Numeric computations deal exclusively with numeric values. Adding two numbers and finding a sine or a logarithm are all numeric operations, but converting bases and converting a number to a string or a string to a number are not.

Numeric Data Types

There are two numeric data types available in HP Instrument BASIC: INTEGER, and REAL. Any numeric variable not declared INTEGER is a REAL variable. This section covers these data types.

INTEGER Data Type

An INTEGER variable can have any whole-number value from −32 768 through +32 767.

REAL Data Type

A REAL variable can be any value from $-1.797\ 693\ 134\ 862\ 315 \times 10^{308}$ through $1.797\ 693\ 134\ 862\ 315 \times 10^{308}$. The smallest non-zero REAL value allowed is approximately $\pm\ 2.225\ 073\ 858\ 507\ 202 \times 10^{-308}$.

A REAL can also have the value of zero.

REAL and INTEGER variables may be declared as arrays.

Declaring Variables

You can declare variables to be of a particular type by using the INTEGER and REAL statements. For example, the statements:

```
INTEGER I, J, Days(5), Weeks(5:17)
REAL X, Y, Voltage(4), Hours(5,8:13)
```

each declare two scalar and two array variables. A scalar variable represents a single value. An array is a subscripted variable that contains multiple values accessed by subscripts. You can specify both the lower and upper bounds of an array or specify the upper bound only, and use the default lower bound of 0. You can also declare an array using the DIM statement.

```
DIM R(4,5)
```

Assigning Variables

The most fundamental numeric operation is the assignment operation, achieved with the LET statement. The LET statement may be used with or without the keyword LET. Thus the following statements are equivalent:

```
LET A = A + 1
A = A + 1
```

Implicit Type Conversions

The computer will automatically convert between REAL and INTEGER values in assignment statements and when parameters are passed by value in function and subprogram calls. When a value is assigned to a variable, the value is converted to the data type of that variable.

For example, the following program shows a REAL value being converted to an INTEGER:

```
100 REAL Real_var
110 INTEGER Integer_var
120 Real_var = 2.34
130 Integer_var = Real_var ! Type conversion occurs here.
140 DISP Real_var, Integer_var
150 END
```

Executing this program returns the following result:

```
2.34        2
```

When parameters are passed by value, the type conversion is from the data type of the calling statement's parameter to the data type of the subprogram's parameter. When parameters are passed by reference, the type conversion is not made and a TYPE MISMATCH error will be reported if the calling parameter and the subprogram parameters are different types.

When a REAL number is converted to an INTEGER, the fractional part is lost and the REAL number is rounded to the closest INTEGER value. Converting the number back to a REAL will not restore the fractional part. Also, because of the differences in ranges between these two data types, not all REAL values can be rounded into an equivalent INTEGER value. This problem can generate INTEGER OVERFLOW errors.

The rounding problem does not generate an execution error. The range problem *can* generate an execution error, and you should protect yourself from this possibility.

The following program segment shows a method to protect against INTEGER overflow errors (note that the variable X is REAL):

```
200 IF (-32768<=X) AND (X<=32767) THEN
210    Y = X
220 ELSE
230    GOSUB Out_of_range
240 END IF
```

It is possible to achieve the same effect using MAX and MIN functions:

```
200 Y=MAX(MIN(x,32767),-32768)
```

Both these methods avoid the overflow errors, but only the first does not lose the fact that the values were originally out of range. If out-of-range is a meaningful condition, an error handling trap is more appropriate.

Evaluating Scalar Expressions

This section covers the following topics as they relate to evaluating scalar expressions.

- Hierarchy of expression evaluation
- HP Instrument BASIC operators: monadic, dyadic, and relational

The Hierarchy

If you look at the expression 2+4/2+6, it can be interpreted several ways:

- 2+(4/2)+6 = 10
- (2+4)/2+6 = 9
- 2+4/(2+6) = 2.5
- (2+4)/(2+6) = .75

To eliminate this ambiguity HP Instrument BASIC uses a hierarchy for evaluating expressions. In order to understand how HP Instrument BASIC evaluates these expressions, let's examine the valid elements in an expression and the evaluation hierarchy (the order of evaluation of the elements).

Six items can appear in a numeric expression:

- Operators (+, −, etc.)—modify other elements of the expression.
- Constants (7.5, 10, etc.)—represent literal, non-changing numeric values.
- Variables (Amount, X_coord, etc.)—represent changeable numeric values.
- Intrinsic functions (SQRT, DIV, etc.)—return a value which replaces them in the evaluation of the expression.
- User-defined functions (FNMy_func, FNReturn_val, etc.)—also return a value which replaces them in the evaluation of the expression.
- Parentheses—are used to modify the evaluation hierarchy.

The following table defines the hierarchy used by the computer in evaluating numeric expressions.

Math Hierarchy

Precedence	Operator
Highest	Parentheses; they may be used to force any order of operation
	Functions, both user-defined and intrinsic
	Exponentiation: ^
	Multiplication and division: * / MOD DIV MODULO
	Addition, subtraction, monadic plus and minus: + −
	Relational Operators: = <> < > <= >=
	NOT
	AND
Lowest	OR, EXOR

When an expression is being evaluated it is read from left to right and operations are performed as encountered, unless a higher precedence operation is found immediately to the right of the operation encountered, or unless the hierarchy is modified by parentheses. If HP Instrument BASIC cannot deal immediately with the operation, it is stacked, and the evaluator continues to read until it encounters an operation it can perform. It is easier to understand if you see an example of how an expression is actually evaluated.

The following expression is complex enough to demonstrate most of what goes on in expression evaluation.

```
A = 5+3*(4+2)/SIN(X)+X*(1>X)+FNNeg1*(X<5 AND X>0)
```

In order to evaluate this expression, it is necessary to have some historical data. We will assume that DEG has been executed, that `X= 90`, and that `FNNeg1` returns `-1`. Evaluation proceeds as follows:

```
5+3*(4+2)/SIN(X)+X*(1>X)+FNNeg1*(X<5 AND X>0)

5+3*6/SIN(X)+X*(1>X)+FNNeg1*(X<5 AND X>0)

5+18/SIN(X)+X*(1>X)+FNNeg1*(X<5 AND X>0)

5+18/1+X*(1>X)+FNNeg1*(X<5 AND X>0)

5+18+X*(1>X)+FNNeg1*(X<5 AND X>0)

23+X*(1>X)+FNNeg1*(X<5 AND X>0)

23+X*0+FNNeg1*(X<5 AND X>0)

23+0+FNNeg1*(X<5 AND X>0)

23+FNNeg1*(X<5 AND X>0)

23+-1*(X<5 AND X>0)

23+-1*(0 AND X>0)

23+-1*(0 AND 1)

23+-1*0

23+0

23
```

Operators

There are three types of operators in HP Instrument BASIC: monadic, dyadic, and relational.

- A **monadic** operator performs its operation on the expression immediately to its right. + - NOT
- A **dyadic** operator performs its operation on the two values it is between. The operators are as follows: ^, *, /, MOD, DIV, +, -, =, <>, <, >, <=, >=, AND, OR, and EXOR.
- A **relational** operator returns a 1 (true) or a 0 (false) based on the result of a relational test of the operands it separates. The relational operators are a subset of the dyadic operators that are considered to produce **boolean** results. These operators are as follows: <, >, <=, >=, =, and <>.

While the use of most operators is obvious from the descriptions in the language reference, some of the operators have uses and side-effects that are not always apparent.

Expressions as Pass Parameters

All numeric expressions are passed by value to subprograms. Thus 5+X is obviously passed by value. Not quite so obviously, +X is also passed by value. The monadic operator makes it an expression.

For more information on pass parameters, read the chapter entitled "Subprograms and User-Defined Functions."

Strings in Numeric Expressions

String expressions can be directly included in numeric expressions if they are separated by relational operators. The relational operators always yield boolean results, and boolean results are numeric values in HP Instrument BASIC. For example:

```
110  Day_number=1*(Day$="Sun")+2*(Day$="Mon")
```

Executing the program line above would result in Day_number being equal to 1 if Day$ equals "Sun" and 2 if Day$ equals "Mon" (or 0 otherwise).

Step Functions

The comparison operators are useful for conditional branching (IF ... THEN statements), but are also valuable for creating numeric expressions representing step functions. For example, suppose you want to output certain values depending on the value, or range of values. of a single variable. This is shown as follows:

- If variable $<$ 0 then output =0
- If $0 \le$ variable $<$ 1 then output equals the square root of (A^2 + $^{B}2$).
- If variable $\ge$ 1 then output = 15

It is possible to generate the required response through a series of IF ... THEN statements, but it can also be done with the following expression (where X is the variable and Y is the output):

```
Y=(X<0)*0+(X>=0 AND X<1)* SQR(A^2+B^2)+(X>=1)*15
```

The boolean expressions each return a 1 or 0 which is then multiplied by the accompanying expression. Expressions not matching the selection return 0, and are not included in the result. The value assigned to the variable (X) before the expression is evaluated determines the computation placed in the result.

Comparing REAL Numbers

When you compare INTEGER numbers, no special precautions are necessary since these values are represented exactly. However, when you compare REAL numbers, especially those which are the results of calculations and functions, it is possible to run into problems due to rounding. For example, consider the use of comparison operators in IF ... THEN statements to check for equality in any situation resembling the following:

```
100   DEG
110   A=25.3765477
120   IF SIN(A)^2+COS(A)^2=1.0 THEN
130     PRINT "Equal"
140   ELSE
150     PRINT "Not Equal"
160   END IF
```

You will find that the equality test fails due to rounding errors. Irrational numbers and most repeating decimals cannot be represented exactly in any finite machine; and most rational decimal numbers with fractional parts cannot be represented exactly with binary numbers, which HP Instrument BASIC uses internally.

Resident Numerical Functions

The resident functions are the functions that are part of the HP Instrument BASIC language. Numerous functions are included to make mathematical operations easier. This section covers these functions by placing them in the categories given below.

- Arithmetic Functions
- Exponential Functions
- Trigonometric Functions
- Binary Functions
- Limit Functions
- Rounding Functions
- Random Number Function
- Base Conversion Functions
- General Functions

Arithmetic Functions

HP Instrument BASIC provides you with the following functions:

ABS — Returns the absolute value of an expression. Takes a REAL, or INTEGER number as its argument.

FRACT — Returns the "fractional" part of the argument.

INT — Returns the greatest integer that is less than or equal to an expression. The result is of the same type (INTEGER or REAL) as the original number.

MAXREAL — Returns the largest positive REAL number available in HP Instrument BASIC. Its value is approximately 1.797 693 134 862 32E+308.

MINREAL — Returns the smallest positive REAL number available in HP Instrument BASIC. Its value is approximately 2.225 073 858 507 24E−308.

SQRT or SQR — Return the square root of an expression. Takes a REAL or INTEGER number as their argument.

SGN — Returns the sign of an expression: 1 if positive, 0 if 0, −1 if negative.

Exponential Functions

These functions determine the natural and common logarithm of an expression, as well as the Napierian e raised to the power of an expression. Note that all exponential functions take REAL, or INTEGER numbers as their argument.

EXP — Raise the Napierian e to an power. e = 2.718 281 828 459 05.

LGT — Returns the base 10 logarithm of an expression.

LOG — Returns the natural logarithm (Napierian base e) of an expression.

Trigonometric Functions

Six trigonometric functions and the constant π are provided for dealing with angles and angular measure. Note that all trigonometric functions take REAL or INTEGER numbers as their argument.

ACS Returns the arccosine of an expression.

ASN Returns the arcsine of an expression.

ATN Returns the arctangent of an expression.

COS Returns the cosine of the angle represented by the expression.

SIN Returns the sine of the angle represented by an expression.

TAN Returns the tangent of the angle represented by an expression.

PI Returns the constant 3.141 592 653 589 79, an approximate value for pi.

Trigonometric Modes: Degrees and Radians

The default mode for all angular measure is radians. Degrees can be selected with the DEG statement. Radians may be re-selected by the RAD statement. It is a good idea to explicitly set a mode for any angular calculations, even if you are using the default (radian) mode. This is especially important in writing subprograms, as the subprogram inherits the angular mode from the context that calls it. The angle mode is part of the calling context.

Binary Functions

All operations that HP Instrument BASIC performs use a binary number representation. You usually don't see this, because HP Instrument BASIC changes decimal numbers you input into its own binary representation, performs operations using these binary numbers, and then changes them back to their decimal representation before displaying or printing them.

The following HP Instrument BASIC functions deal with binary numbers:

BINAND Returns the bit-by-bit "logical and" of two arguments.

BINCMP Returns the bit-by-bit "complement" of its argument.

BINEOR Returns the bit-by-bit "exclusive or" of two arguments.

BINIOR Returns the bit-by-bit "inclusive or" of two arguments.

BIT Returns the state of a specified bit of the argument.

ROTATE Returns a value obtained by shifting an INTEGER representation of an argument a specific number of bit positions, *with* wraparound.

SHIFT Returns a value obtained by shifting an INTEGER representation of an argument a specific number of bit positions, *without* wraparound.

When any of these functions are used, the arguments are first converted to INTEGER (if they are not already in the correct form) and then the specified operation is performed. It is best to restrict bit-oriented binary operations to declared INTEGERs. If it is necessary to operate on a REAL, make sure the precautions described under "Conversions," at the beginning of this chapter, are employed to avoid INTEGER overflow.

Limit Functions

It is sometimes necessary to limit the range of values of a variable. HP Instrument BASIC provides two functions for this purpose:

`MAX` Returns a value equal to the greatest value in the list of arguments.

`MIN` Returns a value equal to the least value in the list of arguments.

Rounding Functions

Sometimes it is necessary to round a number in a calculation to eliminate unwanted resolution. There are two basic types of rounding, rounding to a total number of decimal digits and rounding to a number of decimal places (limiting fractional information). Both types of rounding have their own application in programming.

The functions which perform the types of rounding mentioned above are as follows:

`DROUND` Rounds a numeric expression to the specified number of digits. If the specified number of digits is greater than 15, no rounding takes place. If the number of digits specified is less than 1, zero is returned.

`PROUND` Returns the value of the argument rounded to a specified power of ten.

Random Number Function

The RND function returns a pseudo-random number between 0 and 1. Since many applications require random numbers with arbitrary ranges, it is necessary to scale the numbers.

```
200  R= INT(RND*Range)+Offset
```

The above statement will return an integer between Offset and Offset + Range.

The random number generator is seeded with the value 37 480 660 at power-on, SCRATCH, SCRATCH A, and prerun. The pattern period is $2^{31} - 2$. You can change the seed with the RANDOMIZE statement, which will give a new pattern of numbers.

Time and Date Functions

The following functions return the time and date in seconds:

`TIMEDATE` Returns the current clock value (in Julian seconds).

(If there is no battery-backed clock, the clock value set at power-on is 2.086 629 12E+11, which represents midnight March 1, 1900.

For example, the statement

```
TIMEDATE
```

returns a value in seconds similar to the following:

```
2.11404868285E+11
```

Base Conversion Functions

The two functions IVAL and DVAL convert a binary, octal, decimal, or hexadecimal string value into a decimal number.

IVAL returns the INTEGER value of a binary, octal, decimal, or hexadecimal 16-bit integer. The first argument is a string and the second argument is the radix or base to convert from. For example, executing this statement

```
IVAL("12740",8)
```

returns the following numeric value:

```
5600
```

DVAL returns the decimal whole number value of a binary, octal, decimal, or hexadecimal 32-bit integer. The first argument is a string and the second argument is the radix or base to convert from. For example, executing this statement

```
DVAL("11111111111111111111111111111100",2)
```

returns the following numeric value:

```
-4
```

For more information and examples of these functions, read the section "Number-Base Conversion" found in the "String Manipulation" chapter.

General Functions

When you are specifying select code and device selector numbers, it is more descriptive to use a function to represent that device as opposed to a numeric value. For example, the following command allows you to enter a numeric value from the keyboard.

```
ENTER 2;Numeric_value
```

The above statement used in a program is not as easy to read as this one is:

```
ENTER KBD;Numeric_value
```

where you know the function KBD must stand for keyboard.

Functions which return a select code or device selector are listed below:

CRT Returns the INTEGER 1. This is the select code of the internal CRT.

KBD Returns the INTEGER 2. This is the select code of the keyboard.

PRT Returns the INTEGER 701. This is the default (factory set) device selector for an external printer (connected through the built-in HP-IB interface at select code 7).

4

Numeric Arrays

An array is a multi-dimensioned structure of variables that are given a common name. The array can have one through six dimensions. Each location in an array can contain one variable value, and each value has the characteristics of a single variable, depending on whether the array consists of REAL, or INTEGER values (string arrays are discussed in the chapter, "String Manipulation.")

A one-dimensional array consists of n elements, each identified by a single subscript. A two-dimensional array consists of m times n elements where m and n are the maximum number of elements in the two respective dimensions. Arrays require a subscript in each dimension, in order to locate a given element of the array. Up to six dimensions can be specified for any array in a program. REAL arrays require eight bytes of memory for each element, plus overhead memory for each element, plus overhead. It is easy to see that large arrays can demand massive memory resources.

An undeclared array is given as many dimensions as it has subscripts in its lowest-numbered occurrence. Each dimension of an undeclared array has an upper bound of ten. Space for these elements is reserved whether you use them or not.

Dimensioning an Array

Before you use an array, you should tell the system how much memory to reserve for it. This is called **dimensioning** an array. You can dimension arrays with the DIM, COM, INTEGER, or REAL statements. For example:

```
REAL Array_real(2,4)
```

An array is a type of variable and as such follows all rules for variable names. Unless you explicitly specify INTEGER type in the dimensioning statement, arrays default to REAL type. The same array can only be dimensioned once in a context.

Dimensioning reserves space in internal memory for the array. The system also sets up a table used to locate each element in the array. The location of each element is designated by a unique combination of subscripts, one subscript for each dimension. For a two-dimensional array, for instance, each element is identified by two subscript values. An example of declaring a two-dimensional array is:

```
DIM Array(3,5)
```

This statement dimensions a 4 × 6 array, with the first subscript representing four rows (0,1,2,3) and the second subscript representing six columns (0,1,2,3,4,5). The locations of each element is designated by a unique combination of subscripts, one subscript for each dimension.

The actual size of an array is governed by the number of dimensions and the subscript range of each dimension. If A is a three-dimensional array with a subscript range of 1 through 4 for each dimension,

```
DIM A(1:4,1:4,1:4)
```

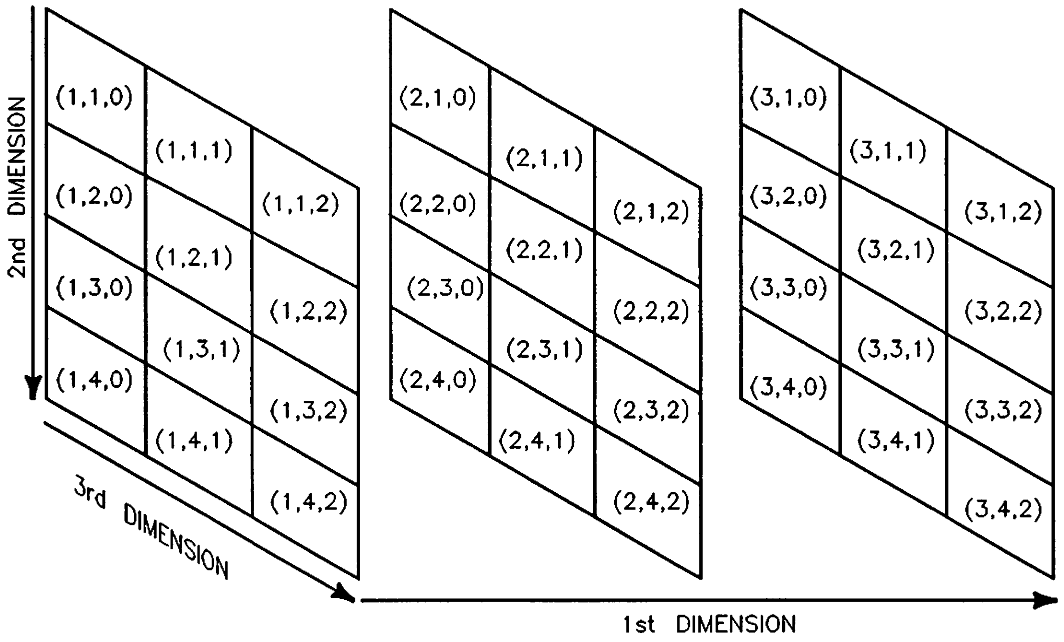
then its size is 4×4×4, 64 elements. Note that 1 on the left side of the colon in the dimension statement above is the lower bound and 4 on the right is the upper bound.

When you dimension an array, therefore, you must give not only the number of dimensions but also the subscript range of each dimension. Subscript ranges can be specified by giving the lower and upper bounds, as shown above, or by giving just the upper bound. If you give only the upper bound, the lower bound defaults to 0.

Some Examples of Arrays

The following examples illustrate some of the flexibility you have in dimensioning arrays.

```
    10  DIM A(1:3,1:4,2)
```



Planes of a Three-Dimensional REAL Array

Dimension	Size	Lower Bound	Upper Bound
1st	3	1	3
2nd	4	1	4
3rd	3	0	2

In this example we portray the first dimension as planes, the second dimension as rows, and the third dimension as columns. In general, the last two dimensions of any array always refer to rows and columns, respectively. When we discuss two-dimensional arrays, the first dimension will always represent rows, and the second dimension will always represent columns.

Note also in the above example that the first two dimensions use the default setting of 1 for the lower bound, while the third dimension explicitly defines 0 as the lower bound. The numbers in parentheses are the subscript values for the particular elements. These are the numbers you use to identify each array element.

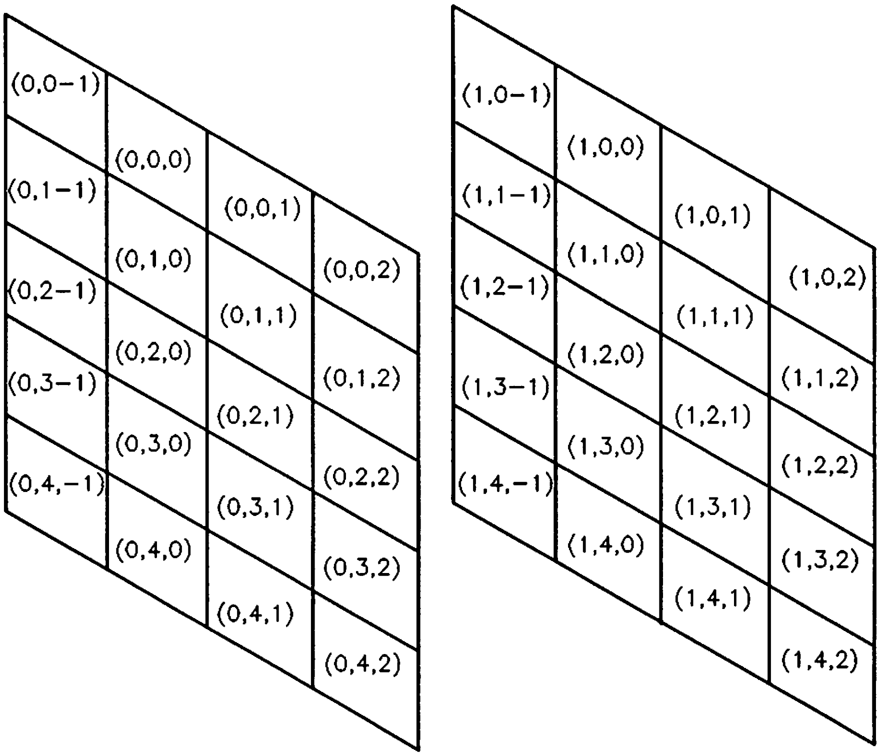
```
20   INTEGER C(2:4,-2:2)
```

A Two-Dimensional INTEGER Array

(2,-2)	(2,-1)	(2,0)	(2,1)	(2,2)
(3,-2)	(3,-1)	(3,0)	(3,1)	(3,2)
(4,-2)	(4,-1)	(4,0)	(4,1)	(4,2)

Dimension	Size	Lower Bound	Upper Bound
1st	3	2	4
2nd	5	−2	2

```
30   COM INTEGER F(1,4,-1:2)
```



A Three-Dimensional INTEGER Array, in Common

Dimension	Size	Lower Bound	Upper Bound
1st	2	0	1
2nd	5	0	4
3rd	4	−1	2

Arrays are limited to six dimensions, and the subscript range for each dimension must lie between -32767 and 32767. For the most part, we use only two-dimensional examples since they are easier to illustrate. However, the same principles apply to arrays of more than two dimensions as well.

Note Unless explicitly specified otherwise, HP Instrument BASIC uses the base of 0 for arrays.

As an example of a four-dimensional array, consider a five-story library. On each floor there are 20 stacks, each stack contains 10 shelves, and each shelf holds 100 books. To specify the location of a particular book you would give the number of the floor, the stack, the shelf, and the particular book on that shelf. We could dimension an array for the library with the statement:

```
DIM Library(1:5,1:20,1:10,1:100)
```

This means that there are 100,000 book locations. To identify a particular book you would specify its subscripts. For instance, `Library(2,12,3,35)` would identify the 35th book on the 3rd shelf of the 12th stack on the 2nd floor.

You can imagine accessing a particular page of a book by using a 5-dimensional array. For instance, if you dimension an array,

```
DIM Page(1:5,1:20,1:10,1:100,1:200)
```

then `Page(1,7,2,19,130)` would designate page 130 of the 19th book on the 2nd shelf of the 7th stack on the 1st floor.

You could specify words on pages by using a 6-dimensional array. Six dimensions is the maximum, though, so we could not specify letters of words.

Also, you can dimension more than one array in a single statement by separating the declarations with a comma. For instance,

```
10 DIM A(1,3,4),B(-2:0,2:5),C(5)
```

would dimension all three arrays: `A`, `B`, and `C`.

Problems with Implicit Dimensioning

In any environment, an array must have a dimensioned size. This size can be passed into an environment through a passed parameter list or a COM statement. It may be explicitly dimensioned through COM, INTEGER, or REAL. It can also be implicitly dimensioned through a subscripted reference in a program statement.

Finding Out the Dimensions of an Array

There are a number of statements that allow you to determine the size and shape of an array. To find out how many dimensions are in an array, use the RANK function. For instance:

```
   10 DIM F(1,4,-1:2)
   20 PRINT RANK (F)
```

would print 3.

The SIZE function returns the size (number of elements) of a particular dimension. For instance,

```
SIZE (F,2)
```

would return 5, the number of elements in F's second dimension.

To find out what the lower bound of a dimension is, use the BASE function. Referring again to array F,

```
BASE (F,1)
```

would return a 0, while,

```
BASE (F,3)
```

would return a -1.

By using the SIZE and BASE functions together, you can determine the upper bounds of any dimension (e.g., Upper Bound=SIZE+BASE-1).

It may seem pointless to have all these functions that return the dimension specifications which you yourself assigned. After all, if you assigned the dimensions, you should know what they are; and if you forget, you can always look at the appropriate dimensioning statement. However, these functions are powerful tools for writing programs that perform functions on an array regardless of the array's size or shape.

Using Individual Array Elements

This section deals with assigning and extracting individual elements from arrays.

Assigning an Individual Array Element

Once an array has been dimensioned, the next step is to fill it with useful values. Initially, every element in an array equals zero. There are a number of different ways to change these values. The most obvious is to assign a particular value to each element. This is done by specifying the element's subscripts. For example, the statement,

```
A(3,4)=13
```

assigns the value 13 to the element in the third row and fourth column of A. You must give enough subscripts for the system to identify a single element. All subscripts must lie within the dimensioned range. If you use out-of-range subscripts, the system returns an error.

Extracting Single Values From Arrays

There are a number of ways to extract values from array elements. To extract the value of a particular element, simply specify the element's subscripts. For instance, the statement,

```
X=A(3,4,2)
```

assigns the value of the element occupying the given location in A to the variable X. The system will automatically convert variable types. For example, if you assign an element from a REAL array to an INTEGER variable, the system will perform the necessary rounding.

Filling Arrays

Using the READ Statement to Fill an Entire Array

You can assign values to an array by using the READ and DATA statements. The DATA statement allows you to create a stream of data items, and the READ statement enables you to enter the data stream into an array. For example:

```
10   DIM A(3,3)
20   DATA -4,36,2.3,5,89,17,-6,-12,42
30   READ A(*)
40   PRINT USING "3(3DD.DD,3DD.DD,3DD.DD,/)";A(*)
50   END
```

The asterisk in line 40 is used to designate the entire array rather than a single element. Note also that the right-most subscript varies fastest. In this case, it means that the system fills an entire row before going to the next one. The READ/DATA statements are discussed further in the chapter "Data Storage and Retrieval".

Executing the above program produces the following results:

```
-4.00    36.00     2.30
 5.00    89.00    17.00
-6.00   -12.00    42.00
```

Printing Arrays

Once an array has been filled with values, it is nice to know what those values are. The best way to do this is to display them on the screen or printer. This section provides information on how to perform this task for REAL and INTEGER values.

Printing an Entire Array

Certain operations (e.g., PRINT, OUTPUT, ENTER and READ) allow you to access all elements of an array merely by using an asterisk in place of the subscript list. The statement,

```
PRINT A(*);
```

displays every element of A on the current PRINTER IS device. The elements are displayed in order, with the rightmost subscripts varying fastest. The semi-colon at the end of the statement is equivalent to putting a semi-colon between each element. When they are displayed, therefore, they will be separated by a space. The default is to place elements in successive columns.

Passing Entire Arrays

The asterisk is also used to pass an array as a parameter to a function or subprogram. For instance, to pass an array `A` to the `Printmat` subprogram listed earlier, you would write:

```
Printmat (A(*))
```

5

String Manipulation

It is often desirable to store non-numerical information in the computer. A word, a name or a message can be stored in the computer as a **string.** Any sequence of characters may be used in a string. Quotation marks delimit the beginning and ending of the string. The following are valid string assignments.

```
LET A$="COMPUTER"
Fail$="The test has failed."
File_name$="INVENTORY"
Test$=Fail$[5,8]
```

The left-hand side of the assignment (the variable name) is equated to the right-hand side of the assignment (the literal). String variable names are identical to numeric variable names with the exception of a dollar sign ($) appended to the end of the name.

The **length** of a string is the number of characters in the string. In the previous example, the length of A$ is 8 since there are eight characters in the literal "COMPUTER". A string with length 0 (i.e., that contains no characters) is known as a **null** string.

HP Instrument BASIC allows the dimensioned length of a string to range from 1 to 32 767 characters. The current length (number of characters in the string) ranges from zero to the dimensioned length.

The default dimensioned length of a string is 18 characters. The DIM and COM statements define string lengths up to the maximum length of 32 767 characters. An error results whenever a string variable is assigned more characters than its dimensioned length.

A string may contain any character. The only special case is when a quotation mark needs to be in a string. Two quotes, in succession, will embed a quote within a string.

```
10  Quote$="The time is ""NOW""."
20  PRINT Quote$
30  END
```

Produces: `The time is "NOW".`

String Storage

Strings whose length exceeds the default length of 18 characters must have space reserved before assignment. The following statements may be used.

`DIM Long$[400]`	Reserve space for a 400 character string.
`COM Line$[80]`	Reserve an 80 character common variable.

The DIM statement reserves storage for strings.

```
DIM Part_number$[10],Description$[64],Cost$[5]
```

The COM statement defines common variables that can be used by subprograms.

```
COM Name$[40],Phone$[14]
```

Strings that have been dimensioned but not assigned return the null string.

String Arrays

Large amounts of text are easily handled in arrays. For example:

```
DIM File$(1:1000)[80]
```

reserves storage for 1000 lines of 80 characters per line. Do not confuse the brackets, which define the length of the string, with the parentheses which define the number of strings in the array. Each string in the array can be accessed by an index. For example:

```
PRINT File$(27)
```

prints the 27th element in the array. Since each character in a string uses one byte of memory and each string in the array requires as many bytes as the length of the string, string arrays can quickly use a lot of memory.

A program saved on a disc as an ASCII type file can be entered into a string array, manipulated, and written back out to disc.

Evaluating Expressions Containing Strings

This section covers the following topics:

- Evaluation Hierarchy
- String Concatenation
- Relational Operations

Evaluation Hierarchy

Evaluation of string expressions is simpler than evaluation of numerical expressions. The three allowed operations are extracting a substring, concatenation, and parenthesization. The evaluation hierarchy is presented in the following table.

Order	Operation
High	Parentheses
—	Substrings and Functions
Low	Concatenation

String Concatenation

Two separate strings are joined together by using the concatenation operator "&". The following program combines two strings into one.

```
10 One$="WRIST"
20 Two$="WATCH"
30 Concat$=One$&Two$
40 PRINT One$,Two$,Concat$
50 END
```

Prints:

```
WRIST    WATCH    WRISTWATCH
```

The concatenation operation, in line 30, appends the second string to the end of the first string. The result is assigned to a third string. An error results if the concatenation operation produces a string that is longer than the dimensioned length of the string being assigned.

Relational Operations

Most of the relational operators used for numeric expression evaluation can also be used for the evaluation of strings.

The following examples show some of the possible tests.

"ABC" = "ABC"	*True*
"ABC" = " ABC"	*False*
"ABC" < "AbC"	*True*
"6" > "7"	*False*
"2" < "12"	*False*
"long" <= "longer"	*True*
"RE-SAVE" >= "RESAVE"	*False*

Any of these relational operators may be used: <, >, <=, >=, =, <>.

Testing begins with the first character in the string and proceeds, character by character, until the relationship has been determined.

The outcome of a relational test is based on the characters in the strings not on the length of the strings. For example:

```
"BRONTOSAURUS" < "CAT"
```

This relationship is true since the letter "C" is higher in ASCII value than the letter "B".

Substrings

You can append a subscript to a string variable name to define a **substring.** A substring may comprise all or just part of the original string. Brackets enclose the subscript which can be a constant, variable, or numeric expression. For instance:

```
String$[4]
```

Specifies a substring starting with the fourth character of the original string. The subscript must be in the range: 1 to the current length of the string plus 1. Note that the brackets now indicate the substring's starting position instead of the total length of the string as when reserving storage for a string. Subscripted strings may appear on either side of the assignment.

Single-Subscript Substrings

When a substring is specified with only one numerical expression, enclosed with brackets, the expression is evaluated and rounded to an integer indicating the position of the first character of the substring within the string.

The following examples use the variable A$ which has been assigned the literal "DICTIONARY".

Statement	Output
PRINT A$	DICTIONARY
PRINT A$[0]	(error)
PRINT A$[1]	DICTIONARY
PRINT A$[5]	IONARY
PRINT A$[10]	Y
PRINT A$[11]	(null string)
PRINT A$[12]	(error)

When you use a single subscript it specifies the starting character position, within the string, of the substring. An error results when the subscript evaluates to zero or greater than the current length of the string plus 1. A subscript that evaluates to 1 plus the length of the string returns the null string ("") but does not produce an error.

Double-Subscript Substrings

A substring may have two subscripts, within brackets, to specify a range of characters. When a comma is used to separate the items within brackets, the first subscript marks the beginning position of the substring, while the second subscript is the ending position of the substring. The form is: A$[Start,End]. For example, if A$ = "JABBERWOCKY", then

`A$[4,6]` Specifies the substring: BER

When a semicolon is used in place of a comma, the first subscript again marks the beginning position of the substring, while the second subscript is now the length of the substring. The form is: A$[Start;Length].

`A$[4;6]` Specifies the substring: BERWOC

In the following examples the variable B$ has been assigned the literal "ENLIGHTENMENT":

Statement	Output
`PRINT B$`	`ENLIGHTENMENT`
`PRINT B$[1,13]`	`ENLIGHTENMENT`
`PRINT B$[1;13]`	`ENLIGHTENMENT`
`PRINT B$[1,9]`	`ENLIGHTEN`
`PRINT B$[1;9]`	`ENLIGHTEN`
`PRINT B$[3,7]`	`LIGHT`
`PRINT B$[3;7]`	`LIGHTEN`
`PRINT B$[13,13]`	`N`
`PRINT B$[13;1]`	`N`
`PRINT B$[13,26]`	(error)
`PRINT B$[13;13]`	(error)
`PRINT B$[14;1]`	(null string)

An error results if the second subscript in a comma separated pair is greater than the current string length plus 1 *or* if the sum of the subscripts in a semicolon separated pair is greater than the current string length plus 1.

Specifying the position just past the end of a string returns the null string.

Special Considerations

All substring operations allow a subscript to specify the first position past the end of a string. This allows strings to be concatenated without the concatenation operator. For instance:

```
10    A$="CONCAT"
20    A$[7]="ENATION"
30    PRINT A$
40    END
```

Produces: CONCATENATION

The substring assignment is only valid if the substring already has characters up to the specified position. Access beyond the first position past the end of a string results in the error:

```
ERROR 18  String ovfl. or substring err
```

Its a good practice to dimension all strings including those shorter than the default length of eighteen characters.

String-Related Functions

Several intrinsic functions are available in HP Instrument BASIC for the manipulation of strings. These functions include conversions between string and numeric values.

Current String Length

The "length" of a string is the number of characters in the string. The LEN function returns an integer whose value is equal to the string length. The range is from 0 (null string) through 32 767. For example:

```
PRINT LEN("HELP ME")
```

Prints: 7

Substring Position

The "position" of a substring within a string is determined by the POS function. The function returns the value of the starting position of the substring or zero if the entire substring was not found. For instance:

```
PRINT POS("DISAPPEARANCE","APPEAR")
```

Prints: 4

Note that POS returns the first occurrence of a substring within a string. By adding a subscript, and indexing through the string, the POS function can be used to find all occurrences of a substring.

String-to-Numeric Conversion

The VAL function converts a string expression into a numeric value. The number will be converted to and from scientific notation when necessary. For example:

```
PRINT VAL("123.4E3")
```

Prints: 123400

The string must evaluate to a valid number or error 32 will result.

```
ERROR 32 String is not a valid number
```

The NUM function converts a single character into its equivalent numeric value. The number returned is in the range: 0 to 255. For example:

```
PRINT NUM("A")
```

Prints: 65

Numeric-to-String Conversion

The VAL$ function converts the value of a numeric expression into a character string. The string contains the same characters (digits) that appear when the numeric variable is printed. For example:

```
PRINT 1000000,VAL$(1000000)
```

Prints: 1.E+6   1.E+6

The CHR$ function converts a number into an ASCII character. The number can be of type INTEGER or REAL since the value is rounded, and a modulo 255 is performed. For example:

```
PRINT CHR$(97);CHR$(98);CHR$(99)
```

Prints: abc

String Functions

This section covers string functions which perform the following tasks:

- Reversing the characters in a string,
- Repeating a string a given number of times,
- Trimming the leading and trailing blanks in a string,
- Converting string characters to the desired case.

String Reverse

The REV$ function returns a string created by reversing the sequence of characters in the given string.

```
PRINT REV$("Snack cans")
```

Prints: `snac kcanS`

String Repeat

The RPT$ function returns a string created by repeating the specified string, a given number of times.

```
PRINT RPT$("* *",10)
```

Prints: `* ** ** ** ** ** ** ** ** ** *`

Trimming a String

The TRIM$ function returns a string with all leading and trailing blanks (ASCII spaces) removed.

```
PRINT "*";TRIM("   1.23   ");"*"
```

Prints: `*1.23*`

Case Conversion

The case conversion functions, UPC$ and LWC$, return strings with all characters converted to the proper case. UPC$ converts all lowercase characters to their corresponding uppercase characters and LWC$ converts any uppercase characters to their corresponding lowercase characters.

```
10    DIM Word$[160]
20    INPUT "Enter a few characters",Word$
30    PRINT
40    PRINT "You typed: ";Word$
50    PRINT "Uppercase: ";UPC$(Word$)
60    PRINT "Lowercase: ";LWC$(Word$)
70    END
```

Number-Base Conversion

Utility functions are available to simplify the calculations between different number bases. The two functions IVAL and DVAL convert a binary, octal, decimal, or hexadecimal string value into a decimal number. The IVAL$ and DVAL$ functions convert a decimal number into a binary, octal, decimal, or hexadecimal string value. The IVAL and IVAL$ functions are restricted to the range of INTEGER variables (-32 768 through 32 767). The DVAL and DVAL$ functions allow "double length" integers and thus allow larger numbers to be converted (-2 147 483 648 through 2 147 483 647).

Each function has two parameters: the number or string to be converted and the radix. The radix is limited to the values 2, 8, 10 and 16, and represents the numeric base of the conversion.

The following statements show valid usage of these functions:

```
PRINT DVAL("FF5900",16)
PRINT IVAL("AA",16")
PRINT DVAL$(100,8)
PRINT IVAL$(-1,16)
```

6

Subprograms and User-Defined Functions

One of the most powerful constructs available in any language is the subprogram. A subprogram can do everything a main program can do except that it must be invoked or "called" before it is executed, whereas a main program is executed by an operator. This chapter describes the benefits of using subprograms, and shows many of the details of using them.

A user-defined function is simply a special form of subprogram.

Benefits of Subprograms

A subprogram has its own "context" or state that is distinct from a main program and all other subprograms. This means that every subprogram has its own set of variables, its own softkey definitions, its own DATA blocks, and its own line labels. There are several benefits to be realized by taking advantage of subprograms:

- The subprogram allows the programmer to take advantage of the **top-down design** method of programming.
- The program is much easier to read using the subprogram calls.
- By using subprograms and *testing each one independently* of the others, it is easier to locate and fix problems.
- You may want to perform the same task from several different areas of your program.
- Libraries of commonly used subprograms can be assembled for widespread use.

A Closer Look at Subprograms

This section shows a few of the details of using subprograms.

Calling and Executing a Subprogram

A SUB subprogram is invoked explicitly using the CALL statement. A nuance of SUB subprograms is that the CALL keyword is optional when invoking a SUB subprogram.

The omission of the CALL keyword when invoking a SUB subprogram is left solely to the discretion of the programmer; some will find it more aesthetic to omit CALL, others will prefer its inclusion. There are, however, three instances which require the use of CALL when invoking a subprogram:

CALL is required:

1. If the subprogram is called from the keyboard,
2. If the subprogram is called after the THEN keyword in an IF statement, or
3. In an ON..event..CALL statement.

Differences Between Subprograms and Subroutines

A **subroutine** and a **subprogram** are very different in HP Instrument BASIC.

- The GOSUB statement transfers program execution to a subroutine. A subroutine is a segment of program lines *within the current context.* No parameters need to be passed, since it has access to all variables in the context (which is also the context in which the "calling" segment exists).
- The CALL statement transfers program execution to a subprogram, which is in a *separate context.* Subprograms can have pass parameters, and they can have their own set of local variables which are separate from all variables in all other contexts.

Subprogram Location

A subprogram is located after the body of the main program, following the main program's END statement. (The END statement must be the last statement in the main program except for comments.) Subprograms may not be nested within other subprograms, but are physically delimited from each other with their heading statements (SUB or DEF) and ending statements (SUBEND or FNEND).

Subprogram and User-Defined Function Names

A subprogram has a name which may be up to 15 characters long, just as with line labels and variable names. Here are some legal subprogram names:

```
Initialize
Read_dvm
Sort_2_d_array
Plot_data
```

Because up to 15 characters are allowed for naming subprograms, it is easy and convenient to name subprograms in such a way as to reflect the purpose for which the subprogram was written.

Difference Between a User-Defined Function and a Subprogram

A SUB subprogram (as opposed to a function subprogram) is invoked explicitly using the CALL statement. A function subprogram is called implicitly by using the function name in an expression. It can be used in a numeric or string expression the same way a constant would be used, or it can be invoked from the keyboard. A function's purpose is to return a single value (either a REAL number or a string).

There are several functions that are built into the HP Instrument BASIC language which can be used to return values, such as SIN, SQR, EXP, etc.

```
Y=SIN(X)+Phase
Root1=(-B+SQR(B*B-4*A*C))/(2*A)
```

User defined functions can extend HP Instrument BASIC if you need a feature that is not provided.

```
X=FNFactorial(N)
Angle=FNAtn2(Y,X)
```

A general guideline, if you want to take a set of data and analyze it to generate a single value, then implement the subprogram as a function. On the other hand, if you want to actually change the data itself, generate more than one value as a result of the subprogram, or perform any I/O activity, it is better to use a SUB subprogram.

REAL Precision Functions and String Functions

A function is allowed to return either a REAL or a string value. Let's examine one which returns a string. There are two primary differences: the first is that a $ must be added to the name of a function which is to return a string. This is used both in the definition of the function (the DEF statement) and when the function is invoked. The second difference is that the RETURN statement in the function returns a string instead of a number.

```
200   PRINT FNAscii_to_hex$(A$)
      .
      .
      .
1550  DEF FNAscii_to_hex$(A$)
1560  ! Each ASCII byte consists of two hex
1570  !   digits; pretty formatting dictates that
1580  !   a space be inserted between every pair
1590  !   of hex digits.  Thus, the output string
1600  !   will be three times as long as the input
1610  !   string.
1620  !
1630  ! upper four bits     lower four bits
1640  ! UUUU LLLL           UUUU LLLL
1650  ! shift 4 bits        0000 1111 mask (15)
1660  ! 0000 UUUU           0000 LLLL final
1670  !
1680  INTEGER I,Length,Hexupper,Hexlower
1690  Length=LEN(A$)
1695  Length=3*Length
1700  DIM Temp$[Length]
1710  FOR I=1 TO Length
1720     Hexupper=SHIFT(NUM(A$[I]),4)
1730     Hexlower=BINAND(NUM(A$[I],15)
1740     Temp$[3*I-2;1]=FNHex$(Hexupper)
1750     Temp$[3*I-1;1]=FNHex$(Hexlower)
1760     Temp$[3*I;1]=" "
1770  NEXT I
1780  RETURN Temp$
1790  FNEND
1800  DEF FNHex$(INTEGER X)
1810  ! Assume 0<=X<=15)
1820  ! Return ASCII representation of the
1830  !   hex digit represented by the four
1840  !   bits of X.
1850  ! If X is between 0 and 9, return
1860  !   "0" ... "9"
1870  ! If X > 9, return "A" ... "F"
1880  IF X<=9 THEN
1890     RETURN CHR$(48+X) ! ASCII 48 through 57
```

```
1900                          !   represent "0" - "9"
1910  ELSE
1920     RETURN CHR$(55+X)  ! ASCII 65 through 70
1930                          !   represent "A" - "F"
1940  END IF
1950  FNEND
```

Lines 200, 1740, and 1750 show examples of how to call a string function. Lines 1550 and 1800 show where the two string function subprograms begin. Notice that the program could be optimized slightly by deleting lines 1720 and 1730 and modifying lines 1740 and 1750:

```
1740     Temp$[3*I-2;1]=FNHex$(SHIFT(NUM(A$[I]),4))
1750     Temp$[3*I-1;1]=FNHex$(BINAND(NUM(A$[I],15))
```

Thus it is perfectly legal to use expressions in the pass parameter list of a subprogram.

Program/Subprogram Communication

As mentioned earlier, there are two ways for a subprogram to communicate with the main program or with other subprograms:

- By passing parameters
- By sharing blocks of common (COM) variables.

Parameter Lists

There are two places where parameter lists occur:

- The **pass parameter list** is in the CALL statement or FN call:

  ```
  30  CALL Build_array(Numbers(*),20)    ! Subprogram call.

  50  PRINT FNSum_array(Numbers(*),20)  ! User-defined function call.
  ```

  It is known as the pass parameter list because it specifies what information is to be passed to the subprogram.

- The **formal parameter list** is in the SUB or DEF FN statement that begins the subprogram's definition:

  ```
  70   SUB Build_array(X(*),N)   ! Subprogram "Build_array".

  410  DEF FNSum_array(A(*),N)   ! User-defined function "Sum_array".
  ```

  This is known as the formal parameter list because it specifies the form of the information that can be passed to the subprogram.

Formal Parameter Lists

The formal parameter list is part of the subprogram's definition, just like the subprogram's name. The formal parameter list defines:

- The *number of values* that may be passed to a subprogram
- The *types of those values* (string, INTEGER, or REAL, and whether they are simple or array variables; or I/O path names)

- The *variable names the subprogram will use* to refer to those values. (This allows the name in the subprogram to be different from the name used in the calling context.)

The subprogram has the power to demand that the calling context match the types declared in the formal parameter list—otherwise, an error results.

Pass Parameter Lists

The calling context provides a pass parameter list which corresponds with the formal parameter list provided by the subprogram. The pass parameter list provides:

- The *actual values* for those inputs required by the subprogram.
- *Storage* for any values to be returned by the subprogram (pass by reference parameters only).

It is perfectly legal for both the formal and pass parameter lists to be null (non-existent).

Passing By Value vs. Passing By Reference

There are two ways for the calling context to pass values to a subprogram:

- Pass by value—the calling context supplies a value and nothing more.
- Pass by reference—the calling context actually gives the subprogram access to the calling context's value area (which is essentially access to the calling context's variable).

The distinction between these two methods is that a subprogram cannot alter the value of data in the calling context if the data is passed by value, while the subprogram *can* alter the value of data in the calling context if the data is passed by reference.

The subprogram has no control over whether its parameters are passed by value or passed by reference. That is determined by the calling context's pass parameter list. For instance, in the example below, the array Numbers(*) is passed by reference, while the quantity 20 is passed by value.

```
30 CALL Build_array(Numbers(*),20) ! Subprogram call.
```

The general rules for passing parameters are as follows:

- In order for a parameter to be passed *by reference,* the pass parameter list (in the calling context) must use a *variable* for that parameter.
- In order for a parameter to be passed *by value,* the pass parameter list must use an *expression* for that parameter.

Note that enclosing a variable in parentheses is sufficient to create an expression and that literals are expressions. Using pass by value, it is possible to pass an INTEGER expression to a REAL formal parameter (the INTEGER is converted to its REAL representation) without causing a type mismatch error. Likewise, it is possible to pass a REAL expression to an INTEGER formal parameter (the value of the expression is rounded to the nearest INTEGER) without causing a type mismatch error (an integer overflow error is generated if the expression is out of range for an INTEGER).

Example Pass and Corresponding Formal Parameter Lists

Here is a sample **formal** parameter list showing which types each parameter demands:

```
SUB Read_dvm(@Dvm,A(*),INTEGER Lower,Upper,Status$,Errflag)
```

@Dvm — This is an I/O path name which may refer to either an I/O device or a mass storage file. Its name here implies that it is a voltmeter, but it is perfectly legal to redirect I/O to a file just by using a different ASSIGN with @Dvm.

A(*) — This is a REAL array. Its size is declared by the calling context. The parameters Lower and Upper contain its limits.

Lower Upper — These are declared here to be INTEGERs. Thus, when the calling program invokes this subprogram, it must supply either INTEGER variables or INTEGER expressions, or an error will occur.

Status$ — This is a simple string which presumably could be used to return the status of the voltmeter to the main program. The length of the string is defined by the calling context.

Errflag — This is a REAL number. The declaration of the string Status$ has limited the scope of the INTEGER keyword which caused Lower and Upper to require INTEGER pass parameters.

Let's look at our previous example from the calling side (which shows the **pass** parameter list):

```
CALL Read_dvm(@Voltmeter,Readings(*),1,400,Status$,Errflag)
```

@Voltmeter — This is the pass parameter which matches the formal parameter @Dvm in the subprogram. I/O path names are always passed by reference, which means the subprogram can close the I/O path or assign it to a different file or device.

Readings(*) — This matches the array A(*) in the subprogram's formal parameter list. Arrays, too, are always passed by reference.

1, 400 — These are the values passed to the formal parameters Lower and Upper. Since constants are classified as expressions rather than variables, these parameters have been passed by value. Thus, if the subprogram used either Lower or Upper on the left-hand side of an assignment operator, no change would take place in the calling context's value area.

Status$ — This is passed by reference here. If it were enclosed in parentheses, it would be passed by value. Notice that if it were passed by value, it would be totally useless as a method for returning the status of the voltmeter to the calling context.

Errflag — This is passed by reference.

COM Blocks

Since we've discussed parameter lists in detail, let's turn now to the other method a subprogram has of communicating with the main program or with other subprograms, the COM block.

There are two types of COM (or common) blocks: blank and labeled. Blank COM is simply a special case of labeled COM (it is the COM whose name is nothing) with the exception that blank COM must be declared in the main program, while labeled COM blocks don't have to be declared in the main program. Both types of COM blocks simply declare blocks of data which are accessible to any context having matching COM declarations.

A blank COM block might look like this:

```
20   COM Conditions(15),INTEGER,Cmin,Cmax,@Nuclear_pile,Pile_status$[20],
Tolerance
```

A labeled COM might look like this:

```
30 COM /Valve/ Main(10),Subvalves(10,15),@Valve_ctrl
```

A COM block's name, if it has one, will immediately follow the COM keyword, and will be set off with slashes, as shown above. The same rules used for naming variables and subprograms are used for naming COM blocks.

Any context need only declare those COM blocks which it needs to have access to. If there are 150 variables declared in 10 COM blocks, it isn't necessary for every context to declare the entire set—only those blocks that are necessary to each context need to be declared. COM blocks with matching names must have matching definitions. As in parameter lists, matching COM blocks is done by position and type, not by name.

COM vs. Pass Parameters

There are several characteristics of COM blocks which distinguish them from parameter lists as a means of communications between contexts:

- COM survives pre-run. In general, any numeric variable is set to 0, strings are set to the null string, and I/O path names are set to undefined after instructing the program to run, or upon entering a subprogram. This is true of COM the first time the program runs, but after COM block variables are defined, they retain their values until:
  - □ SCRATCH A or SCRATCH C is executed,
  - □ A statement declaring a COM block is modified by the user, or
  - □ A new program is brought into memory using the GET command which doesn't match the declaration of a given COM block, or which doesn't declare a given COM block at all.
- COM blocks can be arbitrarily large. One limitation on parameter lists (both pass and formal parameter lists) is that they must fit into a single program line along with the line's number, possibly a label, the invocation or subprogram header, and possibly (in the case of a function) a string or numeric expression. Depending upon the situation, this can impose a restriction on the size of your parameter lists.
- COM blocks can take as many statements as necessary. COM statements can be interwoven with other statements (though this is considered a poor practice). All COM statements within a context which have the same name will be part of the definition of that COM block.

- COM blocks can be used for communicating between contexts that do not invoke each other.
- COM blocks can be used to communicate between subprograms that are not in memory simultaneously.
- COM blocks can be used to retain the value of "local" variables between subprogram calls.
- COM blocks allow subprograms to share data without the intervention of the main program.

Hints for Using COM Blocks

Any COM blocks needed by your program must be resident in memory at prerun time, executing a RUN command, executing GET from the program, or executing a GET from the keyboard and specifying a run line. Thus if you want to create libraries of subprograms which share their own labeled COM blocks, it is wise to collect all the COM declarations together in one subprogram to make it easy to append them to the rest of the program for inclusion at prerun time. (The subprogram need not contain anything but the COM declarations.)

COM can be used to communicate between programs which overlay each other using GET statements, if you remember a few rules:

1. COM blocks which match each other exactly between the two programs will be preserved intact. "Matching" requires that the COM blocks are named identically (except blank COM), and that corresponding blocks have exactly the same number of variables declared, and that the types and sizes of these variables match.
2. Any COM blocks existing in the old program which are not declared in the new program (the one being brought in with the GET) are destroyed.
3. Any COM blocks which are named identically, but which do not match variables and types identically, are defined to match the definition of the new program. All values stored in that COM block under the old program are destroyed.
4. Any new COM blocks declared by the new program (including those mentioned above in #3) are initialized implicitly. Numeric variables and arrays are set to zero, strings are set to the null string, and I/O path names are set to undefined.

The first occurrence in memory of a COM block is used to define or set up the block. Subsequent occurrences of the COM block must match the defining block, both in the number of items, and the types of the items. In the case of strings and arrays, the actual sizes need be specified only in the defining COM blocks. Subsequent occurrences of the COM blocks may either explicitly match the size specifications by re-declaring the same size, or they may implicitly match the size specifications. In the case of strings, this is done by not declaring any size, just declaring the string name. In the case of arrays, this is done by using the (*) specifier for the dimensions of the array instead of explicitly re-declaring the dimensions.

Consider the following COM block definition:

```
10   COM /Dvm_state/ INTEGER Range,Format,N,REAL
Delay,Lastdata(1:40),Status$[20]
```

The following occurrence of the same COM block within a subprogram matches the COM block explicitly and is legal:

```
2000 COM /Dvm_state/ INTEGER Range,Format,N,REAL
Delay,Lastdata(1:40),Status$[20]
```

The following block within a different subprogram uses implicit matching and is also legal:

```
4010 COM /Dvm_state/ INTEGER Range,Format,N,REAL Delay,Lastdata(*),Status$
```

In general, the implicit size matching on arrays and strings is preferable to the explicit matching because it makes programs easier to modify. If it becomes necessary to change the size of an array or string in a COM block, it only needs to be changed in one statement, the one which defines the COM block. If all other occurrences of the COM block use the (*) specifier for arrays, and omit the length field in strings, none of those statements will have to be changed as a result of changing an array or string size.

Context Switching

A subprogram has its own **context** or state which is distinct from a main program and all other subprograms. In between the time that a CALL statement is executed (or an FN name is used) and the time that the first statement in the subprogram gets executed, the computer performs a "prerun" on the subprogram. This "entry" phase is what defines the context of the subprogram. The actions performed at subprogram entry are similar, but not identical, to the actual prerun performed at the beginning of a program. Here is a summary:

- The calling context has a DATA pointer which points to the next item in the current DATA block which will be used the next time a READ is executed (assuming of course that a DATA block even exists in the calling program). This pointer is saved away whenever a subprogram is called, and then the DATA pointer is reset to the first DATA statement in the new subprogram context.
- The RETURN stack for any GOSUBs in the current context is saved and set to the empty stack in the new context.
- The system priority of the current context is saved, and the called subprogram inherits this value. Any change to the system priority which takes place within the subprogram (or any of the subprograms which it calls in turn) is purely local, since the system priority is restored to its original value upon subprogram exit.
- Any event-initiated GOTO/GOSUB statements are disabled for the duration of the subprogram. If any of the specified events occur, this will be logged, but no action will be taken. (The fact that an event did occur will be logged, but only once—multiple occurrences of the same event will not be serviced.) Upon exiting the subprogram, these event-initiated conditions will be restored to active status, and if any of these events occurred while the subprogram was being executed, the proper branches will be taken.
- Any event-initiated CALL/RECOVER statements are saved away upon entering a subprogram, but the subprogram still inherits these ON conditions since CALL/RECOVER are global in scope. However, it is legal for the subprogram to redefine these conditions, in which case the original definitions are restored upon subprogram exit.
- The current DEG or RAD mode for trigonometric operations and graphics rotations is stored away. The subprogram will inherit the current DEG or RAD setting, but if it gets changed within the subprogram, the original setting will be restored when the subprogram is exited.

Variable Initialization

Space for all arrays and variables declared is set aside, whether they are declared explicitly with DIM, REAL, or INTEGER, or implicitly just by using the variable. The entire value area is initialized as part of the subprogram's prerun. All numeric values are set to zero, all strings are set to the null string, and all I/O path names are set to undefined.

Subprograms and Softkeys

ON KEYs are a special case of the event-initiated conditions that are part of context switching. They are special because they are the only event conditions which give visible evidence of their existence to the user through the softkey labels at the bottom of the CRT. These key labels are saved away just as the event conditions are, and the labels get restored to their original state when the subprogram is exited, regardless of any changes the subprogram made in the softkey definitions. This means the programmer doesn't have to make any special allowances for re-enabling his keys and their associated labels after calling a subprogram which changes them—the language system handles this automatically.

Subprograms and the RECOVER Statement

The event-initiated RECOVER statement allows the programmer to cause the program to resume execution at any given place in the context defining the ON ... RECOVER as a result of a specified event occurring, regardless of subprogram nesting.

Thus, if a main program executes an ON ... RECOVER statement (for example a softkey or an external interrupt from the SRQ line on an HP-IB), and then calls a subprogram, which calls a subprogram, which calls a subprogram, etc., program execution can be caused to immediately resume within the main program as a result of the specified event happening.

Editing Subprograms

Inserting Subprograms

There are some rules to remember when inserting SUB and DEF FN statement in the middle of the program. All DEF FN and SUB statements must be appended to the *end* of the program. If you want to insert a subprogram in the middle of your program because your prefer to see it listed in a given order, you must perform the following sequence:

1. SAVE the program.
2. Delete all lines *above* the point where you want to insert your subprogram.
3. SAVE the remaining segment of the program in a new file.
4. GET the original program stored in step 1.
5. Delete all lines *below* the point where you want to insert your subprogram.
6. Type in the new subprogram.
7. Do a GET from the new file created in step 3.

Deleting Subprograms

It is not possible to delete either DEF FN or SUB statements unless you first delete all the other lines in the subprogram. This includes any comments after the SUBEND or FNEND. Another way to delete DEF FN and SUB statements is to delete the entire subprogram, up to, but *not* including, the next SUB or DEF FN line (if any).

Merging Subprograms

If you want to merge two subprograms together, first examine the two subprograms carefully to insure that you don't introduce conflicts with variable usage and logic flow. If you've convinced yourself that merging the two subprograms is really necessary, here's how you go about it:

1. SAVE everything in your program *after* the SUB or DEF FN statement you want to delete.
2. Delete everything in your program from the unwanted SUB statement to the end.
3. GET the program segment you saved in step 1 back into memory, taking care to number the segment in such a way as not to overlay the part of the program already in memory.

SUBEND and FNEND

The SUBEND and FNEND statements must be the last statements in a SUB or function subprogram, respectively. These statements don't ever have to be executed; SUBEXIT and RETURN are sufficient for exiting the subprogram. (If SUBEND is executed, it will behave like a SUBEXIT. If FNEND is executed, it will cause an error.) Rather, SUBEND and FNEND are delimiter statements that indicate to the language system the boundaries between subprograms. The only exception to this rule is the comment statements (either REM or !), which are allowed after SUBEND and FNEND.

Recursion

Both function subprograms and SUB subprograms are allowed to call themselves. This is known as recursion. Recursion is a useful technique in several applications.

The simplest example of recursion is the computation of the factorial function. The factorial of a number N is denoted by N! and is defined to be N × (N−1)! where 0!=1 by definition. Thus N! is simply the product of all the whole numbers from 1 through N inclusive. A recursive function which computes N factorial is:

```
100  DEF FNFactorial(INTEGER N)
110  IF N=0 THEN RETURN 1
120  RETURN N*FNFactorial(N-1)
130  FNEND
```

7

Data Storage and Retrieval

This chapter describes some useful techniques for storing and retrieving data.

- First we describe how to store and retrieve *data that is part of the HP Instrument BASIC program.* With this method, **DATA statements** specify data to be stored in the memory area used by HP Instrument BASIC programs; thus, the data is always kept with the program, even when the program is stored in a mass storage file. The data items can be retrieved by using READ statements to assign the values to variables. This is a particularly effective technique for small amounts of data that you want to maintain in a program file.
- For larger amounts of data, and for data that will be generated or modified by a program, **mass storage files** are more appropriate. Files provide means of storing data on mass storage devices. The two types of data files available with HP Instrument BASIC are described in this chapter.
    - □ ASCII—used for general text and numeric data storage. (These are the interchange method with many other HP systems.)
    - □ BDAT—provide the most compact and flexible data storage mechanism.

More details about these files, including how to choose a file type and how to access each, are described in this chapter.

Storing Data in Programs

This section describes a number of ways to store values in memory. In general, these techniques involve using program variables to store data. The data are kept with the program when it is stored on a mass storage device (with SAVE). These techniques allow extremely fast access of the data. They provide good use of the computer's memory for storing relatively small amounts of data.

Storing Data in Variables

Probably the simplest method of storing data is to use a simple assignment, such as the following LET statements:

```
100   LET Cm_per_inch=2.54
110   Inch_per_cm=1/Cm_per_inch
```

The data stored in each variable can then be retrieved simply by specifying the variable's name. This technique works well when there are only a relatively few items to be stored or when several data values are to be computed from the value of a few items. The program will execute faster when variables are used than when expressions containing constants are used; for instance, using the variable `Inch_per_cm` in the preceding example would be faster than using the constant expression 1/2.54. In addition, it is easier to modify the value of an item when it appears in only one place (i.e., in the LET statement).

Data Input by the User

You also can assign values to variables at run-time with the INPUT statement as shown in the following examples.

```
100   INPUT "Type in the value of X, please.",Id
200   DISP "Enter the value of X, Y, and Z.";
210   INPUT "",X,Y,Z
```

Note that with this type of storage, the values assigned to the corresponding variables are *not* kept with the program when it is stored; they must be entered each time the program is run. This type of data storage can be used when the data are to be checked or modified by the user each time the program is run. As with the preceding example, the data stored in each variable can then be retrieved simply by specifying the variable's name.

Using DATA and READ Statements

The DATA and READ statements provide another technique for storing and retrieving data from the computer's read/write (R/W) memory. The DATA statement allows you to store a stream of data items in memory, and the READ statement allows you retrieve data items from the stream.

You can have any number of READ and DATA statements in a program in any order you want. When you RUN a program, the system concatenates all DATA statements in the same context into a single "data stream." Each subprogram has its own data stream. The following DATA statements distributed in a program would produce the following data stream.

```
100 DATA 1,A,50
      .
      .
      .
200 DATA "BB",20,45
      .
      .
      .
300 DATA X,Y,77
```

DATA STREAM:	1	A	50	BB	20	45	X	Y	77

As you can see from the example above, a data stream can contain both numeric and string data items; however, each item is stored as if it were a string.

Each data item must be separated by a comma and can be enclosed in optional quotes. Strings that contain a comma, exclamation mark, or quote mark must be enclosed in quotes. In addition, you must enter two quote marks for every one you want in the string. For example, to enter the string QUOTE"QUO"TE into a data stream, you would write:

```
100 DATA "QUOTE""QUO""TE"
```

To retrieve a data item, assign it to a variable with the READ statement. Syntactically, READ is analogous to DATA; but instead of a data list, you use a variable list. For instance, the statement:

```
100 READ X,Y,Z$
```

would read three data items from the data stream into the three variables. Note that the first two items are numeric and the third is a string variable.

Numeric data items can be READ into either numeric or string variables. If the numeric data item is of a different type than the numeric variable, the item is converted (i.e., REALs are converted to INTEGERs, and INTEGERs to REALs). If the conversion cannot be made, an error is returned. Strings that contain non-numeric characters must be READ into string variables. If the string variable has not been dimensioned to a size large enough to hold the entire data item, the data item is truncated.

The system keeps track of which data item to READ next by using a "data pointer." Every data stream has its own data pointer which points to the next data item to be assigned to the next variable in a READ statement. When you run a program segment, the data pointer is placed initially at the first item of the data stream. Every time you READ an item from the stream, the pointer is moved to the next data item. If a subprogram is called by a context, the position of the data pointer is recorded and then restored when you return to the calling context.

Starting from the position of the data pointer, data items are assigned to variables one by one until all variables in a READ statement have been given values. If there are more variables than data items, the system returns an error, and the data pointer is moved back to the position it occupied before the READ statement was executed.

Examples

The following example shows how data is stored in a data stream and then retrieved. Note that DATA statements can come after READ statements even though they contain the data being READ. This is because DATA statements are linked during program pre-run, whereas READ statements aren't executed until the program actually runs.

```
10  DATA November,26
20  READ Month$,Day,Year$
30  DATA 1981,"The date is"
40  READ Str$
50  Print Str$;Month$,Day,Year$
60  END

  The date is November 26 1981
```

Storage and Retrieval of Arrays

In addition to using READ to assign values to string and numeric variables, you can also READ data into arrays. The system will match data items with variables one at a time until it has filled a row. The next data item then becomes the first element in the next row. You must have enough data items to fill the array or you will get an error. In the example below, we show how DATA values can be assigned to elements of a 3-by-3 numeric array.

```
10    DIM Example1(2,2)
20    DATA 1,2,3,4,5,6,7,8,9,10,11
30    READ Example1(*)
40    PRINT USING "3(K,X),/";Example1(*)
```

```
50    END

 1 2 3
 4 5 6
 7 8 9
```

The data pointer is left at item 10; thus, items 10 and 11 are saved for the next READ statement.

Moving the Data Pointer

In some programs, you will want to assign the same data items to different variables. To do this, you have to move the data pointer so that it is pointing at the desired data item. You can accomplish this with the RESTORE statement. If you don't specify a line number or label, RESTORE returns the data pointer to the first data item in the data stream. If you do include a line identifier in the RESTORE statement, the data pointer is moved to the first data item in the first DATA statement at or after the identified line. The example below illustrates how to use the RESTORE statement.

```
100   DIM Array1(1:3)   ! Dimensions a 3-element array.
110   DIM Array2(0:4)   ! Dimensions a 5-element array.
120   DATA 1,2,3,4      ! Places 4 items in stream.
130   DATA 5,6,7        ! Places 3 items in stream.
140   READ A,B,C        ! Reads first 3 items in stream.
150   READ Array2(*)    ! Reads next 5 items in stream.
160   DATA 8,9          ! Places 2 items in stream.
170                     !
180   RESTORE           ! Re-positions pointer to 1st item.
190   READ Array1(*)    ! Reads first 3 items in stream.
200   RESTORE 140       ! Moves data pointer to item "8".
210   READ D            ! Reads "8".
220                     !
230   PRINT "Array1 contains:";Array1(*);" "
240   PRINT "Array2 contains:";Array2(*);" "
250   PRINT "A,B,C,D equal:";A;B;C;D
260   END

Array1 contains: 1 2 3
Array2 contains: 4 5 6 7 8
A,B,C,D equal: 1 2 3 8
```

File Input and Output (I/O)

The rest of this chapter describes the second general class of data storage and retrieval—that of using mass storage files. It presents HP Instrument BASIC programming techniques used for accessing files.

- The first section gives a brief introduction to the *general* steps you might take to:
  - □ Choose a file type.
  - □ Store data in any file.
- Subsequent sections describe *details* of these steps with ASCII, BDAT, and HP-UX or DOS files.

Brief Comparison of Available File Types

With HP Instrument BASIC, there are three different types of files in which you can store and retrieve data, ASCII, BDAT, and HPUX or DOS. Understanding the characteristics of each file type will help you choose the one best suited for your specific application.

Note Note that not every system will implement all of these file types.

- ASCII—used for general text and numeric data storage.

  Here are the *advantages* of this type of file:

  - □ There is less chance of reading the contents into the wrong data type (which is possible with BDAT and HP-UX files). Thus, it is the easiest file to read when you don't know how it was written.
  - □ The file format provides fairly compact storage for string data.
  - □ ASCII files are compatible with other HP computers that support this file type. (The full name of ASCII files is "LIF ASCII." LIF stands for Logical Interchange Format, a directory and data storage format that is used by many HP computers.)
  - □ ASCII files containing HP Instrument BASIC program lines can be read with GET and written with SAVE.

  The main *disadvantages* of ASCII files are that:

  - □ They can be accessed *serially* but not *randomly.*
  - □ They can be written in *only default ASCII format* (no formatting is possible, and the data cannot be stored in internal representation. It is possible, however, to format data by first sending it to a string variable (with OUTPUT..USING), and then OUTPUT this string's contents to the file. See the subsequent section called "Formatted OUTPUT with ASCII Files" for examples.)
- BDAT—provide the most compact and flexible data storage mechanism.

  These files have several *advantages:*

  - □ They can be *randomly or serially* accessed.
  - □ More *flexibility* in data formats and access methods.

  - □ *Faster* transfer rates.
  - □ Generally more *space-efficient* than ASCII files (except for string data items).
  - □ They allow data to be stored in ASCII format, internal format, or in a "custom" format (which you can define with IMAGE specifiers).

  The *disadvantages* are that:

  - □ You *must* know how the data items were written (as INTEGERs, REALs, strings, etc.) in order to correctly read the data back.
  - □ These data files cannot be *interchanged* with as many other systems as can ASCII files.

- HP-UX—similar to BDAT files in structure, but also have some of the advantages of ASCII files:
  - □ Like BDAT files, they can also be accessed randomly or serially, and they can use ASCII, internal, or custom data representations.
  - □ Like ASCII files, they are useful for data-file interchange; however, the set of computers with which they can be interchanged is slightly different than LIF ASCII files. HP-UX files can be interchanged with any other system that uses the Hierarchical File System (HFS) format for mass storage volumes (such as HP-UX systems, and HP Series 200/300 Pascal systems beginning with version 3.2).
  - □ HP-UX files containing HP Instrument BASIC program lines can be read with GET and written with RE-SAVE.
- DOS—identical to HP-UX files, they provide file compatibility with MS-DOS.

If in doubt about the type of file to use, choose a BDAT file because of its speed and compact data storage.

Creating Data Files

You can use three BASIC statements to create data files. Use **CREATE ASCII** to create an ASCII file, **CREATE BDAT** to create a BDAT file, or simply **CREATE** to create an HP-UX or DOS file. Note that the **CREATE** statement creates a DOS file on a DOS file system. Otherwise, it creates an HP-UX file.

For example, the statements:

```
CREATE ASCII "Text",100
CREATE BDAT "Text",100
CREATE "Data_file",100
```

all create a data file with a length of 100 records in the current mass storage volume and directory. The file type is ASCII for the first statement, BDAT for the second, and HP-UX or DOS for the third.

Note that you can use CREATE, CREATE ASCII, and CREATE BDAT to create files within LIF volumes, HFS volumes and DOS volumes. Each of these statements contains a file specifier which can include a volume and directory specification. If no volume or directory is specified, it creates the file in the current volume and directory as determined by the last MASS STORAGE IS statement.

Overview of File I/O

Storing data in files requires a few simple steps. The following program segment shows a simple example of placing several items in a data file.

```
100  REAL Real_array1(1:50,1:25),Real_array2(1:50,1:25)
110  INTEGER Integer_var
120  DIM String$[100]
       .
       .
390  ! Specify default mass storage.
400  MASS STORAGE IS ":,700,1"
410  !
420  ! Create BDAT data file with ten (256-byte) records
430  ! on the specified mass storage device (:,700,1).
440  CREATE BDAT "File_1",10
450  !
460  ! Assign (open) an I/O path name to the file.
470  ASSIGN @Path_1 TO "File_1"
480  !
490  ! Write various data items into the file.
500  OUTPUT @Path_1;"Literal"       ! String literal.
510  OUTPUT @Path_1;Real_array1(*)  ! REAL array.
520  OUTPUT @Path_1;255             ! Single INTEGER.
530  !
540  ! Close the I/O path.
550  ASSIGN @Path_1 TO *
     .
     .
     .
790  ! Open another I/O path to the file (assume same default drive).
800  ASSIGN @F_1 TO "File_1"
810  !
820  ! Read data into another array (same size and type).
830  ENTER @F_1;String_var$         ! Must be same data types
840  ENTER @F_1;Real_array2(*)      ! used to write the file.
850  ENTER @F_1;Integer_var         ! "Read it like you wrote it."
860  !
870  ! Close I/O path.
880  ASSIGN @F_1 TO *
```

Line 400 specifies the *default mass storage device,* which is to be used whenever a mass storage device is *not explicitly specified* during subsequent mass storage operations. The term **mass storage volume specifier** (msvs) describes the string expression used to uniquely identify which device is to be the mass storage. In this case, ":,700,1" is the msvs.

In order to store data in mass storage, a data file must be created (or already exist) on the mass storage media. In this case, line 440 creates a BDAT file; the file created contains 10 defined records of 256 bytes each. (Defined records and record size are discussed later in this chapter.)

The term **file specifier** describes the string expression used to uniquely identify the file. In this example, the file specifier is simply File_1, which is the file's name. If the file is to be created (or already exists) on a mass storage device *other than the default mass storage,* the appropriate msus must be appended to the file name. If that device has a hierarchical directory format (such as HFS or MS-DOS discs), then you may also have to specify a directory path (such as /USERS/MARK/PROJECT_1 for LIF or \USERS\MARK\PROJECT_1 for MS-DOS).

Then, in order to store data in (or retrieve data from) the file, you must assign an I/O path name to the file. Line 470 shows an example of assigning an I/O path name to the file (also called opening an I/O path to the file). Lines 500 through 520 show data items of various types being written into the file through the I/O path name.

The I/O path name is closed after all data have been sent to the file. In this instance, closing the I/O path may have been optional, because a *different* I/O path name is assigned to the file later in the program. (All I/O path names are automatically closed by the system at the end of the program.) Closing an I/O path to a file updates the file pointers.

Since these data items are to be retrieved from the file, another ASSIGN statement is executed to open the file (line 800). Notice that a different I/O path name was arbitrarily chosen. Opening this I/O path name to the file sets the file pointer to the beginning of the file. (Re-opening the I/O path name @File_1 would have also reset the file pointer.)

Notice also that the msvs is *not* included with the file name. This shows that the current default mass storage device, here ":,700,1", is assumed when a mass storage device is not specified.

The subsequent ENTER statements read the data items into variables; *with BDAT and HP-UX files,* the *data type of each variable must match the data type type of each data item.* With ASCII files, for instance, you can read INTEGER items into REAL variables and not have problems.

This is a fairly simple example; however, it shows the general steps you must take to access files.

A Closer Look at General File Access

Before you can access a data file, you must assign an I/O path name to the file. Assigning an I/O path name to the file sets up a table in computer memory that contains various information describing the file, such as its type, which mass storage device it is stored on, and its location on the media. The I/O path name is then used in I/O statements (OUTPUT, and ENTER) which move the data to and from the file.

Opening an I/O Path

I/O path names are similar to other variable names, except that I/O path names are preceded by the "@" character. When an I/O path name is used in a statement, the system looks up the contents of the I/O path name and uses them as required by the situation.

To open an I/O path to a file (to set the validity flag to Open), assign the I/O path name to a file specifier by using an ASSIGN statement. For example, executing the following statement:

```
ASSIGN @Path1 TO "Example"
```

assigns an I/O path name called "@Path1" to the file "Example". The file that you open must already exist and must be a data file. If the file does not satisfy one of these requirements, the system will return an error. If you do not use an msus in the file specifier, the system will look for the file on the current MASS STORAGE IS device. If you want to access a different device, use the msus syntax described earlier. For instance, the statement:

```
ASSIGN @Path2 TO "Example:HP9122,700"
```

open an I/O path to the file "Example" on the specified mass storage device. You must include the protect code or password, if the LIF file has one.

Once an I/O path has been opened to a file, you always use the path name to access the file. An I/O path name is only valid in the context in which it is opened, unless you pass it as a parameter or put it in the COM area. To place a path name in the COM area, simply specify the path name in a COM statement before you ASSIGN it. For instance the two statements below would declare an I/O path name in an unnamed COM area and then open it:

```
100   COM @Path3
110   ASSIGN @Path3 TO "File1"
```

Assigning Attributes

When you open an I/O path, certain attributes are assigned to it which define the way data is to be read and written. There are two attributes which control how data items are represented: FORMAT ON and FORMAT OFF.

- With FORMAT ON, ASCII data representations are used.
- With FORMAT OFF, HP Instrument BASIC's internal data representations are used.

Additional attributes are available, which provide control of such functions as changing end-of-line (EOL) sequences. See ASSIGN in the *HP Instrument BASIC Language Reference* for further details.

As mentioned in the tutorial section, BDAT files can use either data representation; however, ASCII files permit only ASCII-data format. Therefore, if you specify FORMAT OFF for an I/O path to an ASCII file, the system ignores it. The following ASSIGN statement specifies a FORMAT attribute:

```
ASSIGN @Path1 TO "File1";FORMAT OFF
```

If `File1` is a BDAT or HP-UX file, the FORMAT OFF attribute specifies that the internal data formats are to be used when sending and receiving data through the I/O path. If the file is of type ASCII, the attribute will be ignored. *Note that FORMAT OFF is the default FORMAT attribute for BDAT and HP-UX files.*

Executing the following statement directs the system to use the ASCII data representation when sending and receiving data through the I/O path:

```
ASSIGN @Path2 TO "File2";FORMAT ON
```

If `File2` is a BDAT or HP-UX file, data will be written using ASCII format, and data read from it will be interpreted as being in ASCII format. For an ASCII file, this attribute is redundant since ASCII-data format is the only data representation allowed anyway.

If you want to change the attribute of an I/O path, you can do so by specifying the I/O path name and attribute in an ASSIGN statement while excluding the file specifier. For instance, if you wanted to change the attribute of `@Path2` to FORMAT OFF, you could execute:

```
ASSIGN @Path2;FORMAT OFF
```

Alternatively, you could re-enter the entire statement:

```
ASSIGN @Path2 TO "File2";FORMAT OFF
```

These two statements, however, are not identical. The first one only changes the FORMAT attribute. The second statement resets the entire I/O path table (e.g., resets the file pointer to the beginning of the file).

Closing I/O Paths

I/O path names not in the COM area are closed whenever the system moves into a stopped state (e.g., STOP, END, SCRATCH, EDIT, etc.). I/O path names local to a context are closed when control is returned to the calling context. Re-ASSIGNing an I/O path name will also cancel its previous association.

You can also explicitly cancel an I/O path by ASSIGNing the path name to an * (asterisk). For instance, the statement:

```
ASSIGN @Path2 TO *
```

closes @Path2. @Path2 cannot be used again until it is re-assigned. You can re-assign a path name to the same file or to a different file.

A Closer Look at Using ASCII Files

You have already been introduced to general file I/O techniques in the example of writing and reading a BDAT file in the preceding section. This section gives you a closer look at ASCII file I/O techniques.

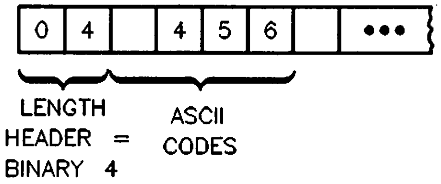
Example of ASCII File I/O

Storing data in ASCII files requires a few simple steps. The following program segment shows a simplistic example of placing several items in an ASCII data file. Note that it is *nearly identical* to the first example in the preceding "Overview of File I/O" section, except for changes to the CREATE statement (line 440) and file name.

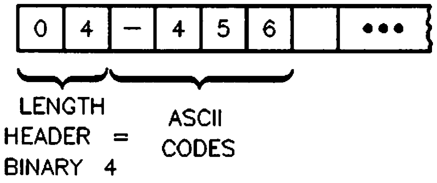
```
100  REAL Real_array1(1:50,1:25),Real_array2(1:50,1:25)
110  INTEGER Integer_var
120  DIM String$[100]
       .
       .
390  ! Specify "default" mass storage device.
400  MASS STORAGE IS ":,700,1"
410  !
420  ! Create ASCII data file with 10 sectors
430  ! on the "default" mass storage device.
440  CREATE ASCII "File_2",10
450  !
460  ! Assign (open) an I/O path name to the file.
470  ASSIGN @Path_1 TO "File_2"
480  !
490  ! Write various data items into the file.
500  OUTPUT @Path_1;"Literal"       ! String literal.
510  OUTPUT @Path_1;Real_array1(*)  ! REAL array.
520  OUTPUT @Path_1;255             ! Single INTEGER.
530  !
540  ! Close the I/O path.
550  ASSIGN @Path_1 TO *
       .
       .
790  ! Open another I/O path to the file (assume same default drive).
800  ASSIGN @F_1 TO "File_2"
810  !
820  ! Read data into another array (same size and type).
830  ENTER @F_1;String_var          ! Must be same data types.
840  ENTER @F_1;Real_array2(*)
850  ENTER @F_1;Integer_var
860  !
       .
       .
870  ! Close I/O path.
880  ASSIGN @F_1 TO *
       .
       .
```

Data Representations in ASCII Files

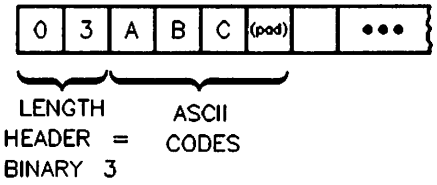
In an ASCII file, every data item, whether string or numeric, is represented by ASCII characters; one byte represents one ASCII character. Each data item is preceded by a two-byte length header which indicates how many ASCII characters are in the item. However, there is no "type" field for each item; data items contain no indication (in the file) as to whether the item was stored as string or numeric data. For instance, the number 456 would be stored as follows in an ASCII file:



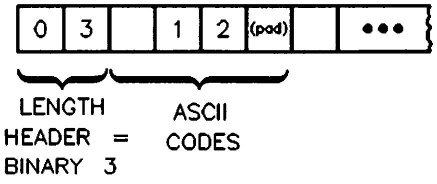
Note that there is a space at the beginning of the data item. This signifies that the number is positive. If a number is negative, a minus sign precedes the number. For instance, the number −456, would be stored as follows:



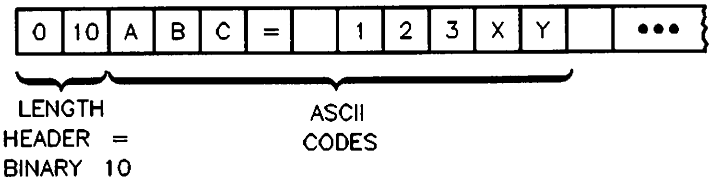
If the length of the data item is an odd number, the system "pads" the item with a space to make it come out even. The string "ABC", for example, would be stored as follows:



There is often a relatively large amount of overhead for numeric data items. For instance, to store the integer 12 in an ASCII file requires the following six bytes:



Similarly, reading numeric data from an ASCII file can be a complex and relatively slow operation. The numeric characters in an item must be entered and evaluated individually by the system's "number builder" routine, which derives the number's internal representation. (Keep in mind that this routine is called automatically when data are entered into a numeric variable.) For example, suppose that the following item is stored in an ASCII file:



Although it may seem obvious that this is not a numeric data item, the system has no way of knowing this since *there is no type-field stored with the item.* Therefore, if you attempt to enter this item into a numeric variable, the system uses the number-builder routine to strip away all non-numeric characters and spaces and assign the value 123 to the numeric variable. When you add to this the intricacies of real numbers and exponential notation, the situation becomes more complex. For more information about how the number builder works, see the chapter called "Entering Data" in *HP Instrument BASIC Interfacing Techniques.*

Because ASCII files require so much overhead (for storage of "small" items), and because retrieving numeric data from ASCII files is sometimes a complex process, they are not the preferred file type for numeric data when compactness is an important criteria. However, as we mentioned before, ASCII files are interchangeable with many other HP products.

In this chapter, we refer to the data representation described above as ASCII-data format. As mentioned earlier, you can also store data in BDAT files in ASCII format (by using the FORMAT ON attribute). Be careful not to confuse the ASCII-*file type* with the ASCII-*data format.* The ASCII format used in BDAT files when FORMAT ON is specified differs from the format used in ASCII files in several respects. Each item output to an ASCII file has its own length header; there are no length headers in a FORMAT ON BDAT file. At the end of each OUTPUT statement an end-of-line sequence is written to a FORMAT ON BDAT file unless suppressed by an IMAGE or EOL OFF. No end-of-line sequence is written to an ASCII file at the end of an OUTPUT statement.

In general, you should only use ASCII files when you want to transport data between HP Instrument BASIC and other machines. There may be other instances where you will want to use ASCII files, but you should be aware that they cause a *noticeable transfer rate degradation* compared to BDAT and HP-UX files (especially for numeric data items).

Formatted OUTPUT with ASCII Files

As mentioned in the "Brief Comparison of File Types," you cannot format items sent to ASCII files; that is, you *cannot* use the following statement with an ASCII file:

```
OUTPUT @Ascii_file USING "#,DD.D,4X,5A";Number,String$
```

You can, however, direct the output to a string variable first, and then OUTPUT this formatted string to an ASCII file:

```
OUTPUT String_var$ USING "#,DD.D,4X,5A";Number,String$
OUTPUT @Ascii_file;String_var$
```

When a string variable is specified as the destination of data in an OUTPUT statement, source items are evaluated individually and placed into the variable according to the free-field rules or the specified image, depending on which type of OUTPUT statement is used. Thus, item terminators may or may not be placed into the variable. The ASCII data representation is always used during outputs to string variables; in fact, *data output to string variables is exactly like that sent to devices through I/O paths with the FORMAT ON attribute.*

When using OUTPUT to a string, characters are always placed into the variable beginning at the first position; no other position can be specified as the beginning position at which data will be placed. Thus, *random access of the information in string variables is not allowed* from OUTPUT and ENTER statements; all data must be accessed serially. For instance, if the characters "1234" are output to a string variable by one OUTPUT statement, and a subsequent OUTPUT statement outputs the characters "5678" to the same variable, the second output *does not* begin where the first one left off (i.e., at string position five). The second OUTPUT statement begins placing characters in position one, just as the first OUTPUT statement did, overwriting the data initially output to the variable by the first OUTPUT statement.

The string variable's length header (2 bytes) is updated and compared to the dimensioned length of the string as characters are output to the variable. If the string is filled before all items have been output, an error is reported; however, the string contains the first *n* characters output (where *n* is the dimensioned length of the string).

The following example program shows how outputs to string variables can be used to reduce the overhead required in ASCII data files. To do this, the program compares two possible methods for storing data in an ASCII data file. The first method stores 64 two-byte items in a file one at a time. Each two-byte item is preceded by a two-byte length header. The second method stores 64 two-byte items in a string array which is output to a string variable. The string variable is then output to an ASCII data file with only one two-byte length header being used. Since the second method used only one two-byte length header to store 64 two-byte items, it can easily be seen that the second method required less overhead. Note that the second method is also the *only way to format data sent to ASCII data files.*

```
100   PRINTER IS CRT
110   !
120   ! Create a file 1 record long (=256 bytes).
130   ON ERROR GOTO File_exists
140   CREATE ASCII "TABLE",1
150 File_exists:   OFF ERROR
160                !
170                !
180   ! First method outputs 64 items individually..
190   ASSIGN @Ascii TO "TABLE"
200   FOR Item=1 TO 64  ! Store 64 2-byte items.
```

```
210        OUTPUT @Ascii;CHR$(Item+31)&CHR$(64+RND*32)
220        STATUS @Ascii,5;Rec,Byte
230        DISP USING Image_1;Item,Rec,Byte
240    NEXT Item
250 Image_1: IMAGE "Item ",DD,"  Record ",D,"  Byte ",3D
260    DISP
270    Bytes_used=256*(Rec-1)+Byte-1
280    PRINT Bytes_used;" bytes used with 1st method."
290    PRINT
300    PRINT
310    !
320    !
330    ! Second method consolidates items.
340    DIM Array$(1:64)[2],String$[128]
350    ASSIGN @Ascii TO "TABLE"
360    !
370    FOR Item=1 TO 64
380        Array$(Item)=CHR$(Item+31)&CHR$(64+RND*32)
390    NEXT Item
400    !
410    OUTPUT String$;Array$(*); ! Consolidate in string variable.
420    OUTPUT @Ascii;String$     ! OUTPUT to file as 1 item.
430    !
440    STATUS @Ascii,5;Rec,Byte
450    Bytes_used=256*(Rec-1)+Byte-1
460    PRINT Bytes_used;" bytes used with 2nd method."
470    !
480    END
```

The program shows many of the features of using ASCII files and string variables. The first method of outputting the data items shows how the file pointer varies as data are sent to the file. Note that the file pointer points to the *next* file position at which a subsequent byte will be placed. In this case, it is incremented by four by every OUTPUT statement (since each item is a two-byte quantity preceded by a two-byte length header).

The program could have used a BDAT file, which would have resulted in using slightly less disc-media space; however, using BDAT files usually saves much more disc space than would be saved in this example. The program does not show that *ASCII files cannot be accessed randomly;* this is one of the major differences between using ASCII and BDAT (and HP-UX) files.

Using VAL$

The VAL$ function (or a user-defined function subprogram) and outputs made to string variables can be used to generate the string representation of a number. The advantage of the latter method is you can explicitly specify the number's image. The following program compares a string generated by the VAL$ function to that generated by outputting a number to a string variable.

```
100   X=12345678
110   !
120   PRINT VAL$(X)
130   !
140   OUTPUT Val$ USING "#,3D.E";X
150   PRINT Val$
160   !
170   END
```

Printed Results

```
    1.2345678E+7
    123.E+05
```

Formatted ENTER with ASCII Files

Data is entered from string variables in much the same manner as output to the variable. For example,

```
     ENTER @File;String$
     ENTER String$;Var1, Var2$
```

All ENTER statements that use string variables as the data source interpret the data according to the FORMAT ON attribute. Data is read from the variable beginning at the first string position; if a subsequent ENTER statement reads characters from the variable, the read also begins at the first position. If more data is to be entered from the string than is contained in the string, an error is reported; however, all data entered into the destination variable(s) before the end of the string was encountered remain in the variable(s) after the error occurs.

When entering data from a string variable, the computer keeps track of the number of characters taken from the variable and compares it to the string length. Thus, statement-termination conditions are *not* required; the ENTER statement automatically terminates when the last character is read from the variable. However, *item* terminators are still required *if* the items are to be separated *and* the lengths of the items are not known. If the length of each item is known, an image can be used to separate the items.

A Closer Look at BDAT and HP-UX or DOS Files

As mentioned earlier, BDAT and HP-UX files are designed for flexibility (random and serial access, choice of data representations), storage-space efficiency, and speed. This chapter provides several examples of using these types of files.

Data Representations Available

The data representations available are:

- HP Instrument BASIC internal formats (allow the fastest data rates and are generally the most space-efficient)
- ASCII format (the most interchangeable)
- Custom formats (design your own data representations using IMAGE specifiers)

The remainder of this section gives more details for each type of data representation.

Random vs. Serial Access

Random access means that you can directly read from and write to any record within the file, while serial access only permits you to access the file in order, from the beginning. That is, you must read records 1, 2, ... , $n-1$ before you can read record n. Serial access can waste a lot of time if you're trying to access data at the end of a file. On the other hand, if you want to access the entire file sequentially, you are better off using serial access than random access, because it generally requires less programming effort and often uses less file space. BDAT and HP-UX files can be accessed both randomly and serially, while ASCII files can be accessed only serially.

Data Representations Used in BDAT Files

BDAT files allow you to store and retrieve data using internal format, ASCII format, or user-defined formats.

- With internal format (FORMAT OFF), items are represented with the same format the system uses to store data in internal computer memory. (This is the default FORMAT for BDAT and HP-UX files.)
- With ASCII format (FORMAT ON), items are represented by ASCII characters.
- User-defined formats are implemented with programs that employ OUTPUT and ENTER statements that reference IMAGE specifiers (items are represented by ASCII characters).

Complete descriptions of ASCII and user-defined formats are given in *HP Instrument BASIC Interfacing Techniques.* This section shows the details of internal (FORMAT OFF) representations of numeric and string data.

BDAT Internal Representations (FORMAT OFF)

In most applications, you will use internal format for BDAT files. Unless we specify otherwise, you can assume that when we talk about retrieving and storing data in BDAT files, we are also talking about internal format. This format is synonymous with the FORMAT OFF attribute, which is described later in this chapter.

Because FORMAT OFF assigned to BDAT files uses almost the same format as internal memory, very little interpretation is needed to transfer data between the computer and a FORMAT OFF file. FORMAT OFF files, therefore, not only save space but also save time.

Data stored in internal format in BDAT files require the following number of bytes per item:

Data Type	Internal Representation
INTEGER	2 bytes
REAL	8 bytes
String	4-byte length header; 1 byte per character (plus 1 pad byte if string length is an odd number)

INTEGER values are represented in BDAT files which have the FORMAT OFF attribute by using a 16-bit, two's-complement notation, which provides a range −32 768 through 32 767. If bit 15 (the MSB) is 0, the number is positive. If bit 15 equals 1, the number is negative; the value of the negative number is obtained by changing all ones to zeros, and all zeros to ones, and then adding one to the resulting value.

Binary Representation	Decimal Equivalent
00000000 00010111	23
11111111 11101000	−24
10000000 00000000	−32768
01111111 11111111	32767
11111111 11111111	−1
00000000 00000001	1
00100011 01000111	9031
11011100 10111001	−9031

REAL values are stored in BDAT files by using their internal format (when FORMAT OFF is in effect): the IEEE-standard, 64-bit, floating-point notation. Each REAL number is comprised of two parts: an exponent (11 bits), and a mantissa (53 bits). The mantissa uses a sign-and-magnitude notation. The sign bit for the mantissa is not contiguous with the rest of the mantissa bits; it is the most significant bit (MSB) of the entire eight bytes. The 11-bit exponent is offset by 1 023 and occupies the 2nd through the 12th MSB's. Every REAL number is internally represented by the following equation. (Note that the mantissa is in binary notation):

$$-1^{\texttt{mantissa sign}} \times 2^{\texttt{exponent} - \texttt{1023}} \times \texttt{1.mantissa}$$

String data are stored in FORMAT OFF BDAT files in their internal format.

Every character in a string is represented by one byte which contains the character's ASCII code. A 4-byte length header contains a value that specifies the length of the string. If the length of the string is odd, a pad character is appended to the string to get an even number of characters; however, the length header does not include this pad character.

The string "A" would be stored:

```
00000000 00000000 00000000 00000001  01000001 00100000
         Length = 0001 (binary)      ASCII 65 ASCII 32
```

In this case, the space character (ASCII code 32) is used as the pad character; however, not all operations use the space as the pad character.

ASCII and Custom Data Representations

When using the ASCII data format for BDAT files, all data items are represented with ASCII characters. With user-defined formats, the image specifiers referenced by the OUTPUT or ENTER statement are used to determine the data representation (which is ASCII characters).

```
OUTPUT @File USING "SDD.DD,XX,B,#";Number,Binary_value
ENTER  @File USING "B,B,40A,%";Bin_val1,Bin_val2,String$
```

Using both of these formats with BDAT files produce results identical to using them with devices. The entire subject is described fully in *HP Instrument BASIC Interfacing Techniques.*

Data Representations with HP-UX and DOS Files

HP-UX and DOS files are *very similar to BDAT files.* The *only differences* between them are:

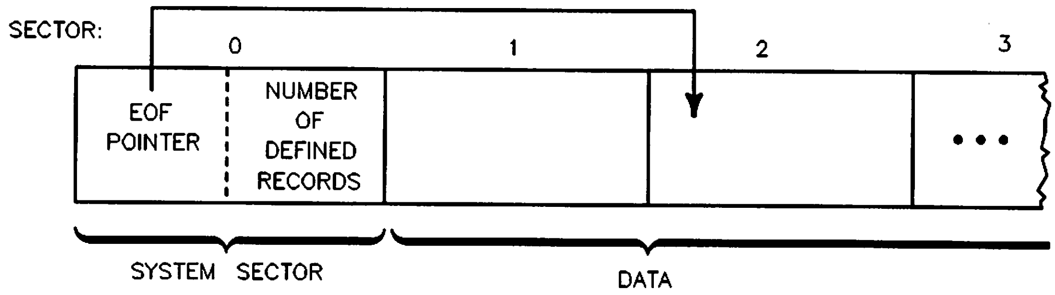
- The internal representation (FORMAT OFF) of strings is slightly different:
  - HP-UX and DOS FORMAT OFF strings have no length header; instead, they are terminated by a null character, CHR$(0).
  - BDAT FORMAT OFF strings have a 4-byte length header;
- HP-UX and DOS files have a *fixed record length of 1.* (BDAT files allow user-definable record lengths.)
- HP-UX and DOS files have *no system sector* like BDAT files do (see the next section for details).

The FORMAT ON representations for HP-UX files are the same as for devices. The entire subject is described fully in *HP Instrument BASIC Interfacing Techniques.*

Note Throughout this section, you should be able to assume that—unless otherwise stated—the techniques shown will apply to HP-UX and DOS as well as BDAT files.

BDAT File System Sector

On the disc, every BDAT file is preceded by a system sector that contains an end-of-file (EOF) pointer and the number of defined records in the file. All data is placed in succeeding sectors. You cannot directly access the system sector. However, as you shall see later, it is possible to indirectly change the value of an EOF pointer.



EOF Pointer:
- number of sectors from beginning of file (32-bit binary number)
- number of bytes from beginning of sector (32-bit binary number)

Number of defined records: See description below (32-bit binary number)

Defined Records

To access a BDAT file randomly, you specify a particular defined record. Records are the smallest units in a file directly addressable by a random OUTPUT or ENTER.

- With BDAT files, defined records can be anywhere from 1 through 65 534 bytes long.
- With HP-UX and DOS files, defined records are always 1 byte long.

Specifying Record Size (BDAT Files Only)

Both the length of the file and the length of the defined records in it are specified when you create a BDAT file. This section shows how to specify the record length of a BDAT file. (The next section talks about how to choose the record length.)

For example, the following statement would create a file called `Example` with 7 defined records, each record being 128 bytes long:

```
CREATE BDAT "Example",7,128
```

If you don't specify a record length in the CREATE BDAT statement, the system will set each record to the default length of 256 bytes.

Both the record length and the number of records are rounded to the nearest integer.

For example, the statement:

```
CREATE BDAT "Odd",3.5,28.7
```

would create a file with 4 records, each 30 bytes long. On the other hand, the statement:

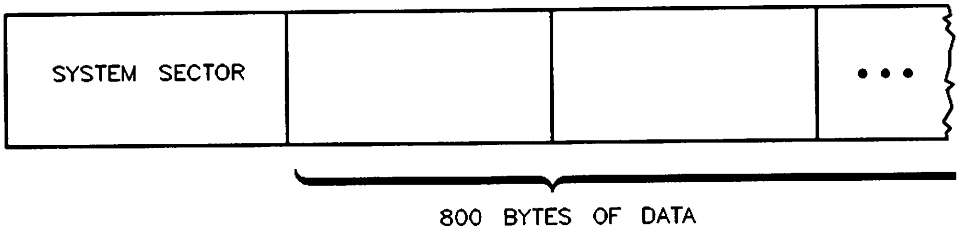
```
CREATE "Odder",3.49,28.3
```

would create a file with 3 records, each 28 bytes long.

Once a file is created, you cannot change its length, or the length of its records. You must therefore calculate the record size and file size required *before* you create a file.

Choosing A Record Length (BDAT Files Only)

Record length is important only for random OUTPUTs and ENTERs. It is not important for serial access. The most important consideration in selecting of a proper record length is the type of data being stored and the way you want to retrieve it. Suppose, for instance, that you want to store 100 real numbers in a file, and be able to access each number individually. Since each REAL number uses 8 bytes, the data itself will take up 800 bytes of storage.



The question is how to divide this data into records. If you define the record length to be 8 bytes, then each REAL number will fill a record. To access the 15th number, you would specify the 15th record. If the data is organized so that you are always accessing two data items at a time, you would want to set the record length to 16 bytes.

The worst thing you can do with data of this type is to define a record length that is not evenly divisible by eight. If, for example, you set the record length to four, you would only be able to randomly access half of each real number at a time. In fact, the system will return an End-Of-Record condition if you try to randomly read data into REAL variables from records that are less than 8 bytes long.

So far, we have been talking about a file that contains only REAL numbers. For files that contain only INTEGERs, you would want to define the record length to be a multiple of two. To access each INTEGER individually, you would use a record length of two; to access two INTEGERs at a time, you would use a record length of four, and so on.

Files that contain string data present a slightly more difficult situation since strings can be of variable length. If you have three strings in a row that are 5, 12, and 18 bytes long, respectively, there is no record length less than 22 that will permit you to randomly access each string. If you select a record length of 10, for instance, you will be able to randomly access the first string but not the second and third.

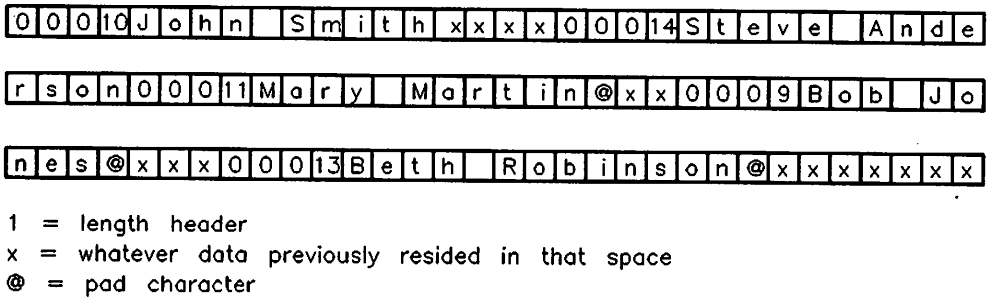
If you want to access strings randomly, therefore, you should make your records long enough to hold the largest string. Once you've done this, there are two ways to write string data to a BDAT file. The first, and easiest, is to output each string in random mode. In other words, select a record length that will hold the longest string and then write each string into its own record. Suppose, for example, that you wanted to OUTPUT the following 5 names into a BDAT file and be able to access each one individually by specifying a record number.

```
John Smith
Steve Anderson
Mary Martin
Bob Jones
Beth Robinson
```

The longest name, "Steve Anderson", is 14 characters. To store it in a BDAT file would require 18 bytes (four bytes for the length header). So you could create a file with record length of 18 and then OUTPUT each item into a different record:

```
100   CREATE BDAT "Names",5,18           ! Create a file.
110   ASSIGN @File TO "Names"            ! Open the file (FORMAT OFF).
120   OUTPUT @File,1;"John Smith"        ! Write names to
130   OUTPUT @File,2;"Steve Anderson"    !  successive records
140   OUTPUT @File,3;"Mary Martin"       !  in file.
150   OUTPUT @File,4;"Bob Jones"
160   OUTPUT @File,5;"Beth Robinson"
```

On the disc, the file Names would look like the figure below. The four-byte length headers show the decimal value of the bytes in the header. The data are shown in ASCII characters.



```
|0|0|0|10|J|o|h|n| |S|m|i|t|h|x|x|x|x|0|0|0|14|S|t|e|v|e| |A|n|d|e|

|r|s|o|n|0|0|0|11|M|a|r|y| |M|a|r|t|i|n|@|x|x|0|0|0|9|B|o|b| |J|o|

|n|e|s|@|x|x|x|0|0|0|13|B|e|t|h| |R|o|b|i|n|s|o|n|@|x|x|x|x|x|x|x|
```

1 = length header
x = whatever data previously resided in that space
@ = pad character

The unused portions of each record contain whatever data previously occupied that physical space on the disc.

Writing Data to BDAT, HP-UX and DOS Files

Data is always written to a file with an OUTPUT statement via an I/O path. You can OUTPUT numeric and string variables, numeric and string expressions, and numeric and string arrays. When you OUTPUT data with the FORMAT OFF, data items are written to the file in internal format (described earlier).

There is no limit to the number of data items you can write in a single OUTPUT statement, except that program statements are limited to two CRT lines. Also, if you try to OUTPUT more data than the file can hold, or the record can hold (if you are using random access), the system will return an EOF or EOR condition. If an EOF or EOR condition occurs, the file retains any data output before the end condition occurred.

There is also no restriction on mixing different types of data in a single OUTPUT statement. The system decides which data type each item is before it writes the item to the disc. Any item enclosed in quotes is a string. Numeric variables and expressions are OUTPUT according to their type (8 bytes for REAL values, and 2 bytes for INTEGER values). Arrays are written to the file in row-major order (right-most subscript varies quickest).

Each data item in an OUTPUT statement should be separated by either a comma or semi-colon (there is no operational difference between the two separators with FORMAT OFF). Punctuation at the end of an OUTPUT statement is ignored with FORMAT OFF.

Sequential (Serial) OUTPUT

Data is written serially to BDAT and HP-UX files whenever you do not specify a record number in an OUTPUT statement. When writing data serially, each data item is stored immediately after the previous item (with FORMAT OFF in effect, there are no separators between items). Sector and record boundaries are ignored. Data items are written to the file one by one, starting at the current position of the file pointer. As each item is written, the file pointer is moved to the byte following the last byte of the preceding item. After all of the data items have been OUTPUT, the file pointer points to the byte following the last byte just written.

There are a number of circumstances where it is faster and easier to use serial access instead of random access. The most obvious case is when you want to access the entire file sequentially. If, for example, you have a list of data items that you want to store in a file and you know that you will never want to read any of the items individually, you should write the data serially. The fastest way to write data serially is to place the data in an array and then OUTPUT the entire array at once.

Another situation where you might want to use serial access is if the file is so small that it can fit entirely into internal memory at once. In this case, even if you want to change individual items, it might be easier to treat the entire file as one or more arrays, manipulate as desired, and then write the entire array(s) back to the file.

Random OUTPUT

Random OUTPUT allows you to write to one record at a time. As with serial OUTPUT, there are EOF and file pointers that are updated after every OUTPUT. The EOF pointers follow the same rules as in serial access. The file pointer positioning is also the same, except that it is moved to the beginning of the specified record before the data is OUTPUT. If you wish to write randomly to a newly created file, start at the beginning of the file and write some "dummy" data into every record.

If you attempt to write more data to a record than the record will hold, the system will report an End-Of-Record (EOR) condition. An EOF condition will result if you try to write data more than one record past the EOF position. EOR conditions are treated by the system just like EOF conditions, except that they return Error 60 instead of 59. Data already written to the file before an EOR condition arises will remain intact.

Reading Data From BDAT, HP-UX and DOS Files

Data is read from files with the ENTER statement. As with OUTPUT, data is passed along an I/O path. You can use the same I/O path you used to OUTPUT the data or you can use a different I/O path.

You can have several variables in a single ENTER statement. Each variable must be separated from the other variables by either a comma or semi-colon. It is extremely important to make sure that your variable types agree with the data types in the file. If you wrote a REAL number to a file, you should ENTER it into a REAL variable; INTEGERs should be entered into INTEGER variables; and strings into string variables. The rule to remember is:

Read it the way you wrote it.

That is the *only* technique that is always guaranteed to work.

In addition to making sure that data types agree, it is also advisable to make sure that access modes agree. If you wrote data serially, you should read it serially; and if you wrote it randomly, you should read it randomly. There are a few exceptions to this rule which we discuss later. However, you should be aware that mixing access modes can lead to erroneous results unless you are aware of the precise mechanics of the file system.

Reading String Data From a File

When reading string data from a file, you must enter it into a string variable. How the system does this depends on file type and FORMAT attribute assigned to the file:

- With FORMAT OFF assigned to a BDAT file, the system reads and interprets the first four bytes after the file pointer as a length header. It will then try to ENTER as many characters as the length header indicates. If the string has been padded by the system to make its length even, the pad character is not read into the variable.
- With FORMAT OFF assigned to an HP-UX file, strings have no length header. Instead, they are assumed to be null-terminated; that is, entry into the string terminates when a null character, CHR$(0), is encountered.
- With FORMAT ON assigned to either type of file, the system reads and interprets the bytes as ASCII characters. The rules for item and ENTER-statement termination match those for devices (see the "Entering Data" chapter of *HP Instrument BASIC Interfacing Techniques* for details.)

After an ENTER statement has been executed, the file pointer is positioned to the next unread byte. If the last data item was a padded string (written to a BDAT file when using FORMAT OFF), the file pointer is positioned after the pad. If you use the same I/O path name to read and write data to a file, the file pointer will be updated after every ENTER and OUTPUT statement. If you use different I/O path names, each will have its own file pointer which is independent of the other. However, be aware that each also has its own EOF pointer and that these pointers may not match, which can cause problems.

Entering data does not affect the EOF pointers. If you attempt to read past an EOF pointer, the system will report an EOF condition.

Serial ENTER

When you read data serially, the system enters data into variables starting at the current position of the file pointer and proceeds, byte by byte, until all of the variables in the ENTER statement have been filled. If there is not enough data in the file to fill all of the variables, the system returns an EOF condition. All variables that have already taken values before the condition occurs retain their values.

The following program creates a BDAT file, assigns an I/O path name to the file (with default FORMAT OFF attribute), writes five data items serially, and then retrieves the data items.

```
10    CREATE BDAT "STORAGE",1    ! Could also be an HP-UX file.
20    ASSIGN @Path TO "STORAGE"
30    INTEGER Num,First,Fourth
40    Num=5
60    OUTPUT @Path;Num,"squared"," equals",Num*Num,"."
70    ASSIGN @Path TO "STORAGE"
80    ENTER @Path;First,Second$,Third$,Fourth,Fifth$
90    PRINT First;Second$;Third$,Fourth,Fifth$
100   END
```

```
5 squared equals 25.
```

Note that we re-ASSIGNed the I/O path in line 70. This was done to re-position the file pointer to the beginning of the file. If we had omitted this statement, the ENTER would have produced an EOF condition.

Random ENTER

When you ENTER data in random mode, the system starts reading data at the beginning of the specified record and continues reading until either all of the variables are filled or the system reaches the EOR or EOF. If the system comes to the end of the record before it has filled all of the variables, an EOR condition is returned.

In the following example, we randomly OUTPUT data to 5 successive records, and then ENTER the data into an array in reverse order.

```
10     CREATE BDAT "SQ_ROOTS",5,2*8
20     ASSIGN @Path TO "SQ_ROOTS"  ! Default is FORMAT OFF.
30     FOR Inc=1 to 5
40        OUTPUT @Path,Inc;Inc,SQR(Inc) ! Outputs two 8-byte REALs each time.
50     NEXT Inc
60     FOR Inc=5 TO 1 STEP -1
70        ENTER @Path,Inc;Num(Inc),Sqroot(Inc)
80     NEXT Inc
90     PRINT "Number","Square Root"
100    FOR Inc=1 TO 5
110       PRINT Num(Inc),Sqroot(Inc)
120    NEXT Inc
130    END
```

```
Number     Square Root
 1          1
 2          1.41421356237
 3          1.73205080757
 4          2
 5          2.2360679775
```

In this example, there was no need to re-ASSIGN the I/O path because the random ENTER automatically re-positions the file pointer.

Line 40 of the above program outputs two 8-byte REALs to the BDAT file called SQ_ROOTS. Note that this line would have to be changed for outputs made to HP-UX files because HP-UX files always have a record length of one. For example, the OUTPUT statement would look like this:

```
OUTPUT @Path,((Inc-1)*2*8)+1;Inc,SQR(Inc)
```

And the ENTER statement would look like this:

```
ENTER @Path,((Inc-1)*2*8)+1;Num(Inc),Sqroot(Inc)
```

Executing a random ENTER without a variable list has the effect of moving the file pointer to the beginning of the specified record. This is useful if you want to serially access some data in the middle of a file. Suppose, for instance, that you have a BDAT file containing 100 8-byte records, and each record has a REAL number in it. If you want to read the last 50 data items, you can position the file pointer to the 51st record and then serially read the remainder of the file into an array.

```
100     REAL Array(50)
110     ENTER @Realpath,51;     ! 51*8 is HP-UX record number.
120     ENTER @Realpath;Array(*)
```

Accessing Files with Single-Byte Records

With BDAT files, you can define records to be just one byte long (defined records in HP-UX files are always 1 byte long). In this case, it doesn't make sense to read or write one record at a time since even the shortest data type requires two bytes to store a number.

Random access to one-byte records, therefore, has its own set of rules. When you access a one-byte record, the file pointer is positioned to the specified byte. From there, the access proceeds in serial mode. Random OUTPUTs write as many bytes as the data item requires, and random ENTERs read enough bytes to fill the variable.

The example below illustrates how you can read and write randomly to one-byte records.

```
10      INTEGER Int
20      CREATE BDAT "BYTE",100,1
30      ASSIGN @Bytepath TO "BYTE"
40      OUTPUT @Bytepath,1;3.67
50      OUTPUT @Bytepath,9;3
60      OUTPUT @Bytepath,11;"string"
70      ENTER @Bytepath,9;Int
80      ENTER @Bytepath,1;Real
90      ENTER @Bytepath,11;Str$
100     PRINT Real
110     PRINT Int
120     PRINT Str$
130     END
```

```
 3.67
 3
 string
```

Note that we had to declare the variable `Int` as an INTEGER. If we hadn't, the system would have given it the default type of REAL and would therefore have required 8 bytes.

Accessing Directories

A directory is merely an index to the files on a mass storage media. The HP Instrument BASIC language has several features that allow you to obtain information from the directories of mass storage media. This section presents several techniques that will help you access this information.

To get a catalog listing of a directory, you will use the CAT statement. Executing CAT with no media specifier directs the system to get a catalog of the current system mass storage directory.

```
CAT
```

Including a media specifier directs the system to get a catalog of the specified mass storage. Here are some examples:

```
CAT ":HP9122,700"
CAT ":,700,0"
CAT "\BLP\PROJECTS"      DOS Volumes Only
CAT "/WORK/PROJECTS"     HFS Volumes Only
```

Both of the preceding statements sent the catalog listing to the current system printer (either specified by the last PRINTER IS statement, or defaulting to CRT).

Sending Catalogs to External Printers

The CAT statement normally directs its output to the current PRINTER IS device. The CAT statement can also direct the catalog to a specified device, as shown in the following examples:

```
CAT TO #726
CAT TO #External_prtr
CAT TO #Device_selector
```

The paramenter following the # is known as a device selector.

8

Using a Printer

Sooner or later a program needs to print something. A wide range of printers are supported by HP Instrument BASIC. This chapter covers the statements commonly used to communicate with external printers.

Selecting the System Printer

The PRINT statement normally directs text to the screen of the CRT where one is present on the instrument. Text may be re-directed to an external printer by using the PRINTER IS statement.

After the printer is switched on and the computer and printer have been connected via an interface cable, there is only one piece of information needed before printing can begin. The computer needs to know the correct **device selector** for the printer. This is analogous to knowing the correct telephone number before making a call.

Device Selectors

A device selector is a number that uniquely identifies a particular device connected to the computer. When only one device is allowed on a given interface, it is uniquely identified by the **interface select code.** In this case, the device selector is the same as the interface select code.

For example, the internal CRT is the only device at the interface whose select code is 1. To direct the output of PRINT statements to the CRT, use one of the following statements:

```
PRINTER IS 1
PRINTER IS CRT
```

These statements define the screen of the CRT to be the system printer. Until changed, the output of PRINT statements will appear on the screen of the CRT. (See your instrument-specific HP Instrument BASIC manual for information regarding the CRT display usage.)

Note In order to view data on the CRT of some host instruments running HP Instrument BASIC, you may need to allocate a display partition. Refer to your instrument-specific HP Instrument BASIC manual for information on display partitions.

When more than one device can be connected to an interface, such as the internal HP-IB interface (interface select code 7), the interface select code no longer uniquely identifies the printer. Extra information is required. This extra information is the **primary address.**

Using Device Selectors to Select Printers

A device selector is used by several different statements. In each of the following, the numeric expressions are device selectors.

`PRINTER IS 701`
`PRINTER IS PRT` Specifies a printer with interface select code 7 and primary address 01 (PRT is a numeric function whose value is always 701).

`PRINTER IS 1407` Specifies a printer with interface select code 14 and primary address 07.

`CAT TO #701` Prints a disc catalog on the printer at device selector 701.

`LIST #701` Lists the program in memory to a printer at 701.

Most statements allow a device selector to be assigned to a variable. Either INTEGER or REAL variables may be used.

```
PRINTER IS Hal
CAT TO #Dog
```

The following three-letter mnemonic functions have been assigned useful values.

Mnemonic	Value
PRT	701
KBD	2
CRT	1

The mnemonic may be used anywhere the numeric device selector can be used.

Another method may be used to identify the printer within a program. An I/O path name may be assigned to the printer; the printer is subsequently referenced by the I/O path name.

Using Control Characters and Escape Sequences

Most ASCII characters are printed on an external printer just as they appear on the screen of the CRT. Depending on your printer, there will be exceptions. Several printers will also support an alternate character set: either a foreign character set, a graphics character set, or an enhanced character set. If your printer supports an alternate character set, it usually is accessed by sending a special command to the printer.

Control Characters

In addition to a "printable" character set, printers usually respond to control characters. These non-printing characters produce a response from the printer. The following table shows some of the control characters and their effect.

Typical Printer Control Characters

Printer's Response	Control Character	ASCII Value
Ring printer's bell	CTRL-G	7
Backspace one character	CTRL-H	8
Horizontal tab	CTRL-I	9
Line-feed	CTRL-J	10
Form-feed	CTRL-L	12
Carriage-return	CTRL-M	13

One way to send control characters to the printer is the CHR$ function. Execute the following:

```
PRINT CHR$(12)
```

Refer to the appropriate printer manual for a complete listing of control characters and their effect on your printer.

Escape-Code Sequences

Similar in use to control characters, escape-code sequences allow additional control over most printers. These sequences consist of the escape character, CHR$(27), followed by one or more characters.

Since each printer may respond differently to control characters and escape code sequences, check the manual that came with your printer.

Formatted Printing

For many applications the PRINT statement provides adequate formatting. The simplest method of print formatting is by specifying a comma or semicolon between printed items.

When the comma is used to separate items, the printer will print the items on field boundaries. Fields start in column one and occur every ten columns (columns 1, 11, 21, 31, ...). Using the following values in a PRINT statement: A=1.1, B=-22.2, C=3E+5, D=5.1E+8.

```
10 PRINT RPT$("1234567890",4)
20 PRINT A,B,C,D
```

Produces:

```
1234567890123456789012345678901234567890
 1.1      -22.2      300000    5.1E+8
```

Note the form of numbers in a normal PRINT statement. A positive number has a leading and a trailing space printed with the number. A negative number uses the leading space position for the "–" sign. This is why the positive numbers in the previous example appear to print one column to the right of the field boundaries. The next example shows how this form prevents numeric values from running together.

```
10  PRINT RPT$("1234567890",4)
20  PRINT A;B;C;D

1234567890123456789012345678901234567890
 1.1 -22.2  300000  5.1E+8
```

Using the semicolon as the separator caused the numbers to be printed as closely together as the "compact" form allows. The compact form always uses one leading space (except when the number is negative) and one trailing space.

The comma and semicolon are often all that is needed to print a simple table. By using the ability of the PRINT statement to print the entire contents of of a array, the comma or semicolon can be used to format the output.

If each array element contained the value of its subscript, the statement:

```
PRINT Array(*);
```

Produces:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14...
```

Another method of aligning items is to use the tabbing ability of the PRINT statement.

```
PRINT TAB(25);-1.414

123456789012345678901234567890123
                        -1.414
```

While PRINT TAB works with an external printer, PRINT TABXY may not. PRINT TABXY may be used to specify both the horizontal and vertical position when printing to an internal CRT.

A more powerful formatting technique employs the ability of the PRINT statement to allow an image to specify the format.

Using Images

Just as a mold is used for a casting, an image can be used to format printing. An image specifies how the printed item should appear. The computer then attempts to print to item according to the image.

One way to specify an image is to include it in the PRINT statement. The **image specifier** is enclosed within quotes and consists of one or more **field specifiers.** A semicolon then separates the image from the items to be printed.

```
PRINT USING "D.DDD";PI
```

This statement prints the value of pi (3.141592659 ...) rounded to three digits to the right of the decimal point.

```
3.142
```

For each character "D" within the image, one digit is to be printed. Whenever the number contains more non-zero digits to the right of the decimal than provided by the field specifier, the last digit is rounded. If more precision is desired, more characters can be used within the image.

```
PRINT USING "D.10D";PI

3.1415926536
```

Instead of typing ten "D" specifiers, one for each digit, a shorter notation is to specify a repeat factor before each field specifier character. The image "DDDDDD" is the same as the image "6D".

The image specifier can be included in the PRINT statement or on it's own line. When the specifier is on a different line, the PRINT statement accesses the image by either the line number or the line label.

```
100   Format: IMAGE 6Z.DD
110   PRINT USING Format;A,B,C
120   PRINT USING 100;D,E,F
```

Both PRINT statements use the image in line 100.

Numeric Image Specifiers

Several characters may be used within an image to specify the appearance of the printed value.

Numeric Image Specifiers

Image Specifier	Purpose
D	Replace this specifier with one digit of the number to be printed. If the digit is a leading zero, print a space. if the value is negative, the position may be used by the negative sign.
Z	Same as "D" except that leading zeros are printed.
E	Prints two digits of the exponent after printing the sequence "E+". This specifier is equal to "ESZZ". See the *HP Instrument BASIC Language Reference* for more details.
K	Print the entire number without leading or trailing spaces.
S	Print the sign of the number: either a "+" or "-".
M	Print the sign if the number is negative; if positive, print a space.
.	Print the decimal point.
H	Similar to K, except the number is printed using the European number format (comma radix).
R	Print the comma (European radix).
*	Like Z, except that asterisks are printed instead of leading zeros.

To better understand the operation of the image specifiers examine the following examples and results.

Examples of Numeric Image Specifiers

Statement	Output
PRINT USING "K";33.666	33.666
PRINT USING "DD.DDD";33.666	33.666
PRINT USING "DDD.DD";33.666	33.67
PRINT USING "ZZZ.DD";33.666	033.67
PRINT USING "ZZZ";.444	000
PRINT USING "ZZZ";.555	001
PRINT USING "SD.3DE";6.023E+23	+6.023E+23
PRINT USING "S3D.3DE";6.023E+23	+602.300E+21
PRINT USING "S5D.3DE";6.023E+23	+60230.000E+19
PRINT USING "H";3121.55	3121,55
PRINT USING "DDRDD";19.95	19,95
PRINT USING "***";.555	**1

To specify multiple fields within the image, the field specifiers are separated by commas.

Multiple-Field Numeric Image Specifiers

Statement	Output
PRINT USING "K,5D,5D";100,200,300	100 200 300
PRINT USING "DD,ZZ,DD";1,2,3	102 3

If the items to be printed can use the same image, the image need be listed only once. The image will then be re-used for the subsequent items.

```
PRINT USING "5D.DD";3.98,5.95,27.50,139.95

123456789012345678901234567890123
    3.98    5.95   27.50  139.95
```

The image is re-used for each value. An error will result if the number cannot be accurately printed by the field specifier.

String Image Specifiers

Similar to the numeric field image characters, several characters are provided for the formatting of strings.

String Image Specifiers

Image Specifier	Purpose
A	Print one character of the string. If all characters of the string have been printed, print a trailing blank.
K	Print the entire string without leading or trailing blanks.
X	Print a space.
"literal"	Print the characters between the quotes.

The following examples show various ways to use string specifiers.

```
PRINT USING "5X,10A,2X,10A";"Tom","Smith"

12345678901234567890123456789
     Tom              Smith

PRINT USING "5X,""John"",2X,10A";"Smith"

12345678901234567890123456789
     John  Smith

PRINT USING """PART NUMBER"",2x,10d";90001234

12345678901234567890123456789
PART NUMBER    90001234
```

Additional Image Specifiers

The following image specifiers serve a special purpose.

Additional Image Specifiers

Image Specifier	Purpose
B	Print the corresponding ASCII character. This is similar to the CHR$ function.
#	Suppress automatic end-of-line (EOL) sequence.
L	Send the current end-of-line (EOL) sequence; with IO, see the PRINTER IS statement in the *HP Instrument BASIC Language Reference* for details on re-defining the EOL sequence.
/	Send a carriage-return and a linefeed.
@	Send a formfeed.
+	Send a carriage-return as the EOL sequence. (Requires IO)
−	Send a linefeed as the EOL sequence. (Requires IO)

For example:

`PRINT USING "@,#"` outputs a formfeed.

`PRINT USING "D,X,3A,""OR NOT"",X,B,X,B,B";2,"BE",50,66,69`

Special Considerations

If nothing prints, check if the printer is ON LINE. When the printer if OFF LINE the computer and printer can communicate but no printing will occur.

Sending text to a non-existent printer will cause the computer to wait indefinitely for the printer to respond. ON TIMEOUT may be used within a program to test for the printer.

Since the printer's device selector may change, keep track of the locations in the program where a device selector is used.

If the program must use the PRINTER IS statement frequently, assign the device selector to a variable; then if the device selector changes, only one program line will need to be changed.

9

Handling Errors

Most programs are subject to errors at run time. This chapter describes how HP Instrument BASIC programs can respond to these errors, and shows how to write programs that attempt to either correct the problem or direct the program user to take some sort of corrective action.

There are three courses of action that you may choose to take with respect to errors:

1. Try to prevent the error from happening in the first place.
2. Once an error occurs, try to recover from it and continue execution.
3. Do nothing—let the system stop the program when an error happens.

The remainder of this chapter describes how to implement the first two alternatives.

The last alternative, which may seem frivolous at first glance, is certainly the easiest to implement, and the nature of HP Instrument BASIC is such that this is often a feasible choice. Upon encountering a run-time error, the computer will pause program execution and display a message giving the error number and the line in which the error happened, and the programmer can then examine the program in light of this information and fix things up. The key word here is "programmer." If the person running the program is also the person who wrote the program, this approach works fine. If the person running the program did not write it, or worse yet, does not know how to program, some attempt should be made to prevent errors from happening in the first place, or to recover from errors and continue running.

Anticipating Operator Errors

When you write a program, you know exactly what the program is expected to do, and what kinds of inputs make sense for the problem. Sometimes you overlook the possibility that other people using the program might *not* understand the boundary conditions. You have no choice but to assume that every time a user has the opportunity to feed an input to a program, a mistake can be made and an error can be caused. You should make an effort to make the program resistant to errors.

Boundary Conditions

A classic example of anticipating an operator error is the "division by zero" situation. An INPUT statement is used to get the value for a variable, and the variable is used as a divisor later in the program. If the operator should happen to enter a zero, accidentally or intentionally, the program pauses with an error 31. It is far better to plan for such an occurrance. One method is shown in the following example.

```
100  INPUT "Miles traveled and total hours",Miles,Hours
110  IF Hours=0 THEN
120    BEEP
130    PRINT "Improper value entered for hours."
140    PRINT "Try again!"
150    GOTO 100
160  END IF
170  Mph=Miles/Hours
```

Trapping Errors

Despite the programmer's best efforts at screening the user's inputs in order to avoid errors, errors will still happen. It is still possible to recover from run-time errors, provided the programmer predicts the places where errors are most likely to happen.

ON/OFF ERROR

The ON ERROR statement sets up a branching condition which will be taken any time a recoverable error is encountered at run time. The branch action taken may be GOSUB, GOTO, CALL or RECOVER. GOTO and GOSUB are purely local in scope—that is, they are active only within the context in which the ON ERROR is declared. CALL and RECOVER are global in scope—after the ON ERROR is setup, the CALL or RECOVER will be executed any time an error occurs, regardless of subprogram environment.

Choosing a Branch Type

The type of branch that you choose (GOTO vs. GOSUB, etc.) depends on how you want to handle the error.

- Using GOSUB indicates that you want to return to the statement that caused the error (RETURN).
- GOTO, on the other hand, may indicate that you do not want to re-attempt the operation after attempting to correct the source of the error.

ON ERROR Execution at Run-Time

When an ON ERROR statement is executed, HP Instrument BASIC will make sure that the specified line or subprogram exists in memory before the program will proceed. If GOTO, GOSUB, or RECOVER is specified, then the *line identifier* must exist in the current context (at pre-run). If CALL is used, then the specified *subprogram* must currently be in memory (at run-time). In either case, if the system can't find the given line, error 49 is reported.

ON ERROR Priority

ON ERROR has a priority of 16, which means that it will *always* take priority over any other ON-event branch, since the highest user-specifiable priority is 15.

Disabling Error Trapping(OFF ERROR)

The OFF ERROR statement will cancel the effects of the ON ERROR statement, and no branching will take place if an error is encountered.

The DISABLE statement has no effect on ON ERROR branching.

ERRN, ERRLN, ERRL, ERRDS, ERRM$

ERRN is a function which returns the error number which caused the branch to be taken. ERRN is a global function, meaning it can be used from the main program or from any subprogram, and it will always return the number of the most recent error.

```
100  IF ERRN=80 THEN ! Media not present in drive.
110      PRINT "Please insert the 'Examples' disc,"
120      PRINT "and press the 'Continue' key (f2)."
130      PAUSE
140  END IF
```

ERRLN is a function which returns the *line number* of the program line in which the most recent error has occurred.

```
100  IF ERRLN<1280 THEN GOSUB During_init
110  IF (ERRLN>=1280 AND ERRLN<=2440) THEN GOSUB During_main
120  IF ERRLN>2440 THEN GOSUB During_Last
```

You can use this function, for instance, in determining what sort of situation-dependent action to take. As in the above example, you may want to take a certain action if the error occurred while "initializing" your program, another if during the "main" segment of your program, and yet another if during the "last" part of the program.

Note that program statements using ERRLN may not behave correctly following a renumber operation. To avoid this problem, use the ERRL function whenever possible (see below).

ERRL is another function which is used to find the line in which the error was encountered; however, the difference between this and the ERRLN function is that ERRL is a boolean function. The program gives it a line identifier, and either a 1 or a 0 is returned, depending upon whether or not the specified identifier indicates the line which caused the error.

```
100  IF ERRL(1250) OR ERRL(1270) THEN GOSUB Fix_12xx
110  IF ERRL(1470) THEN GOSUB Fix_1470
120  IF ERRL(2450) OR ERRL(2530) THEN GOSUB Fix_24xx
```

ERRL is a **local** function, which means it can only be used in the same environment as the line which caused the error. This implies that ERRL *cannot* be used in conjunction with ON ERROR CALL, but it *can* be used with ON ERROR GOTO and ON ERROR GOSUB. ERRL can be used with ON ERROR RECOVER only if the error did not occur in a subprogram which was called by the environment which set up the ON ERROR RECOVER.

Line number parameters to ERRL are renumbered properly by a renumber operation.

The ERRL function will accept either a *line number* or a *line label:*

```
1140  DISP ERRL(710)
```

```
910   IF ERRL(Compute) THEN Fix_compute
```

ERRM$ is a string function which returns the text of the error which caused the branch to be taken.

```
100 DISP ERRM$ ! Display default message.
```

```
ERROR 31 in 10  Division (or MOD) by zero
```

ON ERROR GOSUB

The ON ERROR GOSUB statement is used when you want to return to the program line where the error occurred.

Note that if you do not correct the problem and subsequently use RETURN, HP Instrument BASIC will repeatedly re-execute the problem-causing line (which will result in an infinite loop between the ON ERROR GOSUB branch and the RETURN).

When an error triggers a branch as a result of an ON ERROR GOSUB statement being active, system priority is set at the highest possible level (16) until the RETURN statement is executed, at which point the system priority is restored to the value it was when the error happened.

```
100   Radical=B*B-4*A*C
110   Imaginary=0
120   ON ERROR GOSUB Esr
130   Partial=SQRT(Radical)
140   OFF ERROR
      .
      .
      .
350 Esr: IF ERRN=30 THEN
360         Imaginary=1
370         Radical=ABS(Radical)
380      ELSE
390         BEEP
400         DISP "Unexpected Error (";ERRN;")"
410         PAUSE
420      END IF
430      RETURN
```

Note You cannot trap errors with ON ERROR while in an ON ERROR GOSUB service routine.

ON ERROR GOTO

The ON ERROR GOTO statement is often more useful than ON ERROR GOSUB, especially if you are trying to service more than one error condition. However, ON ERROR GOTO does not change system priority.

As with ON ERROR GOSUB, one error service routine can be used to service all the error conditions in a given context. By testing both the ERRN (what went wrong) and the ERRLN (where it went wrong) functions, you can take proper recovery actions.

One advantage of ON ERROR GOTO is that you can use another ON ERROR statement in the service routine (which you cannot use with ON ERROR GOSUB). This technique, however, requires that you re-establish the original error service routine after correcting any errors (by re-executing the original ON ERROR GOTO statement). The disadvantage is that more programming may be necessary in order to resume execution at the appropriate point after each error service.

ON ERROR CALL

ON ERROR CALL is global, meaning once it is activated, the specified subprogram will be called immediately whenever an error is encountered, *regardless of the current context.* System priority is set to level 17 inside the subprogram, and remains that way until the SUBEXIT is executed, at which time the system priority will be restored to the value it was when the error happened.

As with ON ERROR GOSUB, you will generally use the ON ERROR CALL statement when you want to return to the program where the error occurred.

Remember that if you do not correct the problem, the SUBEXIT statement will repeatedly re-execute the problem-causing line (which will result in an infinite loop between the ON ERROR CALL branch and the SUBEXIT).

Note You cannot trap errors with ON ERROR while in an ON ERROR CALL service routine.

Using ERRLN and ERRL in Subprograms

You can use the ERRLN function in any context, and it returns the line number of the most recent error. However, the ERRL function will not work in a different environment than the one in which the ON ERROR statement is declared. For instance, the following two statements will only work in the context in which the specified lines are defined:

```
100  IF ERRL(40) THEN GOTO Fix40
100  IF ERRL(Line_label) THEN Fix_line_label
```

The line identifier argument in ERRL will be modified properly when the program is renumbered (such as explicitly by REN or implicitly by GET); however, that is not true of expressions used in comparisons with the value returned by the ERRLN function.

So when using an ON ERROR CALL, you should set things up in such a manner that the line number either doesn't matter, or can be guaranteed to always be the same one when the error occurs. This can be accomplished by declaring the ON ERROR immediately before the line in question, and immediately using OFF ERROR after it.

```
5010  ON ERROR CALL Fix_disc
5020  ASSIGN @File TO "Data_file"
5030  OFF ERROR
      .
      .
      .
7020  SUB Fix_disc
7030  SELECT ERRN
7040  CASE 80
7050      DISP "No disc in drive -- insert disc and continue"
7060      PAUSE
7080  CASE 83
7090      DISP "Write protected -- fix and continue"
7100      PAUSE
7120  CASE 85
7130      DISP "Disc not initialized -- fix and continue"
7140      PAUSE
7160  CASE 56
7170      DISP "Creating Data_file"
7180      CREATE BDAT "Data_file",20
7190  CASE ELSE
7200      DISP "Unexpected error ";ERRN
7210      PAUSE
7220  SUBEND
```

ON ERROR RECOVER

The ON ERROR RECOVER statement sets up an immediate branch to the specified line whenever an error occurs. The line specified must be in the context of the ON ... RECOVER statement. ON ERROR RECOVER is global in scope—it is active not only in the environment in which it is defined, but also in any subprograms called by the segment in which it is defined.

If an error is encountered while an ON ERROR RECOVER statement is active, the system will restore the context of the program segment which actually set up the branch, including its system priority, and will resume execution at the given line.

```
      .
      .
      .
3250  ON ERROR RECOVER Give_up
3260  CALL Model_universe
3270  DISP "Successfully completed"
3280  STOP
3290 Give_up:  DISP "Failure ";ERRN
3300  END
      .
      .
      .
```

10

Keyword Guide to Porting

The following sections summarize the HP Instrument BASIC keywords by categories. All keywords are used by both HP Instrument BASIC and HP Series 200/300 BASIC languages, although some features of certain keywords are not supported by HP Instrument BASIC. Where differences exist between HP Instrument BASIC and recent versions of HP Series 200/300 BASIC the most significant discrepancies are listed. This chapter is intended only as a quick reference to the keywords and their compatibility. For detailed information, refer to *HP Instrument BASIC Keyword Reference* and your HP Series 200/300 BASIC Language Reference Manual.

Program	HP BASIC Function	HP Instrument BASIC
Entry/Editing		
LIST	Lists program lines to the system printer.	No support for softkey listing.
REM and !	Allows comments on program lines.	Full support.
Debugging		
ERRL	Indicates whether an error occurred during execution of a specified line.	No support for TRANSFER or Data Communications
ERRLN	Returns the program-line number of the most recent error.	No support for TRANSFER, Data Communications, CLEAR ERROR, or LOAD.
ERRM$	Returns the text of the last error message.	No support for TRANSFER, CLEAR ERROR, or LOAD.
ERRN	Return the most recent program execution error.	No support for TRANSFER, CLEAR ERROR, or softkeys.
Memory Allocation		
COM	Dimensions and reserves memory for variables in a common area for access by more than one context.	No support for OPTION BASE, BUFFER, COMPLEX, LOAD, or subarrays.
DIM	Dimensions and reserves memory for REAL numeric arrays and strings.	No support for OPTION BASE, BUFFER, COMPLEX, or subarrays.
INTEGER	Dimensions and reserves memory for INTEGER variables and arrays.	No support for OPTION BASE, BUFFER, or subarrays.
REAL	Dimensions and reserves memory for full-precision (REAL) variables and arrays.	No support for OPTION BASE, BUFFER, or subarrays.
Relational Operators		
=	Equality	Full Support.
<>	Inequality	Full Support.
<	Less than	Full Support.
<=	Less than or equal to	Full Support.
>	Greater than	Full Support.
>=	Greater than or equal to	Full Support.

Program	HP BASIC Function	HP Instrument BASIC
General Math		
+	Addition operator	Full Support.
–	Subtraction operator	Full Support.
×	Multiplication operator	Full Support.
/	Division operator	Full Support.
^	Exponentiation operator	Full Support.
ABS	Returns an argument's absolute value.	No support for COMPLEX.
DIV	Divides one argument by another and returns the integer portion of the quotient.	Full support.
DROUND	Returns the value of an expression, rounded to a specified number of digits.	Full support.
EXP	Raises the base e to a specified number of digits.	No support for COMPLEX.
FRACT	Returns the fractional portion of an expression.	Full support.
INT	Returns the integer portion of an expression.	Full support.
LET	Assigns values to variables.	Full support.
LGT	Returns the logarithm (base 10) of an argument	No support for COMPLEX.
LOG	Returns the natural logarithm (base e) of an argument	No support for COMPLEX.
MAX	Returns the largest value in a list of arguments	Full support.
MAXREAL	Returns the largest number available.	Full support.
MIN	Returns the smallest value in a list of arguments	Full support.
MINREAL	Returns the smallest number available.	Full support.
MOD	Returns the remainder of integer division.	Full support.
MODULO	Returns the modulo of division.	Full support.
PI	Returns an approximation of pi.	Full support.
PROUND	Returns the value of an expression, rounded to the specified power of ten.	Full support.

Program	HP BASIC Function	HP Instrument BASIC
General Math (continued)		
RANDOMIZE	Modifies the seed used by the RND function.	Full support.
RND	Returns a pseudo-random number.	Full support.
SGN	Returns the sign of an argument.	Full support.
SQRT (or SQR)	Returns the square root of an argument	No support for COMPLEX.
Binary Functions		
BINAND	Returns the bit-by-bit logical-and of two arguments.	Full support.
BINCMP	Returns the bit-by-bit complement of an argument.	Full support.
BINEOR	Returns the bit-by-bit exclusive-or of two arguments.	Full support.
BINIOR	Returns the bit-by-bit inclusive-or of two arguments.	Full support.
BIT	Returns the state of a specified bit of an argument.	Full support.
ROTATE	Returns a value obtained by shifting an argument's binary representation a number of bit positions, with wrap-around.	Full support.
SHIFT	Returns a value obtained by shifting an argument's binary representation a number of bit positions, without wrap-around.	Full support.
Trigonometric		
ACS	Returns the arccosine of an argument.	No support for COMPLEX.
ASN	Returns the arcsine of an argument.	No support for COMPLEX.
ATN	Returns the arctangent of an argument.	No support for COMPLEX.
COS	Returns the cosine of an argument.	No support for COMPLEX.
DEG	Sets the degrees mode.	Full support.
RAD	Sets the radians mode.	Full support.
SIN	Returns the sine of an argument.	No support for COMPLEX.
TAN	Returns the tangent of an argument.	No support for COMPLEX.

Program	HP BASIC Function	HP Instrument BASIC
String Operations		
&	Concatenates two string expressions	Full support.
CHR$	Converts a numeric value into an ASCII character.	Full support.
DVAL	Converts an alternate-base representation into a numeric value.	Full support.
DVAL$	Converts a numeric value into alternate-base representation.	Full support.
IVAL	Converts an alternate-base representation into an INTEGER number.	Full support.
IVAL$	Converts an INTEGER into alternate-base representation.	Full support.
LEN	Returns the number of characters in a string expression.	Full support.
LWC$	Returns the lowercase value of a string expression.	STANDARD lexical order is ASCII
NUM	Returns the decimal value of the first character in a string.	Full support.
POS	Returns the position of a string within a string expression.	Full support.
REV$	Reverses the order of the characters in a string expression.	Full support.
RPT$	Repeats the characters in a string expression a specified number of times.	Full support.
TRIM$	Removes the leading and trailing blanks from a string expression.	Full support.
UPC$	Returns the uppercase value of a string expression.	STANDARD lexical order is ASCII
VAL	Converts a string of numerals into a numeric value.	Full support.
VAL$	Returns a string expression representing a specified numeric value.	Full support.

Program	HP BASIC Function	HP Instrument BASIC
Logical Operators		
AND	Returns 1 or 0 based on the logical AND of two arguments.	Full support.
EXOR	Returns 1 or 0 based on the logical exclusive-or of two arguments.	Full support.
NOT	Returns 1 or 0 based on the logical complement of an argument.	Full support.
OR	Returns 1 or 0 based on the logical inclusive-or of two arguments.	Full support.
Mass Storage		
ASSIGN	Assigns an I/O path name and attributes to a file.	No support for BUFFER, BYTE, WORD, CONVERT, RETURN, PARITY, DELAY, and SRM. The device selector must be a single I/O device or mass storage file.
CAT	Lists the contents of the mass storage media's directory.	No support for SRM, NAMES, EXTEND, PROTECT, SELECT, SKIP, COUNT, NO HEADER, or PROG files.
COPY	Provides a method of copying mass storage files and volumes.	Full support.
CREATE	Creates an HP-UX or MS-DOS-type file on the mass storage media.	No support for SRM.
CREATE ASCII	Creates an ASCII-type file on the mass storage media.	No support for SRM.
CREATE BDAT	Creates an BDAT-type file on the mass storage media.	No support for SRM.
CREATE DIR	Creates an HFS or MS-DOS-type directory on the mass storage media.	No support for SRM.
GET	Reads an ASCII file into memory as a program.	No support for SRM.
INITIALIZE	Formats a mass storage media and places a LIF or DOS directory on the media.	No support for EPROM.
MASS STORAGE IS/ MSI	Specifies the default mass storage device.	No support for SRM, DCOMM, BUBBLE, or EPROM.
PURGE	Deletes a file entry from the directory.	No support for SRM.

Program	HP BASIC Function	HP Instrument BASIC
Mass Storage (continued)		
RENAME	Changes a file's name.	No support for SRM.
SAVE/ RE-SAVE	Creates an ASCII file and copies program lines from memory into the file.	No support for SRM.
Program Control		
CALL	Transfers program execution to a specified subprogram and passes parameters.	No support for CSUB subprograms, COMPLEX, or ON END.
DEF FN/ FNEND	Defines the bounds of a user-defined function subprogram.	No support for COMPLEX, BUFFER, NPAR, or OPTIONAL.
END	Terminates program execution and marks the end of the main program segment.	Full support.
FN	Invokes a user-defined function.	No support for COMPLEX.
FOR ... NEXT	Defines a loop which is repeated a specified number of times.	Full support.
GOSUB	Transfers program execution to a specified subroutine.	Full support.
GOTO	Transfers program execution to a specified line.	Full support.
IF ... THEN ELSE	Provides a conditional execution of a program segment.	Full support.
LOOP/ EXIT IF/ END LOOP	Provides looping with conditional exit.	Full support.
PAUSE	Suspends program execution.	No support for ON END or ON KNOB.
REPEAT ... UNTIL	Allows execution of a program segment until the specified condition is true.	Full support.
RETURN	Transfers program execution from a subroutine to the line following the invoking GOSUB.	Full support.
SELECT ... CASE	Allows execution of one program segment of several.	Full support.
STOP	Terminates execution of the program.	Full support.

Program	HP BASIC Function	HP Instrument BASIC
Program Control (continued)		
SUB/ SUBEND	Defines the bounds of a subprogram.	No support for COMPLEX, OPTIONAL or BUFFER.
SUBEXIT	Transfers control from within a subprogram to the calling context.	Full support.
WAIT	Causes program execution to wait a specified number of seconds.	Full support.
WHILE	Allows execution of a program segment while the specified condition is true.	Full support.
Event-Initiated Branching		
ENABLE/ DISABLE	Enables or disables even-initiated branching (except for ON ERROR, and ON TIMEOUT).	Full support.
ENABLE INTR/ DISABLE	Enables or disables interrupts defined by the ON INTR statement.	Bit mask value is ignored.
ON CYCLE/ OFF CYCLE	Enables or disables an event-initiated branch to be taken each time the specified number of seconds has elapsed.	Full support.
ON ERROR/ OFF ERROR	Sets up an event-initiated branch when a trappable program error occurs.	No support for CSUB.
ON INTR/ OFF INTR	Sets up an event-initiated branch when a specified interface cared generates an interrupt.	No support for CSUB.
ON KEY ... LABEL/ OFF KEY	Sets up an event-initiated branch when a specified softkey is pressed.	No support for CSUB, LINPUT, or ENTER KBD. Key selector range is 0-9.
ON TIMEOUT/ OFF TIMEOUT	Sets up an event-initiated branch when an I/O timeout occurs on a specified interface.	No support for CSUB, PRINTALL IS, PLOTTER IS, READIO, WRITEIO, or SRM.
SYSTEM PRIORITY	Sets a minimum level of system priority for event-initiated branches.	Full support.
Graphics Control		
GCLEAR	Clears the graphics area.	No support for external plotter or Multi-Plane displays.

Program	HP BASIC Function	HP Instrument BASIC
Graphics Plotting		
DRAW	Draws a line to a specified point.	No support for PIVOT.
MOVE	Updates the logical pen position.	No support for PIVOT.
PEN	Selects the pen number on plotting device.	Full support.
HP-IB Control		
ABORT	Terminates bus activity and asserts IFC.	Full support.
CLEAR	Places specified devices in a device-dependent state.	No support for Data Communications Interface.
LOCAL	Returns specified devices to their local state.	Full support.
LOCAL LOCKOUT	Sends the LLO message, disabling all device's front-panel controls.	Full support.
PASS CONTROL	Passes Active Controller capability to another device.	Full support.
REMOTE	Sets specified devices to their remote state.	Full support.
SPOLL	Returns a serial poll byte from a specified device.	Full support.
TRIGGER	Sends the trigger message to specified devices.	Full support.
Clock and Calendar		
TIMEDATE	Returns the value of the real-time clock.	Full support.
General Device Input/Output		
ASSIGN	Associates an I/O path name and attributes with a mass storage file, device or group of devices.	No support for SRM, BUFFER, BYTE, WORD, CONVERT, PARITY, TRANSFER, LOAD, or RETURN. I/O path name is limited to a single device or mass storage file.
BEEP	Produces an audible tone of a defined frequency and duration.	No support for HIL.
CRT	Returns the device selector of the CRT.	Full support.

Program	HP BASIC Function	HP Instrument BASIC
General Device Input/Output (continued)		
DATA	Specifies data accessible via READ statements.	Full support.
DISP	Outputs items to the CRT display line.	No support for COMPLEX.
ENTER	Inputs data from a device, file or string to a list of variables.	No support for COMPLEX, BUFFER, TRANSFER, CRT as source, or SRM.
IMAGE	Provides formats for use with ENTER, OUTPUT, DISP, and PRINT operations.	Full support.
INPUT	Inputs data from the front-panel (keyboard) to a list of variables.	No support for COMPLEX or specific keys.
KBD	Returns the device selector of the keyboard.	Full support.
OUTPUT	Outputs items to a specified device, file, string, or buffer.	No support for COMPLEX, BUFFER, TRANSFER, or SRM.
PRINT	Outputs items to the current PRINTER IS device.	No support for COMPLEX.
PRINTER IS	Specifies a device for PRINT, CAT, and LIST statements.	No support for DELAY, or SRM.
PRT	Returns 701, usually the device selector of external printer.	Full support.
READ	Inputs data from DATA lists to variables.	No support for COMPLEX.
RESTORE	Causes a READ statement to access the specified DATA statement.	Full support.
TAB	Moves the print position ahead to a specified point; used within PRINT and DISP statements.	Full support.
TABXY	Specifies the print position on the internal CRT; used with PRINT statements.	Full support.

Program	HP BASIC Function	HP Instrument BASIC
Display and Keyboard Control		
CLEAR SCREEN/ CLS	Clears the alpha display screen.	Full support.
CRT	Returns 1, which is the select code of the CRT display.	Full support.
KBD	Returns 2, which is the select code of the keyboard.	Full support.
Array Operations		
BASE	Returns the lower bound of a dimension of an array.	Full support.
RANK	Returns the number of dimensions in an array.	Full support.
SIZE	Returns the number of elements in a dimension of an array.	Full support.

Index

Contents

1

Manual Overview

Introduction

This manual presents the concepts of computer interfacing that are relevant to programming in HP Instrument BASIC. Note that not all features described in this manual may be implemented on your instrument. Please consult your instrument-specific manual for a description of implemented features. The topics presented herein will increase your understanding of interfacing the host instrument and external devices and computers with HP Instrument BASIC programs.

Manual Organization

This manual is organized by topics and is designed as a learning tool, not a reference. The text is arranged to focus your attention on interfacing concepts rather than to present only a serial list of the HP Instrument BASIC language I/O statements. Once you have read this manual and are familiar with the general and specific concepts involved, you can use either this manual or the *HP Instrument BASIC Language Reference* when searching for a particular detail of how a statement works.

This manual is designed for easy access by both experienced programmers and beginners.

Beginners may want to begin with Chapter 2, "Interfacing Concepts", before reading about general or interface-specific techniques.

Experienced programmers may decide to go directly to the chapter in your instrument-specific manual that describes the particular interface to be used. It is also usually helpful to become familiar with display and keyboard I/O operations, since these are helpful in seeing results while testing I/O programs.

If you need more background as you read about a particular topic, consult chapters 3 through 9 for a detailed explanation.

The brief descriptions in the next section will help you determine which chapters you will need to read for your particular application.

Chapter Previews

This manual is intended to provide background and tutorial information for programmers who have not written HP Instrument BASIC I/O programs before. It presents topics and programming techniques applicable to all interfaces.

Chapter 2: Interfacing Concepts

This chapter presents a brief explanation of relevant interfacing concepts and terminology. This discussion is especially useful for beginners as it covers much of the "why" and "how" of interfacing. Experienced programmers may also want to skim this material to better understand the terminology used in this manual.

Chapter 3: Directing Data Flow

This chapter describes how to specify which instrument resource is to send data to or receive data. The use of device selectors, string variable names, and "I/O path names" in I/O statements are described.

Chapter 4: Outputting Data

This chapter presents methods of outputting data to devices. All details of this process are discussed, and several examples of free-field output and output using images are given. Since this chapter completely describes outputting data to devices, you may only need to read the sections relevant to your application.

Chapter 5: Entering Data

This chapter presents methods of entering data from devices. All details of this process are discussed, and several examples of free-field enter and enter using images are given. As with Chapter 4, you may only need to read sections of this chapter relevant to your application.

Chapter 6: I/O Path Attributes

This chapter presents several powerful capabilities of the I/O path names provided by the BASIC language system. Interfacing to devices is compared to interfacing to mass storage files, and the benefits of using the same statements to access both types of resources are explained. This chapter is also highly recommended to all readers.

Specific Interfaces

Since each host instrument for HP Instrument BASIC implements the display, keyboard and other interfaces in slightly different manners, this manual does not cover the operation of interfaces. For specific details on the operation of interfaces with HP Instrument BASIC, consult the instrument-specific manual for your host instrument.

2

Interfacing Concepts

This chapter describes the functions and requirements of interfaces between the host instrument and its resources. Concepts in this chapter are presented in an informal manner. *All* levels of programmers can gain useful background information that will increase their understanding of the *why* and *how* of interfacing.

Terminology

These terms are important to your understanding of the text of this manual. The purpose of this section is to make sure that our terms have the same meanings.

computer is herein defined to be the processor, its support hardware, and the HP Instrument BASIC-language system of the *host instrument;* together these system elements *manage* all computer resources.

hardware describes both the electrical connections and electronic devices that make up the circuits within the computer; any piece of hardware is an actual physical device.

software describes the user-written, BASIC-language programs.

firmware refers to the pre-programmed, machine-language programs that are invoked by BASIC-language statements and commands. As the term implies, firmware is not usually modified by BASIC users. The machine-language routines of the operating system are firmware programs.

computer resource is herein used to describe all of the "data-handling" elements of the system. Computer resources include: internal memory, display, keyboard, and disc drive, and any external devices that are under computer control.

I/O is an acronym that comes from "Input and Output"; it refers to the process of copying data to or from computer memory.

output involves moving data from computer memory to another resource. During output, the **source** of data is computer memory and the **destination** is any resource, including memory.

input is moving data from a resource to computer memory; the source is any resource and the destination is a variable in computer memory. *Inputting data is also referred to as "entering data" in this manual* for the sake of avoiding confusion with the INPUT statement.

bus refers to a common group of hardware lines that are used to transmit information between computer resources. The computer communicates directly with the internal resources through the data and control buses.

computer backplane is an extension of these internal data and control buses. The computer communicates indirectly with the external devices through interfaces connected to the backplane hardware.

Why Do You Need an Interface?

The primary function of an interface is to provide a communication path for data and commands between the computer and its resources. Interfaces act as intermediaries between resources by handling part of the "bookkeeping" work, ensuring that this communication process flows smoothly. The following paragraphs explain the need for interfaces.

First, even though the computer bus is driven by electronic hardware that generates and receives electrical signals, this hardware was not designed to be connected directly to external devices. The internal hardware has been designed with specific electrical logic levels and drive capability in mind.

Second, you cannot be assured that the connectors of the computer and peripheral are compatible. In fact, there is a good probability that the connectors may not even mate properly, let alone that there is a one-to-one correspondence between each signal wire's function.

Third, assuming that the connectors and signals are compatible, you have no guarantee that the data sent will be interpreted properly by the receiving device. Some peripherals expect single-bit serial data while others expect data to be in 8-bit parallel form.

Fourth, there is no reason to believe that the computer and peripheral will be in agreement as to when the data transfer will occur; and when the transfer does begin the transfer rates will probably not match. As you can see, interfaces have a great responsibility to oversee the communication between computer and its resources.

Electrical and Mechanical Compatibility

Electrical compatibility must be ensured before any thought of connecting two devices occurs. Often the two devices have input and output signals that do not match; if so, the interface serves to match the electrical levels of these signals before the physical connections are made.

Mechanical compatibility simply means that the connector plugs must fit together properly. The interfaces connect with the computer buses. The peripheral end of the interfaces have connectors that match those on peripherals.

Data Compatibility

Just as two people must speak a common language, the computer and peripheral must agree upon the form and meaning of data before communicating it. As a programmer, one of the most difficult requirements to fulfill before exchanging data is that the format and meaning of the data being sent is identical to that anticipated by the receiving device. Even though some interfaces format data, most do not; most interfaces merely move data to or from computer memory. The computer must make the necessary changes, if any, so that the receiving device gets meaningful information.

Timing Compatibility

Since all devices do not have standard data-transfer rates, nor do they always agree as to when the transfer will take place, a consensus between sending and receiving device must be made. If the sender and receiver can agree on both the transfer rate and beginning point (in time), the process can be made readily.

If the data transfer is not begun at an agreed-upon point in time and at a known rate, the transfer must proceed one data item at a time with acknowledgement from the receiving device that it has the data and that the sender can transfer the next data item; this process is known as a "handshake". Both types of transfers are utilized with different interfaces and both will be fully described as necessary.

Additional Interface Functions

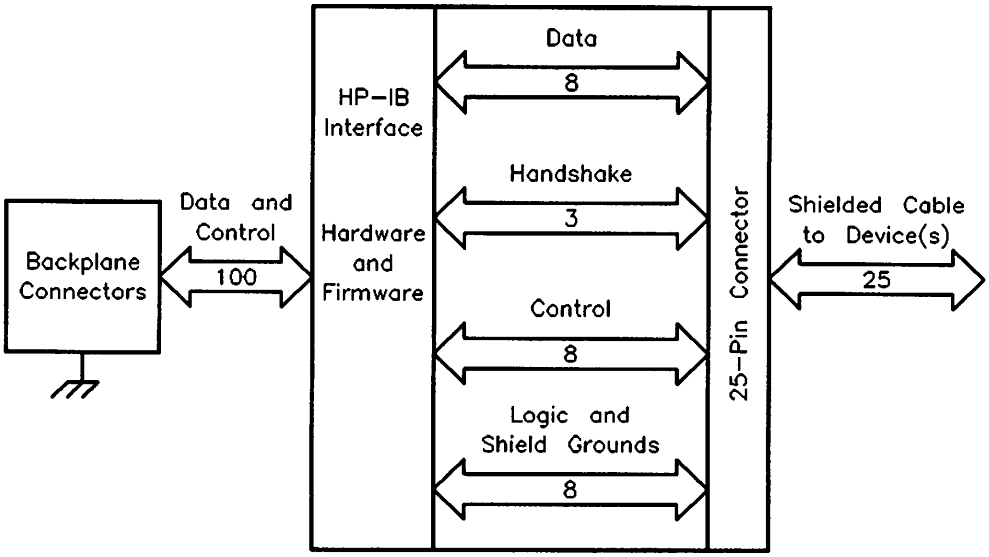
Another powerful feature of some interfaces is to relieve the computer of low-level tasks, such as performing data-transfer handshakes. This distribution of tasks eases some of the computer's burden and also decreases the otherwise-stringent response-time requirements of external devices. The actual tasks performed by each type of interface vary widely and are described in the next section of this chapter.

Interface Overview

Now that you see the need for interfaces, you should see what kinds of interfaces are available for the computer. Each of these interfaces is specifically designed for specific methods of data transfer; each interface's hardware configuration reflects its function.

The HP-IB Interface

This interface is Hewlett-Packard's implementation of the IEEE-488 1978 Standard Digital Interface for Programmable Instrumentation. The acronym "HP-IB" comes from Hewlett-Packard Interface Bus, often called the "bus".



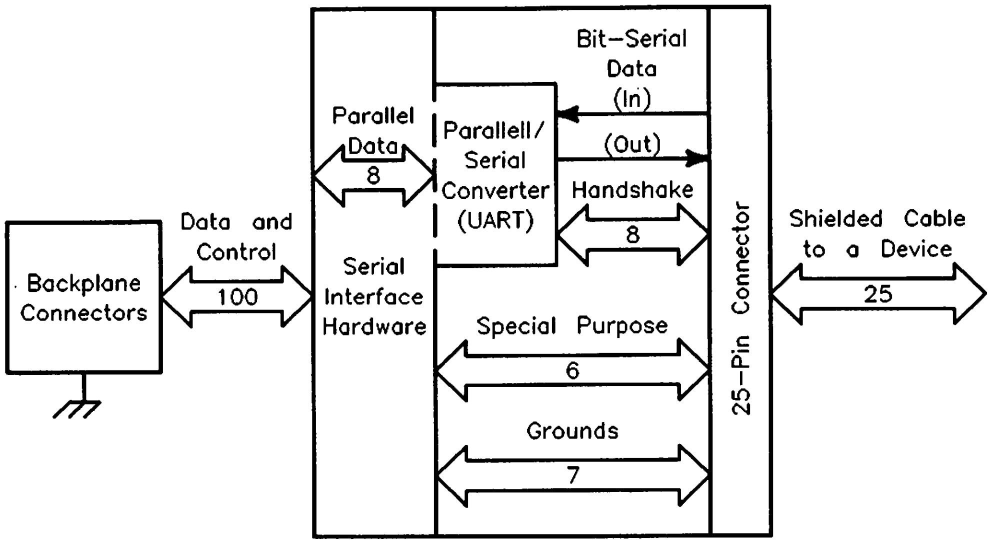
Block Diagram of the HP-IB Interface

The HP-IB interface fulfills all four compatibility requirements (hardware, electrical, data, and timing) with no additional modification. Just about all you need to do is connect the interface cable to the desired HP-IB device and begin programming. All resources connected to the computer through the HP-IB interface must adhere to this IEEE standard.

The "bus" is somewhat of an independent entity; it is a communication arbitrator that provides an organized protocol for communications between several devices. The bus can be configured in several ways. The devices on the bus can be configured to act as senders or receivers of data and control messages, depending on their capabilities.

The RS-232C Serial Interface

The serial interface changes 8-bit parallel data into bit-serial information and transmits the data through a two-wire (usually shielded) cable; data is received in this serial format and is converted back to parallel data. This use of two wires makes it more economical to transmit data over long distances than to use 8 individual lines.



Block Diagram of the Serial Interface

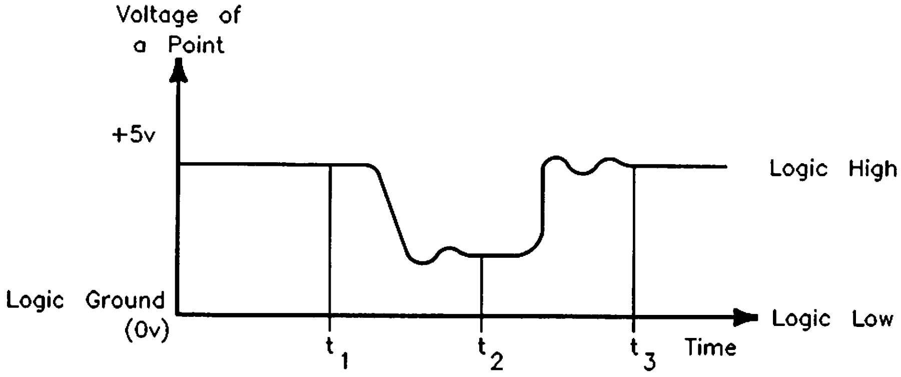
Data is transmitted at several programmable rates using either a simple data handshake or no handshake at all. The main use of this interface is in communicating with simple devices.

Data Representations

As long as data is only being used internally, it really makes little difference how it is represented; the computer always understands its own representations. However, when data is to be moved to or from an external resource, the data representation is of paramount importance.

Bits and Bytes

Computer memory is no more than a large collection of individual bits (*bi*nary digi*ts*), each of which can take on one of two logic levels (high or low). Depending on how the computer interprets these bits, they may mean on or not on (off), true or not true (false), one or zero, busy or not busy, or any other bi-state condition. These logic levels are actually voltage levels of hardware locations within the computer. The following diagram shows the voltage of a point versus time and relates the voltage levels to logic levels.



Voltage and Positive-True Logic

In some cases, you want to determine the state of an individual bit (of a variable in computer memory, for instance). The logical binary functions (BIT, BINCMP, BINIOR, BINEOR, BINAND, ROTATE, and SHIFT) provide access to the individual bits of data.

In most cases, these individual bits are not very useful by themselves, so the computer groups them into multiple-bit entities for the purpose of representing more complex data. Thus, all data in computer memory are somehow represented with binary numbers.

The computer's hardware accesses groups of sixteen bits at one time through the internal data bus; this size group is known as a **word.** With this size of bit group, 65 536 (65 $536=2^{16}$) different bit patterns can be produced. The computer can also use groups of eight bits at a time; this size group is known as a **byte.** With this smaller size of bit group, 256 ($256=2^8$) different patterns can be produced. How the computer and its resources interpret these combinations of ones and zeros is very important and gives the computer all of its utility.

Representing Numbers

The following binary weighting scheme is often used to represent numbers with a single data byte. Only the non-negative integers 0 through 255 can be represented with this particular scheme.

Most-Significant Bit   Least-Significant Bit

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	0	1	0	1	1	0
Value=128	Value=64	Value=32	Value=16	Value=8	Value=4	Value=2	Value=1

Notice that the value of a 1 in each bit position is equal to the power of two of that position. For example, a 1 in the 0th bit position has a value of 1 ($1=2^0$), a 1 in the 1st position has a value of 2 ($2=2^1$), and so forth. The number that the byte represents is then the total of all the individual bit's values.

$0 \times 2^0 = 0$
$1 \times 2^1 = 2$
$1 \times 2^2 = 4$ Number represented =
$0 \times 2^3 = 0$
$1 \times 2^4 = 16$ $2 + 4 + 16 + 128 = 150$
$0 \times 2^5 = 0$
$0 \times 2^6 = 0$
$1 \times 2^7 = 128$

The preceding representation is used by the "NUM" function when it interprets a byte of data. The next section explains why the character "A" can be represented by a single byte.

```
100  Number=NUM("A")
110  PRINT " Number = ";Number
120  END
```

```
Number = 65
```

Representing Characters

Data stored for humans is often alphanumeric-type data. Since less than 256 characters are commonly used for general communication, a single data byte can be used to represent a character. The most widely used character set is defined by the ASCII standard. ASCII stands for "American Standard Code for Information Interchange". This standard defines the correspondence between characters and bit patterns of individual bytes. Since this standard only defines 128 patterns (bit 7 = 0), 128 additional characters are defined by the computer (bit 7 = 1). The entire set of the 256 characters on the computer is hereafter called the "extended ASCII" character set.

When the CHR$ function is used to interpret a byte of data, its argument must be specified by its binary-weighted value. The single (extended ASCII) character returned corresponds to the bit pattern of the function's argument.

```
100   Number=65             ! Bit pattern is "01000001"
110   PRINT " Character is ";
120   PRINT CHR$(Number)
130   END
```

```
Printed Result: Character is A
```

The I/O Process

When using statements that move data between memory and internal computer resources, you do not usually need to be concerned with the details of the operations. However, you may have wondered how the computer moves the data. This section takes you "behind the scenes" of I/O operations to give you a better intuitive feel for how the computer outputs and enters data.

I/O Statements and Parameters

The I/O process begins when an I/O statement is encountered in a program. The computer first determines the type of I/O statement to be executed (such as, OUTPUT, ENTER USING, etc.). Once the type of statement is determined, the computer evaluates the statement's parameters.

Specifying a Resource

Each resource must have a unique specifier that allows it to be accessed to the exclusion of all other resources connected to the computer. The methods of uniquely specifying resources (output destinations and enter sources) are device selectors, string variable names, and I/O path names. These specifiers are further described in the next chapter.

For instance, before executing an OUTPUT statement, the computer first evaluates the parameter which specifies the destination resource. The source parameter of an ENTER statement is evaluated similarly.

```
OUTPUT Dest_parameter;Source_item

ENTER Sourc_parameter;Dest_item
```

Data Handshake

Each byte (or word) of data is transferred with a procedure known as a data-transfer handshake (or simply "handshake"). It is the means of moving one byte of data at a time when the two devices are not in agreement as to the rate of data transfer or as to what point in time the transfer will begin. The steps of the handshake are as follows.

1. The sender signals to get the receiver's attention.
2. The receiver acknowledges that it is ready.
3. A data byte (or word) is placed on the data bus.
4. The receiver acknowledges that it has gotten the data item and is now busy. No further data may be sent until the receiver is ready.

5. Repeat these steps if more data items are to be moved.

3

Directing Data Flow

Data can be moved between computer memory and several resources. These resources include:

- Computer memory
- Internal and external devices
- Mass storage files

This chapter describes in general terms how devices and string variables are specified in I/O statements. Each of these topics is covered in more detail in subsequent chapters. This chapter also describes the use of I/O pathnames in specifying devices for later use in I/O statements.

Specifying a Resource

Each resource must have a specifier that allows it to be accessed to the exclusion of all other computer resources. String variables are specified by variable name, while devices can be specified by either their device selector or a data type known as an I/O path name. This section describes how to specify these resources in OUTPUT and ENTER statements.

String-Variable Names

Data is moved to and from string variables by specifying the string variable's name in an OUTPUT or ENTER statement. Examples of each are shown below:

```
200  OUTPUT To_string$;Data_out$; ! ";" suppresses CR/LF.
240  ENTER From_string$;To_string$
```

Data is always copied to the destination string (or from the source string) beginning at the first position of the variable; subscripts cannot be used to specify any other beginning position within the variable.

Formatted String I/O

The use of outputting to and entering from string variables is a very powerful method of buffering data to be output to other resources. With OUTPUT and ENTER statements that use images, the data sent to the string variables can be explicitly formatted before being sent to (or while being received from) the variable.

Device Selectors

Devices include an internal CRT, keyboard, external printers and instruments, and all other physical entities that can be connected to the computer through an interface. Each interface has a unique number by which it is identified, known as its **interface select code.**

In order to send data to or receive data from a device, merely specify the select code of its interface in an OUTPUT or ENTER statement. Examples of using select codes to access devices are shown below.

```
OUTPUT 1;"Data to CRT"
ENTER CRT;Crt_line$

HPib_device=722
OUTPUT 722;"F1R1"
ENTER Hpib_device;Reading
```

The following pages explain select codes and device selectors.

Select Codes of Built-In Interfaces

The internal devices are accessed with the following, permanently assigned interface select codes.

Note Some host instruments may not contain all of the following interfaces.

Select Codes of Built-In Devices

Built-In Interface/Device	Permanent Select Code
Alpha Display	1
Keyboard	2
Built-in HP-IB interface	7
Built-in serial interface	9

The host instrument may have other built-in interfaces. See your instrument-specific HP Instrument BASIC manual for information regarding these interfaces and their select codes.

HP-IB Device Selectors

Each device on the HP-IB interface has a **primary address** by which it is uniquely identified; each address must be unique so that only one device is accessed when one address is specified. The device selector is then a combination of the interface select code and the device's address. Some examples are shown below.

HP-IB Device Selector Examples

Device Location	Device Selector	Example I/O Statement
interface select code 7, primary address 22	722	`OUTPUT 722;"Data" ENTER 722;Number`
interface select code 10, primary address 01	1001	`OUTPUT 1001;"Data" ENTER 1001;Number`

I/O Paths

All data entered and output via an interface to files or devices is moved through an "I/O Path." The I/O paths to devices and mass storage files can be assigned special names called **I/O path names.** I/O paths to strings cannot use I/O path names. The next section describes how to use I/O path names along with the benefits of using them.

I/O Path Names

An I/O path name is a data type that describes an I/O resource. With HP Instrument BASIC, you can assign I/O path names to either a device or a data file on a mass storage device. The following examples show how this is done.

Devices	`ASSIGN @Device TO 722`
Files	`ASSIGN @File TO "MyFile"`

Once assigned, the I/O path names can be used in place of the device selectors to specify the resource with which communication is to take place. For example:

`ASSIGN @Display TO 1`	Assigns the I/O path name @Display to the CRT.
`OUTPUT @Display;"Data"`	Sends characters to the display.
`ASSIGN @Printer TO 701`	Assigns @Printer to HP-IB device 701.
`OUTPUT @Printer;"Data"`	Sends characters to the printer.
`ASSIGN @Gpio TO 12`	Assigns @Gpio to the interface at select code 12.
`ENTER @Gpio;A_number`	Enters one numeric value from the interface.

Note HP Instrument BASIC does not support assigning an I/O path name to multiple devices.

Since an I/O path name is a data type, a fixed amount of memory is allocated for the variable, similar to the manner in which memory is allocated to other program variables (integer, real and string). This I/O path information is only accessible to the context in which it was allocated, unless it is passed as a parameter or appears in the proper COM statements.

Re-Assigning I/O Path Names

If an I/O path name already assigned to a resource is to be re-assigned to another resource, the preceding form of the ASSIGN statement is also used. The resultant action is that the the I/O path name to the device is implicitly closed. A new assignment is then made just as if the first assignment never existed.

```
100   ASSIGN @Printer TO 1    ! Initial assignment.
110   OUTPUT @Printer;"Data1"
120   !
130   ASSIGN @Printer TO 701  ! 2nd ASSIGN closes 1st
140   OUTPUT @Printer;"Data2" ! and makes a new assignment.
150   PAUSE
160   END
```

The result of running the program is that "Data1" is sent to the CRT, and "Data2" is sent to HP-IB device 701.

Closing I/O Path Names

A second use of the ASSIGN statement is to *explicitly close* the name assigned to an I/O path. For example, to close the path name @Printer you would use the following statement:

```
ASSIGN @Printer TO *
```

After executing this statement for a particular I/O path name, the name cannot be used in subsequent I/O statements until it is reassigned.

I/O Path Names in Subprograms

When a subprogram (either a SUB subprogram or a user-defined function) is called, the "context" is changed to that of the called subprogram. The statements in ht subprogram only have access to the data of the new context. Thus, in order to use an I/O path name in any statement within a subprogram, one of the following conditions must be true:

- The I/O path name must already be assigned within the context (i.e., the same instance of the subprogram)
- The I/O path name must be assigned in another context and passed to this context by reference (i.e., specified in both the formal-parameter and pass-parameter lists)
- The I/O path name must be declared in a variable common (with COM statements) and already be assigned within a context that has access to that common block

The following paragraphs and examples further describe using I/O path names in subprograms.

Assigning I/O Path Names Locally Within Subprograms

Any I/O path name can be used in a subprogram if it has first been assigned to an I/O path within the subprogram. A typical example is shown below.

```
10   CALL Subprogram_x
20   END
30   !
40   SUB Subprogram_x
50   ASSIGN @Log_device TO 1 ! CRT.
60   OUTPUT @Log_device;"Subprogram"
70   SUBEND
```

When the subprogram is exited, all I/O path names assigned locally within the subprogram are automatically closed. If the program (or subprogram) that called the exited subprogram attempts to use the I/O path name, an error results. An example of this closing local I/O path naems upon return from a subprogram is shown below.

```
10   CALL Subprogram_x
11   OUTPUT @Log_device;"Main" ! inserted line
20   END
30   !
40   SUB Subprogram_x
50   ASSIGN @Log_device TO 1 ! CRT.
60   OUTPUT @Log_device;"Subprogram"
70   SUBEND
```

When the above program is run, error 177, *Undefined I/O path name,* occurs in line 11.

Each context has its own set of local variables, which are not automatically accessible to any other context. Consequently, if the same I/O path name is assigned to I/O paths in separate contexts, the assignment local to the context is used while in that context. Upon return to the calling context, any I/O path names accessible to this context remain assigned as before the context was changed.

```
1    ASSIGN @Log_device to 701        ! Inserted line
2    OUTPUT @Log_device;"First Main"  ! Inserted line
10   CALL Subprogram_x
11   OUTPUT @Log_device;"Second Main" ! Changed line
20   END
30   !
40   SUB Subprogram_x
50   ASSIGN @Log_device TO 1 ! CRT.
60   OUTPUT @Log_device;"Subprogram"
70   SUBEND
```

The results of the above program are that the outputs "First Main" and "Second Main" are directed to device 701, while the output "Subprogram" is directed to the CRT. Notice that the original assignment of @Log_device made to interface select code 1 was local to the subprogram.

Passing I/O Names as Parameters

I/O path names can be used in subprograms if they are assigned and have been passed to the called subprogram by reference; they cannot be passed by value. The I/O path names(s) to be used must appear in both the pass-parameter and formal-parameter lists.

```
1    ASSIGN @Log_device to 701
2    OUTPUT @Log_device;"First Main"
10   CALL Subprogram_x(@Log_device)   ! Add pass parameter
11   OUTPUT @Log_device;"Second Main"
20   END
30   !
40   SUB Subprogram_x(@Log)           ! Add formal parameter
50   ASSIGN @Log TO 1 ! CRT.
60   OUTPUT @Log;"Subprogram"
70   SUBEND
```

Upon returning to the calling routine, any changes made to the assignment of the I/O path name passed by reference are maintained; the assignment local to the calling context is not restored as in the preceding example, since the I/O path name is accessible to both contexts. In this example, @Log_device remains assigned to interface select code 1; thus, "Subprogram" and "Second Main" are both directed to the CRT.

Declaring I/O Path Names in Common

An I/O path name can also be accessed by a subprogram if it has been declared in a COM statement (labeled or unlabeled) common to calling and called contexts, as shown in the following example.

```
1    COM @Log_device                  ! Insert COM statement
3    ASSIGN @Log_device to 701
4    OUTPUT @Log_device;"First Main"
10   CALL Subprogram_x                ! Parameters not necessary
11   OUTPUT @Log_device;"Second Main"
20   END
30   !
40   SUB Subprogram_x                 ! Parameters not necessary
41   COM @ Log_device                 ! Insert COM statement
50   ASSIGN @Log_device TO 1 ! CRT.
60   OUTPUT @Log_device;"Subprogram"
70   SUBEND
```

If an I/O path name is common is modified in any way, the assignment is changed for all subsequent contexts; the original assignement is not "restored" upon exiting the subprogram. In this example, "First Main" is sent to the HP-IB device 701, but "Subprogram" and "Second Main" are both directed to the CRT. This is identical to the preceding action when the I/O path name was passed by reference.

Benefits of Using I/O Path Names

Assigning names to I/O paths provide improvements in performance and additional capabilities over using device selectors. These advantages fall in the following areas:

- Execution speed
- Re-directing data to or from other destinations
- Access to mass storage files
- Attribute control

Execution Speed

When a device selector is used in an I/O statement to specify the I/O path to a device, first the numeric expression must be evaluated, then the corresponding attributes of the I/O path must be determined before the I/O path can be used. If an I/O path name is specified in an OUTPUT or ENTER statement, all of this information has already been determined at the time the I/O path name was assigned. Thus, an I/O statement containing an I/O path name executes slightly faster than using the corresponding I/O statement containing a device selector (for the same set of source-list expressions).

Re-Directing Data

Using numeric-variable device selectors, as with I/O path names, allows a single statement to be used to move data between the computer and several devices. Simple examples of re-directing data in this manner are shown in the following programs.

Example of Re-Directing with Device Selectors

```
100   Device=1
110   GOSUB Data_out
        .
        .
        .
200   Device=701
210   GOSUB Data_out
        .
        .
        .
410   Data_out: OUTPUT Device;Data$
420             RETURN
```

Example of Re-Directing with I/O Path Names

```
100   ASSIGN @Device TO 1
110   GOSUB Data_out
        .
        .
        .
200   ASSIGN @Device TO 9
210   GOSUB Data_out
        .
        .
        .
410   Data_out: OUTPUT @Device;Data$
420             RETURN
```

The preceding two methods of re-directing data execute in approximately the same amount of time.

Access to Mass Storage Files

The third advantage of using I/O path names is that device selectors cannot be used to direct data to or from mass storage files. Therefore, I/O path names are the only access to files. If the data is ever to be directed to a file, you must use I/O path names.

Attribute Control

I/O paths have certain "attributes" which control how the system handles data sent through the I/O path. For example, the FORMAT attribute possessed by an I/O path determines which data representation will be used by the path during communications. If the path possesses the attribute of FORMAT ON, the ASCII data representation will be used. This is the default attribute automatically assigned by the computer when I/O path names are assigned to device selectors. If the I/O path possesses the attribute of FORMAT OFF, the internal data representation is used; this is the default format for BDAT files. Further details of these and additional attributes are discussed in the "I/O Path Attributes" chapter.

The final factor that favors using I/O path names is that you can control which attribute(s) are to be assigned to the I/O path. Attributes can be attached to an I/O path name when it is assigned to a device (via the ASSIGN statement) and can specify data representation (ASCII or internal) as well as the end-of-line sequence for all data using the path. Details of these attributes are discussed in the "I/O Path Attributes" chapter.

4

Outputting Data

Introduction

This chapter describes the topic of outputting data to devices; outputting data to string variables, and mass storage files is described in the "I/O Path Attributes" chapter of this manual, in the "Data Storage and Retrieval" chapter of *HP Instrument BASIC Programming Techniques.*

There are two general types of output operations. The first type, known as "free-field outputs", use the HP Instrument BASIC's default data representations. The second type provides precise control over each character sent to a device by allowing you to specify the exact "image" of the ASCII data to be output.

Free-Field Outputs

Free-field outputs are invoked when the following types of OUTPUT statements are executed.

Examples

```
OUTPUT @Device;3.14*Radius^2

OUTPUT Printer;"String data";Num_1

OUTPUT 9;Test,Score,Student$

OUTPUT Escape_code$;CHR$(27)&"&A1S";
```

The Free-Field Convention

The term "free-field" refers to the number of characters used to represent a data item. During free-field outputs, HP Instrument BASIC does *not* send a *constant* number of ASCII characters for each type of data item, as is done during "fixed-field outputs" which use images (described later). Instead, a special set of rules is used that govern the number and type of characters sent for each source item. The rules used for determining the characters output for numeric and string data are described in the following paragraphs.

Standard Numeric Format

The default data representation for devices is to use ASCII characters to represent numbers. The ASCII representation of each expression in the source list is generated during free-field output operations. Even though all REAL numbers have 15 (and INTEGERs can have up to 5) significant decimal digits of accuracy, not all of these digits are output with free-field OUTPUT statements. Instead, the following rules of the free-field convention are used when generating a number's ASCII representation.

All numbers between 1E−5 and 1E+6 are rounded to 12 significant digits and output in floating-point notation with no leading zeros. If the number is positive, a leading space is output for the sign; if negative, a leading "−" is output.

For example:

```
 32767
-32768
 123456.789012
-.000123456789012
```

If the number is less than 1E−5 or greater than 1E+6, it is rounded to 12 significant digits and output in scientific notation. No leading zeros are output, and the sign character is a space for positive and "−" for negative numbers.

For example:

```
-1.23456789012E+6
 1.23456789012E-5
```

Standard String Format

No leading or trailing spaces are output with the string's characters.

```
String characters.
No leading or trailing spaces.
```

Item Separators and Terminators

Data items are output one byte at a time, beginning with the left-most item in the source list and continuing until all of the source items have been output. Items *in the list* must be *separated* by either a comma or a semicolon. However, items in the data output may or may not be separated by item terminators, depending on the use of item separators in the source lists.

The general sequence of items in the data output is as follows. The end-of-line (EOL) sequence is discussed in the next section.

1st item	item terminator	2nd item	item terminator	...	last item	EOL sequence

Using a *comma separator* after an item specifies that the **item terminator** (corresponding to the type of item) will be output after the last character of this item. A carriage-return, CHR$(13), and a line-feed, CHR$(10), terminate string items.

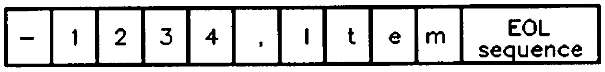
```
OUTPUT Device;"Item",-1234
```

I	t	e	m	CR	LF	−	1	2	3	4	EOL sequence

The default EOL sequence is a CR/LF

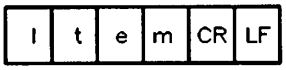
A comma separator specifies that a comma, CHR$(44), terminates numeric items.

```
OUTPUT Device;-1234,"Item"
```

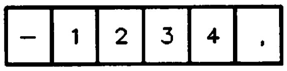


If a separator follows the last item in the list, the proper item terminator will be output *instead* of the EOL sequence.

```
OUTPUT Device;"Item",
```

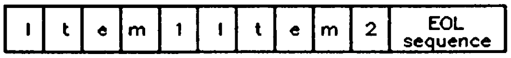


```
OUTPUT Device;-1234,
```

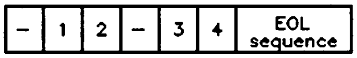


Using a *semicolon separator* suppresses output of the (otherwise automatic) item's terminator.

```
OUTPUT 1;"Item1";"Item2"
```

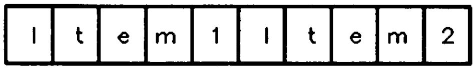


```
OUTPUT 1;-12;-34
```



If a semicolon separator follows the last item in the list, the EOL sequence and item terminators are suppressed.

```
OUTPUT 1;"Item1";"Item2";
```



Neither of the item teminators nor the EOL sequence are output.

If the item is an array, the separator following the array name determines what is output after each array element. (Individual elements are output in row-major order.)

```
110     DIM Array(1:2,1:3)
120     FOR Row=1 TO 2
130       FOR Column=1 TO 3
140         Array(Row,Column)=Row*10+Column
150       NEXT Column
160     NEXT Row
170     !
180     OUTPUT CRT;Array(*)  ! No trailing separator.
190     !
200     OUTPUT CRT;Array(*), ! Trailing comma.
210     !
220     OUTPUT CRT;Array(*); ! Trailing semi-colon.
230     !
240     OUTPUT CRT;"Done"
250     END
```

Resultant Output

```
 11, 12, 13, 21, 22, 23 EOL sequence
 11, 12, 13, 21, 22, 23,
 11 12 13 21 22 23
DONE EOL sequence
```

Item separators cause similar action for string arrays.

```
110     DIM Array$(1:2,1:3)[2]
120     FOR Row=1 TO 2
130       FOR Column=1 TO 3
140         Array$(Row,Column)=VAL$(Row*10+Column)
150       NEXT Column
160     NEXT Row
170     !
180     OUTPUT CRT;Array$(*)  ! No trailing separator.
190     !
200     OUTPUT CRT;Array$(*), ! Trailing comma.
210     !
220     OUTPUT CRT;Array$(*); ! Trailing semi-colon.
230     !
240     OUTPUT CRT;"Done"
250     END
```

Resultant Output

1	1	CR	LF	1	2	CR	LF	1	3	CR	LF	2	1	CR	LF	2	2	CR	LF	2	3	EOL sequence
1	1	CR	LF	1	2	CR	LF	1	3	CR	LF	2	1	CR	LF	2	2	CR	LF	2	3	EOL sequence
1	1	1	2	1	3	2	1	2	2	2	3											
D	O	N	E	EOL sequence																		

Changing the EOL Sequence

An end-of-line (EOL) sequence is normally sent following the last item sent with OUTPUT. The default EOL sequence consists of a carriage-return and line-feed (CR/LF), sent with no device-dependent END indication. It is also possible to define your own special EOL sequences that include sending special characters, and sending an END indication.

In order to define non-default EOL sequences to be sent by the OUTPUT statement, an I/O path must be used. The EOL sequence is specified in one of the ASSIGN statements which describe the I/O path. An example is as follows.

```
ASSIGN @Device TO 7;EOL CHR$(10)&CHR$(10)&CHR$(13)
```

The characters following EOL are the new EOL-sequence characters. Any character in the range CHR$(0) through CHR$(255) may be included in the string expression that defines the EOL characters; however, the length of the sequence is limited to eight characters or less.

If END is included in the EOL attribute, an interface-dependent "END" indication is sent with (or after) the last character of the EOL sequence. However, if no EOL sequence is sent, the END indication is also suppressed. The following statement shows an example of defining the EOL sequence to include an END indication.

```
ASSIGN @Device TO 7;EOL CHR$(13)&CHR$(10) END
```

With the HP-IB Interface, the END indication is an End-or-Identify message (EOI) sent with the last EOL character.

The default EOL sequence is a CR and LF sent with no END indication; this default can be restored by assigning EOL OFF to the I/O path.

EOL sequences can also be sent by using the "L" image specifier. See "Outputs that Use Images" for further details.

Using END in Freefield OUTPUT

The secondary keyword END may be optionally specified following the last source-item expression in a freefield OUTPUT statement. The result is to *suppress the End-of-Line (EOL) sequence* that would otherwise be output after the last byte of the last source item. If a comma is used to separate the last item from the END keyword, the corresponding item terminator will be output as before (carriage-return and line-feed for string items and comma for numeric items).

The END keyword has additional significance when the destination is a mass storage file. See the "Data Storage and Retrieval" chapter of *HP Instrument BASIC Programming Techniques* for further details.

Additional Definition

HP Instrument BASIC defines additional action when END is specified in a freefield OUTPUT statement directed to the HP-IB interface.

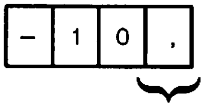
END with HP-IB Interfaces

With HP-IB interfaces, END has the additional function of sending the End-or-Identify signal (EOI) with the last data byte of the last source item; however, *if no data are sent from the last source item, EOI is not sent.*

Examples

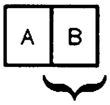
```
ASSIGN @Device TO 701
```

```
OUTPUT @Device;-10,END
```



EOI sent with the last character
(numeric item terminator).

```
OUTPUT @Device;"AB";END
```



EOI sent with the last character of the item.

```
OUTPUT @Device;END
```

```
OUTPUT @Device;""END
```

Neither EOL sequence nor EOI is sent, since no data is sent.

Outputs that Use Images

The free-field form of the OUTPUT statement is very convenient to use. However, there may be times when the data output by the free-field convention is not compatible with the data required by the receiving device.

Several instances for which you might need to format outputs are: special control characters are to be output; the EOL sequence (carriage-return and line-feed) needs to be suppressed; or the exponent of a number must have only one digit. This section shows you how to use image specifiers to create your own, unique data representations for output operations.

The OUTPUT USING Statement

When this form of the OUTPUT statement is used, the data is output according to the format image referenced by the "USING" secondary keyword. This image consists of one or more individual image specifiers which describe the type and number of data bytes (or words) to be output. The image can be either a string literal, a string variable, or the line label or number of an IMAGE statement. Examples of these four possibilities are listed below.

```
100  OUTPUT 1 USING "6A,SDDD.DDD,3X";" K= ",123.45

100  Image_str$="6A,SDDD.DDD,3X"
110  OUTPUT CRT USING Image_str$;" K= ";123.45

100  OUTPUT CRT USING Image_stmt;" K= ";123.45
110  Image_stmt: IMAGE 6A,SDDD.DDD,3X

100  OUTPUT 1 USING 110;" K= ";123.45
110  IMAGE 6A,SDDD.DDD,3X
```

Images

Images are used to specify the format of data during I/O operations. Each image consists of groups of individual image (or "field") specifiers, such as 6A, SDDD.DDD, and 3X in the preceding examples. Each of these field specifiers describe one of the following things:

- It describes the desired format of one item in the source list. (For instance, 6A specifies that a string item is to be output in a "6-character Alpha" field. SDDD.DDD specifies that a numeric item is to be output with Sign, 3 Decimal digits preceding the decimal point, followed by 3 Decimal digits following the decimal point.)
- It specifies that special character(s) are to be output. (For instance, 3X specifies that 3 spaces are to be output.) There is no corresponding item in the source list.

Thus, you can think of the image list as either a precise format description or as a procedure. It is convenient to talk about the image list as a procedure for the purpose of explaining how this type of OUTPUT statement is executed.

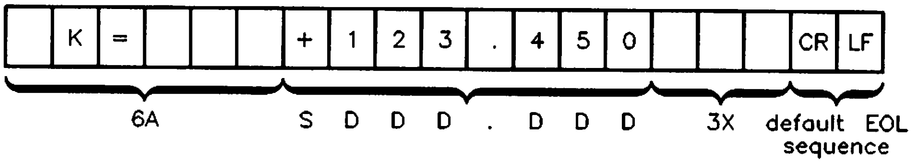
Again, each image list consists of images that each describe the format of data item to be output. The order of images in the list corresponds to the order of data items in the source list. In addition, image specifiers can be added to output (or to suppress the output of) certain characters.

Example of Using an Image

We will use the first of the four, equivalent output statements shown above. Don't worry if you don't understand each of the image specifiers used in the image list; each will be fully described in subsequent sections of this chapter. The main emphasis of this example is that you will see how an image list is used to govern the type and number of characters output.

```
OUTPUT CRT USING "6A,SDDD.DDD,3X";" K= ",123.45
```

The data stream output by the computer is as follows.



Step 1. The computer evaluates the first image in the list. Generally, each group of specifiers separated by commas is an "image"; the commas tell the computer that the image is complete and that it can be "processed". In general, each group of specifiers is processed before going on to the next group. In this case, 6 alphanumeric characters taken from the first item in the source list are to be output.

Step 2. The computer then evaluates the first item in the source list and begins outputting it, one byte (or word) at a time. After the 4th character, the first expression has been "exhausted". In order to satisfy the corresponding specifier, two spaces (alphanumeric "fill" characters) are output.

Step 3. The computer evaluates the next image (note that this image consists of several different image specifiers). The "S" specifier requires that a sign character be output for the number, the "D" specifiers require digits of a number, and the "." specifies where the decimal point will be placed. Thus, the number of digits following the decimal point have been specified. All of these specifiers describe the format of the next item in the source list.

Step 4. The next data item in the source list is evaluated. The resultant number is output one digit at a time, according to its image specifiers. A trailing zero has been added to the number to satisfy the "DDD" specifiers following the decimal point.

Step 5. The next image in the list ("3X")is evaluated. This specifier does not "require" data, so the source list needs no corresponding expression. Three spaces are output by this image.

Step 6. Since the entire image list and source list have been "exhausted", the computer then outputs the current (or default, if none has been specified) "end-of-line" sequence of characters (here we assume that a carriage-return and line-feed are the current EOL sequence).

The execution of the statement is now complete. As you can see, the data specified in the source list must match those specified in the output image in type and in number of items.

Image Definitions During Outputs

This section describes the definitions of each of the image specifiers when referenced by OUTPUT statements. The specifiers have been categorized by data type. It is suggested that you scan through the description of each specifier and then look over the examples. You are also highly encouraged to experiment with the use of these concepts.

Numeric Images

These image specifiers are used to describe the format of numbers.

Sign, Digit, Radix and Exponent Specifiers

Image Specifier	Meaning
S	Specifies a "+" for positive and a "–" for negative numbers is to be output.
M	Specifies a leading space for positive and a "–" for negative numbers is to be output.
D	Specifies one ASCII digit ("0" through "9") is to to be output. Leading spaces and trailing zeros are used as fill characters. The sign character, if any, "floats" to the immediate left of the most-significant digit. If the number is negative and no S or M is used, one digit specifier will be used for the sign.
Z	Same as "D" except that leading zeros are output. This specifier cannot appear to the right of a radix specifier (decimal point or R).
*	Like D, except that asterisks are output as leading fill characters (instead of spaces). This specifier cannot appear to the right of a radix specifier (decimal point or R).
.	Specifies the position of a decimal point radix-indicator (American radix) within a number. There can be only one radix indicator per numeric image item.
R	Specifies the position of a comma radix indicator (European radix) within a number. There can be only one radix indicator per numeric image item.
E	Specifies that the number is to be output using scientific notation. The "E" must be preceded by at least one digit specifier (D, Z, or *). The default exponent is a four-character sequence consisting of an "E", the exponent sign, and two exponent digits, equivalent to an "ESZZ" image. Since the number is left-justified in the specified digit field, the image for a negative number must contain a sign specifier (see the next section).
ESZ	Same as "E" but only 1 exponent digit is output.
ESZZZ	Same as "E" but three exponent digits are output.
K, –K	Specifies that the number is to be output in a "compact" format, similar to the standard numeric format; however, neither leading spaces (that would otherwise replace a "+" sign) nor item terminators (commas) are output, as would be with the standard numeric format.
H, –H	Like K, except that the number is to be output using a comma radix (European radix).

Numeric Examples

```
OUTPUT @Device USING "DDDD";-123.769
```

–	1	2	4	EOL sequence

```
OUTPUT @Device USING "4D";-1.2
```

–	1	EOL sequence

```
OUTPUT @Device USING "ZZ.DD";1.675
```

0	1	.	6	8	EOL sequence

```
OUTPUT @Device USING "Z.D";.35
```

0	.	4	EOL sequence

```
OUTPUT @Device USING "DD.E";12345
```

1	2	.	E	+	0	3	EOL sequence

```
OUTPUT @Device USING "2D.DDE";2E-4
```

2	0	.	0	0	E	–	0	5	EOL sequence

```
OUTPUT @Device USING "K";12.400
```

1	2	.	4	EOL sequence

```
OUTPUT CRT USING "MDD.2D";-12.449
```

–	1	2	.	4	5	EOL sequence

```
OUTPUT CRT USING "MDD.DD";2.09
```

		2	.	0	9	EOL sequence

```
OUTPUT 1 USING "SD.D";2.449
```

+	2	.	4	EOL sequence

```
OUTPUT 1 USING "SZ.DD";.49
```

+	0	.	4	9	EOL sequence

```
OUTPUT CRT USING "SDD.DDE";-2.35
```

–	2	3	.	5	0	E	–	0	1	EOL sequence

```
OUTPUT @Device USING "**.D";2.6
```

*	2	.	6	EOL sequence

```
OUTPUT @Device USING "DRDD";3.1416
```

3	,	1	4	EOL sequence

```
OUTPUT @Device USING "H";3.1416
```

3	,	1	4	1	6	EOL sequence

String Images

These types of image specifiers are used to specify the format of string data items.

Character Specifiers

Image Specifier	Meaning
A	Specifies that one character is to be output. Trailing spaces are used as fill characters if the string contains less than the number of characters specified.
"literal"	All characters placed in quotes form a string literal, which is output exactly as is. Literals can be placed in output images which are part of OUTPUT statements by enclosing them in double quotes.
K, −K, H, −H	Specifies that the string is to be output in "compact" format, similar to the standard string format; however, no item terminators are output as with the standard string format.

String Examples

```
OUTPUT @Device USING "8A";"Characters"
```

C	h	a	r	a	c	t	e	EOL sequence

```
OUTPUT @Device USING "K,""Literal""";"AB"
```

A	B	L	i	t	e	r	a	l	EOL sequence

```
OUTPUT @Device USING "K";"   Hello   "
```

			H	e	l	l	o				EOL sequence

```
OUTPUT @Device USING "5A";"  Hello  "
```

			H	e	EOL sequence

Binary Images

These image specifiers are used to output bytes (8-bit data) and words (16-bit data) to the destination. Typical uses are to output non-ASCII characters or integers in their internal representation.

Binary Specifiers

Image Specifier	Meaning
B	Specifies that one byte (8 bits) of data is to be output. The source expression is evaluated, rounded to an integer, and interpreted MOD 256. If it is less than −32 768, CHR$(0) is output. If is greater than 32 767, CHR$(255) is output.
W	Specifies that one word of data (16 bits) are to be sent as a 16-bit, two's-complement integer. The corresponding source expression is evaluated and rounded to an integer. If it is less than −32 768, then −32 768 is sent; if it is greater than 32 767, then 32 767 is sent. If the destination is a BDAT or HPUX file, or string variable, the WORD attribute is ignored and all data are sent as bytes; however, pad byte(s), CHR$(0), will also be output whenever necessary to achieve alignment on a word boundary. Since HP Instrument BASIC only supports 8-bit interfaces, two bytes are always output, with the most significant byte first. This image specifier has been included primarily to maintain compatibility with HP Series 200/300 BASIC programs that include this specifier.
Y	Like W, except that no pad bytes are output to achieve alignment on a word boundary.

Binary Examples

```
OUTPUT @Device USING "B,B,B";65,66,67
```

A	B	C	EOL sequence

```
OUTPUT @Device USING "B";13
```

CR

```
OUTPUT @Device USING "W";256*65+66
```

A	B	EOL sequence

Special-Character Images

These specifiers require no corresponding data in the source list. They can be used to output spaces, end-of-line sequences, and form-feed characters.

Special-Character Specifiers

Image Specifier	Meaning
X	Specifies that a space character, CHR$(32), is to be output.
/	Specifies that a carriage-return character, CHR$(13), and a line-feed character, CHR$(10), are to be output.
@	Specifies that a form-feed character, CHR$(12), is to be output.

Special-Character Examples

```
OUTPUT @Device USING "A,4X,A";"M","A"
```

M					A	EOL sequence

```
OUTPUT @Device USING "50X"
```

← (50 spaces) →	EOL sequence

```
OUTPUT @Device USING "@,/"
```

FF	CR	LF	EOL sequence

```
OUTPUT @Device USING "/"
```

CR	LF	EOL sequence

Termination Images

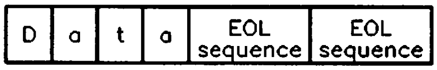
These specifiers are used to output or suppress the end-of-line sequence output after the last data item.

Termination Specifiers

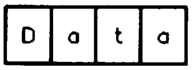
Image Specifier	Meaning
L	Specifies that the current end-of-line sequence is to be output. The default EOL characters are CR and LF; see "Changing the EOL Sequence" for details on how to re-define these characters.
#	Specifies that the EOL sequence that normally follows the last item is to be suppressed.
%	Is ignored in output images but is allowed to be compatible with ENTER images.
+	Specifies that the EOL sequence that normally follows the last item is to be replaced by a single carriage-return character (CR).
–	Specifies that the EOL sequence that normally follows the last item is to be replaced by a single line-feed character (LF).

Termination Examples

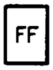
```
OUTPUT @Device USING "4A,L";"Data"
```



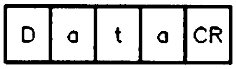
```
OUTPUT @Device USING "#,K";"Data"
```



```
OUTPUT @Device USING "#,B";12
```



```
OUTPUT @Device USING "+,K";"Data"
```



```
OUTPUT @Device USING "-,L,K";"Data"
```

EOL sequence	D	a	t	a	LF

Additional Image Features

Several additional features of outputs which use images are available with the computer. Several of these features, which have already been shown, will be explained here in detail.

Repeat Factors

Many of the specifiers can be repeated without having to explicitly list the specifier as many times as it is to be repeated. For instance, to a character field of 15 characters, you do not need to use “AAAAAAAAAAAAAAA”; instead, you merely specify the number of times that the specifier is to be repeated in front of the image (“15A”). The following specifiers can be repeated by specifying an integer repeat factor; the specifiers not listed cannot be repeated in this manner.

Repeatable Specifiers	Non-Repeatable Specifiers
Z, D, A, X, /, @, L	S, M, ., R, E, K, H, B, W, Y, #, %, +, -

Examples

```
OUTPUT @Device USING "4Z.3D";328.03
```

0	3	2	8	.	0	3	0	EOL sequence

```
OUTPUT @Device USING "6A";"Data bytes"
```

D	a	t	a		b	EOL sequence

```
OUTPUT @Device USING "5X,2A";"Data"
```

					D	a	EOL sequence

OUTPUT @Device USING "2L,4A";"Data"

EOL sequence	EOL sequence	D	a	t	a	EOL sequence

OUTPUT @Device USING "8A,2@";"The End"

T	h	e		E	n	d		FF	FF	EOL sequence

OUTPUT @Device USING "2/"

CR	LF	CR	LF	EOL sequence

Image Re-Use

If the number of items in the source list exceeds the number of matching specifiers in the image list, the computer attempts to re-use the image(s) beginning with the first image.

```
110   ASSIGN @Device TO CRT
120   Num_1=1
130   Num_2=2
140   !
150   OUTPUT @Device USING "K";Num_1,"Data_1",Num_2,"Data_2"
160   OUTPUT @Device USING "K,/";Num_1,"Data_1",Num_2,"Data_2"
170   END
```

Resultant Display

```
1Data_12Data_2
1
Data_1
2
Data_2
```

Since the "K" specifier can be used with both numeric and string data, the above OUTPUT statements can re-use the image list for all items in the source list. If any item cannot be output using the corresponding image item, an error results. In the following example, "Error 100 in 150" occurs due to data mismatch.

```
110   ASSIGN @Device TO CRT
120   Num_1=1
130   Num_2=2
140   !
150   OUTPUT @Device USING "DD.DD";Num_1,Num_2,"Data_1"
160   END
```

Nested Images

Another convenient capability of images is that they can be nested within parentheses. The entire image list within the parentheses will be used the number of times specified by the repeat factor preceding the first parenthesis. The following program is an example of this feature.

```
100   ASSIGN @Device TO 701
110   !
120   OUTPUT @Device USING "3(B),X,DD,X,DD";65,66,67,68,69
130   END
```

Resultant Output

A	B	C		6	8		6	9	EOL sequence

This nesting with parentheses is made with the same hierarchy as with parenthetical nesting within mathematical expressions. Only eight levels of nesting are allowed.

END with OUTPUTs that Use Images

Using the optional secondary keyword END in an OUTPUT statement that uses an image produces results which differ from those of using END in a freefield OUTPUT statement. Instead of always suppressing the EOL sequence, the END keyword *only suppresses the EOL sequence when no data are output from the last source-list expression.* Thus, the "#" image specifier generally controls the suppression of the otherwise automatic EOL sequence, while the END keyword suppresses it only in less common usages.

Examples

```
Device=12

OUTPUT Device USING "K";"ABC",END
OUTPUT Device USING "K";"ABC";END
OUTPUT Device USING "K";"ABC" END
```

A	B	C	EOL sequence

The EOL sequence is not suppressed.

```
OUTPUT Device USING "L,/,""Literal"",X,@"
```

EOL sequence	CR	LF	L	i	t	e	r	a	l		FF	EOL sequence

In this case, specifiers that require no source-item expressions are used to generate characters for the output; there are no source expressions. The EOL sequence is output after all specifiers have been used to output their respective characters. Compare this action to that shown in the next example.

```
OUTPUT Device USING "L,/,""Literal"",X,@";END
```

EOL sequence	CR	LF	L	i	t	e	r	a	l		FF

The EOL sequence is suppressed because no source items were included in the statement; all characters output were the result of specifiers which require no corresponding expression in the source list.

Additional END Definition

The END secondary keyword has been defined to produce additional action when included in an OUTPUT statement directed to HP-IB interfaces.

END with HP-IB Interfaces

With HP-IB interfaces, END has the additional function of sending the End-or-Identify signal (EOI) with the *last character* of either the last source item or the EOL sequence (if sent). As with freefield OUTPUT, *no EOI is sent if no data is sent from the last source item and the EOL sequence is suppressed.*

Examples.

```
ASSIGN @Device TO 701

OUTPUT @Device USING "K";"Data",END
OUTPUT @Device USING "K";"Data","",END
```

D	a	t	a	EOL sequence

EOI sent with last character
of the EOL sequence.

```
OUTPUT @Device USING "#,K";"Data" END
```

D	a	t	a

EOI sent with this character.

EOI is sent with the last character of the last source item when the EOL sequence is suppressed, because the last source item contained data which was used in the output.

```
OUTPUT @Device USING "#,K";"Data","",END
OUTPUT @Device USING """Data""";END
```

D	a	t	a

The EOI was not sent in either case, since no data were sent from the last source item *and* the EOL sequence was suppressed.

5

Entering Data

This chapter discusses the topic of entering data from devices. You may already be familiar with the OUTPUT statement described in the previous chapter; many of those concepts are applicable to the process of entering data. Earlier in this manual, you were told that *the data output from the sender had to match that expected by the receiver.* Because of the many ways that data is represented in external devices, entering data can sometimes require more programming skill than outputting data. In this chapter, you will see what is involved in being the receiving device. Both free-field enters and enters that use images are described, and several examples are given with each topic.

Free-Field Enters

Executing the free-field form of the ENTER invokes conventions which are the "converse" of those used with the free-field OUTPUT statement. In other words, data output using the free-field form of the OUTPUT statement can be readily entered using the free-field ENTER statement; no explicit image specifiers are required. The following statements exemplify this form of the ENTER statement.

For example:

```
ENTER @Voltmeter;Reading
ENTER 724;Readings(*)
ENTER From_string$;Average,Student_name$
ENTER @From_file;Data_code,Str_element$(X,Y)
```

Item Separators

Destination items in ENTER statements can be separated by *either* a comma or a semicolon. Unlike the OUTPUT statement, it makes *no difference* which is used; data will be entered into each destination item in a manner independent of the punctuation separating the variables in the list. However, *no trailing punctuation is allowed.* The first two of the following statements are equivalent, but an error is reported when the third statement is executed.

For example:

```
ENTER @From_a_device;N1,N2,N3
ENTER @From_a_device;N1;N2;N3
```

Item Terminators

Unless the receiver knows exactly how many characters are to be sent, each data item output by the sender must be terminated by special character(s). When entering ASCII data with the free-field form of the ENTER statement, the computer does not know how many characters will be output by the sender.

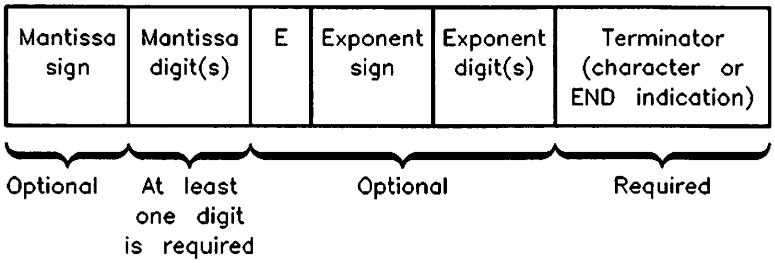
Item terminators must signal the end of each item so that the computer enters data into the proper destination variable. The terminator of the last item may also terminate the ENTER statement (in some cases). The actual character(s) that terminate entry into each type of variable are described in the next sections.

In addition to the termination characters, each item can be terminated (only with selected interfaces) by a device-dependent END indication. For instance, some interfaces use a signal known as EOI (End-or-Identify). The EOI signal is only available with the HP-IB, and keyboard interfaces. EOI termination is further described in the next sections.

Entering Numeric Data with the Number Builder

When the free-field form of the ENTER statement is used, numbers are entered by a routine known as the "number builder". This firmware routine evaluates the incoming ASCII numeric characters and then "builds" the appropriate internal-representation number. This number builder routine recognizes whether data being entered is to be placed into an INTEGER or REAL variable and then generates the appropriate internal representation.

The number builder is designed to be able to enter several formats of numeric data. However, the general format of numeric data must be as follows to be interpreted properly by HP Instrument BASIC.

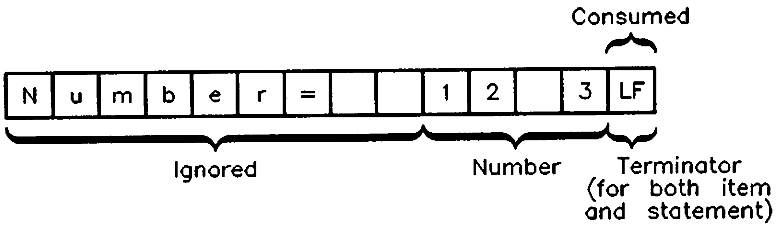


Numeric characters include decimal digits "0" through "9" and the characters ".", "+", "–", "E", and "e". These last five characters must occur in meaningful positions in the data stream to be considered numeric characters; if any of them occurs in a position in which it cannot be considered part of the number, it will be treated as a non-numeric character.

The following rules are used by the number builder to construct numbers from incoming streams of ASCII numeric characters.

1. All leading non-numerics are ignored; all leading and embedded spaces are ignored.

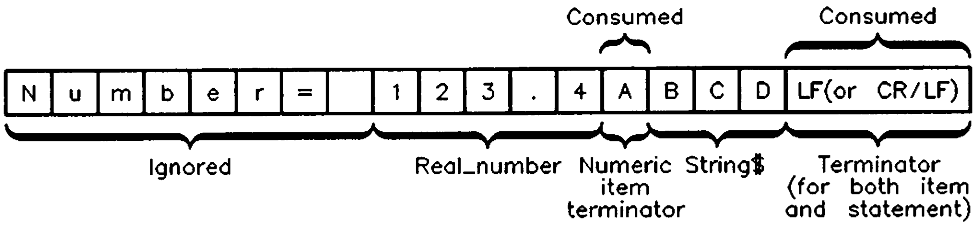
```
100    ASSIGN @Device TO Device_selector
110    ENTER @Device;Number   ! Default is data type REAL.
120    END
```



The result of entering the preceding data with the given ENTER statement is that Number receives a value of 123. The line-feed (statement terminator) is *required* since Number is the last item in the destination list.

2. Trailing non-numerics terminate entry into a numeric variable, and the terminating characters (of *both* string and numeric items) are "consumed". In this manual, "consumed" characters refers to characters *used to terminate* an item but not entered into the variable; "ignored" characters are entered but are *not used.*

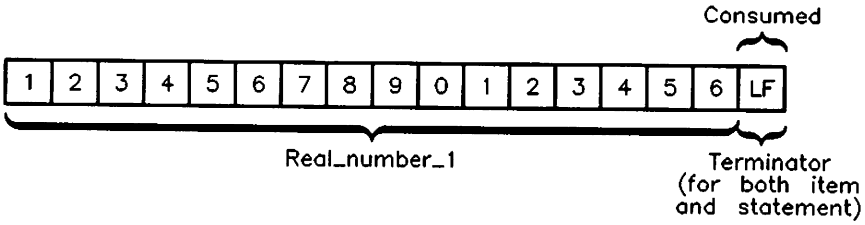
```
ENTER @Device;Real_number,String$
```



The result of entering the preceding data with the given ENTER statement is that Real_number receives the value 123.4 and String$ receives the characters "BCD". The "A" was lost when it terminated the numeric item; the string-item terminator(s) are also lost. The string-item terminator(s) also terminate the ENTER statement, since String$ is the last item in the destination list.

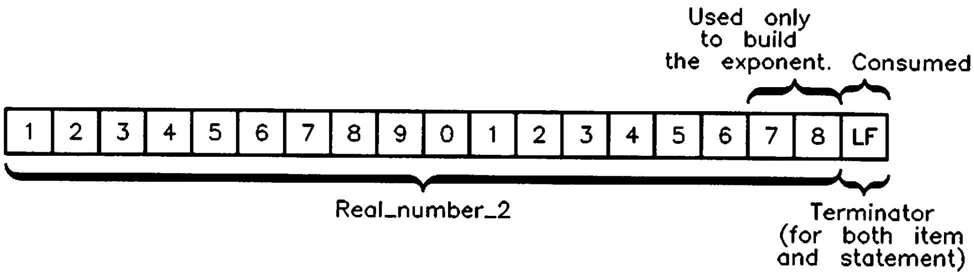
3. If more than 16 digits are received, only the first 16 are used as significant digits. However, all additional digits are treated as trailing zeros so that the exponent is built correctly.

```
ENTER @Device;Real_number_1
```



The result of entering the preceding data with the given ENTER statement is that `Real_number_1` receives the value `1.234567890123456 E+15`.

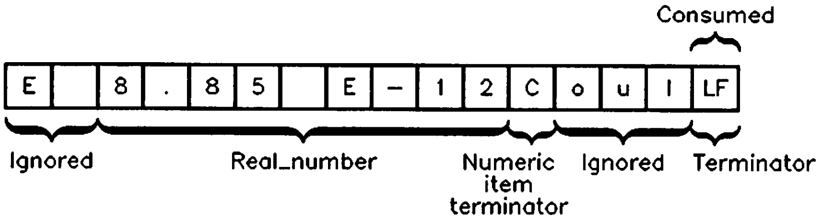
```
ENTER @Device;Real_number_2
```



The result of entering the preceding data with the given ENTER statement is that `Real_number_2` receives the value `1.234567890123456 E+17`.

4. Any exponent sent by the source must be preceded by at least one mantissa digit *and* an E(or e) character. If no exponent digits follow the E (or e), no exponent is recognized, but the number is built accordingly.

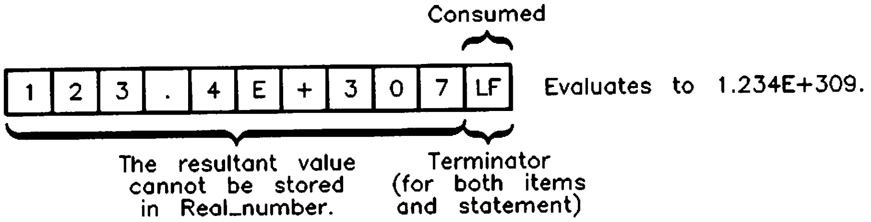
```
ENTER @Device;Real_number
```



The result of entering the preceding data with the given ENTER statement is that Real_number receives a value of 8.85 E–12. The character "C" terminates entry into Real_number, and the characters "oul" are entered (but ignored) in search of the required line-feed statement terminator. If the character "C" is to be entered but not ignored, you must use an image. Using images with the ENTER statement is described later in this chapter.

5. If a number evaluates to a value outside the range corresponding to the type of the numeric variable, an error is reported. If no type has been declared explicitly for the numeric variable, it is assumed to be REAL.

```
ENTER @Device;Real_number
```


Consumed
1
2
3
.
4
E
+
3
0
7
LF
Evaluates to 1.234E+309.
The resultant value cannot be stored in Real_number.
Terminator (for both items and statement)

The data is entered but evaluates to a number outside the range of REAL numbers. Consequently, error 19 is reported, and the variable Real_number retains its former value.

6. If the item is the *last* one in the list, *both* the *item* and the *statement* need to be properly *terminated.* If the numeric **item** is terminated by a non-numeric character, the **statement** will *not* be terminated until it either receives a line-feed character or an END indication (such as EOI signal with a character). The topic of terminating free-field ENTER statements is described later.

Entering String Data

Strings are groups of ASCII characters of varying lengths. Unlike numbers, almost any character can appear in any position within a string; there is not really any defined structure of string data. The routine used to enter string data is therefore much simpler than the number builder. It only needs to keep track of the dimensioned length of the string variable and look for string-item terminators (such as CR/LF, LF, or EOI sent with a character).

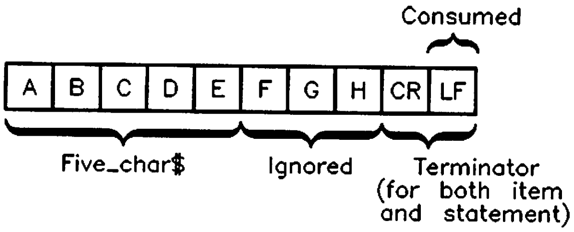
String-item terminator characters are either a line-feed (LF) or a carriage-return followed by a line-feed (CR/LF). As with numeric-item terminators characters, these characters are not entered into the string variable (during free-field enters); they are "lost" when they terminate the entry. The EOI signal also terminates entry into a string variable, but the variable must be the last item in the destination list (during free-field enters).

All characters received from the source are entered directly iemph appropriate string variable until *any* of the following conditions occurs:

- an item terminator character is received.
- the number of characters entered equals the dimensioned length of the string variable.
- the EOI signal is received.

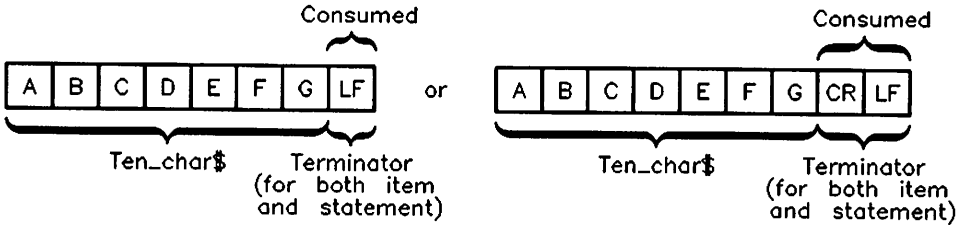
The following statements and resultant variable contents illustrate the first two conditions; the next section describes termination by EOI. Assume that the string variables Five_char$ and Ten_char$ are dimensioned to lengths of 5 and 10 characters, respectively.

```
ENTER @Device;Five_char$
```



The variable Five_char$ only receives the characters "ABCDE", but the characters "FGH" are entered (and ignored) in search of the terminating carriage-return/line-feed (or line-feed).

```
ENTER @Device;Ten_char$
```



The result of entering the preceding data with the given ENTER statement is that Ten_char$ receives the characters "ABCDEFG" and the terminating LF (or CR/LF) is lost.

Terminating Free-Field ENTER Statements

Terminating conditions for free-field ENTER statements are as follows.

1. If the *last item* is terminated by a line-feed or by a character accompanied by EOI, the *entire statement* is properly terminated.
2. If an *END indication* is received while entering data into the *last item,* the statement is properly terminated. Examples of END indications are encountering the last character of a string variable while entering data from the variable, and receiving EOI with a character.
3. If one of the preceding *statement-termination* conditions has *not* occurred *but* entry into the *last item* has been terminated,up to 256 *additional* characters are entered in search of a termination condition. If one is not found, an error occurs.

One case in which this termination condition may not be obvious can occur while entering string data. If the last variable in the destination list is a string *and* the dimensioned length string has been reached *before* a terminator is received, additional characters are entered (but ignored) until the terminator is found. The reason for this action is that the next characters received are still part of this data item, as far as the data *sender* is concerned. These characters are accepted from the sender so that the next enter operation will not receive these "leftover" characters.

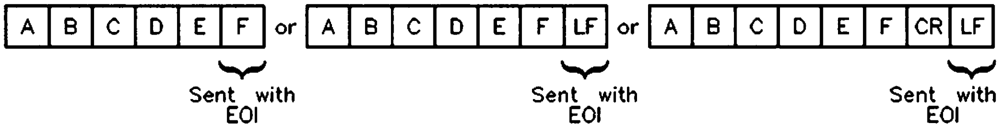
Another case involving numeric data can also occur (see the example given with "rule 4" describing the number builder). If a trailing non-numeric character terminates the last item (which is a numeric variable), additional characters will be entered in search of either a line-feed or a character accompanied by EOI. Unless this terminating condition is found before 256 characters have been entered, an error is reported.

EOI Termination

A termination condition for the HP-IB Interface is the EOI (End-or-Identify) signal. When this message is sent, it immediately terminates the entire ENTER statement, regardless of whether or not all variables have been satisfied. However, if all variable items in the destination list have not been satisfied, an error is reported.

For example:

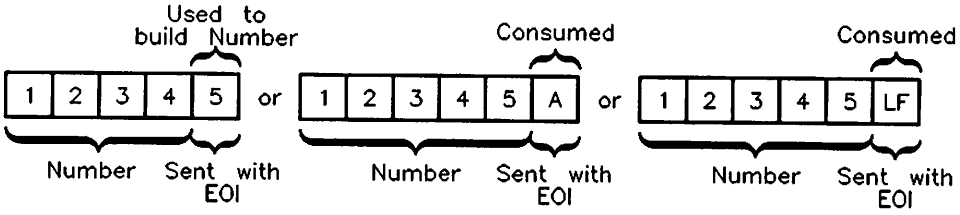
```
ENTER @Device;String$
```



The result of entering the preceding data with the given ENTER statement is that String$ receives the characters "ABCDEF". The EOI signal being received with either the last character or with the terminator character properly terminates the ENTER statement. If the character accompanied by EOI is a string character (not a terminator), it is entered into the variable as usual.

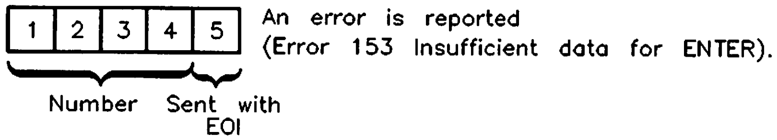
For example:

```
ENTER @Device;Number
```



The result of entering any of the above data streams with the given ENTER statement is that Number receives the value 12345. If the EOI signal accompanies a numeric character, it is entered and used to build the number; if the EOI is received with a numeric terminator, the terminator is lost as usual.

```
ENTER @Device;Number,String$
```



The result of entering the preceding data with the given statement is that an error is reported when the character "5" accompanied by EOI is received. However, Number receives the value 12345, but String$ retains its previous value. An error is reported because *all* variables in the destination list have *not* been satisfied when the EOI is received. Thus, the EOI signal is an *immediate statement terminator during free-field enters.* The EOI signal has a *different* definition during enters that use images, as described later in this chapter.

Enters that Use Images

The free-field form of the ENTER statement is very convenient to use; the computer automatically takes care of placing each character into the proper destination item. However, there are times when you need to design your own images that match the format of the data output by sources. Several instances for which you may need to use this type of enter operations are: the incoming data does not contain any terminators; the data stream is not followed by an end-of-line sequence; or two consecutive bytes of data are to be entered and interpreted as a two's-complement integer.

The ENTER USING Statement

The means by which you can specify how the computer will interpret the incoming data is to reference an image in the ENTER statement. The four general ways to reference the image in ENTER statements are as follows.

```
100  ENTER @Device_x USING "6A,DDD.DD";String_var$,Num_var

100  Image_str$="6A,DDD.DD"
110  ENTER @Device_x USING Image_str$;String_var$,Num_var

100  ENTER @Device USING Image_stmt;String_var$,Num_var
110  Image_stmt: IMAGE 6A,DDD.DD

100  ENTER @Device USING 110;String_var$,Num_var
110  IMAGE 6A,DDD.DD
```

Images

Images are used to specify how data entered from the source is to be interpreted and placed into variables; each image consists of one or more groups of individual image specifiers that determine how the computer will interpret the incoming data bytes (or words). Thus, image lists can be thought of as a description of *either:*

- the format of the expected data, or
- the procedure that the ENTER statement will use to enter and interpret the incoming data bytes.

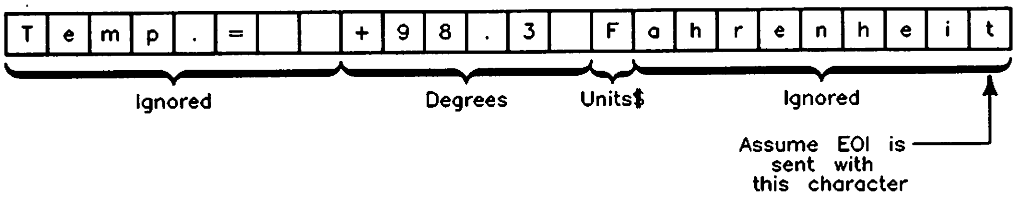
The examples given here treat the image list as a *procedure.*

All of the image specifiers used in image lists are valid for both enters and outputs. However, most of the specifiers have a slightly different meaning for each operation. If you plan to use the same image for output and enter, you must fully understand how both statements will use the image.

Example of an Enter Using an Image

This example is used to show you exactly how the computer uses the image to enter incoming data into variables. Look through the example to get a general feel for how these enter operations work. Afterwards, you should read the descriptions of the pertinent specifier(s).

Assume that the following stream of data bytes are to be entered into the computer.


T e m p . =   + 9 8 . 3   F a h r e n h e i t
Ignored
Degrees
Units$
Ignored
Assume EOI is sent with this character

Given the preceding conditions, let's look at how the computer executes the following ENTER statement that uses the specified IMAGE statement.

```
300  ENTER @Device USING Image_1;Degrees,Units$
310  Image_1:  IMAGE 8X,SDDD.D,A
```

Step 1. The computer evaluates the first image of the IMAGE statement. It is a special image in that it does not correspond to a variable in the destination list. It specifies that eight characters of the incoming data stream are to be ignored. Eight characters, "Temp.= ", are entered and are ignored (i.e., are not entered into any variable).

Step 2. The computer evaluates the next image. It specifies that the next six characters are to be used to build a number. Even though the order of the sign, digit, and radix are explicitly stated in the image, the actual order of these characters in the incoming data stream does not have to match this specifier exactly. Only the *number* of numeric specifiers in the image, here six, is all that is used to specify the data format. When all six characters have been entered, the number builder attempts to form a number.

Step 3. After the number is built, it is placed into the variable "Degrees"; the representation of the resultant number depends on the numeric variable's type (INTEGER, or REAL).

Step 4. The next image in the IMAGE statement is evaluated. It requires that one character be entered for the purpose of filling the variable "Units$". One byte is then entered into Units$.

Step 5. All images have been satisfied; however, the computer has not yet detected a statement-terminating condition. A line-feed or a character accompanied by EOI must be received to terminate the ENTER statement. Characters are then entered, but ignored, in search of one of these conditions. The statement is terminated when the EOI is sent with the "t". For further explanation, see "Terminating Enters that Use Images".

The above example should help you to understand how images are used to determine the interpretation of incoming data. The next section will help you to use each specifier to create your desired images.

Image Definitions During Enter

This section describes the individual image specifiers in detail. The specifiers have been categorized into data and function type.

Numeric Images

Sign, digit, radix, and exponent specifiers are all used identically in ENTER images. The number builder can also be used to enter numeric data.

Numeric Specifiers

Image Specifier	Meaning
D	Specifies that one byte is to be entered and interpreted as a numeric character. If the characters is non-numeric (including leading spaces and item terminators), it will still "consume" one digit of the image item.
Z, *	Same action as D. Keep in mind that A and * can only appear to the left of the radix indicator (decimal point or R) in a numeric image item.
S, M	Same action as D in that one byte is to be entered and interpreted as a numeric character. At least one digit specifier must follow either of these specifiers in an image item.
.	Same action as D in that one byte is to be entered and interpreted as a numeric character. At least one digit specifier must accompany this specifier in an image item.
R	Same action as D in that one byte is to be entered and interpreted as a numeric character; however, when R is used in a numeric image, it directs the number builder to use the comma as a radix indicator and the period as a terminator to the numeric item. At least one digit specifier must accompany this specifier in the image item.
E	Equivalent to 4D, if preceded by at least one digit specifier (Z, *, or D) in the image item. The following specifiers must also be preceded by at least one digit specifier.
ESZ	Equivalent to 3D.
ESZZ	Equivalent to 4D.
ESZZZ	Equivalent to 5D.
K, −K	Specifies that a variable number of characters are to be entered and interpreted according to the rules of the number builder (same rules as used in "free-field" ENTER operations).
H, −H	Like K, except that a comma is used as the radix indicator, and a period is used as the terminator for the numeric item.

Examples of Numeric Images

These 5 are equivalent:

```
ENTER @Device USING "SDD.D";Number
ENTER @Device USING "3D.D";Number
ENTER @Device USING "5D";Number
ENTER @Device USING "DESZZ";Number
ENTER @Device USING "**.DD";Number
```

Use the rules of the number builder:

```
ENTER Device USING "K";Number
```

Enter five characters, using comma as radix:

```
ENTER @Device USING "DDRDD";Number
```

Use the rules of the number builder, but use the comma as radix:

```
ENTER @Device USING "H";Number
```

String Images

The following specifiers are used to determine the number of and the interpretation of data bytes entered into string variables.

String Specifiers

Image Specifier	Meaning
A	Specifies that one byte is to be entered and interpreted as a string character. Any terminators are entered into the string when this specifier is used.
K, H	Specifies that "free-field" ENTER conventions are to be used to enter data into a string variable; characters are entered directly into the variable until a terminating condition is sensed (such as CR/LF, LF, or an END indication).
−K, −H	Like K, except that line-feeds (LF's) do not terminate entry into the string; instead, they are treated as string characters and placed in the variable. Receiving an END indication terminates the image item (for instance, receiving EOI with a character on an HP-IB interface, encountering an end-of-data, or reaching the variable's dimensioned length).
L, @	These specifiers are ignored for ENTER operations; however, they are allowed for compatibility with OUTPUT statements (that is, so that one image may be used for *both* ENTER and OUTPUT statements). Note that it may be necessary to skip characters (with specifiers such as X or /) when ENTERing data which has been sent by including these specifiers in an OUTPUT statement.

Examples of String Images

Enter 10 characters:

```
ENTER @Device USING "10A";Ten_chars$
```

Enter using the free-field rules:

```
ENTER @Device USING "K";Any_string$
```

Enter two strings:

```
ENTER @Device USING "5A,K";String$,Number$
```

Enter a string and a number:

```
ENTER @Device USING "5A,K";String$,Number
```

Enter characters until string is full or END is received:

```
ENTER @Device USING "-K";All_chars$
```

Ignoring Characters

These specifiers are used when one or more characters are to be ignored (i.e., entered but not placed into a string variable).

Specifiers Used to Ignore Characters

Image Specifier	Meaning
X	Specifies that a character is to be entered but ignored (not placed into a variable).
"literal"	Specifies that the number of characters in the literal are to be entered but ignored (not placed into a variable).
/	Specifies that all characters are to be entered but ignored (not placed into a variable) until a line-feed is received. EOI is also ignored until the line-feed is received.

Examples of Ignoring Characters

Ignore first five and use second five characters:

```
ENTER @Device USING "5X,5A";Five_chars$
```

Ignore 6th through 9th characters:

```
ENTER @Device USING "5A,4X,10A";S_1$,S_2$
```

Ignore 1st item of unknown length:

```
ENTER @Device USING "/,K";String2$
```

Ignore two characters:

```
ENTER @Device USING """zz"",AA";S_2$
```

Binary Images

These specifiers are used to enter one byte (or word) that will be interpreted as a number.

Binary Specifiers

Image Specifier	Meaning
B	Specifies that one byte is to be entered and interpreted as an integer in the range 0 through 255.
W	Specifies that one 16-bit word is to be entered and interpreted as a 16-bit, two's complement INTEGER. Since all HP Instrument BASIC interfaces are 8-bit, two bytes are always entered; the first byte entered is most significant. If the source is a file, or string variable, all data are entered as bytes; however, one byte may still be entered and ignored when necessary to achieve alignment on a word boundary.
Y	Like W, except that pad bytes are never entered to achieve word alignment.

Examples of Binary Images

Enter three bytes, then look for LF or END indication:

```
ENTER @Device USING "B,B,B";N1,N2,N3
```

Enter the first two bytes as an INTEGER, then the rest as string data:

```
ENTER @Device USING "W,K";N,N$
```

Terminating Enters that Use Images

This section describes the default statement-termination conditions for enters that use images (for devices). The effects of numeric-item and string-item terminators and the end-or-identify (EOI) signal during these operations are discussed in this section. After reading this section, you will be able to better understand how enters that use images work and how the default statement-termination conditions are *modified* by the #, %, +, and - image specifiers.

Default Termination Conditions

The default statement-termination conditions for enters that use images are very similar to those required to terminate free-field enters. *Either* of the following conditions will properly terminate an ENTER statement that uses an image.

- An END indication (such as the EOI signal or end-of-data) is received *with* the byte that satisfies the last *image item within 256 bytes after* the byte that satisfied the last image item.
- A line-feed is received *as* the byte that satisfies the last *image item* (exceptions are the "B" and "W" specifiers) or *within 256 bytes after* the byte that satisfied the last image item.

EOI Re-Definition

It is important to realize that when an enter uses an image (when the secondary keyword "USING" is specified), the definition of the EOI signal is *automatically modified.* If the EOI signal terminates the *last image item,* the entire statement is properly terminated, as with free-field enters. In addition, *multiple EOI signals are now allowed* and act as *item* terminators; however, the EOI must be received *with* the byte that satisfies each image item. If the EOI is received *before* any image is satisfied, it is *ignored.* Thus, all images must be satisfied, and EOI will not cause early termination of the ENTER-USING-image statement.

The following table summarizes the definitions of EOI during several types of ENTER statement. The statement-terminator modifiers are more fully described in the next section.

Effects of EOI During ENTER Statements

	Free-Field ENTER Statements	ENTER USING without # or %	ENTER USING with #	ENTER USING with %
Definition of EOI	Immediate statement terminator	Item terminator or statement terminator	Item terminator or statement terminator	Immediate statement terminator
Statement Terminator Required?	Yes	Yes	No	No
Early Termination Allowed?	No	No	No	Yes

Statement-Termination Modifiers

These specifiers modify the conditions that terminate enters that use images. The first one of these specifiers encountered in the image list modifies the termination conditions for the ENTER statement. If another of these specifiers is encountered in the image list, it again modifies the terminating conditions for the statement.

Statement-Termination Modifiers

<table>
<tr><th>Image Specifier</th><th>Meaning</th></tr>
<tr><td>#</td><td>Specifies that a statement-termination condition is <i>not</i> required; the ENTER statement is automatically terminated as soon as the <i>last image item</i> is satisfied.</td></tr>
<tr><td>%</td><td>Also specifies that a statement-termination condition is not required. In addition, EOI is re-defined to be an <i>immediate</i> statement terminator, <i>allowing early termination</i> of the ENTER <i>before all</i> image items have been satisfied. However, the statement can only be terminated on a "legal item boundary". The legal boundaries for different specifiers are as follows:
<table>
<tr><th>Specifier</th><th>Legal Boundary</th></tr>
<tr><td>K,−K</td><td>With any character, since this specifies a variable-width field of characters.</td></tr>
<tr><td>S,M,D,E,Z,.,A,
X,<i>literal</i>,B,W</td><td>Only with the last character that satisfies the image (e.g., with the 5th character of a 5A image). If EOI is received with any other character, it is ignored.</td></tr>
<tr><td>/</td><td>Only with the last line-feed character that satisfies the image (e.g., with the 3rd line-feed of a "3/" image); otherwise it is ignored.</td></tr>
</table></td></tr>
<tr><td>+</td><td>Specifies that an END indication is required to terminate the ENTER statement. Line-feeds are ignored as statement terminators; however, they will still terminate items (unless a −K or −H image is used for strings).</td></tr>
<tr><td>−</td><td>Specifies that a line-feed is required to terminate the statement. EOI is ignored, and other END indications (such as EOF or end-of-data) cause an error if encountered before the line-feed.</td></tr>
</table>

Examples of Modifying Termination Conditions

Enter a single byte:

```
ENTER @Device USING "#,B";Byte
```

Enter a single word:

```
ENTER @Device USING "#,W";Integer
```

Enter an array, allowing early termination by EOI:

```
ENTER @Device USING ",K";Array(*)
```

Enter characters into `String$` until line-feed received, then continue entering characters it until END received:

```
ENTER @Device USING "+,K";String$
```

Enter characters until line-feed received; ignore EOI, if received:

```
ENTER @Device USING "-,K";String$
```

Additional Image Features

Several additional image features are available with this BASIC language. Some of these features have already been shown in examples, and all of them resemble the additional features of images used with OUTPUT statements.

Repeat Factors

All of the following specifiers can be preceded by an integer that specifies how many times the specifier is to be used.

Repeatable Specifiers	Non-Repeatable Specifiers
Z, D, A, X, /, @, L	S, M, ., R, E, K, H, B, W, Y, #, %, +, -

Image Re-Use

If there are fewer images than items in the destination list, the list will be re-used, beginning with the first item in the image list. If there are more images than there are items, the additional specifiers will be ignored.

Examples

The "B" is re-used:

```
ENTER @Device USING "#,B";B1,B2,B3
```

The "W" is not used:

```
ENTER @Device USING "2A,2A,W";A$,B$
```

Nested Images

Parentheses can be used to nest images within the image list. The hierarchy is the same as with mathematical operations; evaluation is from inner to outer sets of parentheses. The maximum number of levels of nesting is eight.

Example

```
ENTER @Source USING "2(B,5A,/),/";N1,N1$,N2,N2$
```

6

I/O Path Attributes

This chapter contains two major topics, both of which involve additional features provided by I/O path names.

- The first topic is that I/O path names can be given attributes which control the way that the system handles the data sent and received through the I/O path. Attributes are available for such purposes as controlling data representations, and defining special end-of-line (EOL) sequences.
- The second topic is that one set of I/O statements can access most system resources instead of using a separate set of statements to access each class of resources. This second topic, herein called "unified I/O", may be considered an implicit attribute of I/O path names.

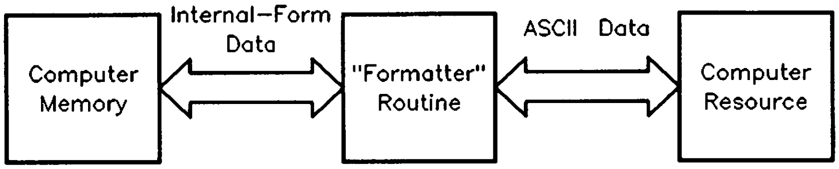
The FORMAT Attributes

All I/O paths possess one of the two following attributes:

- FORMAT ON—means that the data are sent in ASCII representation.
- FORMAT OFF—means that the data are sent in BASIC internal representation.

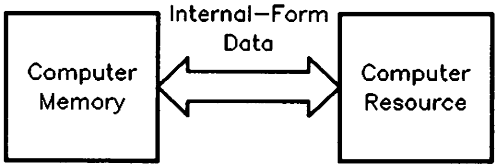
Before getting into how to assign these attributes to I/O paths, let's take a brief look at each one.

With FORMAT ON, internally represented numeric data must be "formatted" into its ASCII representation before being sent to the device. Conversely, numeric data being received from the device must be "unformatted" back into its internal representation. These operations are shown in the diagrams below:



Numeric Data Transformations with FORMAT ON

With FORMAT OFF, however, no formatting is required. The data items are merely copied from the source to the destination. This type of I/O operation requires less time, since fewer steps are involved.



Numeric Data Transfer with FORMAT OFF

The only requirement is that the resource also use the exact same data representations as the internal HP Instrument BASIC representation.

Here are how each type of data item is represented and sent with FORMAT OFF:

- INTEGER: two-byte (16-bit), two's complement.
- REAL: eight-byte (64-bit) IEEE floating-point standard.
- String: four-byte (32-bit) length header, followed by ASCII characters. An additional ASCII space character, CHR$(32), may be sent and received with strings in order to have an even number of bytes.

Here are the FORMAT OFF rules for OUTPUT and ENTER operations:

- No item terminator and no EOL sequence are sent by OUTPUT.
- No item terminator and no statement-termination conditions are required by ENTER.
- If either OUTPUT or ENTER uses an IMAGE (such as with OUTPUT 701 USING "4D.D"), then the FORMAT ON attribute is *automatically* used.

Assigning Default FORMAT Attributes

As discussed in the "Directing Data Flow" chapter, names are assigned to I/O paths between the computer and devices with the ASSIGN statement. Here is a typical example:

```
ASSIGN Any_name TO Device_selector
```

This assignment fills a "table" in memory with information that describes the I/O path. This information includes the device selector, the path's FORMAT attribute, and other descriptive information. When the I/O path name is specified in a subsequent I/O statement (such as OUTPUT or ENTER), this information is used by the system in completing the I/O operation.

Different default FORMAT attributes are given to devices and files:

- **Devices**—since most devices use an ASCII data representation, the default attribute assigned to devices is FORMAT ON. (This is also the default for ASCII files.)
- **BDAT** and **HP-UX** or **DOS** files—the default for BDAT and HPUX or DOS files is FORMAT OFF. (This is because the FORMAT OFF representation requires no translation time for numeric data; this is possible because humans never see the data patterns written to the file, and therefore the items do not have to be in ASCII, or humanly readable, form.)

One of the most powerful features of this BASIC system is that you can change the attributes of I/O paths programmatically.

Specifying I/O Path Attributes

There are two ways of specifying attributes for an I/O path:

Specify the desired attribute(s) when the I/O path name is initially assigned. For example:

```
100  ASSIGN @Device TO Dev_selector; FORMAT ON                    or
100  ASSIGN @Device TO Dev_selector ! Default for devices is FORMAT ON.
```

Specify only the attribute(s) in a subsequent ASSIGN statement:

```
    250  ASSIGN @Device; FORMAT OFF  ! Change only the attribute.
```

The result of executing this last statement is to modify the entry in the I/O path name table that describes which FORMAT attribute is currently assigned to this I/O path. The implicit `ASSIGN @Device TO *`, which is automatically performed when the "TO ... " portion is included, is *not* performed. Also, the I/O path name must currently be assigned (in this context), or an error is reported.

Changing the EOL Sequence Attribute

In addition to the FORMAT attributes, another attribute is available to direct HP Instrument BASIC system to re-define the end-of-line sequence normally sent after the last data item in output operations.

An end-of-line (EOL) sequence is normally sent following the last item sent with free-field OUTPUT statements and when the "L" specifier is used in an OUTPUT that uses an image. The default EOL characters are carriage-return and line-feed (CR/LF), sent with no device-dependent END indication. You can also define your own special EOL sequences that include sending special characters, sending an END indication, and delaying a specified amount of time after sending the last EOL character.

In order to define non-default EOL sequences to be sent by the OUTPUT statement, an I/O path must be used. The EOL sequence is specified in one of the ASSIGN statements which describe the I/O path. Here is an example that changes the EOL sequence to a single line-feed character.

```
    ASSIGN @File TO "file_one";EOL CHR$(10)
```

The characters following the secondary keyword `EOL` are the EOL characters. Any character in the range CHR$(0) through CHR$(255) may be included in the string expression that defines the EOL characters; however, the length of the sequence is limited to eight characters or less.

If END is included in the EOL attribute, an interface-dependent "END" indication is sent with (or after) the last character of the EOL sequence. However, if no EOL sequence is sent, the END indication is also suppressed. The following statement shows an example of defining the EOL sequence to include an END indication.

```
    ASSIGN @Device TO 20;EOL CHR$(13)&CHR$(10) END
```

With the HP-IB Interface, the END indication is an End-or-Identify message (EOI) sent with the last EOL character.

The default EOL sequence is a CR and LF sent with no end indication. This default can be restored by using the EOL OFF attribute.

Restoring the Default Attributes

If any attribute is specified, the corresponding entry in the I/O path name table is changed (as above); no other attributes are affected. However, if no attribute is assigned (as below), then *all* attributes are restored to their default state (such as FORMAT ON for devices.)

```
    340  ASSIGN @Device  !  Restores ALL default attributes.
```

Concepts of Unified I/O

The HP Instrument BASIC language provides the ability to communicate with the several system resources with the OUTPUT and ENTER statements.

The next section of this chapter describes how data can be moved to and from string variables with OUTPUT and ENTER statements.

And, if you have read about mass storage operations (in the "Data Storage and Retrieval" chapter of *HP Instrument BASIC Programming Techniques*), you know that the ENTER and OUTPUT statements are also used to move data between the computer and mass storage files.

This ability to move data between the computer and all of its resources with the same statements is a very powerful capability of the HP Instrument BASIC language.

Before briefly discussing I/O paths to mass storage files, the following discussion will present some background information that will help you understand the rationale behind implementing the two data representations used by the computer. The remainder of this chapter then presents several uses of this language structure.

Data-Representation Design Criteria

As you know, the computer supports two general data representations—the ASCII and the internal representations. This discussion presents the rationale of their design.

The data representations used by the computer were chosen according to the following criteria.

- to maximize the rate at which computations can be made
- to maximize the rate at which the computer can move the data between its resources
- to minimize the amount of storage space required to store a given amount of data
- to be compatible with the data representation used by the resources with which the computer is to communicate

The *internal representations* implemented in the computer are designed according to the *first three of the above criteria*. However, the last criterion must always be met if communication is to be achieved. If the resource uses the ASCII representation, this compatibility requirement takes precedence over the other design criteria. The *ASCII representation* fulfills this *last criterion* for most devices and for the computer operator. The first three criteria are further discussed in the following description of data representations used for mass storage files.

I/O Paths to Files

There are three types of *data files:* ASCII, BDAT, and HP-UX or DOS.

- Only the ASCII data representation is used with ASCII files.
- But either the ASCII (FORMAT ON) or the internal (FORMAT OFF) representation can be used with BDAT and HP-UX or DOS files.

BDAT, HPUX and DOS Files

BDAT, HP-UX and DOS files have been designed to maximize the efficiency with which HP Instrument BASIC moves, stores and manipulates data. Both numeric and string computations are much faster. These internal data representations allow much more data to be stored on a disc because there is no storage overhead (for numeric items); that is, there are no "record headers" for numeric items.

The **transfer rates** for each data type has also been *increased.* Numeric output operations are always much faster because there is no time required for "formatting". Numeric enter operations are also faster because the system does not have to search for item- and statement-termination conditions.

In addition, I/O paths to BDAT and HP-UX files can use either the ASCII (FORMAT ON) or the internal (FORMAT OFF) representation.

The following program shows a few of the features of BDAT files. The program first outputs an internal-form string (with FORMAT ON), and then enters the length header and string characters with FORMAT OFF.

```
110   DIM Length$[4],Data$[256],Int_form$[256]
120   !
130   ! Create a BDAT file (1 record; 256 bytes/record.)
140   ON ERROR GOTO Already_created
150   CREATE BDAT "B_file",1
160   Already_created: OFF ERROR
170   !
180   ! Use FORMAT ON during output.
190   ASSIGN @Io_path TO "B_file";FORMAT ON
200   !
210   Length$=CHR$(0)&CHR$(0)  ! Create length header.
220   Length$=Length$&CHR$(0)&CHR$(252)
230   !
240   ! Generate 256-character string.
250   Data$="01234567"
260   FOR Doubling=1 TO 5
270      Data$=Data$&Data$
280   NEXT Doubling
290   ! Use only 1st 252 characters.
300   Data$=Data$[1,252]
310   !
320   ! Generate internal-form and output.
330   Int_form$=Length$&Data$
340   OUTPUT @Io_path;Int_form$;
350   ASSIGN @Io_path TO *
360   !
370   ! Use FORMAT OFF during enter (default).
380   ASSIGN @Io_path TO "B_file"
390   !
400   ! Enter and print data and # of characters.
```

```
410   ENTER Data$
420   PRINT LEN(Data$);"characters entered."
430   PRINT
440   PRINT Data$
450   ASSIGN @Io_path TO * ! Close I/O path.
460   !
470   END
```

ASCII Files

ASCII files are designed for interchangeability with other HP computer systems. This interchangeability imposes the restriction that the data must be represented with ASCII characters. Each data item sent to these files is a special case of FORMAT ON representation; *each item is preceded by a two-byte length header* (analogous to the internal form of string data). In order to maintain this compatibility, there are two additional restrictions placed on ASCII files:

- The FORMAT OFF attribute *cannot* be assigned to an ASCII file
- You cannot use OUTPUT..USING or ENTER..USING with an ASCII file.

The following program shows the I/O path name @Io_path being assigned to the ASCII file named ASC_FILE. Notice that the file name is in all uppercase letters; this is also a compatibility requirement when using this file with some other systems.

The program creates an ASCII file, and then outputs program lines to the file. The program then gets and runs this newly created program. (If you type in and run this program, be sure to save it on disc, because running the program will load the program it creates, destroying itself in the process.)

```
100   DIM Line$(1:3)[100]  ! Array to store program.
110   !
120   ! Create if not already on disc.
130   ON ERROR GOTO Already_exists
140   CREATE ASCII "ASC_FILE",1  ! 1 record.
150   Already_exists:  OFF ERROR
160   !
170   ASSIGN @Io_path TO "ASC_FILE"
180   STATUS @Io_path,6;Pointer
190   PRINT "Initially:    file pointer=";Pointer
200   PRINT
210   !
220   Line$(1)="100  PRINT ""New program.""  "
230   Line$(2)="110  BEEP"
240   Line$(3)="120  END"
250   !
260   OUTPUT @Io_path;Line$(*)
270   STATUS @Io_path,6;Pointer
280   PRINT "After OUTPUT: file pointer=";Pointer
290   PRINT
300   !
310   GET "ASC_FILE" ! Implicitly closes I/O path.
320   !
330   END
```

Data Representation Summary

The following table summarizes the control that programs have on the FORMAT attribute assigned to I/O paths.

Program Control of the FORMAT Attribute

Type of Resource	Default FORMAT Attribute Used	Can Default FORMAT Attribute Be Changed?
Devices	FORMAT ON	Yes (if an I/O path is used)[1]
BDAT files	FORMAT OFF	Yes
HP-UX or DOS files	FORMAT OFF	Yes
ASCII files	FORMAT ON[2]	No
String variables	FORMAT ON	No

[1] FORMAT ON is *always* used whenever an OUTPUT ... USING or ENTER ... USING statement is used, regardless of the FORMAT attribute assigned to the I/O path.
[2] The data representation used with ASCII files is a special case of the FORMAT ON representation.

Applications of Unified I/O

This section describes two uses of the powerful unified-I/O scheme of the computer. The first application contains further details and uses of I/O operations with string variables. The second application involves using a disc file to simulate a device.

I/O Operations with String Variables

This section describes both the details of and several uses of outputting data to and entering data from string variables.

Outputting Data to String Variables

When a string variable is specified as the destination of data in an OUTPUT statement, source items are evaluated individually and placed into the variable according to the free-field rules or the specified image, depending on which type of OUTPUT statement is used. Thus, item terminators may or may not be placed into the variable. The ASCII data representation is always used during outputs to string variables.

Characters are always placed into the variable beginning at the first position; no other position can be specified as the beginning position at which data will be placed. Thus, *random access of the information in string variables is not allowed* from OUTPUT and ENTER statements; all data must be accessed serially. For instance, if the characters "1234" are output to a string variable by one OUTPUT statement, and a subsequent OUTPUT statement outputs the characters "5678" to the same variable, the second output *does not* begin where the first one left off (i.e., at string position five). The second OUTPUT statement begins placing characters in position one, just as the first OUTPUT statement did, overwriting the data initially output to the variable by the first OUTPUT statement.

The string variable's length header (4 bytes) is updated and compared to the dimensioned length of the string as characters are output to the variable. If the string is filled before all items have been output, an error is reported; however, the string contains the first n characters output (where n is the dimensioned length of the string).

Example

The following program outputs string and numeric data items to a string variable and then calls a subprogram which displays each character, its decimal code, and its position within the variable.

```
100   ASSIGN @Crt TO 1  ! CRT is disp. device.
110   !
120   OUTPUT Str_var$;12,"AB",34
130   !
140   CALL Read_string(@Crt,Str_var$)
150   !
160   END
170   !
180   !
190 SUB Read_string(@Disp,Str_var$)
200     !
210     ! Table heading.
220     OUTPUT @Disp;"---------------------"
230     OUTPUT @Disp;"Character  Code  Pos."
240     OUTPUT @Disp;"---------  ----  ----"
250     Dsp_img$="2X,4A,5X,3D,2X,3D"
260     !
270     ! Now read the string's contents.
280   FOR Str_pos=1 TO LEN(Str_var$)
290      Code=NUM(Str_var$[Str_pos;1])
300     IF Code<32 THEN ! Don't disp. CTRL chars.
310       Char$="CTRL"
320     ELSE
330       Char$=Str_var$[Str_pos;1] ! Disp. char.
340     END IF
350     !
360     OUTPUT @Disp USING Dsp_img$;Char$,Code,Str_pos
370   NEXT Str_pos
380   !
390   ! Finish table.
400   OUTPUT @Disp;"---------------------"
410   OUTPUT @Disp ! Blank line.
420   !
430   SUBEND
```

```
--------------------
Character  Code  Pos.
---------  ----  ----
           32    1
1          49    2
2          50    3
,          44    4
A          65    5
B          66    6
CTRL       13    7
CTRL       10    8
           32    9
3          51    10
4          52    11
CTRL       13    12
CTRL       10    13
--------------------
```

Outputting data to a string and then examining the string's contents is usually a more convenient method of examining output data streams than using a mass storage file. The preceding subprogram may facilitate the search for control characters, because they are not actually displayed, which could otherwise interfere with examining the data stream.

Example

Outputs to string variables can also be used to generate the string representation of a number, rather than using the VAL$ function (or a user-defined function subprogram). The *main advantage* is that you can explicitly specify the number's image while still using only a single program line. The following program compares the string generated by the VAL$ function to that generated by outputting the number to a string variable.

```
100   X=12345678
110   !
120   PRINT VAL$(X)
130   !
140   OUTPUT Val$ USING "#,3D.E";X
150   PRINT Val$
160   !
170   END
```

```
1.2345678E+7    Printed results
123.E+05
```

Entering Data From String Variables

Data are entered from string variables in much the same manner as output to the variable. All ENTER statements that use string variables as the data source interpret the data according to the FORMAT ON attribute. Data is read from the variable beginning at the first string position; if subsequent ENTER statements read characters from the variable, the read also begins at the first position. If more data are to be entered from the string than are contained in the string, an error is reported; however, all data entered into the destination variable(s) before the end of the string was encountered remain in the variable(s) after the error occurs.

When entering data from a string variable, the computer keeps track of the number of characters taken from the variable and compares it to the string length. Thus, *statement-termination* conditions are *not* required; the ENTER statement automatically terminates when the last character is read from the variable. However, *item* terminators are still required *if* the items are to be separated *and* the lengths of the items are not known. If the length of each item is known, an image can be used to separate the items.

Example

The following program shows an example of the need for *either* item terminators *or* length of each item. The first item was not properly terminated and caused the second item to not be recognized.

```
100   OUTPUT String$;"ABC123";  ! OUTPUT w/o CR/LF.
110   !
120   ! Now enter the data.
130   ON ERROR GOTO Try_again
140   !
150  First_try: !
160   ENTER String$;Str$,Num
170   OUTPUT 1;"First try results:"
180   OUTPUT 1;"Str$= ";Str$,"Num=";Num
190   BEEP     ! Report getting this far.
200   STOP
210   !
220  Try_again: OUTPUT 1;"Error";ERRN;" on 1st try"
230             OUTPUT 1;"STR$=";Str$,"Num=";Num
240             OUTPUT 1
250             OFF ERROR  ! The next one will work.
260             !
270   ENTER String$ USING "3A,3D";Str$,Num
280   OUTPUT 1;"Second try results:"
290   OUTPUT 1;"Str$= ";Str$,"Num=";Num
300   !
310   END
```

This technique is convenient when attempting to enter an unknown amount of data or when numeric and string items within incoming data are not terminated. The data can be entered into a string variable and then searched by using images.

Example

ENTERs from string variables can also be used to generate a number from ASCII numeric characters (a recognizable collection of decimal digits, decimal point, and exponent information), rather than using the VAL function. As with outputs to string variables, images can be used to interpret the data being entered.

```
30    Number$="Value= 43.5879E-13"
40    !
50    ENTER Number$;Value
60    PRINT "VALUE=";Value
70    END
```

Index

Contents

1

Using the Language Reference

This section contains an alphabetical reference to all the keywords currently available with the HP Instrument BASIC language. Each entry defines the keyword, shows the proper syntax for its use, gives some example statements, and explains relevant semantic details. A cross reference is provided in the "Keyword Guide to Porting" chapter of the *HP Instrument BASIC Programming Techniques* manual, that groups the keywords into several functional categories.

Legal Usage Table

Above each drawing is a small table indicating the legal uses of the keyword.

Keyboard Executable	Yes
Programmable	No
In an IF..THEN..	No

Keyboard Executable — means that a properly constructed statement containing that keyword can be typed into the keyboard input line and executed by a press of the EXECUTE, ENTER, or Return key. Note that not all host instruments implement a keyboard input line or arbitrary command execution. See your instrument-specific HP Instrument BASIC manual to determine your keyboard capabilities.

Programmable — means that a properly constructed statement containing that keyword can be placed after a line number and stored in a program.

In an IF ... THEN ... — means that a properly constructed statement containing that keyword can be placed after "THEN" in a **single-line** IF ... THEN statement. Keywords that are prohibited in a single-line IF ... THEN are not necessarily prohibited in a multiple-line IF ... THEN structure. Constructs such as IF ... THEN, REPEAT ... UNTIL, and FOR ... NEXT statements are executed conditionally when they are included in a multiple-line IF ... THEN structure. All other prohibited statements (see IF ... THEN) are used only during pre-run. Therefore, the action of those statements will not be conditional, even though the IF ... THEN wording may make them appear to be conditional.

Syntax Drawings Explained

Statement syntax is represented pictorially. All characters enclosed by a rounded envelope must be entered exactly as shown. Words enclosed by a rectangular box are names of items used in the statement.

A description of each item is given either in the table following the drawing, another drawing, or the Glossary.

Statement elements are connected by lines. Each line can be followed in only one direction, as indicated by the arrow at the end of the line. Any combination of statement elements that can be generated by following the lines in the proper direction is syntactically correct. An element is optional if there is a path around it. Optional items usually have default values. The table or text following the drawing specifies the default value that is used when an optional item is not included in a statement.

Comments may be added to any valid line. A comment is created by placing an exclamation point after a statement, or after a line number or line label.

```
100  PRINT "Hello"  ! This is a comment.
110  ! This is also a comment.
```

The text following the exclamation point may contain any characters in any order.

The drawings do not necessarily deal with the proper use of spaces (ASCII blanks). In general, whenever you are traversing a line, any number of spaces may be entered. If two envelopes are touching, it indicates that no spaces are allowed between the two items. However, this convention is not always possible in drawings with optional paths, so it is important to understand the following rules for spacing.

Keywords and Spaces

HP Instrument BASIC uses spaces, as well as required punctuation, to distinguish the boundaries between various keywords, names, and other items. In general, at least one space is required between a keyword and a name if they are not separated by other punctuation. Spaces cannot be placed in the middle of keywords or other reserved groupings of symbols. Also, keywords are recognized whether they are typed in uppercase or lowercase. Therefore, to use the letters of a keyword as a name, the name entered must contain some mixture of uppercase and lowercase letters. The following are some examples of these guidelines.

Space Between Keywords and Names

The keyword NEXT and the variable `Count` are properly entered with a space between them, as in `NEXT Count`. Without the space, the entire group of characters is interpreted as the name `Nextcount`.

No Spaces in Keywords or Reserved Groupings

A function call to "A$" must be entered as `FNA$`, not as `FN A $`. The I/O path name "@Meter" must be entered as `@Meter`, not as `@ Meter`. The "exceptions" are keywords that contain spaces, such as `END IF`.

Using Keyword Letters for a Name

Attempting to store the line `IF X=1 THEN END` will generate an error because `END` is a keyword not allowed in an IF ... THEN. To create a line label called "End", type `IF X=1 THEN ENd`. This or any other mixture of uppercase and lowercase will prevent the name from being recognized as a keyword.

Also note that names may begin with the letters of an infix operator (such as MOD, DIV, and EXOR). In such cases, you should type the name with a case switch in the infix operator portion of the name (e.g., MOdULE, DiVISOR).

Keyboards

The HP Instrument BASIC instruments and computers support many keyboard styles.

Throughout the manuals which document HP Instrument BASIC, specific keys are mentioned. Because many key labels are different on each keyboard, you will not have all the keys mentioned. For example, (ENTER) and (Return) normally have the same meaning, but only one of them appears on any one keyboard. The instrument-specific HP Instrument BASIC manual included with your instrument discusses the keyboard for your device.

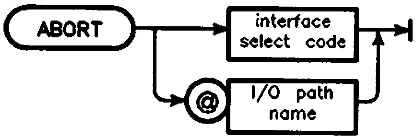
2

Keyword Dictionary

ABORT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement ceases activity on the specified interface.



Item	Description	Range
interface select code	numeric expression, rounded to an integer	5, 7 thru 31 (See instrument manual)
I/O path name	name assigned to an HP-IB interface	—

Example Statements

```
ABORT 7
IF Stop_code THEN ABORT @Source
```

Semantics

Executing this statement ceases activity on the specified HP-IB interface; other interfaces may not be specified. If the computer is the system controller but not currently the active controller, executing ABORT causes the computer to assume active control.

Summary of Bus Actions

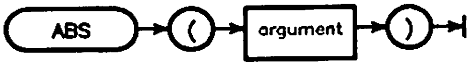
	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active/Controller	IFC (duration ≥100µsec) REN $\overline{\text{ATN}}$	Error	ATN MTA UNL $\overline{\text{ATN}}$	Error
Not Active Controller	IFC (duration ≥100µsec)* REN $\overline{\text{ATN}}$	Error	No Action	Error

**The IFC Message allows a non-active controller(which is the system controller) to become the active controller.*

ABS

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the absolute value of its argument.



Item	Description/Default	Range Restrictions
argument	numeric expression	within valid ranges of INTEGER and REAL data types for INTEGER and REAL arguments

Examples Statements

```
Magnitude=ABS(Vector)
PRINT "Value = ";ABS(-4)
```

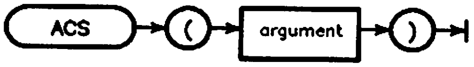
Range Restriction Specifics

Taking the ABS of the INTEGER −32768 will cause an error.

ACS

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the principle value of the angle which has a cosine equal to the argument. This is the arccosine function.



Item	Description/Default	Range Restrictions
argument	numeric expression	−1 thru +1 for INTEGER and REAL arguments

Examples Statements

```
Angle=ACS(Cosine)
PRINT "Angle = ";ACS(0.67)
```

Semantics

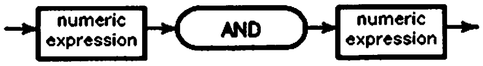
If the argument is REAL or INTEGER, the value returned is REAL.

The angle mode (RAD or DEG) for REAL and INTEGER arguments indicates whether you should interpret the value returned in degrees or radians. If the current angle mode is DEG, the range of the result is 0° to 180°. If the current angle mode is RAD, the range of the result is 0 to π radians. The angle mode is radians unless you specify degrees with the DEG statement.

AND

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This operator returns a 1 or a 0 based upon the logical AND of the arguments.



Example Statements

```
IF Flag AND Test2 THEN Process
Final=Initial AND Valid
```

Semantics

A non-zero value (positive or negative) is treated as a logical 1; only zero is treated as a logical 0.

The logical AND is shown in this table:

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

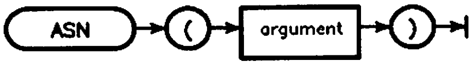
ASCII

See the CREATE ASCII statement.

ASN

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the principle value of the angle which has a sine equal to the argument. This is the arcsine function.



		Range
Item	Description/Default	Restrictions
argument	numeric expression	−1 thru +1 for INTEGER and REAL arguments

Examples Statements

```
Angle=ASN(Sine)
PRINT "Angle = ";ASN(0.98)
```

Semantics

If the argument is REAL or INTEGER, the value returned is REAL.

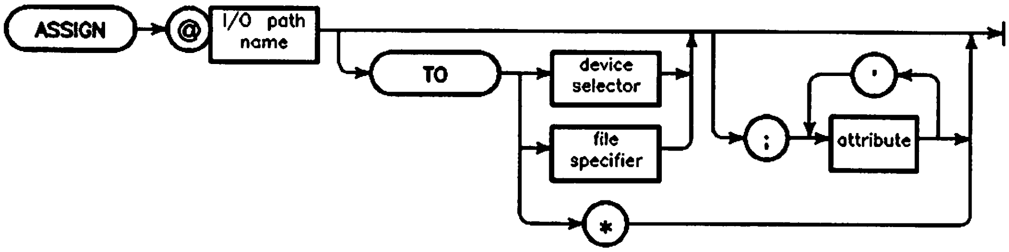
The angle mode (RAD or DEG) for REAL and INTEGER arguments indicates whether you should interpret the value returned in degrees or radians. If the current angle mode is DEG, the range of the result is −90° to 90°. If the current angle mode is RAD, the range of the result is $-\pi\ /2$ to $+\pi\ /2$ radians. The angle mode is radians unless you specify degrees with the DEG statement.

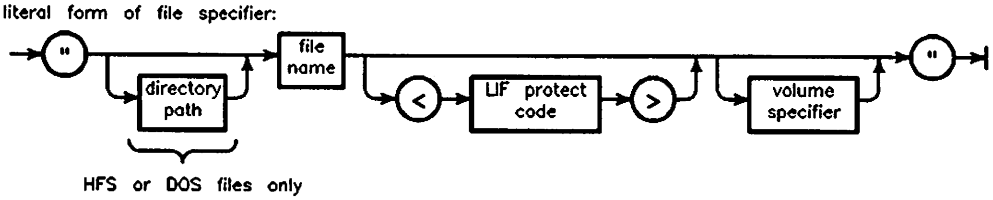
ASSIGN

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

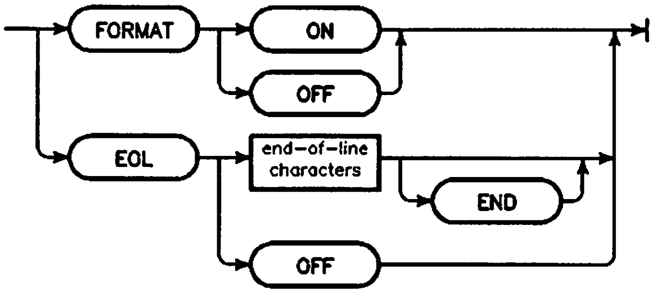
This statement is used to perform one of the following actions:

- assign an I/O path name and attributes to:
  - □ a device,
  - □ a group of devices,
  - □ or a mass storage file;
- change attributes;
- or close an I/O path name.





attribute:



Item	Description	Range
I/O path name	name identifying an I/O path	any valid name
device selector	numeric expression	(see Glossary)
file specifier	string expression	(see drawing)
attribute	attribute to be assigned to the I/O path name	(see drawing)
directory path	literal	(see MASS STORAGE IS)
file name	literal	depends on volume's format (see Glossary)
LIF protect code	literal; first two non-blank characters are significant	> not allowed
volume specifier	literal	(see MASS STORAGE IS)
end-of-line characters	string expression; Default=CR and LF	up to 8 characters

Example Statements

```
ASSIGN @Source TO Isc;FORMAT OFF
ASSIGN @Source;FORMAT ON
ASSIGN @Device TO 724
ASSIGN @Listeners TO 711,712,715
ASSIGN @Dest TO *

ASSIGN @File TO File_name$
ASSIGN @File TO Dir_path$&File_name$&Vol_spec$
ASSIGN @File TO "WorkDir/File"
ASSIGN @File TO "/RootDir/MyDir/MyFile:,700"
ASSIGN @File TO "DIR_JOHN/dir_proj/file1"
```

Semantics

The ASSIGN statement has three primary purposes. Its main purpose is to create an I/O path name and assign that name to an I/O resource and attributes that describe the use of that resource. The statement is also used to change the attributes of an existing I/O path and to close an I/O path.

Associated with an I/O path name is a unique data type that uses 148 bytes of memory. I/O path names can be placed in COM statements and can be passed by reference as parameters to subprograms. They cannot be evaluated in a numeric or string expression and cannot be passed by value.

Once an I/O path name has been assigned to a resource, OUTPUT, and ENTER operations can be directed to that I/O path name. This provides the convenience of re-directing I/O operations in a program by simply changing the appropriate ASSIGN statement. The resource assigned to the I/O path name may be an interface, a device, a group of devices on HP-IB, or a mass storage file.

The FORMAT Attributes

Assigning the FORMAT ON attribute to an I/O path name directs the computer to use its ASCII data representation while sending and receiving data through the I/O path. Assigning the FORMAT OFF attribute to an I/O path name directs the computer to use its internal data representation when using the I/O path.

LIF ASCII format (similar to ASCII representation) is always used with ASCII files; thus, if either FORMAT ON or FORMAT OFF is specified for the I/O path name of an ASCII file, it will be ignored.

If a FORMAT attribute is not explicitly given to an I/O path, a default is assigned. The following table shows the default FORMAT attribute assigned to computer resources.

Resource	Default Attribute
interface/device	FORMAT ON
ASCII file	(always ASCII format)
BDAT file	FORMAT OFF
HP-UX file	FORMAT OFF
DOS file	FORMAT OFF

Using Devices

I/O path names are assigned to devices by placing the device selector after the keyword TO. For example, `ASSIGN @Display TO 1` creates the I/O path name "@Display" and assigns it to the internal CRT display.

A device can have more than one I/O path name associated with it. Each I/O path name can have different attributes, depending upon how the device is used. The specific I/O path name used for an I/O operation determines which set of attributes is used for that operation.

Using Files

Assigning an I/O path name to a file name associates the I/O path with a file on the mass storage media (that is, it "opens" the "file"). The mass storage file must be a data file (a file of type ASCII, BDAT, or HP-UX). The file must already exist on the media, as ASSIGN does not do an implied CREATE.

Data files have a position pointer which is associated with each I/O path name. The position pointer identifies the next byte to be written or read. The position pointer is reset to the beginning of the file when the file is opened, and updated with each ENTER or OUTPUT that uses that I/O path name. (It is best if a file is open with only one I/O path name at a time.)

BDAT and HP-UX files have an additional *physical* end-of-file pointer. This end-of-file pointer (which resides on the media) is read when the file is opened. This end-of-file pointer is updated on the media at the following times:

- When the current end-of-file changes.

- When END is specified in an OUTPUT statement directed to the file.

HFS Permissions

ASSIGN opens any existing ASCII, BDAT, or HP-UX file if you currently have R (read) or W (write) access permission on the file as well as X (search) permission on the parent and all superior directories. Otherwise, error 183 will be reported.

Additional Attributes

The EOL attribute specifies the end-of-line (EOL) sequence sent after all data during normal OUTPUT operations and when the "L" image specifier is used. Up to eight characters may be specified as the EOL characters; an error is reported if the string contains more than eight characters. If END is included in the EOL attribute, an interface-dependent END indication is sent with the last character of the EOL sequence (such as the EOI signal on HP-IB interfaces); however, if no EOL sequence is sent, the END indication is also suppressed. END applies only to devices; it is ignored when a file is the destination. The default EOL sequence consists of sending a carriage-return and a line-feed character with no END indication and no delay period. This default is restored when EOL is OFF.

Changing Attributes

The attributes of a currently valid I/O path may be changed, without otherwise disturbing the state of that I/O path or the resource(s) to which it is assigned, by omitting the "TO resource" clause of the ASSIGN statement. For example, `ASSIGN @File;FORMAT OFF` assigns the FORMAT OFF attribute to the I/O path name "@File" without changing the file pointers (if assigned to a mass storage file).

A statement such as `ASSIGN @Device` restores the default attributes to the I/O path name, if it is currently assigned.

Closing I/O Paths

There are a number of ways that I/O paths are closed and the I/O path names rendered invalid. Closing an I/O path cancels any ON-event actions for that I/O path. I/O path names that are *not* included in a COM statement are closed at the following times:

- When they are explicitly closed; for example, `ASSIGN @File TO *`
- When a currently assigned I/O path name is re-assigned to a resource, the original I/O path is closed before the new one is opened. The re-assignment can be to the same resource or a different resource. No closing occurs when the ASSIGN statement only changes attributes and does not include the "TO ... " clause.
- When an I/O path name is a local variable within a subprogram, it is closed when the subprogram is exited by SUBEND, SUBEXIT, RETURN..`expression`, or ON-`event`..RECOVER.
- When any form of STOP occurs; or an END, or GET is executed.

I/O path names that *are* included in a COM statement remain open and valid during a GET, STOP, END, or simple SCRATCH. I/O path names in COM are only closed at the following times:

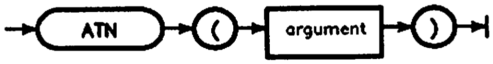
- When they are explicitly closed; for example, `ASSIGN @File TO *`

- When a GET, or EDIT operation brings in a program that has a COM statement that does not exactly match the COM statement containing the open I/O path names.

ATN

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the principal value which has a tangent equal to the argument. This is the arctangent function.



Item	Description/Default	Range Restrictions
argument	numeric expression	within valid ranges of INTEGER or REAL data types for INTEGER and REAL arguments

Examples Statements

```
Angle=ATN(Tangent)
PRINT "Angle = ";ATN(-1.5)
```

Semantics

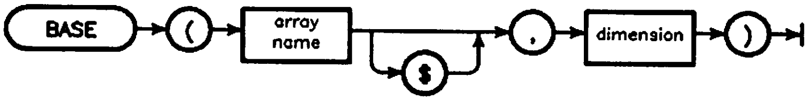
If the argument is REAL or INTEGER, the value returned is REAL.

The angle mode (RAD or DEG) for REAL and INTEGER arguments indicates whether you should interpret the value returned in degrees or radians. If the current angle mode is DEG, the range of the result is −90° to 90°. If the current angle mode is RAD, the range of the result is $-\pi\ /2$ to $+\pi\ /2$ radians. The angle mode is radians unless you specify degrees with the DEG statement.

BASE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the lower subscript bound of a dimension of an array. This value is always an INTEGER.



Item	Description	Range
array name	name of an array	any valid name
dimension	numeric expression, rounded to an integer	1 thru 6; ≤ the RANK of the array

Example Statements

```
Lowerbound=BASE(Array$,1)
Upperbound(2)=BASE(A,2)+SIZE(A,2)-1
```

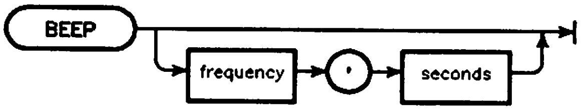
BDAT

See the CREATE BDAT statements.

BEEP

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement produces one of 64 audible tones.



Item	Description	Recommended Range
frequency	numeric expression, rounded to the nearest tone; Default = 1220.7	81 thru 5208
seconds	numeric expression, rounded to the nearest hundredth; Default = 0.2	.01 thru 2.55

Example Statements

```
BEEP 81.38*Tone,.5
BEEP
```

Semantics

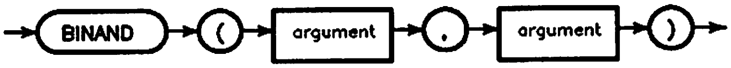
The frequency and duration of the tone are subject to the resolution of the built in tone generator. If the frequency specified is larger than 5167.63, a tone of 5208.32 is produced. If it is less than 40.69, it is considered to be a 0 and no tone is produced.

Note that low frequency tones may be difficult to hear.

BINAND

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the value of a bit-by-bit, logical AND of its arguments.



Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 thru +32 767

Example Statements

```
Low_4_bits=BINAND(Byte,15)
IF BINAND(Stat,8) THEN Bit_3_set
```

Semantics

The arguments for this function are represented as 16-bit two's-complement integers. Each bit in an argument is AND'ed with the corresponding bit in the other argument. The results of all the AND's are used to construct the integer which is returned.

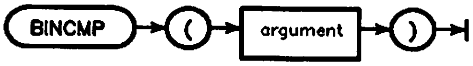
For example, the statement `Ctrl_word=BINAND(Ctrl_word,-9)` clears bit 3 of Ctrl_word without changing any other bits.

```
       bit 15          bit 0
         |               |
         V               V
  12 =  00000000 00001100  old Ctrl_word
  -9 =  11111111 11110111  mask to clear bit 3
        -----------------
   4 =  00000000 00000100  new Ctrl_word
```

BINCMP

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the value of the bit-by-bit complement of its argument.



Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 thru +32 767

Example Statements

```
True=BINCMP(Inverse)
PRINT X,BINCMP(X)
```

Semantics

The argument for this function is represented as a 16-bit, two's-complement integer. Each bit in the representation of the argument is complemented, and the resulting integer is returned.

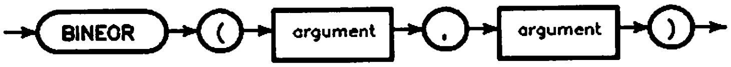
For example, the complement of −9 equals +8:

```
       bit 15          bit 0
        |               |
        V               V
-9 =   11111111 11110111
       -----------------
+8 =   00000000 00001000
```

BINEOR

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the value of a bit-by-bit, exclusive OR of its arguments.



Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 thru +32 767

Example Statements

```
Toggle=BINEOR(Toggle,1)
True_byte=BINEOR(Inverse_byte,255)
```

Semantics

The arguments for this function are represented as 16-bit, two's-complement integers. Each bit in an argument is exclusively OR'ed with the corresponding bit in the other argument. The results of all the exclusive OR's are used to construct the integer which is returned.

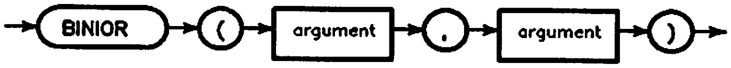
For example, the statement `Ctrl_word=BINEOR(Ctrl_word,4)` inverts bit 2 of Ctrl_word without changing any other bits.

```
         bit 15          bit 0
           |               |
           V               V
    12 =  00000000 00001100  old Ctrl_word
     4 =  00000000 00000100  mask to invert bit 2
           ----------------
     8 =  00000000 00001000  new Ctrl_word
```

BINIOR

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the value of a bit-by-bit, inclusive OR of its arguments.



Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 thru +32 767

Example Statements

```
Bits_set=BINIOR(Value1,Value2)
Top_on=BINIOR(All_bits,2^15)
```

Semantics

The arguments for this function are represented as 16-bit, two's-complement integers. Each bit in an argument is inclusively OR'ed with the corresponding bit in the other argument. The results of all the inclusive OR's are used to construct the integer which is returned.

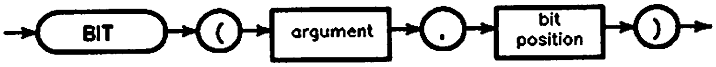
For example, the statement `Ctrl_word=BINIOR(Ctrl_word,6)` sets bits 1 & 2 of Ctrl_word without changing any other bits.

```
        bit 15           bit 0
         |                |
         V                V
  19 =  00000000 00010011  old Ctrl_word
   6 =  00000000 00000110  mask to set bits 1 & 2
        -----------------
  23 =  00000000 00010111  new Ctrl_word
```

BIT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns a 1 or 0 representing the value of the specified bit of its argument.



Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 thru +32 767
bit position	numeric expression, rounded to an integer	0 thru 15

Example Statements

```
Flag=BIT(Info,0)
IF BIT(Word,Test) THEN PRINT "Bit #";Test;"is set"
```

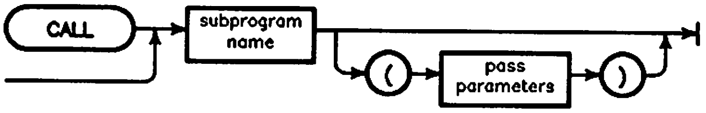
Semantics

The argument for this function is represented as a 16-bit, two's-complement integer. Bit 0 is the least-significant bit, and bit 15 is the most-significant bit.

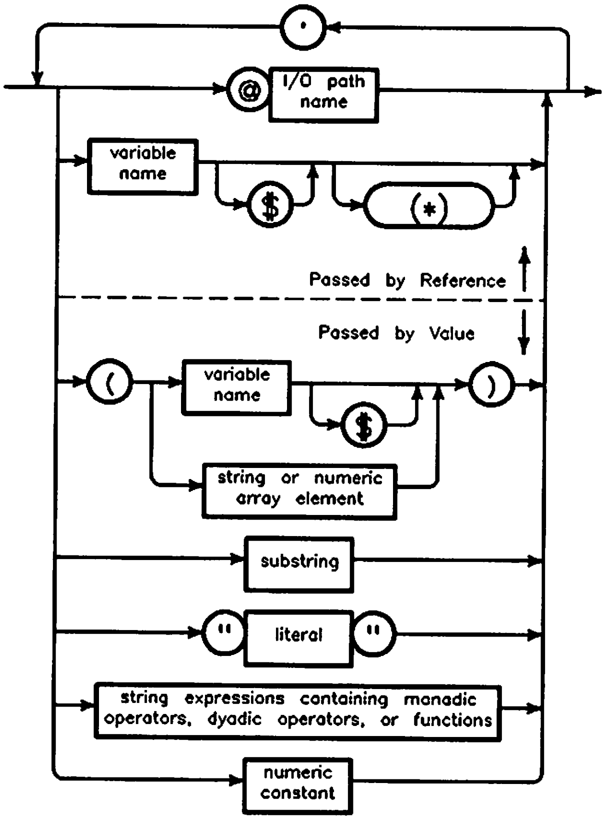
CALL

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement transfers program execution to the specified SUB subprogram and may pass parameters to the subprogram. SUB programs are created with the SUB statement. (Also see the ON ... statements.)



pass parameters:



Item	Description	Range
subprogram name	name of the SUB subprograms to be called	any valid name
I/O path name	name assigned to a device, devices, or mass storage file	any valid name (see ASSIGN)
variable name	name of a string or numeric variable	any valid name
substring	string expression containing substring notation	(see Glossary)
literal	string constant composed of characters from the keyboard	—
numeric constant	numeric quantity expressed using numerals, and optionally a sign, a decimal point, and/or exponent notation	—

Example Statements

```
CALL Process(Ref,(Value),@Path)
Process(Ref,(Value),@Path)
CALL Transform(Array(*))
IF Flag THEN CALL Special
```

Semantics

A subprogram may be invoked by a stored program line, or by a statement executed from the keyboard. Invoking a subprogram changes the program context. Subprograms may be invoked recursively. The keyword CALL may be omitted if it would be the first word in a program line. However, the keyword CALL is required in all other instances (such as a CALL from the keyboard and a CALL in an IF ... THEN ... statement).

The pass parameters must be of the same type (numeric, string, or I/O path name) as the corresponding parameters in the SUB statement. Numeric values passed by value are converted to the numeric type (REAL, or INTEGER) of the corresponding formal parameter. Variables passed by reference must match the corresponding parameter in the SUB statement exactly. An entire array may be passed by reference by using the asterisk specifier.

If there is more than one subprogram with the same name, the lowest-numbered subprogram is invoked by a CALL.

Program execution generally resumes at the line following the subprogram CALL. However, if the subprogram is invoked by an event-initiated branch (such as ON ERROR, ON KEY, etc.), program execution resumes at the point at which the event-initiated branch was permitted.

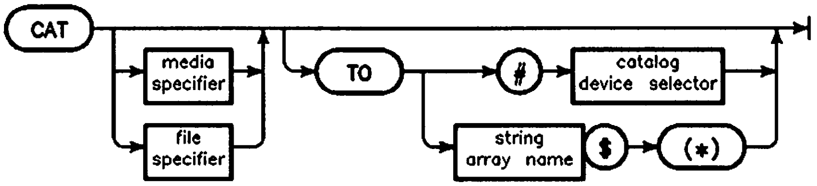
CASE

See the SELECT ... CASE construct.

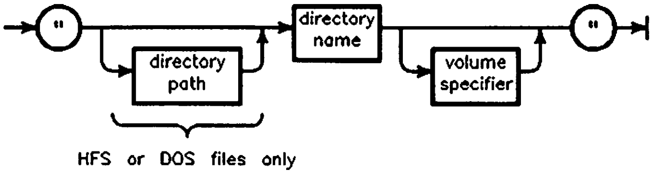
CAT

Keyboard Executable Yes
Programmable Yes
In an IF ... THEN ... Yes

This statement lists the contents of a mass storage directory.



literal form of directory specifier:



Item	Description	Range
directory specifier	string expression; Default=MASS STORAGE IS directory	(see MASS STORAGE IS)
volume specifier	string expression; Default=MASS STORAGE IS volume	(see MASS STORAGE IS)
directory path	literal	(see MASS STORAGE IS)
file name	literal	depends on volume's format (see Glossary)
LIF protect code	literal; first two non-blank characters are significant	> not allowed
catalog device selector	numeric expression, rounded to an integer; Default=PRINTER IS device	(see Glossary)
string array name	name of a string array (see text)	any valid name

Example Statements

```
CAT
CAT TO #701
CAT ":,700,1"

CAT "../../.."
CAT "Dir1/Dir2"

CAT TO String_array$(*)
```

Semantics

A directory entry is listed for each file in the specified directory. The catalog shows information such as the name of each file, whether or not it is protected, the file's type and length, and the number of bytes per logical record.

The file types recognized in HP Instrument BASIC are: ASCII, BDAT (BASIC data), BIN (binary program), HP-UX, PROG (BASIC program), and SYSTM (HP Series 200/300 BASIC operating system). An ID number is listed for any unrecognized file types.

LIF Catalogs

The LIF catalog format is shown below. This catalog format requires that the PRINTER IS device have the capability of displaying 65 or more characters. If the printer width is less than 65, the DATE and TIME columns are omitted.

```
:CS80,700
VOLUME LABEL: B9836
FILE NAME PRO TYPE  REC/FILE BYTE/REC   ADDRESS    DATE    TIME

MyProg        PROG        14      256        16  23-May-87 7:58
VisiComp      ASCII       29      256        30   8-Apr-87 6:00
GRAPH         BIN        171      256        59   1-May-87 1:00
GRAPHX        BIN        108      256       230  10-Aug-87 9:00
```

The first line of the catalog shows the volume specifier (:CS80,700 in this example).

The second line shows the volume label—a name, containing up to 6 characters, stored on the media (B9836 in this example).

The third line labels the columns of the remainder of the catalog. Here is what each column means:

FILE NAME lists the names of the files in the directory (up to 10 characters).

PRO indicates whether the file has a protect code (* is listed in this column if the file has a protect code).

FILE TYPE lists the type of each file.

REC/FILE indicates the number of records in the file.

BYTE/REC indicates the record size.

ADDRESS indicates the number of the beginning sector in the file.

DATE indicates when the date the file was last modified.

TIME indicates the time the file was last modified.

HFS Catalogs

In order to perform a CAT of an HFS directory, you need to have R (read) and X (search) permissions on the directory to be cataloged, as well as X (search) permissions on all superior directories.

In order to perform a CAT of an HFS file, you need to have R (read) permission on the file to be cataloged, as well as X (search) permissions on all superior directories.

Here is a typical catalog listing of an HFS directory. Note that a 50 column display truncates this catalog listing after the column with TIME in it. Therefore, the PERMISSION, OWNER, and GROUP columns will be not be listed.

```
:CS80, 700
LABEL: MyVol
FORMAT: HFS
AVAILABLE SPACE:     60168
                FILE     NUM   REC     MODIFIED
FILE NAME       TYPE    RECS   LEN DATE           TIME PERMISSION OWNER GROUP
=============== ===== ====== ===== =============== ========== ===== =====
lost+found      DIR         0    32 19-Nov-86 10:47  RWXRWXRWX    18     9
FILEIOD         PROG      191   256 21-Nov-86  9:03  RW-RW-RW-    18     9
RBDAT           BDAT        2   256 21-Nov-86  9:10  RW-RW-RW-    18     9
CATTOSTR        PROG        2   256  1-Dec-86  8:02  RW-RW-RW-    18     9
```

The first line of the catalog shows the volume specifier (:CS80,700 in this example).

If the directory path specifier contains more characters than the display width, the last 49 or 79 characters (depending on the display width) are shown. An asterisk (*) as the left-most character in the path specifier indicates that leading characters were truncated for the display.

The second line shows the volume label—a name, containing up to 6 characters, stored on the media (MyVol in this example).

The third line shows the format of the disc (HFS in this example).

The fourth line lists the number of available 256-byte sectors on the disc (60168 in this example). If the sector size is 1024 bytes, then each 1024-byte sector would count as 4 256-byte sectors.

The fifth line labels the columns of the remainder of the catalog. Here is what each column means:

FILE NAME — Lists the name of the file.

FILETYPE — Lists the file's type (for instance, DIR specifies that the file is a directory; PROG specifies a BASIC program file; BDAT specifies a BASIC DATA file; etc.) if you have read permission. If you do not have read permission, the file type is left blank.

NUMRECS — number of **logical** records (the number of records allocated to the file when it was created). For a DIR file, this indicates the number of directory entries.

RECLEN — the **logical** record size (default is 256 bytes; BDAT files can have user-selected record lengths). For a DIR file, this indicates the size of the directory entry. You **cannot** specify record length for ASCII or HP-UX files. The record length for HP-UX files is 1.

MODIFIEDDATE TIME — the day and time when the file was last modified.

PERMISSION — specifies who has access rights to the file:

R — indicates that the file can be read;

W — indicates that the file can be written;

X — indicates that the file can be searched (meaningful for directories only).

There are 3 classes of user permissions for each file:

OWNER	(left-most 3 characters);
GROUP	(center 3 characters);
OTHER	(right-most 3 characters).

OWNER specifies the owner identifier for the file (for BASIC Workstation files, the default owner identifier is always 18). BASIC/UX shows the user id of the user that owns the file.

GROUP specifies the group identifier of the file or directory

MS-DOS Catalogs

Here is a typical catalog listing of an DOS directory.

```
DIRECTORY: \RANDOM:CS80, 700
LABEL: MyVol
FORMAT: DOS
AVAILABLE SPACE:     60168
                FILE    NUM   REC     MODIFIED
FILE NAME       TYPE   RECS   LEN DATE         TIME PERMISSION
============== ===== ====== ===== =============== ==========
lost+found     DIR       0    32 19-Nov-86 10:47 RWXRWXRWX
FILEIOD        PROG    191   256 21-Nov-86  9:03 RW-RW-RW-
RBDAT          BDAT      2   256 21-Nov-86  9:10 RW-RW-RW-
CATTOSTR       PROG      2   256  1-Dec-86  8:02 RW-RW-RW-
```

The first line of the catalog shows the volume specifier (:CS80,700 in this example).

If the directory path specifier contains more characters than the display width, the last 49 or 79 characters (depending on the display width) are shown. An asterisk (*) as the left-most character in the path specifier indicates that leading characters were truncated for the display.

The second line shows the volume label—a name, containing up to 6 characters, stored on the media (MyVol in this example).

The third line shows the format of the disc (DOS in this example).

The fourth line lists the number of available 256-byte sectors on the disc (60168 in this example). If the sector size is 1024 bytes, then each 1024-byte sector would count as 4 256-byte sectors.

The fifth line labels the columns of the remainder of the catalog. Here is what each column means:

FILE NAME Lists the name of the file.

FILETYPE Lists the file's type (for instance, DIR specifies that the file is a directory; PROG specifies a BASIC program file; BDAT specifies a BASIC DATA file; etc.).

NUMRECS number of **logical** records (the number of records allocated to the file when it was created). For a DIR file, this indicates the number of directory entries.

RECLEN the **logical** record size (default is 256 bytes; BDAT files can have user-selected record lengths). For a DIR file, this indicates the size of the directory entry. You **cannot** specify record length for ASCII or HP-UX files. The record length for HP-UX files is 1.

MODIFIEDDATE TIME — the day and time when the file was last modified.

CAT to a Device

When the symbol # is included in a CAT statement, the numeric expression following this symbol must be a device selector. The catalog listing is sent to the device specified by this expression.

CAT to a String Array

The catalog can be sent to a string array. The array must be one-dimensional, and each element of the array must contain at least 80 characters for a directory listing. If the directory information does not fill the array, the remaining elements are set to null strings. If the directory information "overflows" the array, the overflow is not reported as an error. When a CAT of a mass storage directory is sent to a string array, the catalog's format is different than when sent to a device. This format is shown below. Protect status is shown by letters, instead of an asterisk. An unprotected file has the entry MRW in the PUB ACC (public access) column. A protected BDAT file has no entry in that column. Other types of protected files show R (read access). In addition to the standard information, this format also shows OPEN in the OPEN STAT column when a file is currently assigned.

```
:CS80,702,0
VOLUME LABEL: B9836
FORMAT: LIF
AVAILABLE SPACE:    11
                      SYS  FILE   NUMBER   RECORD      MODIFIED      PUB OPEN
FILE NAME            TYPE  TYPE  RECORDS   LENGTH DATE           TIME ACC STAT
================ === ==== ===== ======== ======== ================== === ====
SYSTEM_BA5         1 98X6 SYSTM     1024      256 29 Nov 86 15:24:55 MRW
AUTOST             1 98X6  PROG       38      256 29 Nov 86 09:25:07 MRW
```

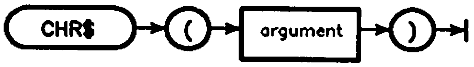
To aid in accessing the catalog information in a string, the following table gives the location of some important fields in the string.

Field	Position (in String)
File Name	1 thru 21
File Type	32 thru 36
Number of Records	37 thru 45
Record Length	46 thru 54
Time Stamp	56 thru 71
Public Access Capabilities	73 thru 75
Open Status	77 thru 80

CHR$

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function converts a numeric value into an ASCII character. The low order byte of the 16-bit integer representation of the argument is used; the high order byte is ignored. A table of ASCII characters and their decimal equivalent values may be found in the back of this book.



Item	Description	Range
argument	numeric expression, rounded to an integer	0 thru 255

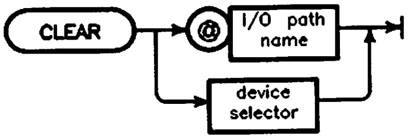
Example Statements

```
A$[Marker;1]=CHR$(Digit+128)
Esc$=CHR$(27)
```

CLEAR

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement clears HP-IB devices or Data Communications interfaces.



Item	Description	Range
I/O path name	name assigned to a device or devices	any valid name (see ASSIGN)
device selector	numeric expression, rounded to an integer	(see Glossary)

Example Statements

```
CLEAR 7
CLEAR Isc*100+Address
CLEAR @Source
```

Semantics

This statement allows the computer to put all or only selected HP-IB devices into a pre-defined, device-dependent state. The computer must be the active controller to execute this statement. When primary addresses are specified, the bus is reconfigured and the SDC (Selected Device Clear) message is sent to all devices which are addressed by the LAG message.

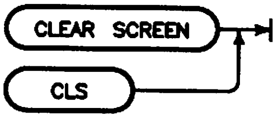
Summary of Bus Actions

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active/Controller	ATN DCL	ATN MTA UNL LAG SDC	ATN DCL	ATN MTA UNL LAG SDC
Not Active Controller	Error	Error	Error	Error

CLEAR SCREEN

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN	Yes

This statement clears the contents of the alpha display.



Example Statements

```
CLS
CLEAR SCREEN
IF Loop_count=1 THEN CLEAR SCREEN
```

Semantics

When this statement is executed it clears alpha memory.

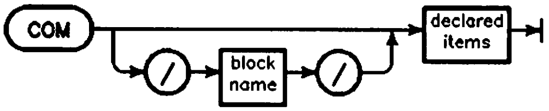
CLS

See the CLEAR SCREEN statement.

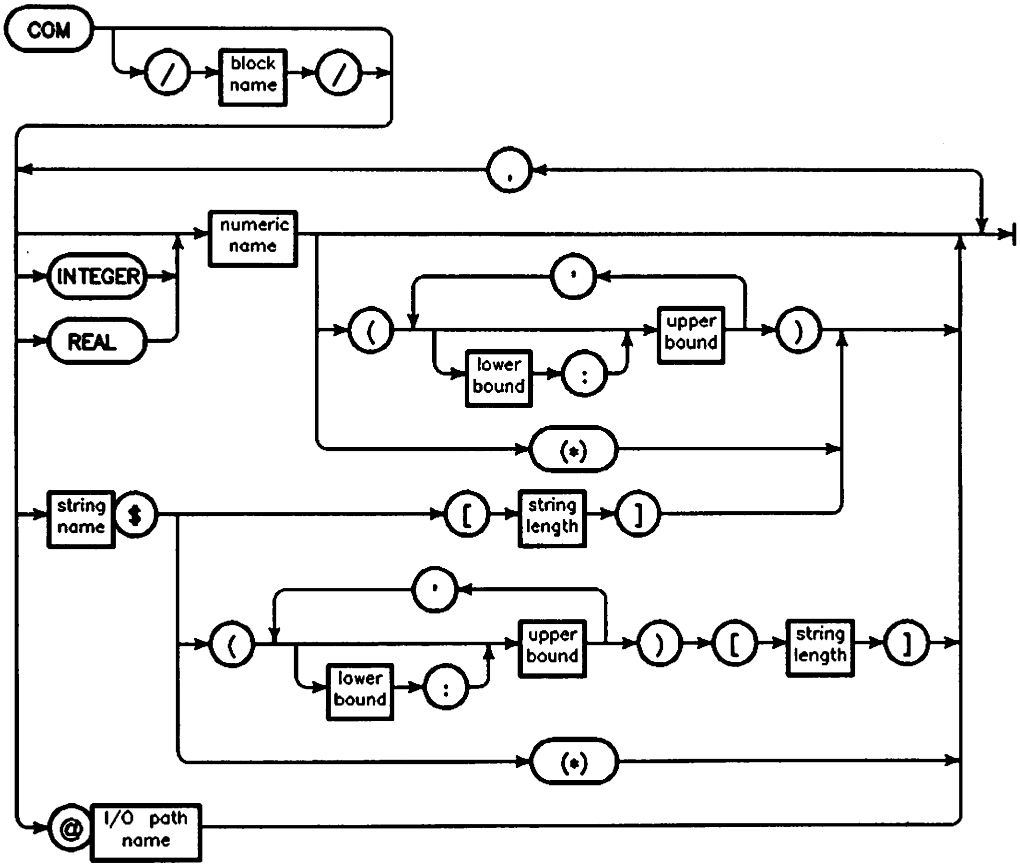
COM

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN	No

This statement dimensions and reserves memory for variables in a special "common" memory area so more than one program context can access the variables.



Expanded diagram:



Item	Description	Range
block name	name identifying a labeled COM area	any valid name
numeric name	name of a numeric variable	any valid name
string name	name of a string variable	any valid name
lower bound	integer constant; Default = 0	−32 767 thru +32 767 (see "array" in Glossary)
upper bound	integer constant	−32 767 thru +32 767 (see "array" in Glossary)
string length	integer constant	1 thru 32 767
I/O path name	name assigned to a device, devices, mass storage file, or buffer	any valid name (see ASSIGN)

Example Statements

```
COM X,Y,Z
COM /Graph/ Title$,@Device,INTEGER Points(*)
COM INTEGER I,J,REAL Array(-128:127)
```

Semantics

Storage for COM is allocated at prerun time. Changing the definition of the COM space is accomplished by a full program prerun. This can be done by:

- Executing a RUN command when no program is running.
- Executing any GET from a program.
- Executing a GET command that tells program execution to begin (such as GET "File",1).

When COM allocation is performed at prerun, the new program's COM area is compared to the COM area currently in memory. When comparing the old and new areas, HP Instrument BASIC looks first at the types and structures declared in the COM statements. If the "text" indicates that there is no way the areas could match, then those areas are considered mismatched. If the declarations are consistent, but the shape of an array in memory does not match the shape in a new COM declaration, the arrays are redimensioned wherever possible to match the new declarations. Any variable values are left intact. All other COM areas are rendered undefined, and their storage area is not recovered by HP Instrument BASIC. New COM variables are initialized at prerun: numeric variables to 0, string variables to the null string.

Each context may have as many COM statements as needed (within the limits stated below), and COM statements may be interspersed between other statements. COM variables do not have to have the same names in different contexts. Formal parameters of subprograms are not allowed in COM statements. A COM mismatch between contexts causes an error.

The total number of COM elements is limited to a maximum memory usage of $2^{24}-1$, or 16 777 215, bytes (or limited by the amount of available memory, whichever is less).

If a COM area requires more than one statement to describe its contents, COM statements defining that block may not be intermixed with COM statements defining other COM areas.

Numeric variables in a COM list can have their type specified as either REAL, or INTEGER. Specifying a variable type implies that all variables which follow in the list are of the same type. The type remains in effect until another type is specified. String variables and I/O path names are considered a type of variable and change the specified type. Numeric variables are assumed to be REAL unless their type has been changed to INTEGER.

COM statements (blank or labeled) in different contexts which refer to an array or string must specify it to be of the same size and shape. The lowest-numbered COM satement containing an array or string name must explicitly specify the subscript bounds and/or string length. Subsequent COM statements can reference a string by name only or an array only by using an asterisk specifier (*).

No array can have more than six dimensions. The lower bound value must be less than or equal to the upper bound value. The default lower bound is 0.

Unlabeled or Blank COM

Blank COM does not contain a block name in its declaration. Blank COM (if it is used) must be created in a main context. The main program can contain any number of blank COM statements (limited only by available memory). Blank COM areas can be accessed by subprograms, if the COM statements in the subprograms agree in type and shape with the main program COM statements.

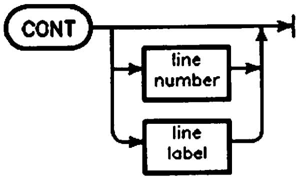
Labeled COM

Labeled COM contains a name for the COM area in its declaration. Memory is allocated for labeled COM at prerun time according to the lowest-numbered occurrence of the labeled COM statement. Each context which contains a labeled COM statement with the same label refers to the same labeled COM block.

CONT

Keyboard Executable	Yes
Programmable	No
In an IF ... THEN ...	No

This command resumes execution of a paused program at the specified line.



Item	Description	Range
line number	integer constant identifying a program line; Default = next program line	1 thru 32 766
line label	name identifying a program line	any valid name

Example Statements

```
CONT 550
CONT Sort
```

Semantics

Continue can be executed by pressing the (CONTINUE) key (if implemented on your keyboard), or by executing a CONT command. Variables retain their current values whenever CONT is executed. CONT causes the program to resume execution at the next statement which would have occurred unless a line is specified.

When a line label is specified, program execution resumes at the specified line, provided that the line is in either the main program or the current subprogram. If a line number is specified, program execution resumes at the specified line, provided that the line is in the current program context. If there is no line in the current context with the specified line number, program execution resumes at the next higher-numbered line. If the specified line label does not exist in the proper context, an error results.

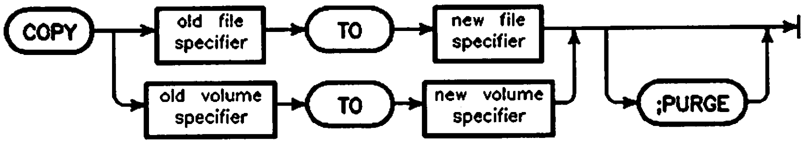
CONTROL

(This keyword is used in PASS CONTROL.)

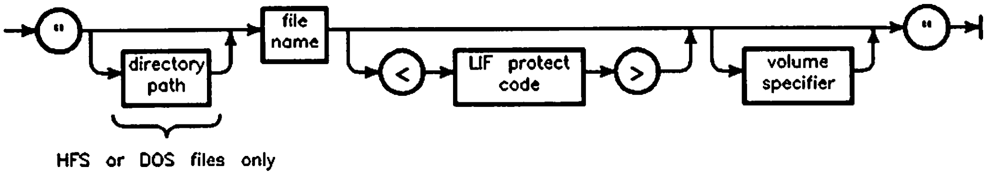
COPY

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement allows copying of an individual file or an entire disc.



literal form of file specifier:



Item	Description	Range
file specifier	string expression	(see drawing)
volume specifier	string expression	(see MASS STORAGE IS)
directory path	literal	(see MASS STORAGE IS)
file name	literal	depends on volume's format (see Glossary)
LIF protect code	literal; first two non-blank characters are significant	> not allowed
volume specifier	string expression	(see MASS STORAGE IS)

Example Statements

```
COPY "OLD_FILE" TO "New_file"
COPY File$ TO File$&Other_volume$

COPY "/Dir_1/File_1" TO "Dir_3/File_1"
COPY Dir_path$&File$&Vol$& TO "File:,700"

COPY Left_disc$ TO Right_disc$
COPY ":,700" TO ":,700,1"
COPY ":,4,1" TO ":,4,0"
```

Semantics

The contents of the old file is copied into the new file, and a directory entry is created. A protect code (LIF directories) may be specified for the new file, to prevent accidental erasure, etc. The old file and the new file can exist in the same directory, but the new file name must be unique.

An error is returned if there is not enough room on the destination device or if the new file name already exists in the destination directory.

If the mass storage volume specifier (msvs) is omitted from a file specifier, the MASS STORAGE IS device is assumed.

If the directory path is also omitted, the MASS STORAGE IS directory is assumed.

Copying an Entire LIF or HFS Volume

LIF and HFS volumes can be duplicated if the destination volume is as large as, or larger than, the source volume. COPY from a larger capacity volume to a smaller capacity volume is only possible when the amount of data on the larger will fit on the smaller. The directory and any files on the destination volume are destroyed. The directory size on the destination volume becomes the same size as that on the source media.

When copying an entire volume, the volume specifiers must be unique. File names are not allowed. Disc-to-disc copy time is dependent on media type and interleave factors.

HFS Permissions

With HFS, COPY allows copying of individual files and volumes. HFS directories cannot be copied.

In order to COPY a file on an HFS volume, you need to have R (read) permission on the source file, as well as X (search) permission on the parent directory and all other superior directories. In addition, you will need W (write) and X (search) permission on the destination file's parent directory, as well as X (search) permission on all other superior directories.

HFS and MS-DOS File Headers

When copying a file from LIF to HFS, a special header is added to the beginning of that file. This action is taken because that is the only way to "type" files (which would otherwise be "typeless"). When copying a file from HFS to LIF volumes, this file header is removed (since these volumes have typed files). Note that HP Instrument BASIC handles the file headers automatically and requires no special treatment in programs that use these files.

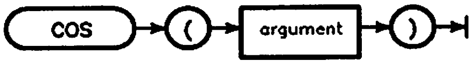
Similarly, when copying a file from LIF to DOS, an additional 512 byte header is added to the beginning of that file. When copying a file from DOS to LIF volumes, this file header is also removed.

Copying HP-UX files to DOS has no effect on the files.

COS

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the cosine of the angle represented by the argument. The range of the returned real value is −1 through +1.



Item	Description/Default	Range Restrictions
argument	numeric expression in current units of angle when INTEGER or REAL argument	host device dependent

Examples Statements

```
Cosine=COS(Angle)
PRINT COS(X+45)
```

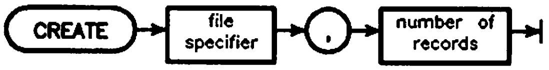
Semantics

If the argument is REAL or INTEGER, the value returned is REAL.

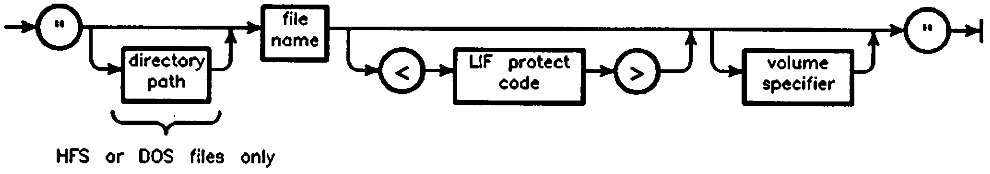
CREATE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement creates an HP-UX or an MS-DOS file.



literal form of file specifier:



Item	Description	Range
file specifier	string expression	(see drawing)
directory path	literal	(see MASS STORAGE IS)
file name	literal	depends on volume's format (see Glossary)
LIF protect code	literal; first two non-blank characters are significant	> not allowed
volume specifier	literal	(see MASS STORAGE IS)
number of records	numeric expression, rounded to an integer	1 thru $2^{31} - 1$

Example Statements

```
CREATE File_spec$,N_records
CREATE "HPUX_file",12
CREATE "OnLIF<pc>",50*N
```

Semantics

CREATE creates a new file of type HP-UX (on an HFS file system) or DOS (on an MS-DOS file system) on the default or specified volume or hierarchical directory. A corresponding directory entry is also made. The name of the newly created file must be unique within its directory. CREATE does not open the file; that is performed by ASSIGN. In the event of an error, no directory entry is made and the file is not created.

The number of records parameter specifies how many **logical** records are to be initially allocated to the file. The logical record size is always 1 for HP-UX files. On LIF volumes, the number of records allocated for the file is fixed; however, with HFS volumes files are extensible (see the following explanation of extensible files). Similarly, DOS files on DOS volumes, are also extensible.

The data representation used in the file depends on the FORMAT option used in the ASSIGN statement used to open the file. See ASSIGN for details.

Extensible Files (HFS and MS-DOS Volumes Only)

If the file is created on an HFS volume, the file is "extensible". With HFS volumes, the initial size of the file is 0, but the file will automatically be extended as many bytes as necessary whenever an OUTPUT operation would otherwise overflow the file. "Preallocating" the file on HFS volumes (initially creating a file of sufficient size) will improve the data transfer rate with extensible files, because the file system will not have to extend the file during data transfer operations.

DOS files created on a DOS volume are also extensible. However, ASCII and BDAT files on DOS volumes are not extensible.

LIF Protect Codes

A protect code is not allowed on an HP-UX file.

HFS Permissions

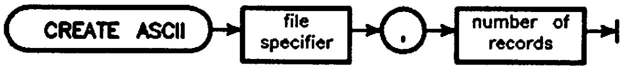
In order to create a file on an HFS volume, you need to have W (write) and X (search) permission of the immediately superior directory, as well as X (search) permission on all other superior directories.

When a file is created on an HFS volume, access permission bits are set to `RW-RW-RW-`.

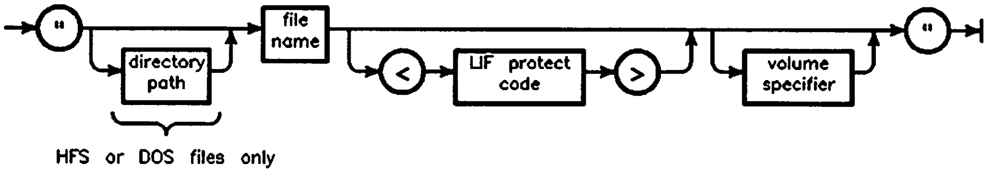
CREATE ASCII

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement creates an ASCII file.



literal form of file specifier:



Item	Description	Range
file specifier	string expression	(see drawing)
number of records	numeric expression, rounded to an integer	1 thru $(2^{31} - 1)/256$
directory path	literal	(see MASS STORAGE IS)
file name	literal	depends on volume's format (see Glossary)
volume specifier	literal	(see MASS STORAGE IS)

Example Statements

```
CREATE ASCII "TEXT",100
CREATE ASCII Name$&":,700,1",Length
CREATE ASCII "/Dir1/Dir2/AsciiFile",25
```

Semantics

CREATE ASCII creates a new ASCII file and directory entry on the mass storage media. The name of the newly created ASCII file must be unique within its containing directory. CREATE ASCII does not open the new file; that is performed by the ASSIGN statement. In the event of an error, no directory entry is made and the file is not created.

The physical records of an ASCII file have a fixed length of 256 bytes; logical records have variable lengths, which are automatically determined when the OUTPUT, SAVE, or RE-SAVE statements are used.

Extensible Files (HFS Volumes Only)

If the file is created on an HFS volume, the file is "extensible". With HFS volumes, the initial size of the file is the size specified in the CREATE ASCII statement, but the file will automatically be extended as many bytes as necessary whenever an OUTPUT operation would otherwise overflow the file. "Preallocating" the file on an HFS volume (initially creating a file of sufficient size) will improve the data transfer rate with extensible files, because the file system will not have to extend the file during data transfer operations.

Note MS-DOS ASCII files are NOT extensible.

LIF Protect Codes

On a LIF disc, a protect code is not allowed on an ASCII file. Including a protect code in the CREATE ASCII statement will give an error.

Permissions and File Headers

In order to create a file on an HFS volume, you need to have W (write) and X (search) permissions on the immediately superior directory, as well as X (search) permissions on all other superior directories.

On HFS volumes, access permission bits are set to `RW-RW-RW-` when an ASCII file is created.

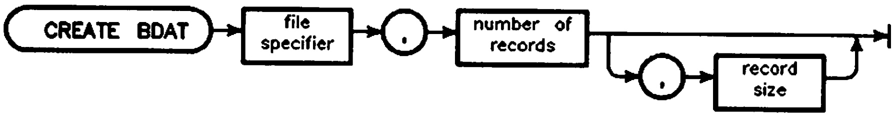
On an HFS volume, the first 512 bytes of an ASCII file are used by the BASIC file system to describe the file's type (this is the only way for BASIC to create a "typed" file on an HFS volume, since HFS files are otherwise "typeless"). This file header is handled automatically by HP Instrument BASIC, but it should be skipped when reading and writing the file with other HP-UX languages.

Note On an MS-DOS volume, CREATE ASCII produces a file with an additional 512 byte header.

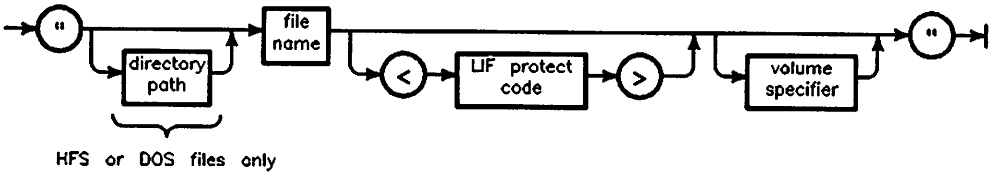
CREATE BDAT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement creates a BDAT file.



literal form of file specifier:



Item	Description	Range
file specifier	string expression	(see drawing)
number of records	numeric expression, rounded to an integer	1 thru (2^{31} – 769)/(record size)
record size	numeric expression, rounded to next even integer (except 1), which specifies bytes/record; Default = 256	1 thru 65 534
directory path	literal	(see MASS STORAGE IS)
file name	literal	depends on volume's format (see Glossary)
LIF protect code	literal; first two non-blank characters are significant	> not allowed
volume specifier	string expression	(see MASS STORAGE IS)

Example Statements

```
CREATE BDAT "File",Records,Rec_size
CREATE BDAT "George",48
CREATE BDAT "Protected<PC>",Length,128
CREATE BDAT Name$&Volume$,Bytes,1
CREATE BDAT "/Dir1/Dir2/BDATfile",25,128
```

Semantics

CREATE BDAT creates a new BDAT file and directory entry on the mass storage media. The name of the newly created BDAT file must be unique within its containing directory. CREATE BDAT does not open the file; that is performed by the ASSIGN statement. In the event of an error, no directory entry is made and the file is not created.

A sector at the beginning of the file is reserved for system use. This sector cannot be directly accessed by HP Instrument BASIC programs.

Extensible Files (HFS Volumes Only)

If the file is created on an HFS volume, the file is "extensible". With HFS volumes, the initial size of the file is the size specified in the CREATE BDAT statement, but the file will automatically be extended as many bytes as necessary whenever an OUTPUT operation would otherwise overflow the file.

Note MS-DOS BDAT files are NOT extensible.

LIF Protect Codes

On LIF volumes, an optional protect code may be specified; the first two characters become the protect code of the file.

Permissions and File Headers

In order to create a file on an HFS volume, you need to have W (write) and X (search) permission of the immediately superior directory, as well as X (search) permission on all other superior directories.

When a file is created on an HFS volume, access permission bits are set to `RW-RW-RW-`.

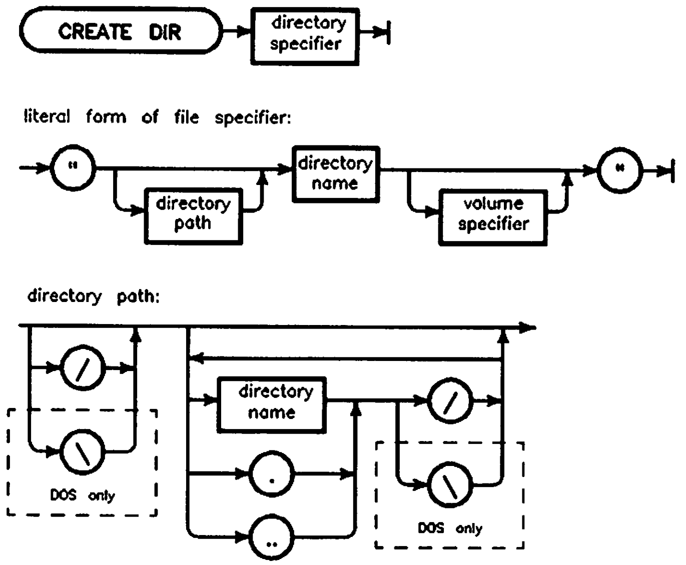
On HFS volumes, the first 512 bytes of a BDAT file are used by the BASIC file system to describe the file's type (this is the only way for HP Instrument BASIC to create a "typed" file on an HFS volume, since HFS files are otherwise "typeless"). This file header is handled automatically by HP Instrument BASIC, but it should be skipped when reading and writing the file with other HP-UX languages.

On a DOS disk, CREATE BDAT produces a file with an additional 512 byte header.

CREATE DIR

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This command creates an HFS directory on an HFS volume, or a DOS directory on a DOS volume.



This statement creates a directory in either the current working directory or a specified directory.

Item	Description	Range
directory specifier	string expression	(see drawing)
directory path	literal	(see drawing)
directory name	literal	depends on volume's format (14 characters for HFS; 255 characters for long file name systems; see Glossary for details)
volume specifier	literal	(see MASS STORAGE IS)

Example Statements

```
CREATE DIR "Under_work_dir"
CREATE DIR "/Level1/Level2/New_dir"
CREATE DIR "Dir3/Dir4:,700"
CREATE DIR "Dir3\Dir4:,700"          DOS Volume Only
```

Semantics

This statement creates a directory and a corresponding directory entry in the current working directory or specified directory. The DIR file, or directory, keeps information on files and directories immediately subordinate to itself. The name of the newly created directory must be unique within its containing directory.

If no directory path is included in the directory specifier, the directory is created within the current working directory (the directory specified in the latest MASS STORAGE IS statement). To specify a target directory other than the current working directory, specify the directory path to the desired directory.

HFS Permissions

In order to create a directory on an HFS volume, you need to have W (write) and X (search) permission of the immediately superior directory, as well as X (search) permission on all other superior directories.

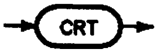
When a directory is created on an HFS volume, access permission bits are set toRWXRWXRWX.

As each directory or data file is created within an HFS directory, a 32-byte record identifying the addition is added to the DIR file. The length of this entry is variable for HFS long file name file systems.

CRT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This INTEGER function returns 1, the device selector of the alpha CRT display.



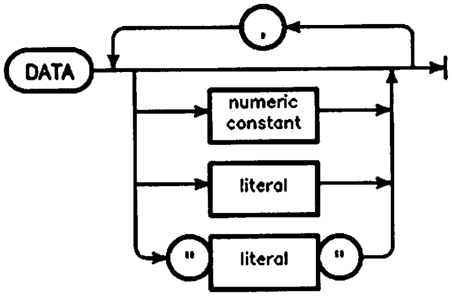
Example Statements

```
PRINTER IS CRT
OUTPUT CRT;Array(*)
```

DATA

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This statement contains data which can be read by READ statements.



Item	Description	Range
numeric constant	numeric quantity expressed using numerals, and optionally a sign, decimal point, or exponent notation	—
literal	string constant composed of characters from the keyboard	—

Example Statements

```
DATA 1,1.414,1.732,2
DATA word1,word2,word3
DATA "ex-point(!)","quote("")","comma(,)"
```

Semantics

A program or subprogram may contain any number of DATA statements at any locations. When a program is run, the first item in the lowest numbered DATA statement is read by the first READ statement encountered. When a subprogram is called, the location of the next item to be read in the calling context is remembered in anticipation of returning from the subprogram. Within the subprogram, the first item read is the first item in the lowest numbered DATA statement within the subprogram. When program execution returns to the calling context, the READ operations pick up where they left off in the DATA items.

A numeric constant must be read into a variable which can store the value it represents. The computer cannot determine the intent of the programmer; although attempting to read a string value into a numeric variable will generate an error, numeric constants will be read into string variables with no complaint. In fact, the computer considers the contents of all DATA statements to be literals, and processes items to be read into numeric variables with a VAL function, which can result in error 32 if the numeric data is not of the proper form (see VAL).

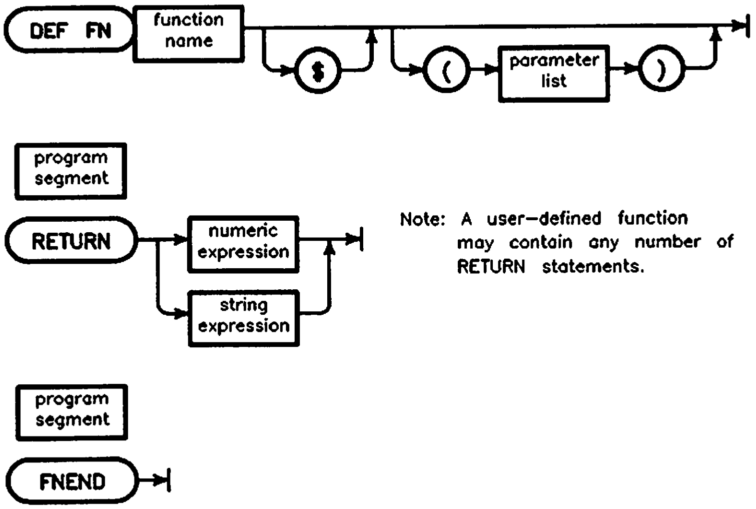
Unquoted literals may not contain quote marks (which delimit strings), commas (which delimit data items), or exclamation marks (which indicate the start of a comment).

Leading and trailing blanks are deleted from unquoted literals. Enclosing a literal in quote marks enables you to include any punctuation you wish, including quote marks, which are represented by a set of two quote marks.

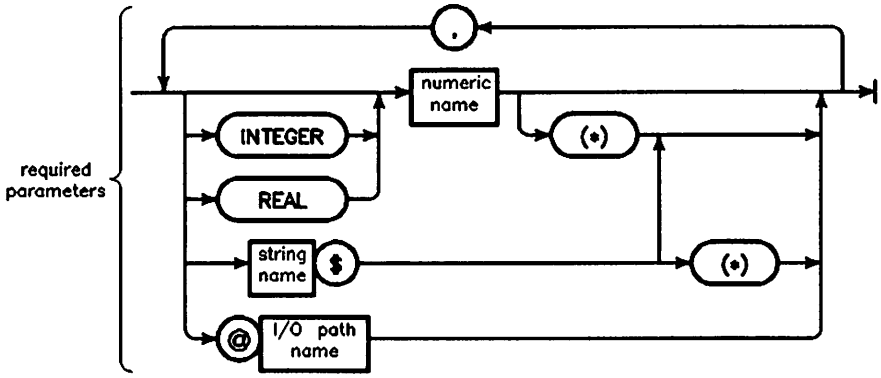
DEF FN

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This statement indicates the beginning of a function subprogram. It also indicates whether the function is string or numeric and defines the formal parameter list.



Note: A user-defined function may contain any number of RETURN statements.



Item	Description	Range
function name	name of the user-defined function	any valid name
numeric name	name of a numeric variable	any valid name
string name	name of a string variable	any valid name
I/O path name	name assigned to a device, devices, or mass storage file	any valid name (see ASSIGN)
program segment	any number of contiguous program lines not containing the beginning or end of a main program or subprogram	—

Example Statements

```
DEF FNTrim$(String$)
DEF FNTransform(@Printer,INTEGER Array(*))
DEF FNInvert (REAL A)
```

Semantics

User-defined functions must appear after the main program. The first line of the function must be a DEF FN statement. The last line must be an FNEND statement. Comments after the FNEND are considered to be part of the function.

Variables in a subprogram's formal parameter list may not be declared in COM or other declaratory statements within the subprogram. A user-defined function may not contain any SUB statements or DEF FN statements. User-defined functions can be called recursively and may contain local variables. A unique labeled COM must be used if the local variables are to preserve their values between invocations of the user-defined function.

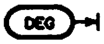
The RETURN <expression> statement is important in a user-defined function. If the program actually encounters an FNEND during execution (which can only happen if the RETURN is missing or misplaced), error 5 is generated. The <expression> in the RETURN statement must be numeric for numeric functions, and string for string functions. A string function is indicated by the dollar sign suffix on the function name. RETURN <integer expression> yields a real function result.

The purpose of a user-defined function is to compute a single value. While it is possible to alter variables passed by reference and variables in COM, this can produce undesirable side effects, and should be avoided. If more than one value needs to be passed back to the program, SUB subprograms should be used.

DEG

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement selects degrees as the unit of measure for expressing angles.



Semantics

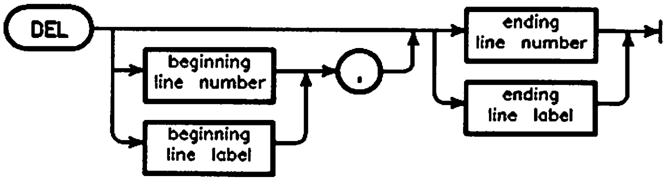
All functions which return an angle will return an angle in degrees. All operations with parameters representing angles will interpret the angle in degrees.

A subprogram "inherits" the angle mode of the calling context. If the angle mode is changed in a subprogram, the mode of the calling context is restored when execution returns to the calling context. If no angle mode is specified in a program, the default is radians (see RAD).

DEL

Keyboard Executable	Yes
Programmable	No
In an IF ... THEN ...	No

This command deletes program line(s).



Item	Description	Range
beginning line number	integer constant identifying a program line	1 thru 32 766
beginning line label	name of a program line	any valid name
ending line number	integer constant identifying a program line	1 thru 32 766
ending line label	name of a program line	any valid name

Example Statements

```
DEL 15
DEL Sort,9999
```

Semantics

DEL cannot be executed while a program is running. If DEL is executed while a program is paused, the computer changes to the stopped state.

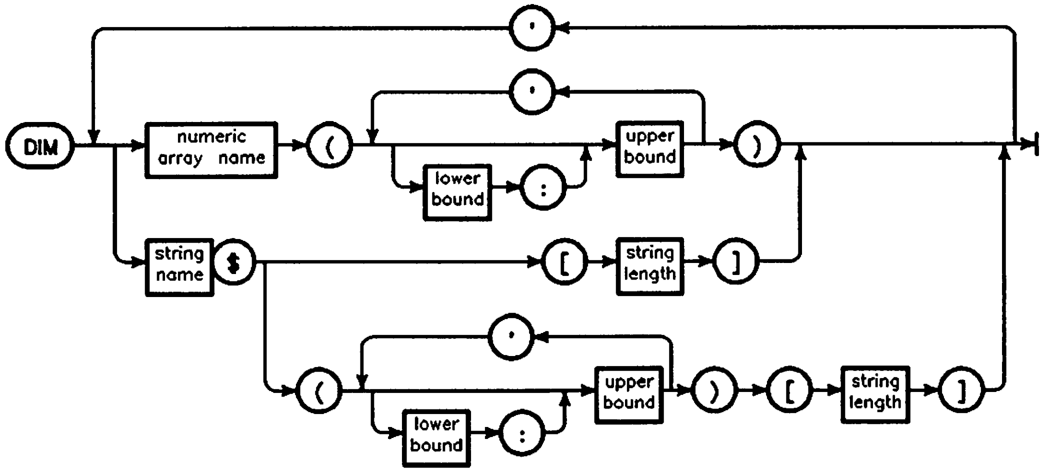
When a line is specified by a line label, the computer uses the lowest numbered line which has the label. If the label does not exist, error 3 is generated. An attempt to delete a non-existent program line is ignored when the line is specified by a line number. An error results if the ending line number is less then the beginning line number. If only one line is specified, only that line is deleted.

When deleting SUB and FN subprograms, the range of lines specified must include the statements delimiting the beginning and ending of the subprogram (DEF FN and FNEND for user-defined function subprograms; SUB and SUBEND for SUB subprograms), as well as all comments following the delimiting statement for the end of the subprogram. Contiguous subprograms may be deleted in one operation.

DIM

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This statement dimensions and reserves memory for REAL numeric arrays, strings and string arrays.



Item	Description	Range
numeric array name	name of a numeric array	any valid name
string name	name of a string variable	any valid name
lower bound	integer constant; Default = 0	−32 767 thru + 32 767 (see "array" in Glossary)
upper bound	integer constant	−32 767 thru +32 767 (see "array" in Glossary)
string length	integer constant	1 thru 32 767

Example Statements

```
DIM String$[100],Name$(12)[32]
DIM Array(-128:127,16)
```

Semantics

A program can have any number of DIM statements. The same variable cannot be declared twice within a program (variables declared in a subprogram are distinct from those declared in a main program, except those declared in COM). The DIM statements can appear anywhere within a program. Dimensioning occurs at pre-run or subprogram entry time.

No array can have more than six dimensions. Each dimension can have a maximum of 32 767 elements.

The total number of variables is limited by the fact that the maximum memory usage for *all* variables—INTEGER, REAL, and string—within any context is $2^{24}-1$, or 16 777 215, bytes (or limited by the amount of available memory, whichever is less).

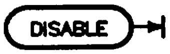
All numeric arrays declared in a DIM statement are REAL, and each element of type REAL requires 8 bytes of storage. A string requires one byte of storage per character, plus two bytes of overhead.

An undeclared array is given as many dimensions as it has subscripts in its lowest-numbered occurrence. Each dimension of an undeclared array has an upper bound of ten. Space for these elements is reserved whether you use them or not. Any time a lower bound is not specified, it defaults to 0.

DISABLE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement disables all event-initiated branches currently defined, except ON ERROR, and ON TIMEOUT.



Semantics

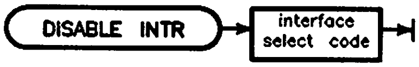
If an event occurs while the event-initiated branches are disabled, only the first occurrence of each event is logged; there is no record of how many of each type of event has occurred.

If event-initiated branches are enabled after being disabled, all logged events will initiate their respective branches if and when system priority permits. ON ERROR, and ON TIMEOUT branches are *not* disabled by DISABLE.

DISABLE INTR

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement disables interrupts from an interface by turning off the interrupt generating mechanism on the interface.



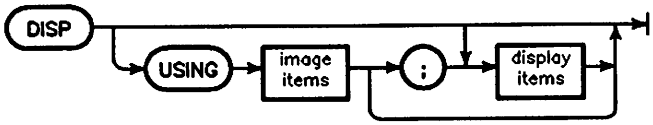
Example Statements

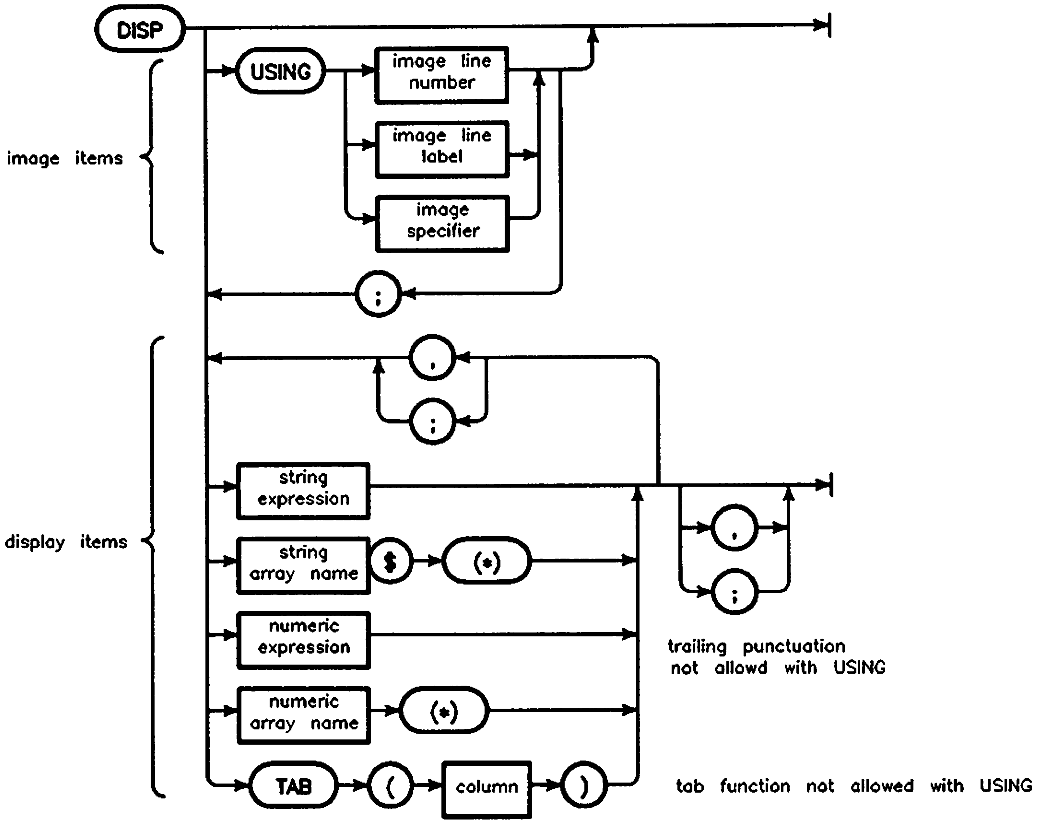
```
DISABLE INTR 7
DISABLE INTR Isc
```

DISP

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement causes the display items to be sent to the display line on the CRT.

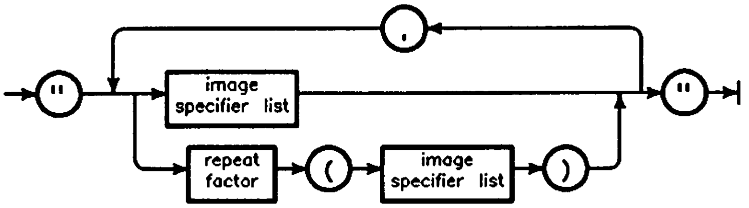


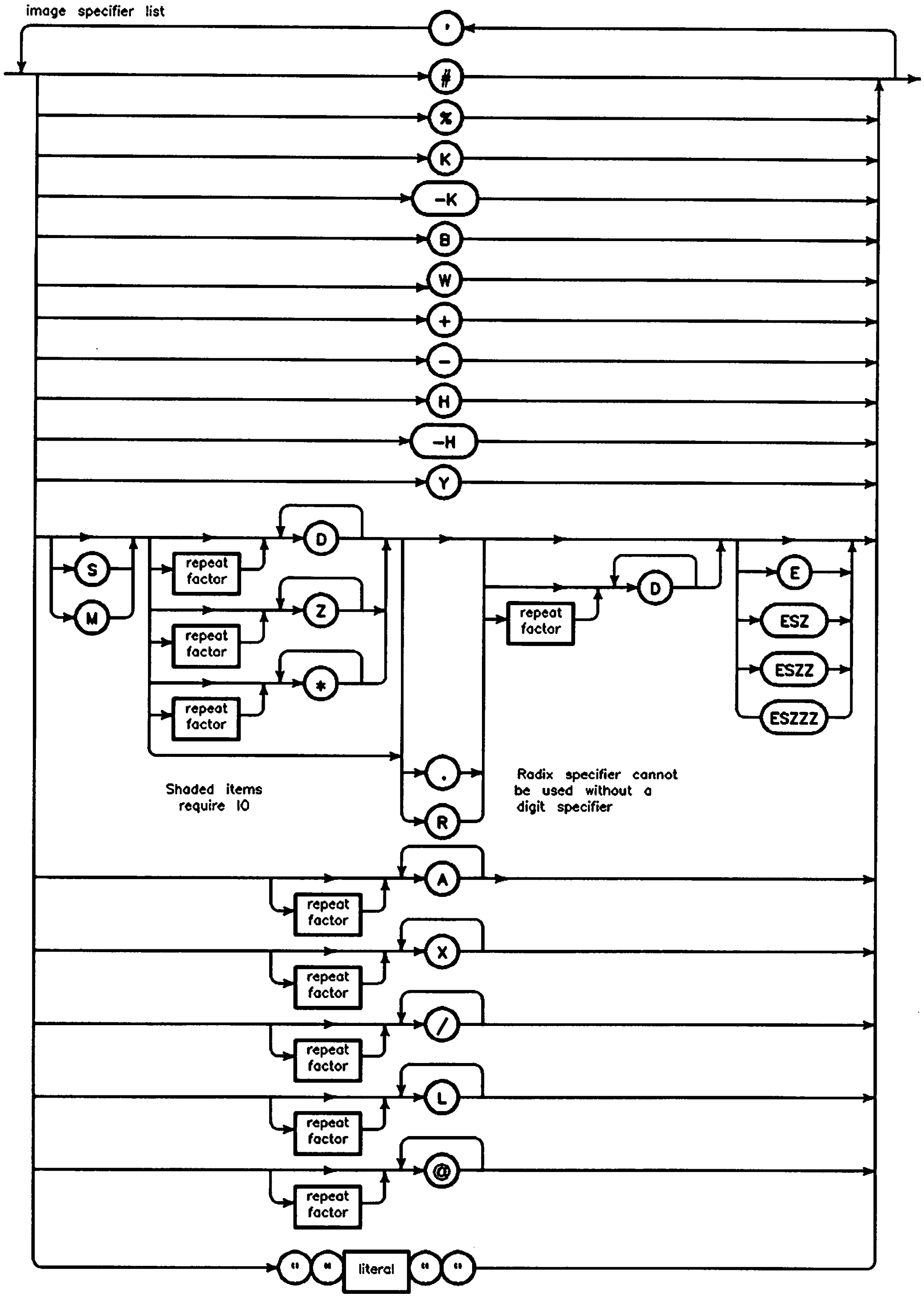


trailing punctuation not allowd with USING

tab function not allowed with USING

literal form of image specifier





Shaded items require IO

Radix specifier cannot be used without a digit specifier

Item	Description	Range
image line label	name identifying an IMAGE statement	any valid name
image line number	integer constant identifying an IMAGE statement	1 thru 32 766
image specifier	string expression	(see diagram)
string array name	name of a string array	any valid name
numeric array name	name of a numeric array	any valid name
column	numeric expression, rounded to an integer	1 thru screenwidth
image specifier list	literal	(see diagram)
repeat factor	integer constant	1 thru 32 767
literal	string constant composed of characters entered from the keyboard	quote mark not allowed

Example Statements

```
DISP Prompt$;
DISP TAB(5),First,TAB(20),Second
DISP USING "5Z.DD";Money
```

Semantics

Standard Numeric Format

The standard numeric format depends on the value of the number being displayed. If the absolute value of the number is greater than or equal to 1E−4 and less than 1E+6, it is rounded to 12 digits and displayed in floating point notation. If it is not within these limits, it is displayed in scientific notation. The standard numeric format is used unless USING is selected, and may be specified by using K in an image specifier.

Automatic End-Of-Line Sequence

After the display list is exhausted, an End Of Line (EOL) sequence is sent to the display line, unless it is suppressed by trailing punctuation or a pound-sign image specifier.

Control Codes

Some ASCII control codes have a special effect in DISP statements:

Character	Keystroke	Name	Action
CHR$(7)	CTRL-G	bell	Sound the beeper
CHR$(8)	CTRL-H	backspace	Move the cursor back one character.
CHR$(12)	CTRL-L	form-feed	Clear the display line.
CHR$(13)	CTRL-M	carriage-return	Move the cursor to column 1. The next character sent to the display clears the display line, unless it is a carriage-return.

Arrays

Entire arrays may be displayed using the asterisk specifier. Each element in an array is treated as a separate item by the DISP statement, as if the items were listed separately, separated by the punctuation following the array specifier. If no punctation follows the array specifier, a comma is assumed. The array is output in row major order (rightmost subscript varies fastest).

Display Without USING

If DISP is used without USING, the punctuation following an item determines the width of the item's display field; a semicolon selects the compact field, and a comma selects the default display field. When the display item is an array with the asterisk array specifier, each array element is considered a separate display item. Any trailing punctation will suppress the automatic EOL sequence, in addition to selecting the display field to be used for the display item preceding it.

The compact field is slightly different for numeric and string items. Numeric items are displayed with one trailing blank. String items are displayed with no leading or trailing blanks.

The default display field displays items with trailing blanks to fill to the beginning of the next 10-character field.

Numeric data is displayed with one leading blank if the number is positive, or with a minus sign if the number is negative, whether in compact or default field.

In the TAB function, a column parameter less than one is treated as one. A column parameter greater than the screen width (in characters) is treated as equal to the screen width.

Display With USING

When the computer executes a DISP USING statement, it reads the image specifier, acting on each field specifier (field specifiers are separated from each other by commas) as it is encountered. If nothing is required from the display items, the field specifier is acted upon without accessing the display list. When the field specifier requires characters, it accesses the next item in the display list, using the entire item. Each element in an array is considered a separate item.

The processing of image specifiers stops when a specifier is encountered that has no matching display item (and the specifier requires a display specifier). If the image specifiers are exhausted before the display items, they are reused, starting at the beginning.

If a numeric item requires more decimal places to the left of the decimal point than are provided by the field specifier, an error is generated. A minus sign takes a digit place if M or S is not used, and can generate unexpected overflows of the image field. If the number contains more digits to the right of the decimal point than specified, it is rounded to fit the specifier.

If a string is longer than the field specifier, it is truncated, and the rightmost characters are lost. If it is shorter than the specifier, trailing blanks are used to fill out the field.

Effects of the image specifiers on the DISP statement are shown in the following table:

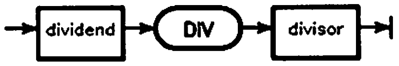
Image Specifier	Meaning
K	Compact field. Displays a number or string in standard form with no leading or trailing blanks.
–K	Same as K.
H	Similar to K, except the number is displayed using the European number format (comma radix).
–H	Same as H.
S	Displays the number's sign (+ or –).
M	Displays the number's sign if negative, a blank if positive.
D	Displays one digit character. A leading zero is replaced by a blank. If the number is negative and no sign image is specified, the minus sign will occupy a leading digit position. If a sign is displayed, it will "float" to the left of the left-most digit.
Z	Same as D, except that leading zeros are displayed.
*	Same as Z, except that asterisks are displayed instead of leading zeros.

Image Specifier	Meaning
.	Displays a decimal-point radix indicator.
R	Displays a comma radix indicator (European radix).
E	Displays an E, a sign, and a two-digit exponent.
ESZ	Displays an E, a sign, and a one-digit exponent.
ESZZ	Same as E.
ESZZZ	Displays an E, a sign, and a three-digit exponent.
A	Displays a string character. Trailing blanks are output if the number of characters specified is greater than the number available in the corresponding string. If the image specifier is exhausted before the corresponding string, the remaining characters are ignored.
X	Displays a blank.
literal	Displays the characters contained in the literal.
B	Displays the character represented by one byte of data. This is similar to the CHR$ function. The number is rounded to an INTEGER, and the least-significant byte is sent. If the number is greater than 32 767, then 255 is used; if the number is less than −32 768, then 0 is used.
W	Displays two characters represented by the two bytes of a 16-bit, two's-complement integer. The corresponding numeric item is rounded to an INTEGER. If it is greater than 32 767, then 32 767 is used; if it is less than −32 768, then −32 768 is used. The most-significant byte is sent first.
Y	Same as W.
#	Suppresses the automatic output of an EOL (End-Of-Line) sequence following the last display item.
%	Ignored in DISP images.
+	Changes the automatic EOL sequence that normally follows the last display item to a single carriage-return.
−	Changes the EOL automatic sequence that normally follows the last display item to a single line-feed.
/	Sends a carriage-return and a line-feed to the display line.
L	Same as /.
@	Sends a form-feed to the display line.

DIV

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This operator returns the integer portion of the quotient of the dividend and the divisor.



Item	Description	Range
dividend	numeric expression	—
divisor	numeric expression	not equal to 0

Example Statements

```
Quotient=Dividend DIV Divisor
PRINT "Hours =";Minutes DIV 60
```

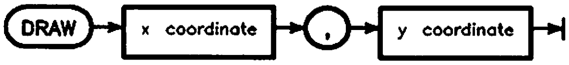
Semantics

DIV returns a REAL value unless both arguments are INTEGER. In the latter case the returned value is INTEGER. A DIV B is identical to SGN(A/B) × INT(ABS(A/B)).

DRAW

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement draws a line from the pen's current position to the specified X and Y coordinate position using the current line type and pen number.



Item	Description	Range
x coordinate	numeric expression, in current units	*
y coordinate	numeric expression, in current units	*

**See your instrument-specific HP Instreument BASIC manual for details on x and y coordinate ranges.*

Example Statements

```
DRAW 10,90
DRAW Next_x,Next_y
```

Semantics

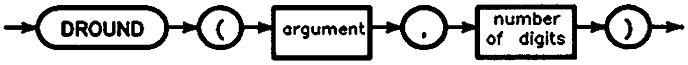
A DRAW to the current position generates a point. DRAW updates the logical pen position at the completion of the DRAW statement, and leaves the pen down on an external plotter. The line is clipped at the current clipping boundary.

If none of the line is inside the current clipping limits, the pen is not moved, but the logical pen position is updated.

DROUND

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function rounds a numeric expression to the specified number of digits. If the specified number of digits is greater than 15, no rounding takes place. If the number of digits specified is less than 1, 0 is returned.



Item	Description	Range
argument	numeric expression	—
number of digits	numeric expression, rounded to an integer	—

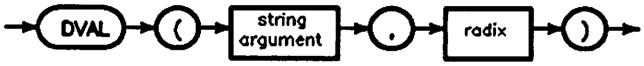
Example Statements

```
Test_real=DROUND(True_real,12)
PRINT "Approx. Volts =";DROUND(Volts,3)
```

DVAL

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function converts a binary, octal, decimal, or hexadecimal character string into a REAL whole number.



Item	Description	Range
string argument	string expression, containing digits valid for the specified base	(see tables)
radix	numeric expression, rounded to an integer	2, 8, 10, or 16

Example Statements

```
Address=DVAL("FF590004",16)
Real=DVAL("01010101010101010101010101010101",2)
Number=DVAL(Octal$,8)
```

Semantics

The radix is a numeric expression that will be rounded to an integer and must evaluate to 2, 8, 10, or 16.

The string expression must contain only the characters allowed for the particular number base indicated by the radix. ASCII spaces are not allowed.

Binary strings are presumed to be in two's-complement form. If all 32 digits are specified and the leading digit is a 1, the returned value is negative.

Octal strings are presumed to be in the octal representation of two's-complement form. If all 11 digits are specified, and the leading digit is a 2 or a 3, the returned value is negative.

Decimal strings containing a leading minus sign will return a negative value.

Hex strings are presumed to be in the hex representation of the two's-complement binary form. The letters A through F may be specified in either uppercase or lowercase letters. If all 8 digits are specified and the leading digit is 8 through F, the returned value is negative.

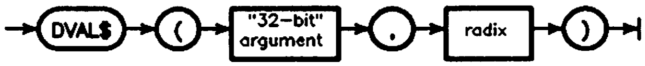
Radix	Base	String Range	String Length
2	binary	0 thru 11111111111111111111111111111111	1 to 32 characters
8	octal	0 thru 37777777777	1 to 11 characters
10	decimal	−2 147 483 648 thru 2 147 483 647	1 to 11 characters
16	hexadecimal	0 thru FFFFFFFF	1 to 8 characters

Radix	Legal Characters	Comments
2	+,0,1	—
8	+,0,1,2,3,4,5,6,7	Range restricts the leading character. Sign, if used, must be a leading character.
10	+,−,0,1,2,3,4,5,6,7,8,9	Sign, if used, must be a leading character.
16	+,0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F,a,b,c,d,e,f	A/a = 10, B/b = 11, C/c = 12, D/d = 13, E/e = 14, F/f = 15

DVAL$

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function converts a whole number into a binary, octal, decimal, or hexadecimal string.



Item	Description	Range
"32-bit" argument	numeric expression, rounded to an integer	-2^{31} thru $2^{31} -1$
radix	numeric expression, rounded to an integer	2, 8, 10, or 16

Example Statements

```
F$=DVAL$(-1,16)
Binary$=DVAL$(Count DIV 256,2)
```

Semantics

The rounded argument must be a value that can be expressed (in binary) using 32 bits or less.

The radix must evaluate to be 2, 8, 10, or 16—representing binary, octal, decimal, or hexadecimal notation, respectively.

If the radix is 2, the returned string is in two's-complement form and contains 32 characters. If the numeric expression is negative, the leading digit will be 1. If the value is zero or positive, there will be leading zeros.

If the radix is 8, the returned string is the octal representation of the two's-complement binary form and contains 11 digits. Negative values return a leading digit of 2 or 3.

If the radix is 10, the returned string contains 11 characters. Leading zeros are added to the string if necessary. Negative values have a leading minus sign.

If the radix is 16, the returned string is the hexadecimal representation of the two's-complement binary form and contains 8 characters. Negative values return with the leading digit in the range 8 thru F.

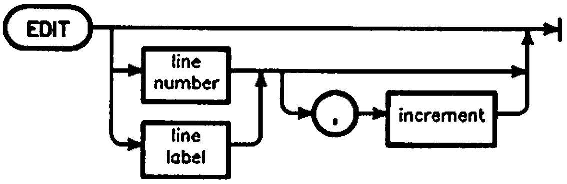
Radix	Base	Range of Returned String	String Length
2	binary	00000000000000000000000000000000 thru 11111111111111111111111111111111	32 characters
8	octal	00000000000 thru 37777777777	11 characters
10	decimal	−2 147 483 648 thru 2 147 483 647	11 characters
16	hexadecimal	00000000 thru FFFFFFFF	8 characters

EDIT

Keyboard Executable	Yes
Programmable	No
In an IF ... THEN ...	No

This command allows you to enter or edit a program.

Note Each host instrument implements the EDIT command in a different manner. See your instrument-specific HP Instrument BASIC manual for specific information regarding the EDIT command on your instrument.



Item	Description	Range
line number	integer constant identifying program line; Default (see Semantics)	1 thru 32 766
line label	name of a program line	any valid name

Example Statements

```
EDIT
EDIT 1000
```

Semantics

The EDIT command allows you to scroll through a program using the keyboard. Lines may be added to the end of a program by going to the bottom of the program. A new line number will be provided automatically. Lines may be added between existing program lines by using the insert line key, and lines may be deleted by using the delete line key. Lines may be modified by typing the desired characters over the existing line, using the insert character and delete character keys as necessary. (ENTER), (EXECUTE) or (Return) are used to store the newly created or modified lines.

Edit mode is exited by pressing (CONTINUE), (CLR SCR), (Clear display), (PAUSE), (Stop), (RESET), (RUN), or (STEP) or by executing CAT, LIST (if PRINTER IS CRT),or GET. In general any PRINT to the CRT (e.g., executing DISP) will exit you from the EDIT mode. If the program was changed while paused, pressing (CONTINUE) will generate an error, since modifying a program moves it to the stopped state.

EDIT Without Parameters

If no program is currently in the computer, the edit mode is entered at line 10, and the line numbers are incremented by 10 as each new line is stored. If a program is in the computer, the line at which the editor enters the program is dependent upon recent history. If an error has paused program execution, the editor enters the program at the line flagged by the error message. Otherwise, the editor enters the program at the line most recently edited (or the beginning of the program after a GET operation).

EDIT With Parameters

If no program is in the computer, a line number (not a label) must be used to specify the beginning line for the program. If the line specified is between two existing lines, the lowest-numbered line greater than the specified line is used. If a line label is used to specify a line, the lowest-numbered line with that label is used. If the label cannot be found, an error is generated.

ELSE

See the IF . . . THEN statement.

ENABLE

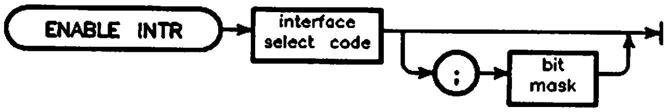
Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement re-enables all event-initiated branches which were suspended by DISABLE. ON ERROR, and ON TIMEOUT are not affected by ENABLE and DISABLE.

ENABLE INTR

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement enables the specified interface to generate an interrupt which can cause end-of-statement branches.



Item	Description	Range
interface select code	numeric expression, rounded to an integer	5, and 7 thru 31 (See your instrument manual)
bit mask	numeric expression, rounded to an integer	–

Example Statements

```
ENABLE INTR 7
ENABLE INTR Isc;Mask
```

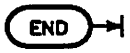
Semantics

The bit mask argument is included for compatibility only. Any bit mask specified in the statement is ignored by HP Instrument BASIC.

END

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This statement marks the end of the main program. (For information about END as a secondary keyword, see the OUTPUT statement.)



Semantics

END must be the last statement (other than comments) of a main program. Only one END statement is allowed in a program. (Program execution may also be terminated with a STOP statement, and multiple STOP statements are allowed.) END terminates program execution, stops any event-initiated branches, and clears any unserviced event-initiated branches. CONTINUE is not allowed after an END statement.

Subroutines used by the main program must occur prior to the END statement. Subprograms and user-defined functions must occur after the END statement.

END IF

See the IF ... THEN statement.

END LOOP

See the LOOP statement.

END SELECT

See the SELECT ... CASE construct.

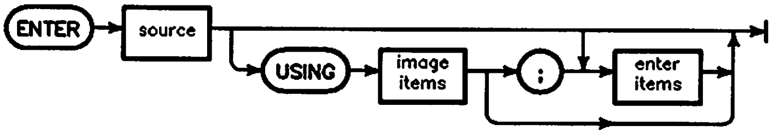
END WHILE

See the WHILE statement.

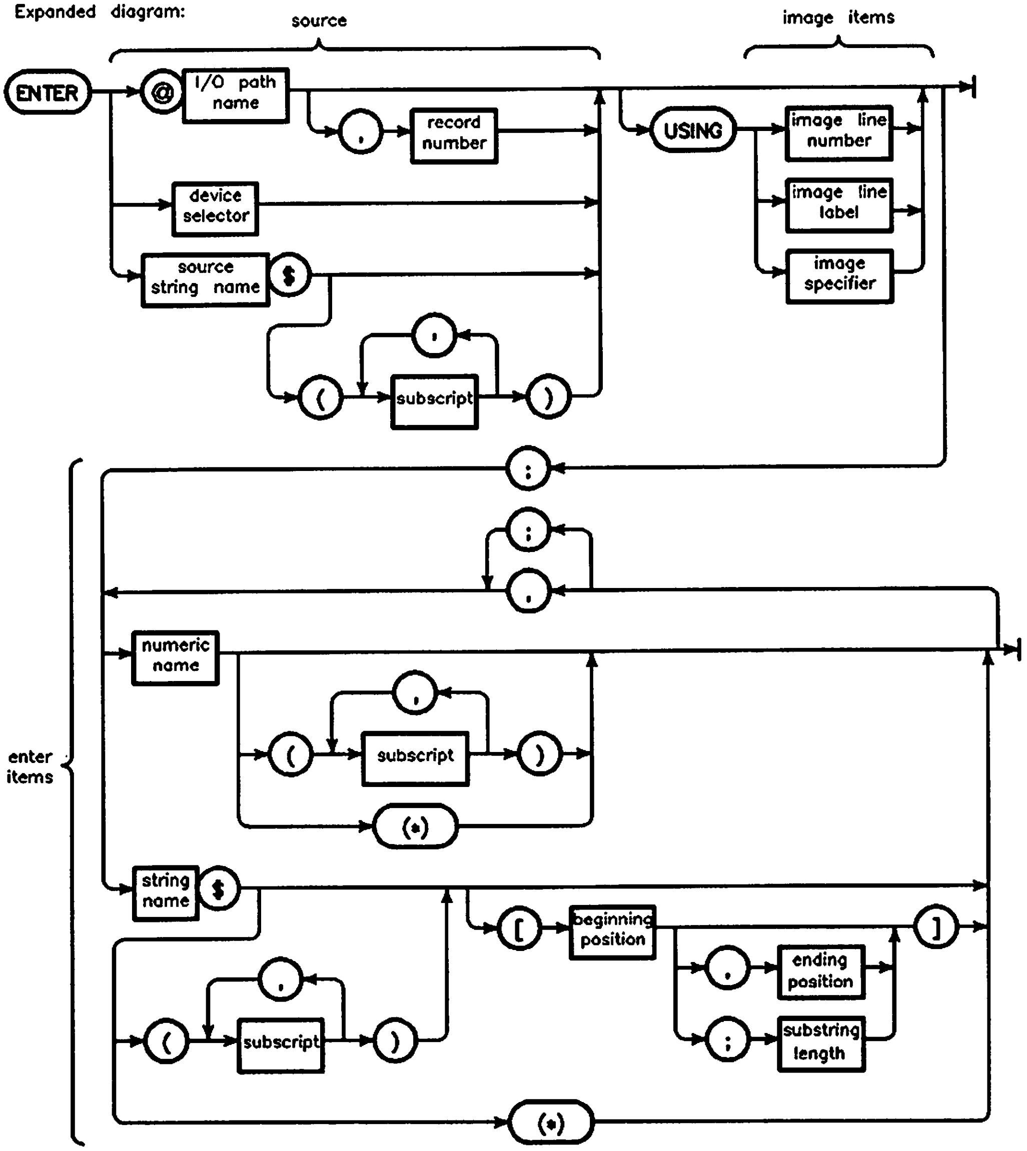
ENTER

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

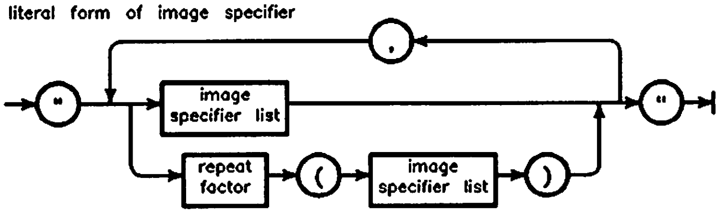
This statement is used to input data from a device, file, or string, and assign the values entered to variables.



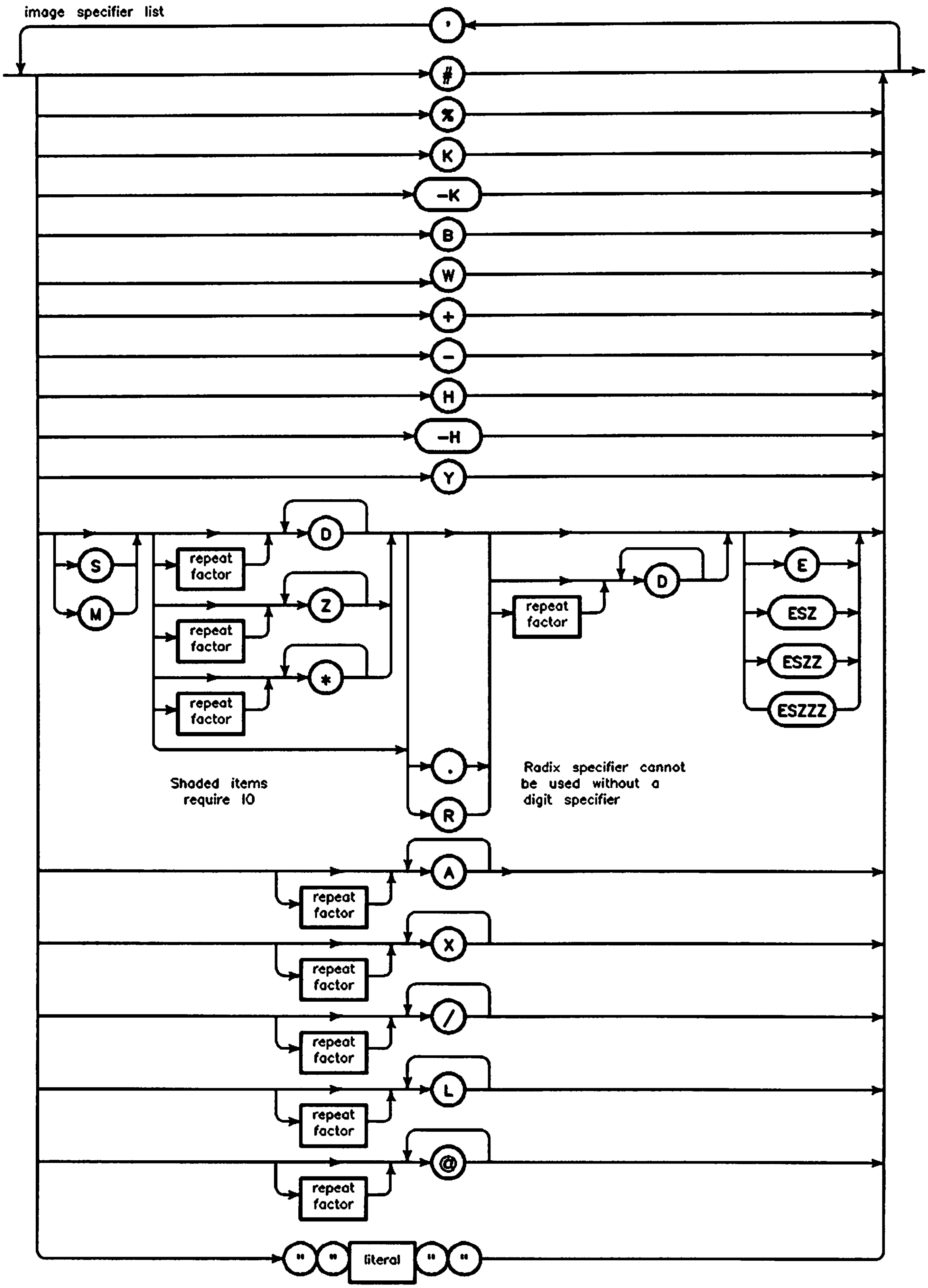
Expanded diagram:



literal form of image specifier



Item	Description	Range
I/O path name	name assigned to a device, devices, mass storage file, or buffer	any valid name (see ASSIGN)
record number	numeric expression, rounded to an integer	1 thru $2^{31}-1$
device selector	numeric expression, rounded to an integer	(see Glossary)
source string name	name of a string variable	any valid name
subscript	numeric expression, rounded to an integer	−32 767 thru +32 767 (see "array" in Glossary)
image line number	integer constant identifying an IMAGE statement	1 thru 32 766
image line label	name identifying an IMAGE statement	any valid name
image specifier	string expression	(see drawing)
numeric name	name of a numeric variable	any valid name
string name	name of a string variable	any valid name
beginning position	numeric expression, rounded to an integer	1 thru 32 767 (see "substring" in Glossary)
ending position	numeric expression, rounded to an integer	0 thru 32 767 (see "substring" in Glossary)
substring length	numeric expression, rounded to an integer	0 thru 32 767 (see "substring" in Glossary)
image specifier list	literal	(see next drawing)
repeat factor	integer constant	1 thru 32 767
literal	string constant composed of characters from the keyboard	quote mark not allowed



Shaded items require IO

Radix specifier cannot be used without a digit specifier

Example Statements

```
ENTER 705;Number,String$
ENTER @File;Array(*)
ENTER @Source USING Fmt5;Item(1),Item(2),Item(3)
ENTER 12 USING "#,6A";A$[2;6]
```

Semantics

The Number Builder

If the data being received is ASCII and the associated variable is numeric, a number builder is used to create a numeric quantity from the ASCII representation. The number builder ignores all leading non-numeric characters, ignores all blanks, and terminates on the first non-numeric character, or the first character received with EOI true. (Numeric characters are 0 thru 9, +, −, decimal point, e, and E, in a meaningful numeric order.) If the number cannot be converted to the type of the associated variable, an error is generated. If more digits are received than can be stored in a variable of type REAL, the rightmost digits are lost, but any exponent will be built correctly. Overflow occurs only if the exponent overflows.

Arrays

Entire arrays may be entered by using the asterisk specifier. Each element in an array is treated as an item by the ENTER statement, as if the elements were listed separately. The array is filled in row major order (rightmost subscript varies fastest).

Files as Source

If an I/O path has been assigned to a file, the file may be read with ENTER statements. The file must be an ASCII, BDAT, or HP-UX file. The attributes specified in the ASSIGN statement are used only if the file is a BDAT or HP-UX file. Data read from an ASCII file is always in ASCII format (i.e., you *cannot* use ENTER..USING); however, you can enter the data into a string variable, and then use ENTER..USING from the string variable. Data read from a BDAT or HP-UX file is considered to be in internal representation with FORMAT OFF, and is read as ASCII characters with FORMAT ON.

Serial access is available for ASCII, BDAT, and HP-UX files. Random access is available for BDAT and HP-UX files. The file pointer is important to both serial and random access. The file pointer is set to the beginning of the file when the file is opened by an ASSIGN. The file pointer always points to the next byte available for ENTER operations.

Random access uses the record number parameter to read items from a specific location in a file. The record specified must be before the end-of-file pointer. The ENTER begins at the beginning of the specified record.

It is recommended that random and serial access to the same file not be mixed. Also, data should be entered into variables of the same type as those used to output it (e.g. string for string, REAL for REAL, etc.).

In order to ENTER from a file on an HFS volume, you need to have R (read) permission on the file, as well as X (search) permission on the immediately superior directory and all other superior directories.

Devices as Source

An I/O path name or a device selector may be used to ENTER from a device. If a device selector is used, the default system attributes are used (see ASSIGN). If an I/O path name is used, the ASSIGN statement determines the attributes used.

If FORMAT ON is the current attribute, the items are read as ASCII.

If FORMAT OFF is the current attribute, items are read from the device in the computer's internal format. Two bytes are read for each INTEGER, eight bytes for each REAL.

Each string entered consists of a four byte header containing the length of the string, followed by the actual string characters. The string must contain an even number of characters; if the length is odd, an extra byte is entered to give alignment on the word boundary.

Keyboard as Source

ENTER from device selector 2 may be used to read the keyboard. An entry can be terminated by pressing ENTER, EXECUTE, Return, CONTINUE, or STEP. Using ENTER, EXECUTE, Return or STEP causes a CR/LF to be appended to the entry. The CONTINUE key adds no characters to the entry and does not terminate the ENTER statement. If an ENTER is stepped into, it is stepped out of, even if the CONTINUE key is pressed.

Strings as Source

If a string name is used as the source, the string is treated similarly to a file. However, there is no file pointer; each ENTER begins at the beginning of the string, and reads serially within the string.

ENTER With USING

When the computer executes an ENTER USING statement, it reads the image specifier, acting on each field specifier (field specifiers are separated from each other by commas) as it is encountered. If no variable is required for the field specifier, the field specifier is acted upon without referencing the enter items. When the field specifier references a variable, bytes are entered and used to create a value for the next item in the enter list. Each element in an array is considered a separate item.

The processing of image specifiers stops when a specifier is encountered that has no matching enter item. If the image specifiers are exhausted before the enter items, the specifiers are reused, starting at the beginning of the specifier list.

Entry into a string variable always terminates when the dimensioned length of the string is reached. If more variables remain in the enter list when this happens, the next character received is associated with the next item in the list.

When USING is specified, all data is interpreted as ASCII characters. FORMAT ON is always assumed with USING, regardless of any attempt to specify FORMAT OFF.

ENTER with USING cannot be used to enter data from an ASCII file. Instead, enter the item(s) into a string variable, and then use ENTER with USING to enter the item(s) from the string variable. For instance, use `ENTER @File;String$` then `ENTER String$ USING "5A,X,5DD"; Str2$,Number`.

Effects of the image specifiers on the ENTER statement are shown in the following table:

Image Specifier	Meaning
K	Freefield Entry. Numeric Entered characters are sent to the number builder. Leading non-numeric characters are ignored. All blanks are ignored. Trailing non-numeric characters and characters sent with EOI true are delimiters. Numeric characters include digits, decimal point, +, −, e, and E when their order is meaningful. String Entered characters are placed in the string. Carriage-return not immediately followed by line-feed is entered into the string. Entry to a string terminates on CR/LF, LF, a character received with EOI true, or when the dimensioned length of the string is reached.
−K	Like K except that LF is entered into a string, and thus CR/LF and LF do not terminate the entry.
H	Like K, except that the European number format is used. Thus, a comma is the radix indicator and a period is a terminator for a numeric item.
−H	Same as −K for strings; same as H for numbers.
S	Same action as D.
M	Same action as D.
D	Demands a character. Non-numerics are accepted to fill the character count. Blanks are ignored, other non-numerics are delimiters.
Z	Same action as D.
*	Same action as D.
.	Same action as D.
R	Like D, R demands a character. When R is used in a numeric image, it directs the number builder to use the European number format. Thus, a comma is the radix indicator and a period is a terminator for the numeric item.
E	Same action as 4D.
ESZ	Same action as 3D.
ESZZ	Same action as 4D.

Image Specifier	Meaning
ESZZZ	Same action as 5D.
A	Demands a string character. Any character received is placed in the string.
X	Skips a character.
literal	Skips one character for each character in the literal.
B	Demands one byte. The byte becomes a numeric quantity.
W	Demands one 16-bit word, which is interpreted as a 16-bit, two's-complement integer. If either an I/O path name with the BYTE attribute or a device selector is used to access an 8-bit interface, two bytes will be entered; the most-significant byte is entered first. If an I/O path name with the BYTE attribute is used to access a 16-bit interface, the BYTE attribute is overridden and one word is entered in a single operation. If an I/O path name with the WORD attribute is used to access a 16-bit interface, one byte is entered and ignored when necessary to achieve alignment on a word boundary. If the source is a file, string variable, or buffer, the WORD attribute is ignored and all data are entered as bytes; however, one byte will be entered and ignored when necessary to achieve alignment on a word boundary.
Y	Like W, except that pad bytes are never entered to achieve word alignment. If an I/O path name with the BYTE is used to access a 16-bit interface, the BYTE attribute is not overridden (as with W specifier above).
#	Statement is terminated when the last ENTER item is terminated. EOI and line-feed are item terminators, and early termination is not allowed.
%	Like #, except that an END indication (such as EOI or end-of-file) is an immediate statement terminator. Otherwise, no statement terminator is required. Early termination is allowed if the current item is satisfied.
+	Specifies that an END indication is required with the last character of the last item to terminate the ENTER statement. Line-feeds are not statement terminators. Line-feed is an item terminator unless that function is suppressed by −K or −H.
−	Specifies that a line-feed terminator is required as the last character of the last item to terminate the statement. EOI is ignored, and other END indications, such as EOF or end-of-data, cause an error if encountered before the line-feed.
/	Demands a new field; skips all characters to the next line-feed. EOI is ignored.
L	Ignored for ENTER.
@	Ignored for ENTER.

ENTER Statement Termination

A simple ENTER statement (one without USING) expects to give values to all the variables in the enter list and then receive a statement terminator. A statement terminator is an EOI, a line-feed received at the end of the last variable (or within 256 characters after the end of the last variable), an end-of-data indication, or an end-of-file. If a statement terminator is received before all the variables are satisfied, or no terminator is received within 256 bytes after the last variable is satisfied, an error occurs. The terminator requirements can be altered by using images.

An ENTER statement with USING, but without a % or # image specifier, is different from a simple ENTER in one respect. EOI is not treated as a statement terminator unless it occurs on or after the last variable. Thus, EOI is treated like a line-feed and can be used to terminate entry into each variable.

An ENTER statement with USING that specifies a # image requires no statement terminator other than a satisfied enter list. EOI and line feed end the entry into individual variables. The ENTER statement terminates when the variable list has been satisfied.

An ENTER statement with USING that specifies a % image allows EOI as a statement terminator. Like the # specifier, no special terminator is required. Unlike the # specifier, if an EOI is received, it is treated as an immediate statement terminator. If the EOI occurs at a normal boundary between items, the ENTER statement terminates without error and leaves the value of any remaining variables unchanged.

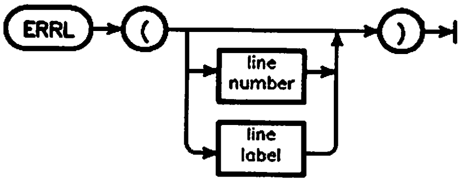
EOL

See the ASSIGN, and PRINTER IS statements.

ERRL

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns a value of 1 if the most recent error occurred in the specified line; otherwise, a value of 0 is returned.



Item	Description	Range
line number	integer constant	1 thru 32 766
line label	name of a program line	any valid name

Example Statements

```
IF ERRL(220) THEN Parse_error
IF NOT ERRL(Parameters) THEN Other
```

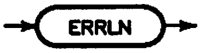
Semantics

The specified line must be in the same context as the ERRL function, or an error will occur.

ERRLN

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the number of the program line on which the most recent error occurred.



Example Statements

```
ERRLN
IF ERRLN=240 THEN GOSUB Fix_240
```

Semantics

If no error has occurred since power-on, pre-run, SCRATCH, SCRATCH A, or GET, this function will return a value of 0.

ERRM$

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the text of the error message associated with the most recent program execution error.

Example Statements

```
PRINT ERRM$
Em$=ERRM$
ENTER Em$;Error_number,Error_line
```

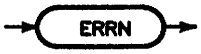
Semantics

If no error has occurred since power on, prerun, SCRATCH, SCRATCH A, or GET, the null string will be returned. The line number and error number returned in the ERRM$ string are the same as those used by ERRN and ERRL. For details on the interaction, see ERRL and ERRN.

ERRN

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the number of the most recent program execution error. If no error has occurred, a value of 0 is returned.



Example Statements

```
IF ERRN=80 THEN Disc_out
DISP "Error Number";ERRN
```

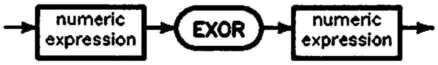
ERROR

See the OFF ERROR and ON ERROR statements.

EXOR

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This operator returns a 1 or a 0 based on the logical exclusive-or of its arguments.



Example Statements

```
Ok=First_pass EXOR Old_data
IF A EXOR Flag THEN Exit
```

Semantics

A non-zero value (positive or negative) is treated as a logical 1; only a zero is treated as a logical 0.

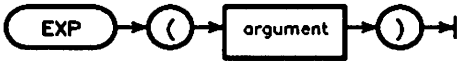
The EXOR function is summarized in this table.

A	B	A EXOR B
0	0	0
0	1	1
1	0	1
1	1	0

EXP

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function raises *e* to the power of the argument. With this system,Napierian *e* ≈ 2.718 281 828 459 05.



Item	Description/Default	Range Restrictions
argument	numeric expression	−708.396 418 532 264 thru +709.782 712 893 383 8 for INTEGER and REAL arguments

Examples Statements

```
Y=EXP(-X^2/2)
PRINT "e to the ";Z;"=";EXP(Z)
```

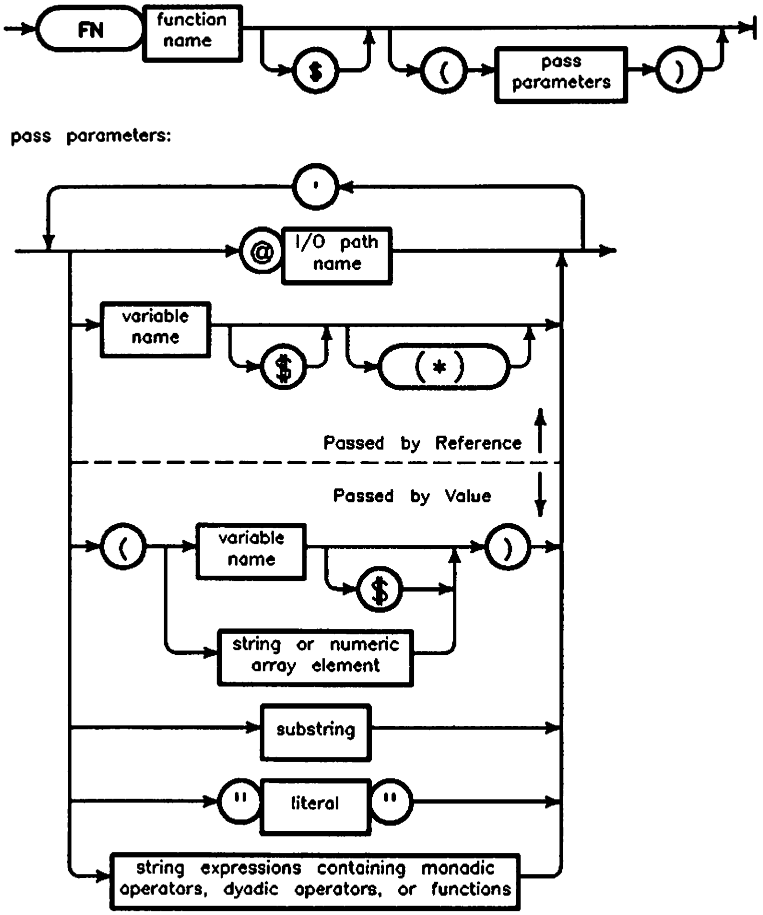
Semantics

If the argument is REAL or INTEGER, the value returned is REAL.

FN

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This keyword transfers program execution to the specified user-defined function and may pass items to the function. The value returned by the function is used in place of the function call when evaluating the statement containing the function call.



Item	Description	Range
function name	name of a user-defined function	any valid name
I/O path name	name assigned to a device, devices, or mass storage file	any valid name (see ASSIGN)
variable name	name of a numeric or string variable	any valid name
substring	string expression containing substring notation	(see Glossary)
literal	string constant composed of characters from the keyboard	—
numeric constant	numeric quantity expressed using numerals, and optionally a sign, decimal point, or exponent notation	—

Example Statements

```
PRINT X;FNChange(X)
Final$=FNTrim$(First$)
Result=FNPround(Item,Power)
```

Semantics

The pass parameters must be of the same type (numeric or string) as the corresponding parameters in the DEF FN statement. Numeric values passed by value are converted to the numeric type (REAL, or INTEGER) of the corresponding formal parameter.

Variables passed by reference must match the type of the corresponding parameter in the DEF FN statement exactly. An entire array may be passed by reference by using the asterisk specifier.

Invoking a user-defined function changes the program context. The functions may be invoked recursively.

If there is more than one user-defined function with the same name, the lowest numbered one is invoked by FN.

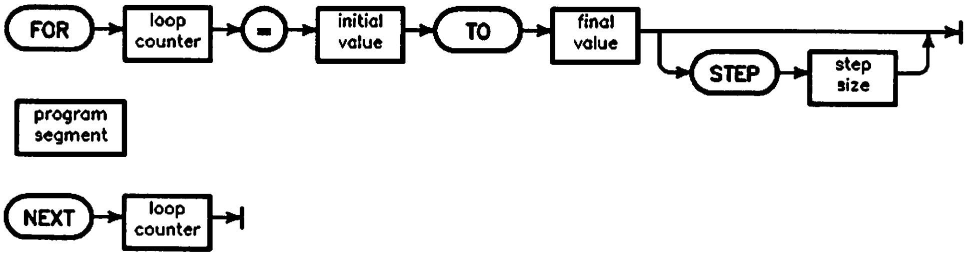
FNEND

See the DEF FN statement.

FOR ... NEXT

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This construct defines a loop which is repeated until the loop counter passes a specific value. The step size may be positive or negative.



Item	Description	Range
loop counter	name of a numeric variable	any valid name
initial value	numeric expression	—
final value	numeric expression	—
step size	numeric expression; Default = 1	—
program segment	any number of contiguous program lines not containing the beginning or end of a main program or subprogram, but which may contain properly nested construct(s).	—

Example Program Segments

```
100   FOR I=4 TO 0 STEP -.1
110     PRINT I;SQR(I)
120   NEXT I

1220   INTEGER Point
1230   FOR Point=1 TO LEN(A$)
1240     CALL Convert(A$[Point;1])
1250   NEXT Point
```

Semantics

The loop counter is set equal to the initial value when the loop is entered. Each time the corresponding NEXT statement is encountered, the step size (which defaults to 1) is added to the loop counter, and the new value is tested against the final value. If the final value has not been passed, the loop is executed again, beginning with the line immediately following the FOR statement. If the final value has been passed, program execution continues at the line following the NEXT statement. Note that the loop counter is not equal to the specified final value when the loop is exited.

The loop counter is also tested against the final value as soon as the values are assigned when the loop is first entered. If the loop counter has already passed the final value in the direction the step would be going, the loop is not executed at all. The loop may be exited arbitrarily (such as with a GOTO), in which case the loop counter has whatever value it had obtained at the time the loop was exited.

The initial, final and step size values are calculated when the loop is entered and are used while the loop is repeating. If a variable or expression is used for any of these values, its value may be changed after entering the loop without affecting how many times the loop is repeated. However, changing the value of the loop counter itself can affect how many times the loop is repeated.

The loop counter variable is allowed in expressions that determine the initial, final, or step size values. The previous value of the loop counter is not changed until after the initial, final, and step size values are calculated.

If the step value evaluates to 0, the loop repeats infinitely and no error is given.

Nesting Constructs Properly

Each FOR statement is allowed one and only one matching NEXT statement. The NEXT statement must be in the same context as the FOR statement. FOR ... NEXT loops may be nested, and may be contained in other constructs, as long as the loops and constructs are properly nested and do not improperly overlap.

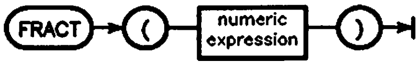
FORMAT

See the ASSIGN statement.

FRACT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns a number greater than or equal to zero and less than 1, representing the "fractional part" of the value of its argument. For all X, X=INT(X)+FRACT(X).



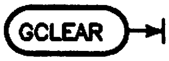
Example Statements

```
PRINT FRACT(X)
Right_digits=FRACT(All_digits)
```

GCLEAR

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

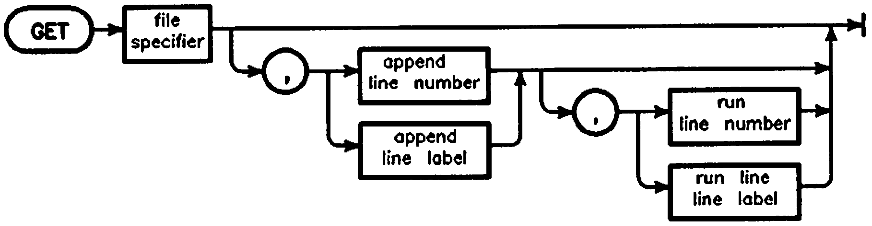
This statement clears the graphics display.



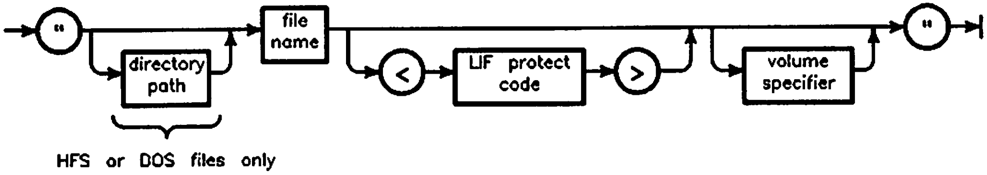
GET

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement reads the specified ASCII or HP-UX file and attempts to store the strings into memory as program lines.



literal form of file specifier:



Item	Description	Range
file specifier	string expression	(see drawing)
append line number	integer constant identifying a program line	1 thru 32 766
append line label	name of a program line	any valid name
run line number	integer constant identifying a program line	1 thru 32 766
run line label	name of a program line	any valid name
directory path	literal	(see MASS STORAGE IS)
file name	literal	depends on volume's format (see Glossary)
volume specifier	literal	(see MASS STORAGE IS)

Example Statements

```
GET "George"
GET Next_prog$,180,10
```

Semantics

The file must be either an ASCII, DOS or an HP-UX file (HP-UX files must contain text written in FORMAT ON representation).

When GET is executed, the first line in the specified file is read and checked for a valid line number. If no valid line number is found, the current program stays in memory and error 68 is reported. If the GET was attempted from a running program, the program remains active and the error 68 can be trapped with ON ERROR. If there is no ON ERROR in effect, the program pauses.

If there is a valid line number at the start of the first line in the file, the GET operation proceeds. Values for all variables except those in COM are lost and the current program is deleted from the append line to the end. If no append line is specified, the entire current program is deleted.

As the file is brought in, each line is checked for proper syntax. The syntax checking during GET is the same as if the lines were being typed from the keyboard, and any errors that would occur during keyboard entry will also occur during GET. Any lines which contain syntax errors are listed on the PRINTER IS device. Those erroneous lines which have valid line numbers are converted into comments and syntax is checked again. If the GET encounters a line longer than 256 characters, the operation is terminated and error 128 is reported. If any line caused any other syntax error, an error 68 is reported at the completion of the GET operation. This error is not trappable because the old program was deleted and the new one is not running yet.

Any line in the main program or any subprogram may be used for the append location. If an append line number is specified, the lines from the file are renumbered by adding an offset to their line numbers. This offset is the difference between the append line number and the first line number in the file. This operation preserves the line-number intervals that exist in the file. When a line containing an error (or an invalid line number caused by renumbering) is printed on the PRINTER IS device, the line number shown is the one the line had in the file. Any programmed references to line numbers that would be renumbered by the renumbering command (REN) are also renumbered by GET. If no append line is specified, the lines from the file are entered without renumbering.

If a successful GET is executed from a program, execution resumes automatically after a prerun initialization (see RUN). If no run line is specified, execution resumes at the lowest-numbered line in the program. If a run line is specified, execution resumes at the specified line. The specified run line must be a line in the main program segment.

When your host-instrument includes a keyboard and the command-line editing capabilities, if a successful GET is executed from the keyboard *and* a run line is specified, a prerun is performed and program execution begins automatically at the specified line. If GET is executed from the keyboard with no run line specified, RUN must be executed to start the program. GET is not allowed from the keyboard while a program is running.

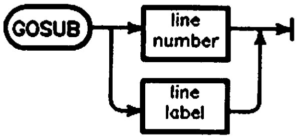
HFS Permissions

In order to GET a file on an HFS volume, you need to have R (read) permission on the file, as well as X (search) permission on the immediately superior directory and all other superior directories.

GOSUB

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement transfers program execution to the subroutine at the specified line. The specified line must be in the current context. The current program line is remembered in anticipation of returning (see RETURN). (Also see the ON ... statements.)



Item	Description	Range
line label	name of a program line	any valid name
line number	integer constant identifying a program line	1 thru 32 766

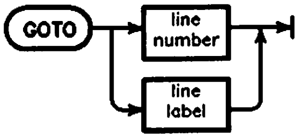
Example Statements

```
GOSUB 120
IF Numbers THEN GOSUB Process
```

GOTO

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement transfers program execution to the specified line. The specified line must be in the current context. (Also see the ON ... statements.)



Item	Description	Range
line label	name of a program line	any valid name
line number	integer constant identifying a program line	1 thru 32 766

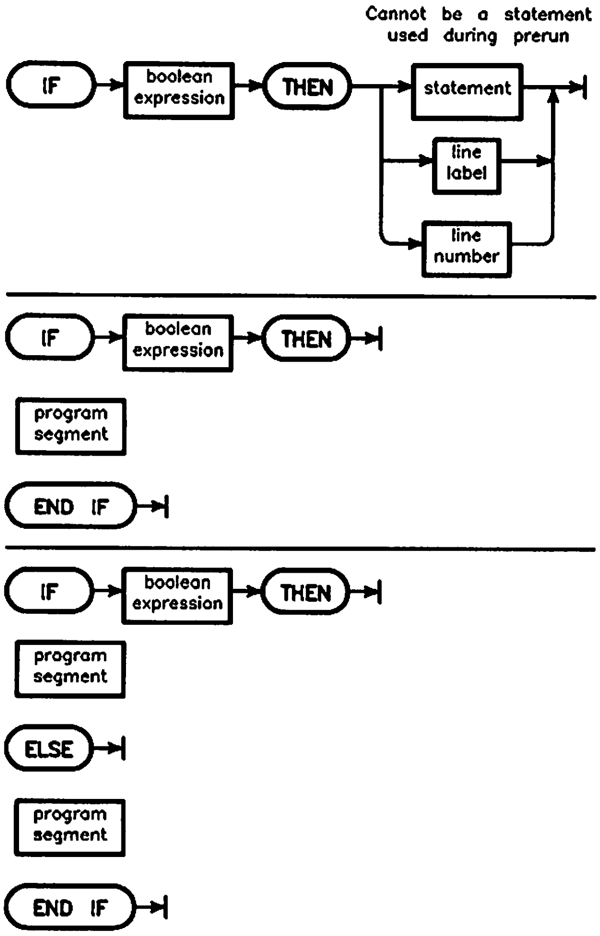
Example Statements

```
GOTO 550
GOTO Loop_start
IF Full THEN Exit       ! (implied GOTO)
```

IF ... THEN

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This statement provides conditional branching.



Item	Description	Range
boolean expression	numeric expression; evaluated as true if non-zero and false if zero	—
line label	name of a program line	any valid name
line number	integer constant identifying a program line	1 thru 32 766
statement	a programmable statement	(see following list)
program segment	any number of contiguous program lines not containing the beginning or end of a main program or subprogram.	—

Example Program Segments

```
150  IF Flag THEN Next_file
160  IF Pointer<1 THEN Pointer=1

580  IF First_pass THEN
590    Flag=0
600    INPUT "Command?",Cmd$
610    IF LEN(Cmd$) THEN GOSUB Parse
620  END IF

1000  IF X<0 THEN
1010    BEEP
1020    DISP "Improper Argument"
1030  ELSE
1040    Root=SQR(X)
1050  END IF
```

Semantics

If the boolean expression evaluates to 0, it is considered false; if the evaluation is non-zero, it is considered true. Note that a boolean expression can be constructed with numeric or string expressions separated by relational operators, as well as with a numeric expression.

Single Line IF ... THEN

If the conditional statement is a GOTO, execution is transferred to the specified line. The specified line must exist in the current context. A line number or line label by itself is considered an implied GOTO. For any other statement, the statement is executed, then program execution resumes at the line following the IF ... THEN statement. If the tested condition is false, program execution resumes at the line following the IF ... THEN statement, and the conditional statement is not executed.

Prohibited Statements

The following statements must be identified at prerun time or are not executed during normal program flow. Therefore, they are not allowed as the statement in a single line IF ... THEN construct.

CASE
CASE ELSE
COM
DATA
DEF FN
DIM
ELSE
END
END IF
END LOOP
END SELECT
END WHILE
EXIT IF
FNEND
FOR
IF
IMAGE
INTEGER
LOOP
NEXT
REAL
REM
REPEAT
SELECT
SUB
SUBEND
UNTIL
WHILE

When ELSE is specified, only one of the program segments will be executed. When the condition is true, the segment between IF ... THEN and ELSE is executed. When the condition is false, the segment between ELSE and END IF is executed. In either case, when the construct is exited, program execution continues with the statement after the END IF.

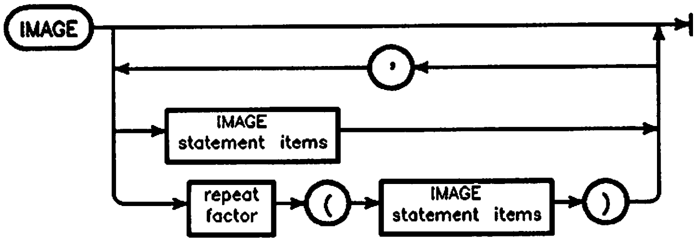
Branching into an IF ... THEN construct (such as with a GOTO) results in a branch to the program line following the END IF when the ELSE statement is executed.

The prohibited statements listed above are allowed in multiple-line IF ... THEN constructs. However, these statements are not executed conditionally. The exceptions are other IF ... THEN statements or constructs such as FOR ... NEXT, REPEAT ... UNTIL, etc. These are executed conditionally, but need to be properly nested. To be properly nested, the entire construct must be contained in one program segment (see drawing).

IMAGE

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This statement provides image specifiers for the ENTER, OUTPUT, DISP, LABEL, and PRINT statements. Refer to the appropriate statement for details on the effect of the various image specifiers.

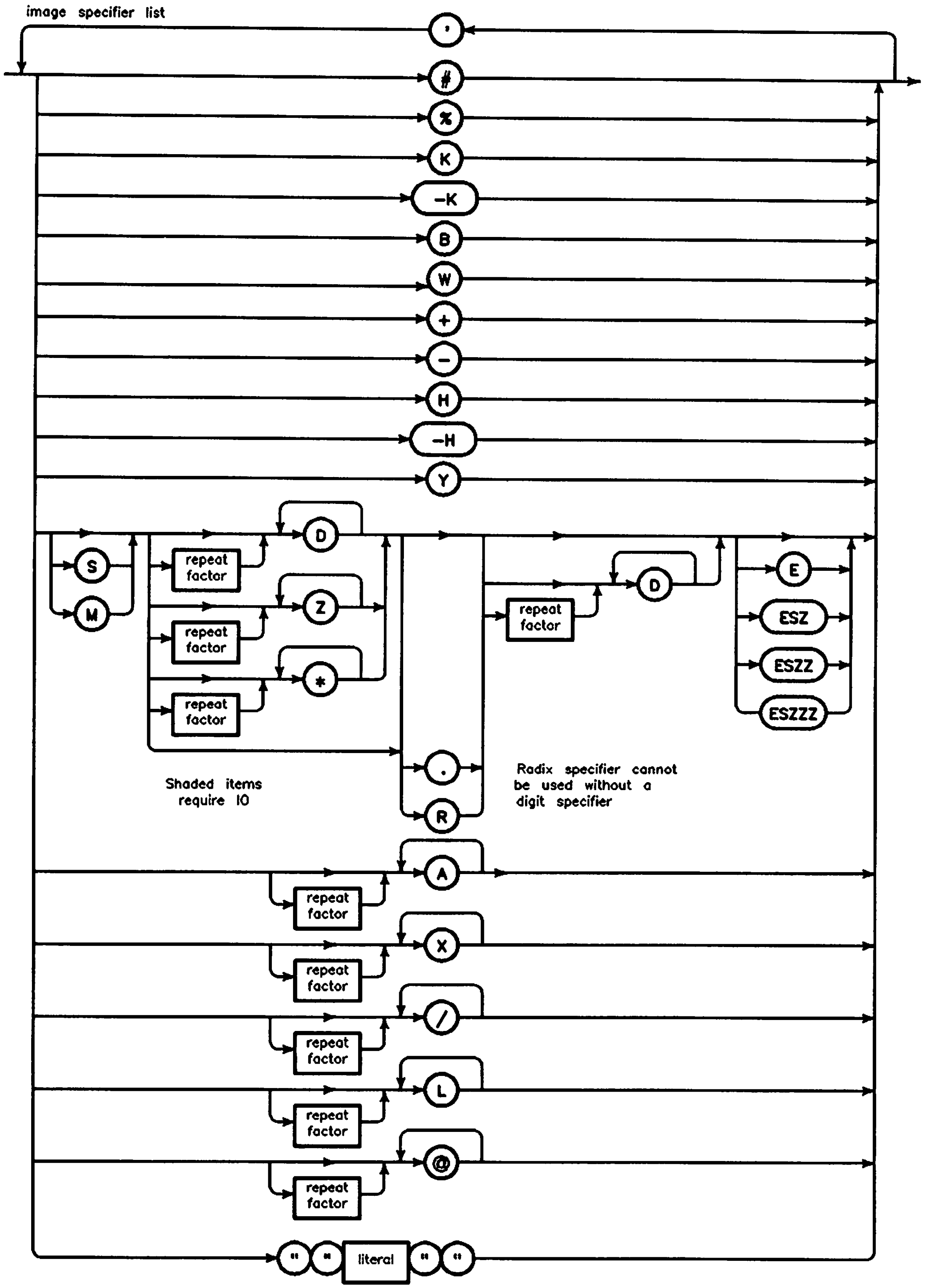


Item	Description	Range
IMAGE statement items	literal	(see drawing)
repeat factor	integer constant	1 thru 32 767
literal	string composed of characters from the keyboard, including those generated using the ANY CHAR key.	quote mark not allowed

Example Statements

```
IMAGE 4Z.DD,3X,K,/
IMAGE "Result = ",SDDDE,3(XX,ZZ)
IMAGE #,B
```

IMAGE



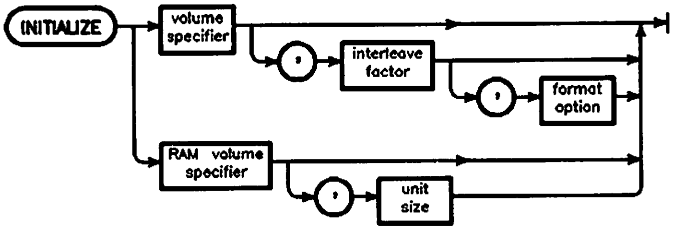
Shaded items require IO

Radix specifier cannot be used without a digit specifier

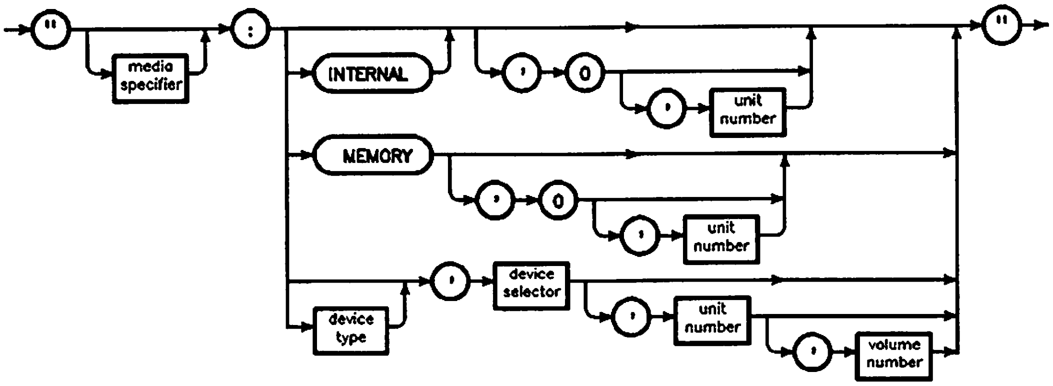
INITIALIZE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement prepares ("formats") mass storage media and places a LIF (Logical Interchange Format) or DOS directory on the media. When INITIALIZE is executed, *any existing files on the media are destroyed.*



literal form of volume specifier:



Item	Description	Range
volume specifier	string expression	(see MASS STORAGE IS)
interleave factor	numeric expression, rounded to an integer; Default = device dependent (see table)	0 thru 15
format option	numeric expression Default = 0	device dependent
media specifier	selects media format; default = LIF	LIF, DOS, lif, dos
RAM volume specifier	string expression	(see drawing)
RAM unit size	numeric expression, rounded to an integer; specifies number of 256-byte sectors;	4 thru 32 767 memory-dependent

Example Statements

```
INITIALIZE ":INTERNAL"
INITIALIZE Disc$,2
INITIALIZE ":,700",0,4
INITIALIZE ":MEMORY,0",Sectors
INITIALIZE "DOS:MEMORY,0",Sectors
INITIALIZE "DOS:,701",0,4
```

Semantics

Any media used by the computer must be initialized before its *first* use. Initialization creates a new LIF or DOS directory, (depending on the media specifier) eliminating any access to old data. The media is partitioned into physical records. The quality of the media is checked during initialization. Defective tracks are "spared" (marked so that they will not be used subsequently).

Interleave Factor

The interleave factor establishes the distance (in physical sectors) between consecutively numbered sectors. The interleave factor is ignored if the mass storage device is not a disc. If you specify 0 for the interleave factor, the default for the device is used.

Note For best performance, use the recommended interleave factor for the disc being used.

Format Option

Some mass storage devices allow you to select the sector or volume size with which the disc is initialized. Omitting this parameter or specifying 0 initializes the disc to the default sizes. Refer to the disc drive manual for options available with your disc drive.

INITIALIZE and HFS Volumes

Since INITIALIZE creates a LIF directory on LIF volumes, it *cannot* alone be used to format an HFS disc; it will still, however, scan the volume for bad sectors.

Recovering MEMORY Volume Space

BASIC RAM disc memory can be reclaimed after initializing the memory volume. To recover this memory, you would execute a line similar to the following:

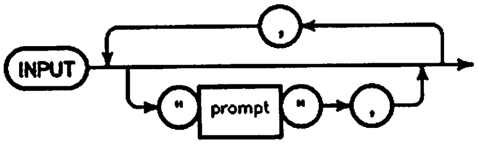
INITIALIZE ":,0,*unit number* ",0

Initializing the volume to 0 sectors removes it from memory.

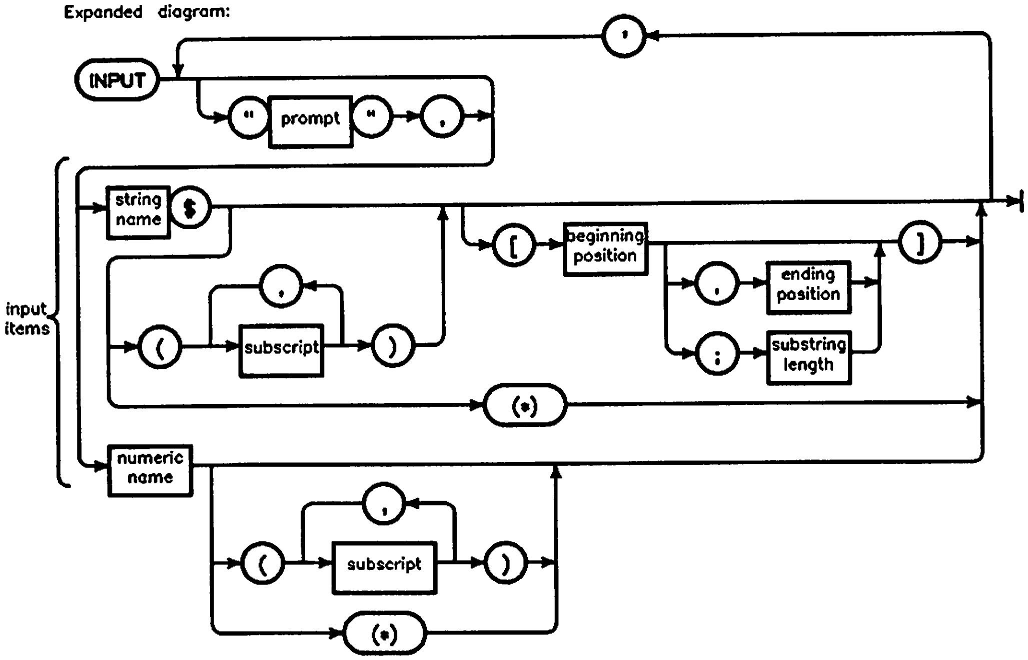
INPUT

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement is used to assign keyboard input to program variables.



Expanded diagram:



Item	Description/Default	Range Restrictions
prompt	a literal composed of characters from the keyboard. Default = question mark	—
string name	name of a string variable	any valid name
subscript	numeric expression, rounded to an integer	−32 768 thru +32 767(see "array" in Glossary)
beginning position	numeric expression, rounded to an integer	1 thru 32 767(see "substring" in Glossary)
ending position	numeric expression, rounded to an integer	0 thru 32 767(see "substring" in Glossary)
substring length	numeric expression, rounded to an integer	0 thru 32 767(see "substring" in Glossary)
numeric name	name of a numeric variable	any valid name

Example Statements

```
INPUT "Name?",N$,"ID Number?",Id
INPUT Array(*)
```

Semantics

Values can be assigned through the keyboard for any numeric or string variable, substring, array, or array element.

A prompt, which is allowed for each item in the input list, appears on the CRT display line. If the last DISP or DISP USING statement suppressed its EOL sequence, the prompt is appended to the current display line contents. If the last DISP or DISP USING did not suppress the EOL sequence, the prompt replaces the current display line contents.

Not specifying a prompt results in a question mark being used as the prompt. Specifying the null string ("") for the prompt suppresses the question mark.

To respond to the prompt, the operator enters a number or a string. Leading and trailing blank characters are deleted. Unquoted strings *may not* contain commas or quotation marks. Placing quotes around an input string allows any character(s) to be used as input. If " is intended to be a character in a quoted string, use "".

Multiple values can be entered individually or separated by commas. Press the CONTINUE, Return, EXECUTE, ENTER or STEP after the final input response. Two consecutive commas cause the corresponding variable to retain its original value. Terminating an input line with a comma retains the old values for all remaining variables in the list.

The assignment of a value to a variable in the INPUT list is done as soon as the terminator (comma or key) is encountered. Not entering data and pressing CONTINUE, ENTER, EXECUTE, Return, or STEP retains the old values for all remaining variables in the list.

If CONTINUE, ENTER, EXECUTE, or Return is pressed to end the data input, program execution continues at the next program line. If STEP is pressed, the program execution continues at the next program line in single step mode. (If the INPUT was stepped into, it is stepped out of, even if (CONTINUE), (ENTER), (EXECUTE), or (Return) is pressed.)

If too many values are supplied for an INPUT list, the extra values are ignored.

An entire array may be specified by the asterisk specifier. Inputs for the array are accepted in row major order (right most subscript varies most rapidly).

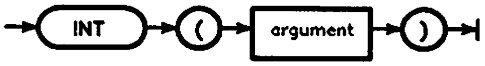
If during an INPUT statement execution, (PAUSE) or (STOP) is pressed the program pauses. The INPUT statement is re-executed, beginning with the first item, when (CONTINUE) or (STEP) is pressed. All values for that particular INPUT statement must be re-entered.

Note The above key sequences will be different for each host instrument. See your instrument-specific HP Instrument BASIC manual for detailed information.

INT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the greatest integer which is less than or equal to the expression. The result will be of the same type (REAL or INTEGER) as the argument.



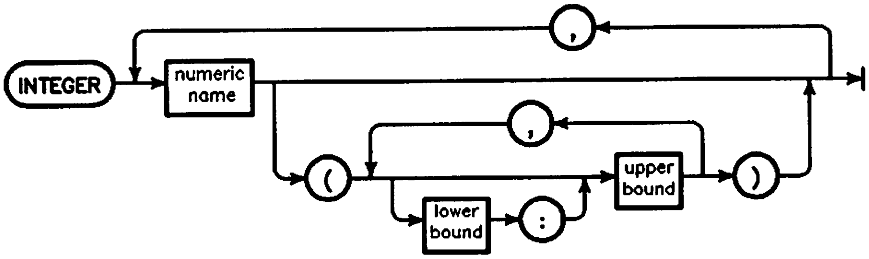
Example Statements

```
Whole=INT(Number)
IF X/2=INT(X/2) THEN Even
```

INTEGER

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This statement declares INTEGER variables, dimensions INTEGER arrays, and reserves memory for them. (For information about INTEGER as a secondary keyword, see the COM, DEF FN, or SUB statements.)



Item	Description	Range
numeric name	name of a numeric variable	any valid name
lower bound	integer constant; Default = 0	–32 767 thru +32 767 (see "array" in Glossary)
upper bound	integer constant	–32 767 thru +32 767 (see "array" in Glossary)

Example Statements

```
INTEGER I,J,K
INTEGER Array(-128:255)
```

Semantics

An INTEGER variable (or an element of an INTEGER array) uses two bytes of storage space. An INTEGER array can have a maximum of six dimensions. No single dimension can have more than 32 767 total elements.

The total number of INTEGER elements is limited by the fact that the maximum memory usage for *all* variables—INTEGER, REAL, and string—within any context is 2^{24}–1, or 16 777 215, bytes (or limited by the amount of available memory, whichever is less).

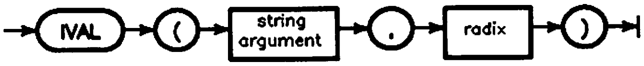
INTR

See the OFF INTR and ON INTR statements.

IVAL

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function converts a binary, octal, decimal, or hexadecimal string expression into an INTEGER.



Item	Description	Range
string argument	string expression, containing digits valid for the specified base	(see table)
radix	numeric expression, rounded to an integer.	2, 8, 10 or 16

Example Statements

```
Number=IVAL("FDF0",16)
I=IVAL("1111111111111110",2)
DISP IVAL(Octal$,8)
```

Semantics

The radix is a numeric expression that will be rounded to an integer and must evaluate to 2, 8, 10, or 16.

The string expression must contain only the characters allowed for the particular number base indicated by the radix. ASCII spaces are not allowed.

Binary strings are presumed to be in two's-complement form. If all 16 digits are specified and the leading digit is a 1, the returned value is negative.

Octal strings are presumed to be in the octal representation of two's-complement form. If all 6 digits are specified, and the leading digit is a 1, the returned value is negative.

Decimal strings containing a leading minus sign will return a negative value.

Hex strings are presumed to be in the hex representation of the two's-complement binary form. The letters A through F may be specified in either upper or lower case. If all 4 digits are specified and the leading digit is 8 through F, the returned value is negative.

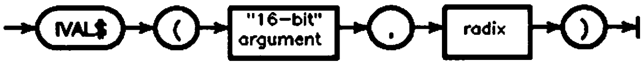
Radix	Base	String Range	String Length
2	binary	0 thru 1111111111111111	1 to 16 characters
8	octal	0 thru 177777	1 to 6 characters
10	decimal	−32 768 thru +32 768	1 to 6 characters
16	hexadecimal	0 thru FFFF	1 to 4 characters

Radix	Legal Characters	Comments
2	+,0,1	—
8	+,0,1,2,3,4,5,6,7	Range restricts the leading character. Sign must be a leading character.
10	+,−,0,1,2,3,4,5, 6,7,8,9	Sign must be a leading character.
16	+,0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F,a,b,c,d,e,f	A/a=10, B/b=11, C/c=12, D/d=13 E/e=14, F/f=15

IVAL$

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function converts an INTEGER into a binary, octal, decimal, or hexadecimal string.



Item	Description	Range
"16-bit" argument	numeric expression, rounded to an integer	(see table)
radix	numeric expression, rounded to an integer	2, 8, 10, or 16

Example Statements

```
F$=IVAL$(-1,16)
Binary$=IVAL$(Count DIV 256,2)
```

Semantics

The rounded argument must be a value that can be expressed (in binary) using 16 bits or less.

The radix must evaluate to be 2, 8, 10, or 16; representing binary, octal, decimal, or hexadecimal notation.

If the radix is 2, the returned string is in two's-complement form and contains 16 characters. If the numeric expression is negative, the leading digit will be 1. If the value is zero or positive, there will be leading zeros.

If the radix is 8, the returned string is the octal representation of the two's-complement binary form and contains 6 digits. Negative values return a leading digit of 1.

If the radix is 10, the returned string contains 6 characters. Leading zeros are added to the string if necessary. Negative values have a leading minus sign.

If the radix is 16, the returned string is the hexadecimal representation of the two's-complement binary form and contains 4 characters. Negative values return a leading digit in the range 8 thru F.

Radix	Base	Range of Returned String	String Length
2	binary	0000000000000000 thru 1111111111111111	16 characters
8	octal	000000 thru 177777	6 characters
10	decimal	−32 768 thru +32 768	6 characters
16	hexadecimal	0000 thru FFFF	4 characters

KBD

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This INTEGER function returns a 2, the select code of the keyboard.

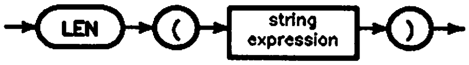
Example Statements

```
OUTPUT KBD;Clear$;
```

LEN

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the current number of characters in the argument. The length of the null string ("") is 0.



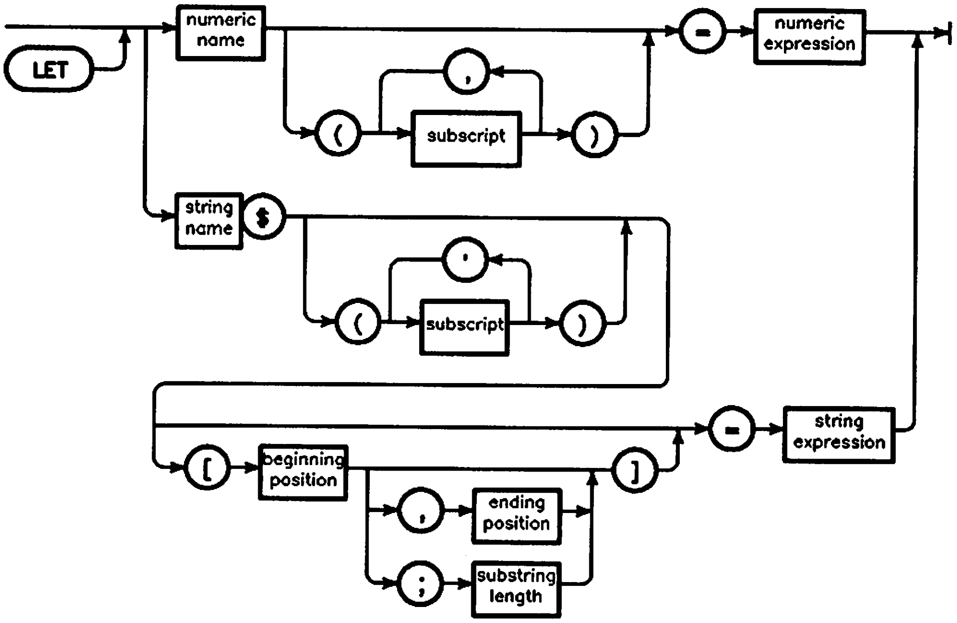
Example Statements

```
Last=LEN(String$)
IF NOT LEN(A$) THEN Empty
```

LET

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This is the assignment statement, which is used to assign values to variables.



Item	Description	Range
numeric name	name of a numeric variable	any valid name
string name	name of a string variable	any valid name
subscript	numeric expression, rounded to an integer	−32 767 thru +32 767 (see "array" in Glossary)
beginning position	numeric expression, rounded to an integer	1 thru 32 767 (see "substring" in Glossary)
ending position	numeric expression, rounded to an integer	0 thru 32 767 (see "substring" in Glossary)
substring length	numeric expression, rounded to an integer	0 thru 32 767 (see "substring" in Glossary)

Example Statements

```
LET Number=33
Array(I+1)=Array(I)/2
String$="Hello Scott"
A$(7)[1;2]=CHR$(27)&"Z"
```

Semantics

The assignment is done to the variable which is to the left of the equals sign. Only one assignment may be performed in a LET statement; any other equal signs are considered relational operators, and must be enclosed in a parenthetical expression (i.e., `A=A+(B=1)+5`). A variable can occur on both sides of the assignment operator (i.e., `I=I+1` or `Source$=Source$&Temp$`).

A real expression will be rounded when assigned to an INTEGER variable, if it is within the INTEGER range. Out-of-range assignments to an INTEGER give an error.

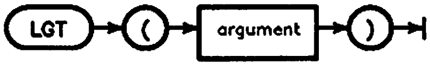
The length of the string expression must be less than or equal to the dimensioned length of the string it is being assigned to. Assignments may be made into substrings, using the normal rules for substring definition. The string expression will be truncated or blank-filled on the right (if necessary) to fit the destination substring when the substring has an explicitly stated length. If only the beginning position of the substring is specified, the substring will be replaced by the string expression and the length of the recipient string variable will be adjusted accordingly; however, error 18 is reported if the expression overflows the recipient string variable.

If the name of the variable to the left of the equal sign begins with AND, DIV, EXOR, MOD or OR (the name of an operator) and the keyword LET is omitted, the prefix must have at least one uppercase letter and one lowercase letter in it.

LGT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the logarithm (base 10) of its argument.



Item	Description/Default	Range Restrictions
argument	numeric expression	> 0 for INTEGER and REAL arguments

Examples Statements

```
Decibel=20*LGT(Volts)
PRINT "Log base 10 of ";X;"=";LGT(X)
```

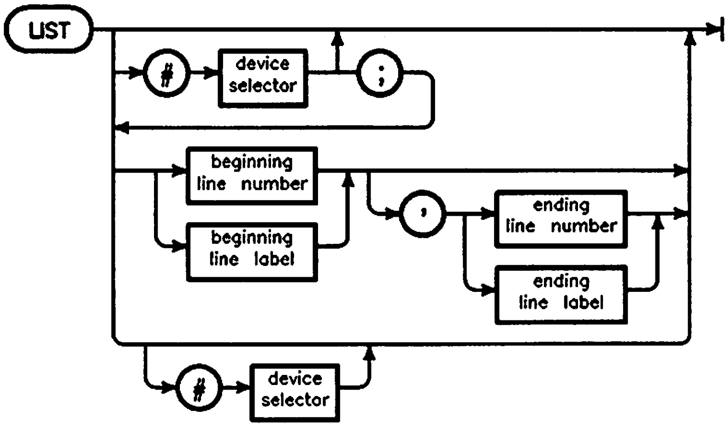
Semantics

If the argument is REAL or INTEGER, the value returned is REAL.

LIST

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement allows you to list the program or the key definitions currently in memory.



Item	Description	Range
device selector	numeric expression; is rounded to an integer. Default is PRINTER IS device.	(see Glossary)
beginning line number	integer constant identifying program line	1 thru 32 766
beginning line label	name of a program line	any valid name
ending line number	integer constant identifying program line	1 thru 32 766
ending line label	name of a program line	any valid name

Example Statements

```
LIST
LIST #701
LIST 100,Label1
```

Semantics

When a label is used as a line identifier, the lowest-numbered line in memory having that label is used. When a number is used as a line identifier, the lowest-numbered line in memory having a number equal to or greater than the specified line is used. An error occurs if the ending line identifier occurs before the beginning line identifier or if a specified line label does not exist in the program.

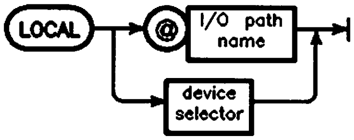
After the listing is finished, the amount of available memory, in bytes, is displayed on the CRT.

Note that the default width of the PRINTER IS device is 80 characters, which means that a carriage-return (CR) and line-feed (LF) character will be sent after 80 characters are printed on any one line. You can change this, however, with the WIDTH attribute of the PRINTER IS statement.

LOCAL

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement returns all specified devices to their local state.



Item	Description	Range
I/O path name	name assigned to a device or devices	any valid name (see ASSIGN)
device selector	numeric expression, rounded to an integer	(see GLOSSARY)

Example Statements

```
LOCAL @Dvm
LOCAL 7
```

Semantics

If only an interface select code is specified by the I/O path name or device selector, all devices on the bus are returned to their local state by setting REN false. Any existing LOCAL LOCKOUT is cancelled.

If a primary address is included, the GTL message (Go To Local) is sent to all listeners. LOCAL LOCKOUT is not cancelled.

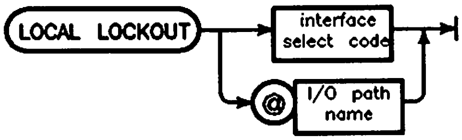
Summary of Bus Actions

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active/Controller	REN $\overline{\text{ATN}}$	ATN MTA UNL LAG	ATN GTL	ATN MTA UNL LAG GTL
Not Active Controller	REN	Error	Error	Error

LOCAL LOCKOUT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This HP-IB statement sends the LLO (local lockout) message, preventing an operator from returning the specified device to local (front panel) control.



Summary of Bus Actions

Item	Description	Range
I/O path name	name assigned to an interface select code	any valid name (see ASSIGN)
interface select code	numeric expression, rounded to an integer	7 thru 31 (See your instrument manual)

Example Statements

```
LOCAL LOCKOUT 7
LOCAL LOCKOUT @Hpib
```

Semantics

The computer must be the active controller to execute LOCAL LOCKOUT.

If a device is in the LOCAL state when this message is sent, it does not take effect on that device until the device receives a REMOTE message and becomes addressed to listen.

LOCAL LOCKOUT does not cause bus reconfiguration, but issues a universal bus command received by all devices on the interface whether addressed or not. The command sequence is ATN and LLO.

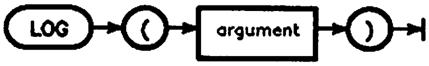
Summary of Bus Actions

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active/Controller	ATN LLO	Error	ATN LLO	Error
Not Active Controller	Error	Error	Error	Error

LOG

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the logarithm (base e) of its argument.



Item	Description/Default	Range Restrictions
argument	numeric expression	> 0

Examples Statements

```
Time=-1*Rc*LOG(Volts/Emf)
PRINT "Natural log of ";Y;"=";LOG(Y)
```

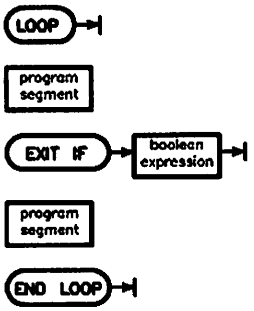
Semantics

If the argument is REAL or INTEGER, the value returned is REAL.

LOOP

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This construct defines a loop which is repeated until the boolean expression in an EXIT IF statement evaluates to be logically true (evaluates to a non-zero value).



Item	Description	Range
boolean expression	numeric expression; evaluated as true if nonzero and false if 0	—
program segment	any number of contiguous program lines not containing the beginning or end of a main program or subprogram, but which may contain properly nested construct(s).	—

Example Program Segments

```
460   LOOP
470   EXIT IF LEN(A$)<2
480     P=POS(A$,Delim$)
490   EXIT IF NOT P
500     A$=A$[1,P-1]&A$[P+2]
510   END LOOP
```

Semantics

The LOOP ... END LOOP construct allows continuous looping with conditional exits which depend on the outcome of relational tests placed within the program segments. The program segments to be repeated start with the LOOP statement and end with END LOOP. Reaching the END LOOP statement will result in a branch to the first program line after the LOOP statement.

Any number of EXIT IF statements may be placed within the construct to escape from the loop. The only restriction upon the placement of the EXIT IF statements is that they must not be part of any other construct which is nested within the LOOP ... END LOOP construct.

If the specified conditional test is true, a branch to the first program line following the END LOOP statement is performed. If the test is false, execution continues with the next program line within the construct.

Branching into a LOOP ... END LOOP construct (via a GOTO) results in normal execution from the point of entry. Any EXIT IF statement encountered will be executed. If execution reaches END LOOP, a branch is made back to the LOOP statement, and execution continues as if the construct had been entered normally.

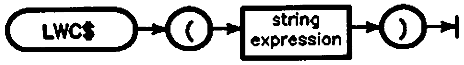
Nesting Constructs Properly

LOOP ... END LOOP may be placed within other constructs, provided it begins and ends before the outer construct can end.

LWC$

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function replaces any uppercase characters with their corresponding lowercase characters.



Example Statements

```
Lower$=LWC$("UPPER")
IF LWC$(Yes$)="y" THEN True_test
```

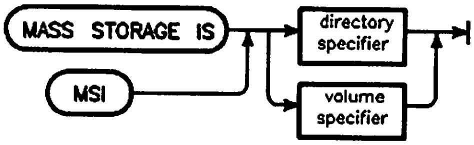
Semantics

The LWC$ function converts only uppercase alphabetic characters to their corresponding lowercase characters and will not alter numerals or special characters.

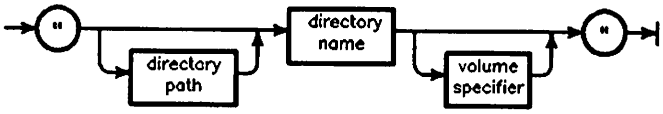
MASS STORAGE IS

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

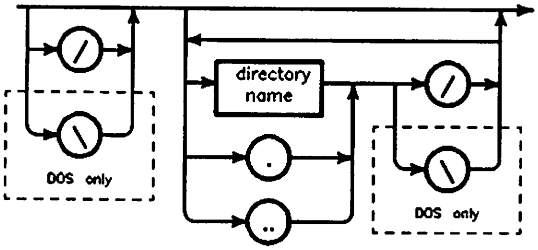
This statement specifies the system mass storage device.



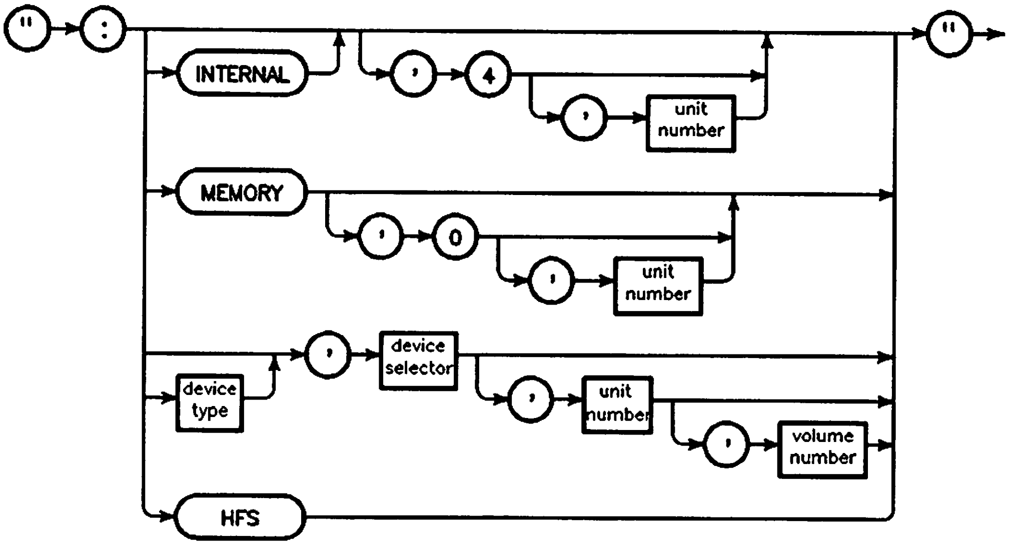
literal form of directory specifier (HFS and DOS volumes only):



directory path:



literal form of volume specifier:



Item	Description	Range
directory specifier	string expression	(see drawing)
volume specifier	string expression	(see drawing)
directory path	literal	(see drawing)
directory name	literal	depends on volume's format (14 characters for HFS; 255 characters for HFS long file name systems; see Glossary for details)
device type	literal	(see Semantics)
device selector	integer constant	(see Glossary)
unit number	integer constant; Default = 0	0 thru 255 (device-dependent)
volume number	integer constant; Default = 0	(device-dependent)

Example Statements

```
MASS STORAGE IS Vol_specifier$
MSI ":,700"
MSI ":INTERNAL,4,1"
MSI ":X,12"
MASS STORAGE IS Dir_path$&Vol_specifier$
MSI "/Dir1/Dir2/MyDir"
MSI "../.."
MSI "\Dir1\Dir2\MyDir"                    DOS Only
```

Semantics

All mass storage operations which do not specify a source or destination by either an I/O path name or volume specifier in the file specifier use the current system mass storage device.

MASS STORAGE IS can be abbreviated as MSI when entering a program line, but a program listing always shows the unabbreviated keywords.

Note The current mass storage device is not associated in anyway with the host device's current storage configuration.

Device Type

Except for the INTERNAL and MEMORY device types, the device type is ignored in HP Instrument BASIC.

Non-Disc Mass Storage

Memory volumes are created by the INITIALIZE statement. They are removed by SCRATCH A or by turning off the power. The unit number for a MEMORY volume must be 0.

MSI with HFS Volumes

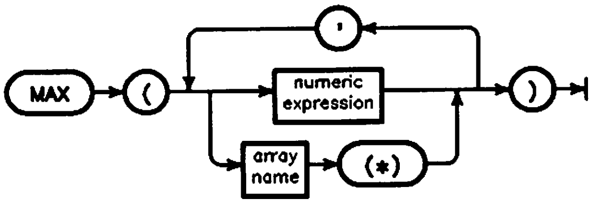
With hierarchical volumes (such as HFS), MASS STORAGE IS can also be used to specify the current working directory.

In order to specify an HFS directory as the current working directory, you need to have X (search) permission of the immediately superior directory as well as on all other superior directories.

MAX

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns a value equal to the largest value in the list of arguments provided. If an array is specified as part of the list of arguments, it is equivalent to listing all the values in the array. An INTEGER is returned if and only if all arguments in the list are INTEGER.



Item	Description	Range
array name	name of a numeric array	any valid name

Example Statements

```
X=MAX(A(*))
X=MAX(A,3,B)
X=MAX(Floor,MIN(Ceiling,Argument))
```

MAXREAL

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the largest positive REAL number available in the range of the machine.

Example Statements

```
A=MAXREAL
IF A*B<MAXREAL/(10^N) THEN GOTO 100
```

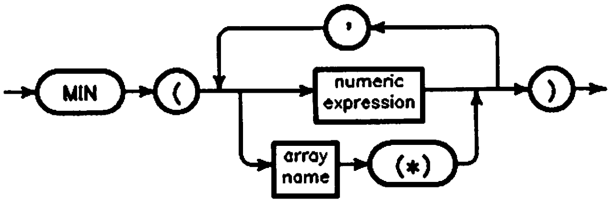
Semantics

The value of MAXREAL is approximately 1.797 693 134 862 32 E+308.

MIN

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns a value equal to the smallest value in the list of arguments provided. If an array is specified as part of the list of arguments, it is equivalent to listing all the values in the array. An INTEGER is returned if and only if all arguments in the list are INTEGER.



Item	Description	Range
array name	name of a numeric array	any valid name

Example Statements

```
X=MIN(A(*))
X=MIN(A,3,B)
X=MIN(Ceiling,MAX(Floor,Argument))
```

MINREAL

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the smallest positive REAL number available in the range of the computer.

Example Statements

```
A=MINREAL
IF A-B>MINREAL*(10^N) THEN GOTO 100
```

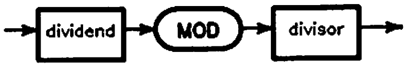
Semantics

The value of MINREAL is approximately 2.225 073 858 507 2 4E–308.

MOD

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This operator returns the remainder of a division.



Item	Description	Range
dividend	numeric expression	—
divisor	numeric expression	not equal to 0

Example Statements

```
Remainder=Dividend MOD Divisor
PRINT "Seconds =";Time MOD 60
```

Semantics

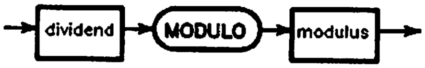
MOD returns an INTEGER value if both arguments are INTEGER. Otherwise the returned value is REAL.

For INTEGERs, MOD is equivalent to X − Y × (X DIV Y). This may return a different result from the modulus function on other computers when negative numbers are involved.

MODULO

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This operator returns the integer remainder resulting from a division.



Item	Description	Range
dividend	numeric expression	range of REAL
modulus	numeric expression	range of REAL, ≠ 0

Example Statements

```
Remainder=Dividend MODULO Modulus
A=B MODULO C
```

Semantics

X MODULO Y is equivalent to X−Y×INT(X/Y).

The result satisfies:

```
0 <= (X MODULO Y) < Y if Y>0
Y < (X MODULO Y) <= 0 if Y<0
```

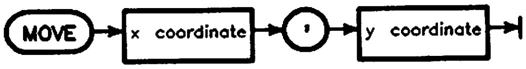
The type of the result is the higher of the types of the two operands. If the modulus is zero error 31 occurs.

MODULO returns the remainder of a division.

MOVE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement moves both the logical and physical pens from the current pen position to the specified X and Y coordinates.



Item	Description	Range
x coordinate	numeric expression in current units	*
y coordinate	numeric expression in current units	*

**See your instrument-specific HP Instrument BASIC manual for details on valid x and y coordinate ranges.*

Example Statements

```
MOVE 10,75
MOVE Next_x,Next_y
```

Semantics

If both current physical pen position and specified pen position are outside current clip limits, no physical pen movement is made; however, the logical pen position is moved to the specified coordinates.

MSI

See the MASS STORAGE IS statement.

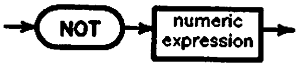
NEXT

See the FOR ... NEXT construct.

NOT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This operator returns 1 if its argument equals 0. Otherwise, 0 is returned.



Example Statements

```
Invert_flag=NOT Std_device
IF NOT Pointer THEN Next_op
```

Semantics

When evaluating the argument, a non-zero value (positive or negative) is treated as a logical 1; only zero is treated as a logical 0.

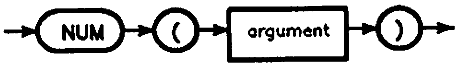
The logical complement is shown below:

A	NOT A
0	1
1	0

NUM

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the decimal value of the ASCII code of the first character in the argument. The range of returned values is 0 thru 255.



Item	Description	Range
argument	string expression	not a null string

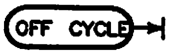
Example Statements

```
Letter=NUM(String$)
A$[I;1]=CHR$(NUM(A$[I])+32)
```

OFF CYCLE

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement cancels event-initiated branches previously defined and enabled by an ON CYCLE statement.



Example Statements

```
OFF CYCLE
IF Kick_stand THEN OFF CYCLE
```

Semantics

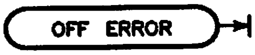
OFF CYCLE destroys the log of any CYCLE event which has already occurred but which has not been serviced.

If OFF CYCLE is executed in a subprogram such that it cancels an ON CYCLE in the calling context, the ON CYCLE definition is restored upon returning to the calling context.

OFF ERROR

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

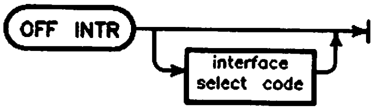
This statement cancels event-initiated branches previously defined and enabled by an ON ERROR statement. Further errors are reported to the user in the usual fashion.



OFF INTR

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement cancels event-initiated branches previously defined by an ON INTR statement.



Item	Description	Range
interface select code	numeric expression, rounded to an integer; Default = all interfaces	5, and 7 thru 31*

**See your instrument-specific HP Instrument BASIC manual for valid ranges for interface select codes.*

Example Statements

```
OFF INTR
OFF INTR Hpib
```

Semantics

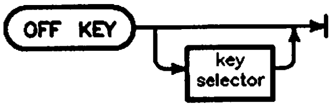
Not specifying an interface select code disables the event-initiated branches for all interfaces. Specifying an interface select code causes the OFF INTR to apply to the event-initiated log entry for the specified interface only.

Any pending ON INTR branches for the effected interfaces are lost and further interrupts are ignored.

OFF KEY

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement cancels event-initiated branches previously defined and enabled by an ON KEY statement.



Item	Description	Range
key selector	numeric expression, rounded to an integer; Default = all keys	0 thru 19*

**See your instrument-specific HP Instrument BASIC manual for valid key selectors.*

Example Statements

```
OFF KEY
OFF KEY 4
```

Semantics

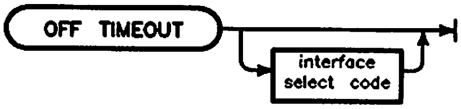
Not specifying a softkey number disables the event-initiated branches for all softkeys. Specifying a softkey number causes the OFF KEY to apply to the specified softkey only. If OFF KEY is executed in a subprogram and cancels an ON KEY in the context which called the subprogram, the ON KEY definitions are restored when the calling context is restored.

Any pending ON KEY branches for the effected softkeys are lost.

OFF TIMEOUT

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement cancels event-initiated branches previously defined and enabled by an ON TIMEOUT statement.



Item	Description	Range
interface select code	numeric expression, rounded to an integer; Default = all interfaces	7 thru 31*

**See your instrument-specific HP Instrument BASIC manual for valid ranges for interface select codes.*

Example Statements

```
OFF TIMEOUT
OFF TIMEOUT Isc
```

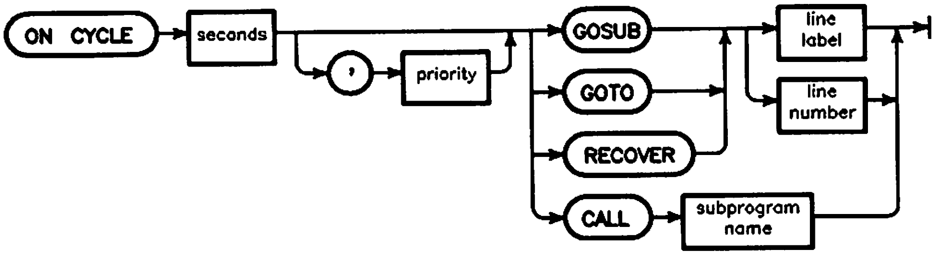
Semantics

Not specifying an interface select code disables the event-initiated branches for all interfaces. Specifying an interface select code causes the OFF TIMEOUT to apply to the event-initiated branches for the specified interface only. When OFF TIMEOUT is executed, no more timeouts can occur on the effected interfaces.

ON CYCLE

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement defines and enables an event-initiated branch to be taken each time the specified number of seconds has elapsed.



Item	Description	Range
seconds	numeric expression, rounded to the nearest 0.02 second	0.01 thru 167 772.16
priority	numeric expression, rounded to an integer; Default=1	1 thru 15
line label	name of a program line	any valid name
line number	integer constant identifying a program line	1 thru 32 766
subprogram name	name of a SUB or CSUB subprogram	any valid name

Example Statements

```
ON CYCLE 1 GOSUB One_second
ON CYCLE 3600,12 CALL Chime
```

Semantics

The most recent ON CYCLE (or OFF CYCLE) definition overrides any previous ON CYCLE definition. If the overriding ON CYCLE definition occurs in a context different from the one in which the overridden ON CYCLE occurs, the overridden ON CYCLE is restored when the calling context is restored, but the time value of the more recent ON CYCLE remains in effect.

The priority can be specified, with the highest priority represented by 15. The highest user-defined priority (15) is less than the priority for ON ERROR, ON END, and ON TIMEOUT (whose priorities are not user-definable). ON CYCLE can interrupt service routines of other event-initiated branches with user-definable priorities, if the ON CYCLE priority is higher than the priority of the service routine (the current system priority). CALL and GOSUB service routines get the priority specified in the ON ... statement which set up the branch that invoked them. The system priority is not changed when a GOTO branch is taken.

Any specified line label or line number must be in the same context as the ON CYCLE statement. CALL and GOSUB will return to the next line that would have been executed if the CYCLE event had not been serviced, and the system priority is restored to that which existed before the ON CYCLE branch was taken. RECOVER forces the program to go directly to the specified line in the context containing that ON CYCLE statement. When RECOVER forces a change of context, the system priority is restored to that which existed in the original (defining) context at the time that context was exited.

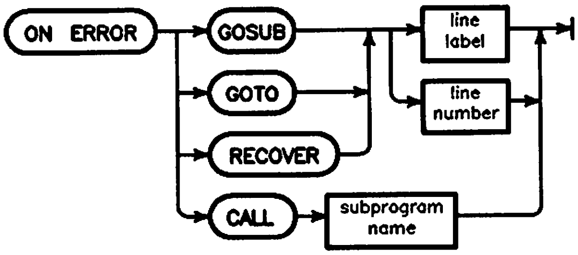
CALL and RECOVER remain active when the context changes to a subprogram, unless the change in context is caused by a keyboard-originated call. GOSUB and GOTO remain active when the context changes to a subprogram, but the branch cannot be taken until the calling context is restored.

ON CYCLE is disabled by DISABLE and deactivated by OFF CYCLE. If the cycle value is short enough that the computer cannot service it, the interrupt will be lost.

ON ERROR

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement defines and enables an event-initiated branch which results from a trappable error. This allows you to write your own error-handling routines.



Item	Description	Range
line label	name of a program line	any valid name
line number	integer constant identifying a program line	1 thru 32 766
subprogram name	name of a SUB subprogram	any valid name

Example Statements

```
ON ERROR GOTO 1200
ON ERROR CALL Report
```

Semantics

The ON ERROR statement has the highest priority of any event-initiated branch. ON ERROR can interrupt any event-initiated service routine.

Any specified line label or line number must be in the same context as the ON ERROR statement. RECOVER forces the program to go directly to the specified line in the context containing the ON ERROR statement.

Returns from ON ERROR GOSUB or ON ERROR CALL routines are different from regular GOSUB or CALL returns. When ON ERROR is in effect, the program resumes at the beginning of the line where the error occurred. If the ON ERROR routine did not correct the cause of the error, the error is repeated. This causes an infinite loop between the line in error and the error handling routine. To avoid a retry of the line which caused the error, use ERROR RETURN instead of RETURN or ERROR SUBEXIT instead of SUBEXIT. When execution returns from the ON ERROR routine, system priority is restored to that which existed before the ON ERROR branch was taken.

CALL and RECOVER remain active when the context changes to a subprogram, GOSUB and GOTO do not remain active when the context changes to a subprogram. If an error occurs, the error is reported to the user, as if ON ERROR had not been executed.

If an execution error occurs while servicing an ON ERROR CALL or ON ERROR GOSUB, program execution stops. If an execution error occurs while servicing an ON ERROR GOTO or ON ERROR RECOVER routine, an infinite loop can occur between the line in error and the GOTO or RECOVER routine.

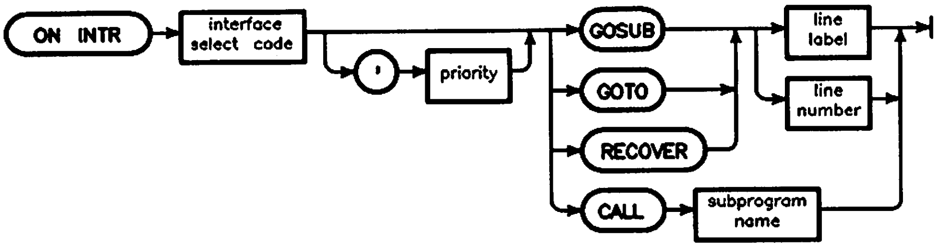
If an ON ERROR routine cannot be serviced because inadequate memory is available for the computer; the original error is reported and program execution pauses at that point.

ON ERROR is deactivated by OFF ERROR. DISABLE does not affect ON ERROR.

ON INTR

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement defines an event-initiated branch to be taken when an interface card generates an interrupt. The interrupts must be explicitly enabled with an ENABLE INTR statement.



Item	Description	Range
interface select code	numeric expression, rounded to an integer	5, 7 thru 31*
priority	numeric expression, rounded to an integer; Default=1	1 thru 15
line label	name of a program line	any valid name
line number	integer constant identifying a program line	1 thru 32 766
subprogram name	name of a SUB subprogram	any valid name

**See your instrument-specific HP Instrument BASIC manual for valid ranges for interface select codes.*

Example Statements

```
ON INTR 7 GOSUB 500
ON INTR Isc,4 CALL Service
```

Semantics

The occurrence of an interrupt performs an implicit DISABLE INTR for the interface. An ENABLE INTR must be performed to re-enable the interface for subsequent event-initiated branches. Another ON INTR is not required, nor must the mask for ENABLE INTR be redefined.

The priority can be specified, with highest priority represented by 15. The highest priority is less than the priority for ON ERROR, ON END, and ON TIMEOUT. ON INTR can interrupt service routines of other event-initiated branches which have user-definable priorities, if the ON INTR priority is higher than the priority of the service routine (the current system priority). CALL and GOSUB service routines get the priority specified in the ON ...

statement which set up the branch that invoked them. The system priority is not changed when a GOTO branch is taken.

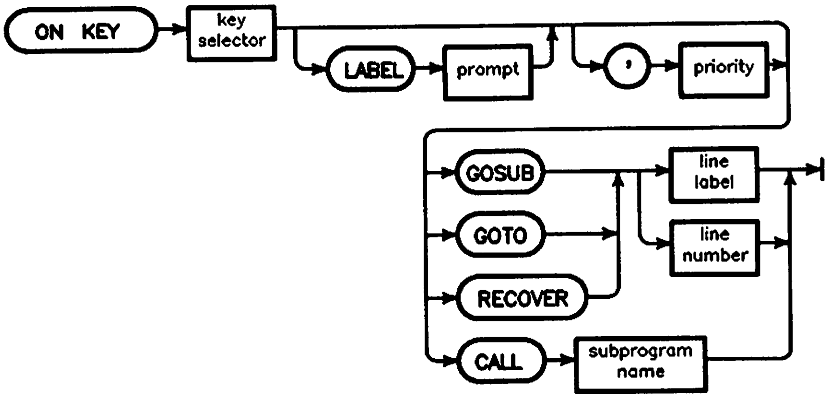
Any specified line label or line number must be in the same context as the ON INTR statement. CALL and GOSUB will return to the next line that would have been executed if the INTR event had not been serviced, and the system priority is restored to that which existed before the ON INTR branch was taken. RECOVER forces the program to go directly to the specified line in the context containing that ON INTR statement. When RECOVER forces a change of context, the system priority is restored to that which existed in the original (defining) context at the time that context was exited.

ON INTR is disabled by DISABLE INTR or DISABLE and deactivated by OFF INTR.

ON KEY

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement defines and enables an event-initiated branch to be taken when a softkey is pressed.



Item	Description	Range
key selector	numeric expression, rounded to an integer	0 thru 23*
prompt	string expression; Default = no label	—
priority	numeric expression, rounded to an integer; Default=1	1 thru 15
line label	name of a program line	any valid name
line number	integer constant identifying a program line	1 thru 32 766
subprogram name	name of a SUB subprogram	any valid name

**See your instrument-specific HP Instrument BASIC manual for valid key selectors.*

Example Statements

```
ON KEY 0 GOTO 150
ON KEY 5 LABEL "Print",3 GOSUB Report
```

Semantics

The most recently executed ON KEY (or OFF KEY) definition for a particular softkey overrides any previous key definition. If the overriding ON KEY definition occurs in a context different from the one in which the overridden ON KEY occurs, the overridden ON KEY is restored when the calling context is restored.

Labels appear on the CRT. The label of any key is bound to the current ON KEY definition. Therefore, when a definition is changed or restored, the label changes accordingly.

The priority can be specified, with the highest priority represented by 15. The highest user-defined priority (15) is less than the priority for ON ERROR, ON END, and ON TIMEOUT (whose priorities are not user-definable). On KEY can interrupt service routines of other event-initiated branches with user-definable priorities, if the ON KEY priority is higher than the priority of the service routine (the current system priority). CALL and GOSUB service routines get the priority specified in the ON ... statement which set up the branch that invoked them. The system priority is not changed when a GOTO branch is taken.

Any specified line label or line number must be in the same context as the ON KEY statement. CALL and GOSUB will return to the next line that would have been executed if the KEY event had not been serviced, and the system priority is restored to that which existed before the ON KEY branch was taken. RECOVER forces the program to go directly to the specified line in the context containing that ON KEY statement. When RECOVER forces a change of context, the system priority is restored to that which existed in the original (defining) context at the time that context was exited.

CALL and RECOVER remain active when the context changes to a subprogram.

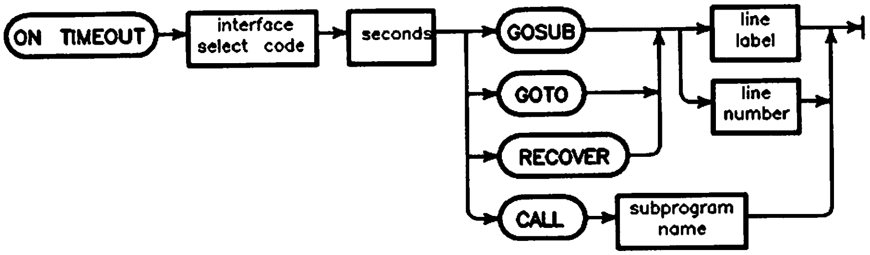
GOSUB and GOTO remain active when the context changes to a subprogram, but the branch cannot be taken until the calling context is restored.

ON KEY is disabled by DISABLE, deactivated by OFF KEY, and temporarily deactivated when the program is paused or executing INPUT, or ENTER KBD.

ON TIMEOUT

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement defines and enables an event-initiated branch to be taken when an I/O timeout occurs on the specified interface.



Item	Description	Range
interface select code	numeric expression, rounded to an integer	7 thru 31*
seconds	numeric expression	0.001 thru 32.767
line label	name of a program line	any valid name
line number	integer constant identifying a program line	1 thru 32 766
subprogram name	name of a SUB subprogram	any valid name

**See your instrument-specific HP Instrument BASIC manual for valid ranges for interface select codes.*

Example Statements

```
ON TIMEOUT 7,2.544 GOTO 770
ON TIMEOUT Printer,Time GOSUB Message
```

Semantics

There is no default system timeout. If ON TIMEOUT is not in effect for an interface, a device can cause the program to wait forever.

The specified branch occurs if an input or output is active on the interface and the interface has not responded within the number of seconds specified. The computer waits at least the specified time before generating an interrupt; however, it may wait up to an additional 25% of the specified time.

Timeouts apply to ENTER and OUTPUT statements, and operations involving the PRINTER IS, PRINTALL IS, and PLOTTER IS devices when they are external. Timeouts do not apply to CRT alpha or graphics I/O, real time clock I/O, keyboard I/O, or mass storage operations.

The priority associated with ON TIMEOUT is higher than priority 15. ON END and ON ERROR have the same priority as ON TIMEOUT, and can interrupt an ON TIMEOUT service routine.

Any specified line label or line number must be in the same context as the ON TIMEOUT statement. CALL and GOSUB will return to the line immediately following the one during which the timeout occurred, and the system priority is restored to that which existed before the ON TIMEOUT branch was taken. RECOVER forces the program to go directly to the specified line in the context containing that ON TIMEOUT statement. When RECOVER forces a change of context, the system priority is restored to that which existed in the original (defining) context at the time that context was exited.

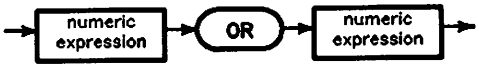
CALL and RECOVER remain active when the context changes to a subprogram, unless the change in context is caused by a keyboard-originated call. GOSUB and GOTO do not remain active when the context changes to a subprogram. The TIMEOUT event does remain active. Unlike other ON events, TIMEOUTs are never logged, they always cause an immediate action. If a TIMEOUT occurs when the ON TIMEOUT branch cannot be taken, an error 168 is generated. This can be trapped with ON ERROR. The functions ERRN and ERRDs are set *only* when the error is generated. They are not set when the ON TIMEOUT branch can be taken.

ON TIMEOUT is deactivated by OFF TIMEOUT. DISABLE does not affect ON TIMEOUT.

OR

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This operator returns a 1 or a 0 based on the logical inclusive-or of the arguments.



Example Statements

```
X=Y OR Z
IF File_type OR Device THEN Process
```

Semantics

An expression which evaluates to a non-zero value is treated as a logical 1. An expression must evaluate to a zero to be treated as a logical 0.

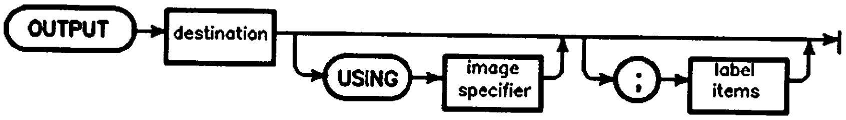
The truth table is:

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

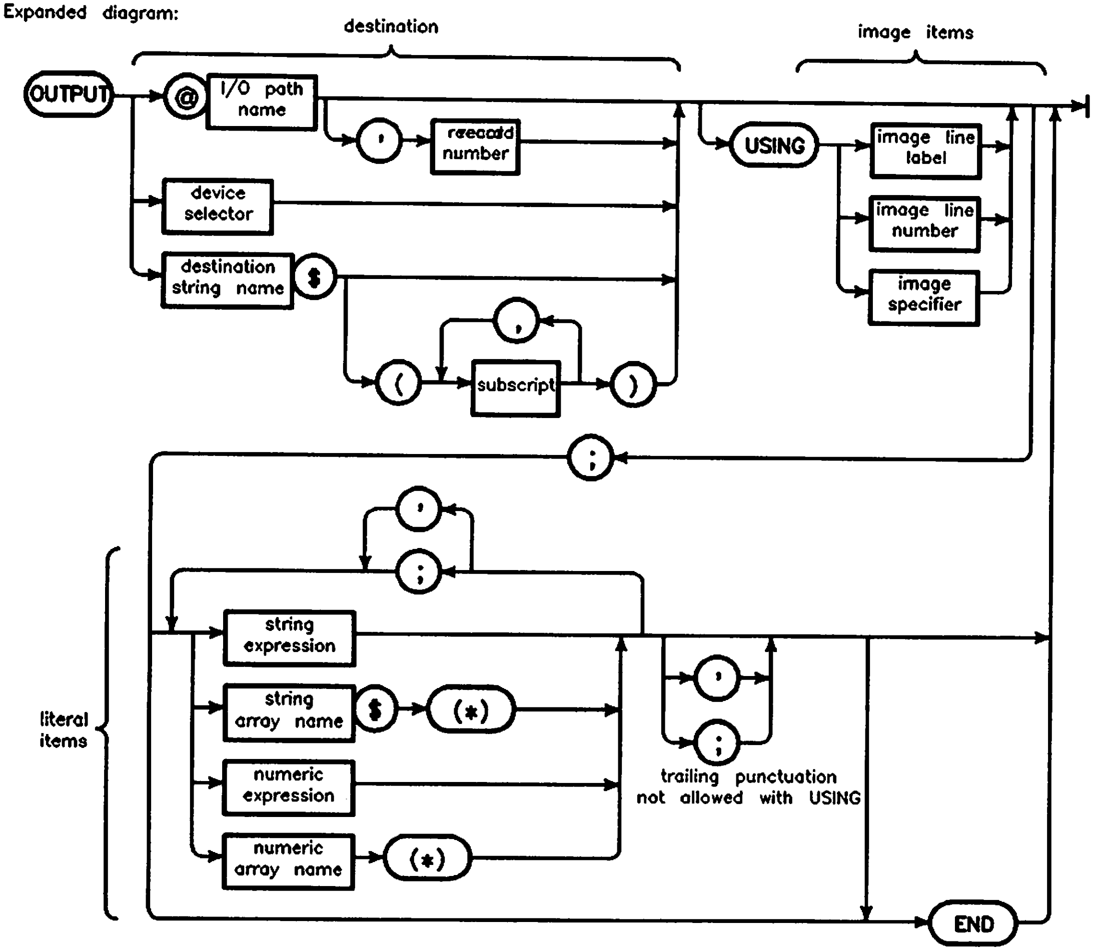
OUTPUT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

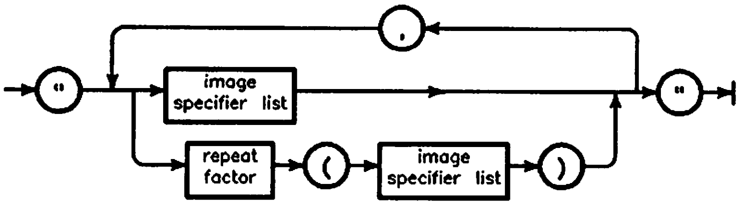
This statement outputs items to the specified destination.



Expanded diagram:



literal form of image specifier



Item	Description	Range
I/O path name	name assigned to a device, or mass storage file	any valid name
record number	numeric expression, rounded to an integer	1 thru $2^{31}-1$
device selector	numeric expression, rounded to an integer	(see Glossary)
destination string name	name of a string variable	any valid name
subscript	numeric expression, rounded to an integer	−32 767 thru +32 767 (see "array" in Glossary)
image line number	integer constant identifying an IMAGE statement	1 thru 32 766
image line label	name identifying an IMAGE statement	any valid name
image specifier	string expression	(see drawing)
string array name	name of a string array	any valid name
numeric array name	name of a numeric array	any valid name
image specifier list	literal	(see next drawing)
repeat factor	integer constant	1 thru 32 767
literal	string constant composed of characters from the keyboard	quote mark not allowed

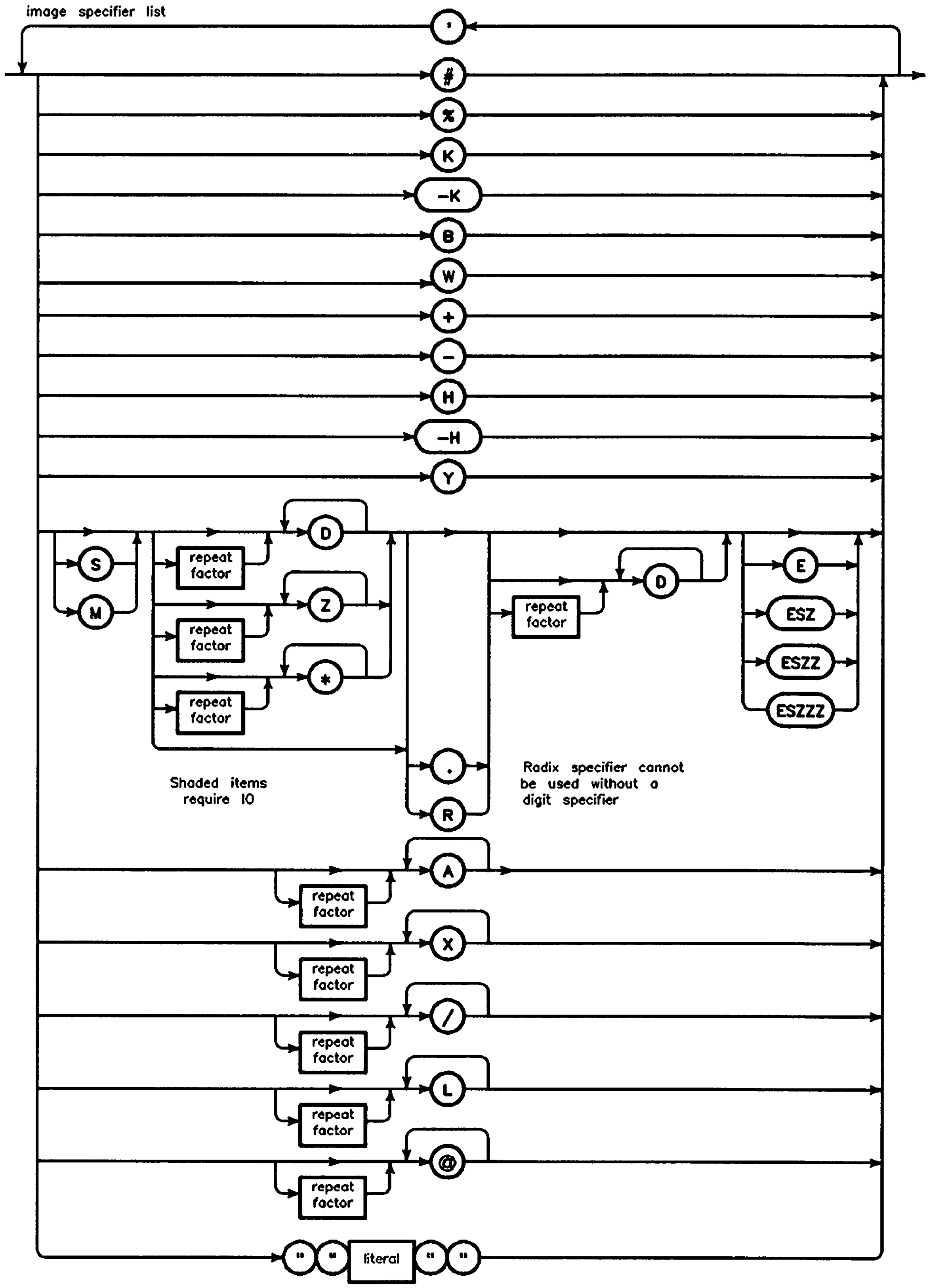


image specifier list
,
#
%
K
-K
B
W
+
-
H
-H
Y
S
M
repeat factor
D
repeat factor
Z
repeat factor
*
.
R
repeat factor
D
E
ESZ
ESZZ
ESZZZ
Shaded items require IO
Radix specifier cannot be used without a digit specifier
repeat factor
A
repeat factor
X
repeat factor
/
repeat factor
L
repeat factor
@
"
"
literal
"
"

Example Statements

```
OUTPUT 701;Number,String$;
OUTPUT @File;Array(*),END
OUTPUT @Rand,5 USING Fmt1;Item(5)
OUTPUT 12 USING "#,6A";B$[2;6]
OUTPUT @Printer;Rank;Id;Name$
```

Semantics

Standard Numeric Format

The standard numeric format depends on the value of the number being displayed. If the absolute value of the number is greater than or equal to 1E-4 and less than 1E+6, it is rounded to 12 digits and displayed in floating point notation. If it is not within these limits, it is displayed in scientific notation. The standard numeric format is used unless USING is selected, and may be specified by using K in an image specifier.

Arrays

Entire arrays may be output by using the asterisk specifier. Each element in an array is treated as an item by the OUTPUT statement, as if the items were listed separately, separated by the punctuation following the array specifier. If no punctation follows the array specifier, a comma is assumed. The array is output in row major order (rightmost subscript varies fastest).

Files as Destination

If an I/O path has been assigned to a file, the file may be written to with OUTPUT statements. The file must be an ASCII, BDAT, or HP-UX file. The attributes specified in the ASSIGN statement are used if the file is a BDAT or HP-UX file (ASCII files are always assigned a special case of the FORMAT ON attribute).

Serial access is available for ASCII, BDAT, and HP-UX files. Random access is available for BDAT and HP-UX files. The end-of-file marker (EOF) and the file pointer are important to both serial and random access. The file pointer is set to the beginning of the file when the file is opened by an ASSIGN. It is updated by OUTPUT operations so that it always points to the next byte to be written.

The EOF pointer is read from the media when the file is opened by an ASSIGN. On a newly created file, EOF is set to the beginning of the file. After each OUTPUT operation, the EOF pointer in the I/O path table is updated to the maximum of the file pointer or the previous EOF value. The EOF pointer on the volume is updated at the following times:

- When the current end-of-file changes.
- When END is specified in an OUTPUT statement directed to the file.

Random access uses the record number parameter to write items to a specific location in a file. The OUTPUT begins at the start of the specified record and must fit into one record. The record specified cannot be beyond the record containing the EOF, if EOF is at the first byte of a record. The record specified can be one record beyond the record containing the EOF, if EOF is not at the first byte of a record. Random access is always allowed to records preceding the EOF record. If you wish to write randomly to a newly created file, either use a

CONTROL statement to position the EOF in the last record, or write some "dummy" data into every record.

When data is written to an ASCII file, each item is sent as an ASCII representation with a 2-byte length header. You cannot use OUTPUT with USING to ASCII files; see the following section, "OUTPUT with USING" for details.

Data sent to a BDAT or HP-UX file is sent in internal format if FORMAT OFF is currently assigned to the I/O path (this is the default FORMAT attribute for these file types), and is sent as ASCII characters if FORMAT ON has been explicitly assigned. (See "Devices as Destination" for a description of these formats.)

OUTPUT to HFS Files

You must have W (write) permission on an HFS file, as well as X (search) permission on all superior directories, to output data to the file. If you do not have these permissions, error 183 is reported.

HFS files are extensible. If the data output to the file with this statement would overflow the file's space allocation, the file is extended. HP Instrument BASIC allocates the additional space needed to store the data being output, provided the disc contains enough unused storage space.

Devices as Destination

An I/O path or a device selector may be used to direct OUTPUT to a device. If a device selector is used, the default system attributes are used (see ASSIGN). If an I/O path is used, the ASSIGN statement used to associate the I/O path with the device also determines the attributes used.

If FORMAT ON is the current attribute, the items are sent in ASCII. Items followed by a semicolon are sent with nothing following them. Numeric items followed by a comma are sent with a comma following them. String items followed by a comma are sent with a CR/LF following them. If the last item in the OUTPUT statement has no punctuation following it, the current end-of-line (EOL) sequence is sent after it. Trailing punctuation eliminates the automatic EOL.

If FORMAT OFF is the current attribute, items are sent to the device in internal format. Punctuation following items has no effect on the OUTPUT. Two bytes are sent for each

INTEGER, eight bytes for each REAL. Each string output consists of a four byte header containing the length of the string, followed by the actual string characters. If the number of characters is odd, an additional byte containing a blank is sent after the last character.

CRT as Destination

If the device selector is 1, the OUTPUT is directed to the CRT. OUTPUT 1 and PRINT differ in their treatment of separators and print fields. The OUTPUT format is described under "Devices as Destination." See the PRINT keyword for a discussion of that format. OUTPUT 1 USING and PRINT USING to the CRT produce similar actions.

Strings as Destination

If a string is used for the destination, the string is treated similarly to a file. However, there is no file pointer; each OUTPUT begins at the beginning of the string, and writes serially within the string.

Using END with Devices

The secondary keyword END may be specified following the last item in an OUTPUT statement. The result, when USING is not specified, is to suppress the EOL (End-of-Line) sequence that would otherwise be output after the last byte of the last item. If a comma is used to separate the last item from the END keyword, the corresponding item terminator is output (CR/LF for string items or comma for numeric items).

With HP-IB interfaces, END specifies an EOI signal to be sent with the last data byte of the last item.. However, if no data is sent from the last output item, EOI is not sent. With Data Communications interfaces, END specifies an end-of-data indication to be sent with the last byte of the last output item.

OUTPUT With USING

When the computer executes an OUTPUT USING statement, it reads the image specifier, acting on each field specifier (field specifiers are separated from each other by commas) as it is encountered. If nothing is required from the output items, the field specifier is acted upon without accessing the output list. When the field specifier requires characters, it accesses the next item in the output list, using the entire item. Each element in an array is considered a separate item.

The processing of image specifiers stops when there is no matching display item (and the specifier requires a display item). If the image specifiers are exhausted before the display items, they are reused, starting at the beginning.

If a numeric item requires more decimal places to the left of the decimal point than are provided by the field specifier, an error is generated. A minus sign takes a digit place if M or S is not used, and can generate unexpected overflows of the image field. If the number contains more digits to the right of the decimal point than specified, it is rounded to fit the specifier.

If a string is longer than the field specifier, it is truncated, and the right-most characters are lost. If it is shorter than the specifier, trailing blanks are used to fill out the field.

OUTPUT with USING cannot be used with output to ASCII files. Instead, direct the OUTPUT with USING to a string variable, and then OUTPUT this variable to the file. For instance,

```
OUTPUT String$ USING "5A,X,6D.D";Chars$,Number
OUTPUT @File;String$
```

Effects of the image specifiers on the OUTPUT statement are shown in the following table:

Image Specifier	Meaning
K	Compact field. Outputs a number or string in standard form with no leading or trailing blanks.
−K	Same as K.
H	Similar to K, except the number is output using the European number format (comma radix).
−H	Same as H.
S	Outputs the number's sign (+ or −).
M	Outputs the number's sign if negative, a blank if positive.
D	Outputs one digit character. A leading zero is replaced by a blank. If the number is negative and no sign image is specified, the minus sign will occupy a leading digit position. If a sign is output, it will "float" to the left of the left-most digit.
Z	Same as D, except that leading zeros are output.
*	Like D, except that asterisks are output instead of leading zeros.
.	Outputs a decimal-point radix indicator.
R	Outputs a comma radix indicator (European radix). (Requires IO)
E	Outputs an E, a sign, and a two-digit exponent.
ESZ	Outputs an E, a sign, and a one-digit exponent.
ESZZ	Same as E.
ESZZZ	Outputs an E, a sign, and a three-digit exponent.
A	Outputs a string character. Trailing blanks are output if the number of characters specified is greater than the number available in the corresponding string. If the image specifier is exhausted before the corresponding string, the remaining characters are ignored.
X	Outputs a blank.
literal	Outputs the characters contained in the literal.
B	Outputs the character represented by one byte of data. This is similar to the CHR$ function. The number is rounded to an INTEGER and the least-significant byte is sent. If the number is greater than 32 767, then 255 is used; if the number is less than −32 768, then 0 is used.

Image Specifier	Meaning
W	Outputs a 16-bit word as a two's-complement integer. The corresponding numeric item is rounded to an INTEGER. If it is greater than 32 767, then 32 767 is sent; if it is less than −32 768, then −32 768 is sent. If a device selector is used to access an 8-bit interface, two bytes will be output; the most-significant byte is sent first.
Y	Like W, except that no pad bytes are output to achieve word alignment.
#	Suppresses the automatic output of the EOL (End-Of-Line) sequence following the last output item.
%	Ignored in OUTPUT images.
+	Changes the automatic EOL sequence that normally follows the last output item to a single carriage-return.
–	Changes the automatic EOL sequence that normally follows the last output item to a single line-feed.
/	Outputs a carriage-return and a line-feed.
L	Outputs the current end-of-line (EOL) sequence. The default EOL characters are CR and LF; see ASSIGN for information on re-defining the EOL sequence. If the destination is an I/O path name with the WORD attribute, a pad byte may be sent after the EOL characters to achieve word alignment.
@	Outputs a form-feed.

END with OUTPUT ... USING

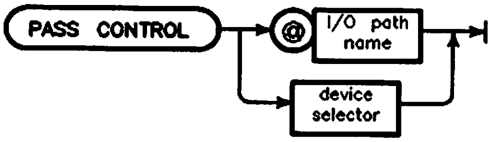
Using the optional secondary keyword END in an OUTPUT ... USING statement produces results which differ from those in an OUTPUT statement without USING. Instead of always suppressing the EOL sequence, the END keyword only suppresses the EOL sequence when no data is output from the last output item. Thus, the # image specifier generally controls the suppression of the otherwise automatic EOL sequence.

With HP-IB interfaces, END specifies an EOI signal to be sent with the last byte output. However, no EOI is sent if no data is sent from the last output item or the EOL sequence is suppressed. With Data Communications interfaces, END specifies an end-of-data indication to be sent at the same times an EOI would be sent on HP-IB interfaces.

PASS CONTROL

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement is used to pass the capability of Active Controller to a specified HP-IB device.



Item	Description	Range
I/O path name	name assigned to an HP-IB device	any valid name
device selector	numeric expression, rounded to an integer	must contain primary address (see Glossary)

Example Statements

```
PASS CONTROL 719
PASS CONTROL @Controller_19
```

Semantics

Executing this statement first addresses the specified device to talk and then sends the Take Control message (TCT), after which Attention is placed in the False state. The computer then assumes the role of a bus device (a non-active controller).

The computer must currently be the active controller to execute this statement, and primary addressing (but not multiple listeners) must be specified.

HP-UX Specifics

You cannot pass control on an interface containing a swap device or mounted file system.

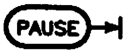
Summary of Bus Actions

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active/Controller	Error	ATN TAD TCT $\overline{\text{ATN}}$	Error	ATN TAD TCT $\overline{\text{ATN}}$
Not Active Controller	Error	Error	Error	Error

PAUSE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	es

This statement suspends program execution. (Also see TRACE PAUSE.)



Semantics

PAUSE suspends program execution before the next line is executed, until CONT is executed. If the program is modified while paused, RUN must be used to restart program execution.

When program execution resumes, the computer attempts to service any ON INTR events that occurred while the program was paused. ON ERROR, or ON TIMEOUT events generate errors if they occur while the program is paused. ON KEY events are ignored while the program is paused.

PEN

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This command selects the pen used for plotting.

PI

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns 3.141 592 653 589 79, which is an approximate value for π .

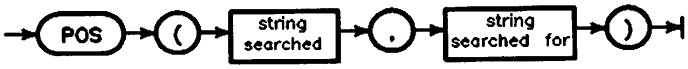
Example Statements

```
Area=PI*Radius^2
PRINT X,X*2*PI
```

POS

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the first position of a substring within a string.



Item	Description	Range
string searched	string expression	—
string searched for	string expression	—

Example Statements

```
Point=POS(Big$,Little$)
IF POS(A$,CHR$(10)) THEN Line_end
```

Semantics

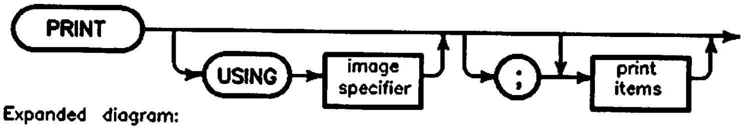
If the value returned is greater than 0, it represents the position of the first character of the string being searched for in the string being searched. If the value returned is 0, the string being searched for does not exist in the string being searched (or the string searched for is the null string).

Note that the position returned is the relative position within the string expression used as the first argument. Thus, when a substring is searched, the position value refers to that substring, not to the parent string from which the substring was taken.

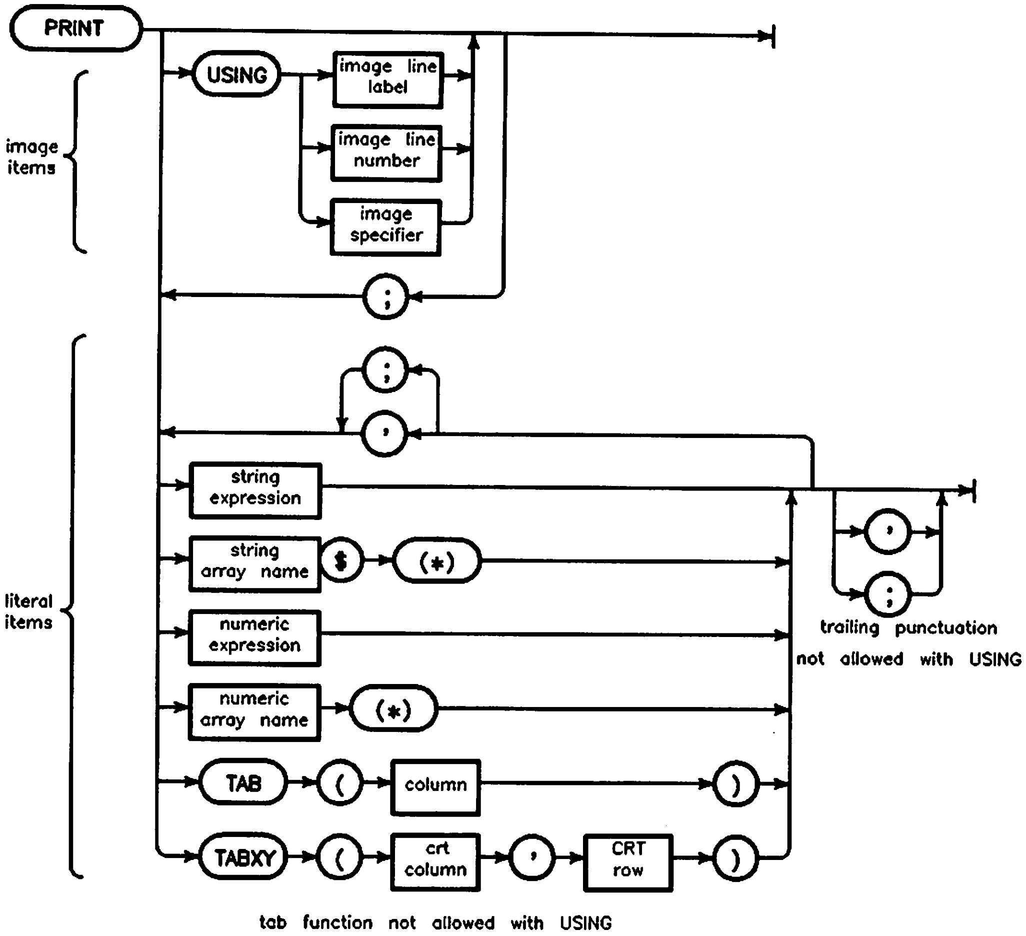
PRINT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

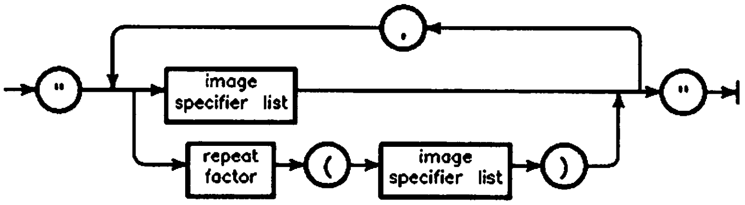
This statement sends items to the PRINTER IS device.



Expanded diagram:



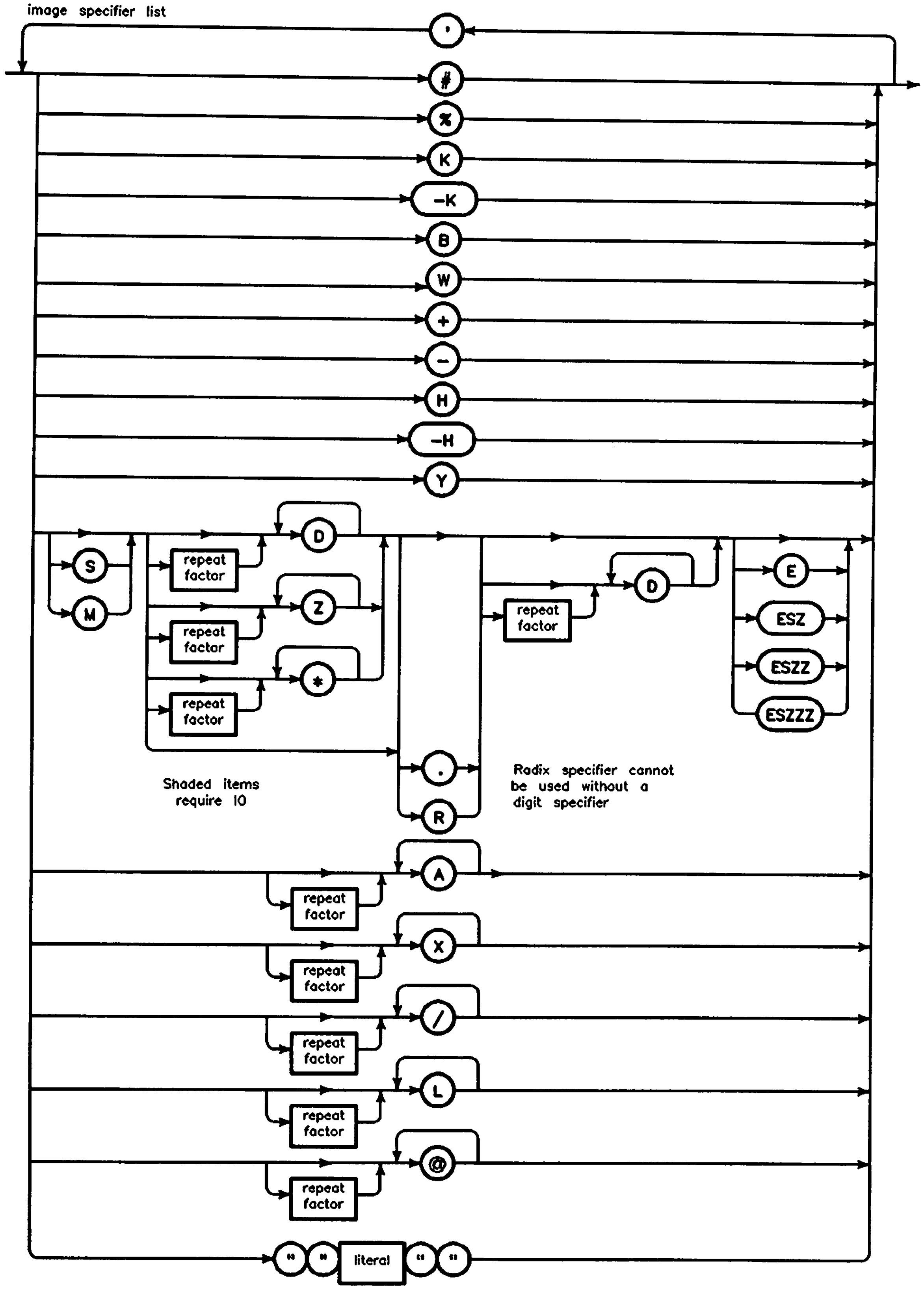
literal form of image specifier



Item	Description	Range
image line number	integer constant identifying an IMAGE statement	1 thru 32 766
image line label	name identifying an IMAGE statement	any valid name
image specifier	string expression	(see drawing)
string array name	name of a string array	any valid name
numeric array name	name of a numeric array	any valid name
column	numeric expression, rounded to an integer	device dependent
CRT column	numeric expression, rounded to an integer	1 thru screen width
CRT row	numeric expression, rounded to an integer	1 thru alpha height
image specifier list	literal	(see next drawing)
repeat factor	integer constant	1 thru 32 767
literal	string constant composed of characters from the keyboard	quote mark not allowed

Example Statements

```
PRINT "LINE";Number
PRINT Array(*);
PRINT TABXY(1,1),Header$,TABXY(Col,3),Message$
PRINT USING "5Z.DD";Money
PRINT USING Fmt3;Id,Item$,Kilograms/2.2
```



Shaded items require IO

Radix specifier cannot be used without a digit specifier

Semantics

Standard Numeric Format

The standard numeric format depends on the value of the number being displayed. If the absolute value of the number is greater than or equal to 1E-4 and less than 1E+6, it is rounded to 12 digits and displayed in floating point notation. If it is not within these limits, it is displayed in scientific notation. The standard numeric format is used unless USING is selected, and may be specified by using K in an image specifier.

Automatic End-Of-Line Sequence

After the print list is exhausted, an End-Of-Line (EOL) sequence is sent to the PRINTER IS device, unless it is suppressed by trailing punctuation or a pound-sign (#) image specifier. The printer width for EOL sequences generation is set to the screen width for CRTs and to 80 for external devices unless the WIDTH attribute of the PRINTER IS statement was specified. WIDTH is off for files. This "printer width exceeded" EOL is not suppressed by trailing punctuation, but can be suppressed by the use of an image specifier.

Control Codes

Some ASCII control codes have a special effect in PRINT statements if the PRINTER IS device is the CRT (device selector=1):

Character	Keystroke	Name	Action
CHR$(7)	CTRL-G	bell	Sounds the beeper
CHR$(8)	CTRL-H	backspace	Moves the print position back one character.
CHR$(10)	CTRL-J	line-feed	Moves the print position down one line.
CHR$(12)	CTRL-L	form-feed	Prints two line-feeds, then advances the CRT buffer enough lines to place the next item at the top of the CRT.
CHR$(13)	CTRL-M	carriage-return	Moves the print position to column 1.

The effect of ASCII control codes on a printer is device dependent. See your printer manual to find which control codes are recognized by your printer and their effects.

Arrays

Entire arrays may be printed using the asterisk specifier. Each element in an array is treated as a separate item by the PRINT statement, as if the items were listed separately, separated by the punctuation following the array specifier. If no punctation follows the array specifier, a comma is assumed. The array is output in row major order (rightmost subscript varies fastest).

PRINT Fields

If PRINT is used without USING, the punctuation following an item determines the width of the item's print field; a semicolon selects the compact field, and a comma selects the default print field. Any trailing punctation will suppress the automatic EOL sequence, in addition to selecting the print field to be used for the print item preceding it.

The compact field is slightly different for numeric and string items. Numeric items are printed with one trailing blank. String items are printed with no leading or trailing blanks.

The default print field prints items with trailing blanks to fill to the beginning of the next 10-character field.

Numeric data is printed with one leading blank if the number is positive, or with a minus sign if the number is negative, whether in compact or default field.

TAB

The TAB function is used to position the next character to be printed on a line. In the TAB function, a column parameter less than one is treated as one. A column parameter greater than zero is subjected to the following formula: TAB position = ((column − 1) MOD width) + 1; where "width" is 80 for all devices. If the TAB position evaluates to a column number less than or equal to the number of characters printed since the last EOL sequence, then an EOL sequence is printed, followed by (TAB position − 1) blanks. If the TAB position evaluates to a column number greater than the number of characters printed since the last EOL, sufficient blanks are printed to move to the TAB position.

TABXY

The TABXY function provides X-Y character positioning on the CRT. It is ignored if a device other than the CRT is the PRINTER IS device. TABXY(1,1) specifies the upper left-hand corner of the CRT. If a negative value is provided for CRT row or CRT column, it is an error. Any number greater than the screen width for CRT column is treated as the last column on the screen. Any number greater than the height of the output area for CRT row is treated as the last line of the output area. If 0 is provided for either parameter, the current value of that parameter remains unchanged.

PRINT With Using

When the computer executes a PRINT USING statement, it reads the image specifier, acting on each field specifier (field specifiers are separated from each other by commas) as it is encountered. If nothing is required from the print items, the field specifier is acted upon without accessing the print list. When the field specifier requires characters, it accesses the next item in the print list, using the entire item. Each element in an array is considered a separate item.

The processing of image specifiers stops when there is no matching display item (and the specifier requires a display item). If the image specifiers are exhausted before the display items, they are reused, starting at the beginning.

If a numeric item requires more decimal places to the left of the decimal point than are provided by the field specifier, an error is generated. A minus sign takes a digit place if M or S is not used, and can generate unexpected overflows of the image field. If the number contains more digits to the right of the decimal point than are specified, it is rounded to fit the specifier.

If a string is longer than the field specifier, it is truncated, and the right-most characters are lost. If it is shorter than the specifier, trailing blanks are used to fill out the field.

Effects of the image specifiers on the PRINT statement are shown in the following table:

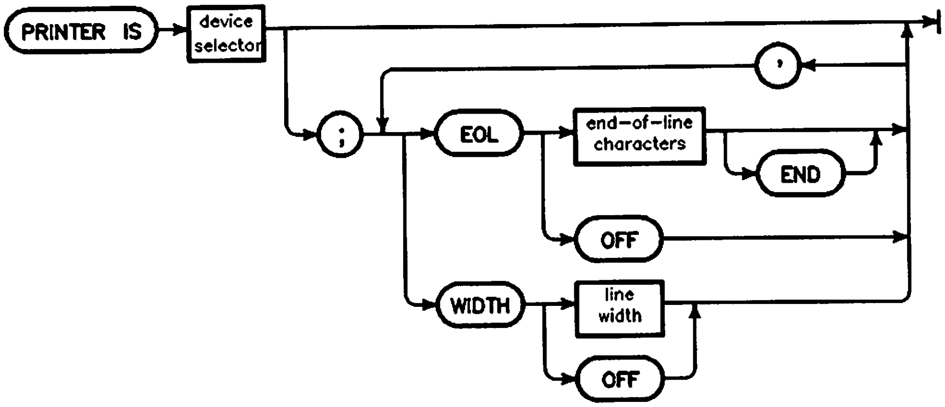
Image Specifier	Meaning
K	Compact field. Prints a number or string in standard form with no leading or trailing blanks.
−K	Same as K.
H	Similar to K, except the number is printed using the European number format (comma radix).
−H	Same as H.
S	Prints the number's sign (+ or −).
M	Prints the number's sign if negative, a blank if positive.
D	Prints one digit character. A leading zero is replaced by a blank. If the number is negative and no sign image is specified, the minus sign will occupy a leading digit position. If a sign is printed, it will "float" to the left of the left-most digit.
Z	Same as D, except that leading zeros are printed.
*	Like Z, except that asterisks are printed instead of leading zeros.
.	Prints a decimal-point radix indicator.
R	Prints a comma radix indicator (European radix).
E	Prints an E, a sign, and a two-digit exponent.
ESZ	Prints an E, a sign, and a one-digit exponent.
ESZZ	Same as E.
ESZZZ	Prints an E, a sign, and a three-digit exponent.
A	Prints a string character. Trailing blanks are output if the number of characters specified is greater than the number available in the corresponding string. If the image specifier is exhausted before the corresponding string, the remaining characters are ignored.

Image Specifier	Meaning
X	Prints a blank.
literal	Prints the characters contained in the literal.
B	Prints the character represented by one byte of data. This is similar to the CHR$ function. The number is rounded to an INTEGER and the least-significant byte is sent. If the number is greater than 32 767, then 255 is used; if the number is less than −32 768, then 0 is used.
W	Prints two characters represented by the two bytes in a 16-bit, two's-complement integer word. The corresponding numeric item is rounded to an INTEGER. If it is greater than 32 767, then 32 767 is used; if it is less than −32 768, then −32 768 is used. On an 8-bit interface, the most-significant byte is sent first. On a 16-bit interface, the two bytes are sent as one word in a single operation.
Y	Same as W.
#	Suppresses the automatic output of the EOL (End-Of-Line) sequence following the last print item.
%	Ignored in PRINT images.
+	Changes the automatic EOL sequence that normally follows the last print item to a single carriage-return.
−	Changes the automatic EOL sequence that normally follows the last print item to a single line-feed.
/	Sends a carriage-return and a line-feed to the PRINTER IS device.
L	Sends the current EOL sequence to the PRINTER IS device. The default EOL characters are CR and LF; see PRINTER IS for information on re-defining the EOL sequence. If the destination is an I/O path name with the WORD attribute, a pad byte may be sent after the EOL characters to achieve word alignment.
@	Sends a form-feed to the PRINTER IS device.

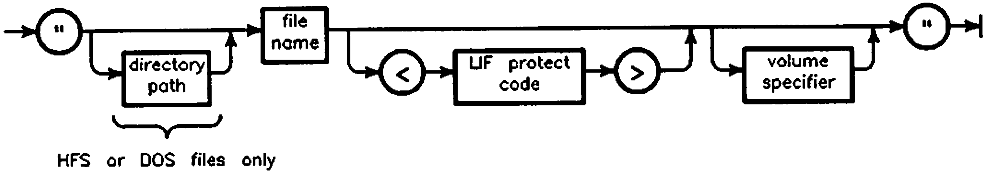
PRINTER IS

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement specifies the system printing device, file, or pipe.



literal form of file specifier:



Item	Description	Range
file specifier	string expression	-
device selector	numeric expression, rounded to an integer	(see Glossary)
end-of-line characters	string expression; Default = CR/LF	0 thru 8 characters
seconds	numeric expression, rounded to the nearest 0.001 seconds; Default=0	0.001 thru 32.767
line width	numeric expression, rounded to an integer; Default = (see text)	1 thru 32 767
directory path	literal	(see MASS STORAGE IS)
file name	literal	depends on volume's format (see Glossary)
LIF protect code	literal; first two non-blank characters are significant	> not allowed
volume specifier	literal	(see MASS STORAGE IS)

Example Statements

```
PRINTER IS 701
PRINTER IS 614
PRINTER IS "debug.out"
PRINTER IS 701;EOL CHR$(13) END,WIDTH 65
PRINTER IS "Myfile";WIDTH 80
PRINTER IS "My_dir/Temp_print";WIDTH 80
```

Semantics

The system printing device or file receives all data sent by the PRINT statement and all data sent by CAT, LIST, and XREF statements in which the destination is not explicitly specified.

The default printing device is the CRT (select code 1) at power-on and after executing SCRATCH A.

The EOL Attribute

The EOL attribute re-defines the end-of-line (EOL) sequence, which is sent at the following times: after the number of characters specified by *line width,* after each line of text, and when an "L" specifier is used in a PRINT USING statement. Up to eight characters may be specified as the EOL characters; an error is reported if the string contains more than eight characters. If END is included in the EOL attribute, an interface-dependent END indication is sent with the last character of the EOL sequence. The default EOL sequence consists of a carriage-return and a line-feed character with no END indication and no delay period. END is ignored for files.

The WIDTH Attribute

The WIDTH attribute specifies the maximum number of characters which will be sent to the printing device before an EOL sequence is automatically sent. The EOL characters are not counted as part of the line width. The default for most devices is 80. Specifying WIDTH OFF sets the width to infinity. If the default is desired, it must be restored explicitly. If the USING clause is included the PRINT statement, the WIDTH attribute is ignored. Default WIDTH for files is OFF.

PRINTER IS file

The file must be a BDAT or HP-UX file.

The PRINTER IS file statement positions the file pointer to the beginning of the file.

The file is closed when another PRINTER IS statement is executed and at SCRATCH A.

You can read the file with ENTER if it is ASSIGNed with FORMAT ON.

An end-of-file error occurs when the end of a LIF file is reached.

HFS Files

In order to write to a PRINTER IS file on an HFS volume, you need to have R (read) and W (write) permission on the file, and X (search) permission on all superior directories.

No end-of-file error occurs when writing to a file on an HFS volume because these files are extensible. That is, if the data output to the file with this statement would otherwise overflow the file's space allocation, HP Instrument BASIC automatically allocates the additional space needed (provided the media contains enough unused storage space).

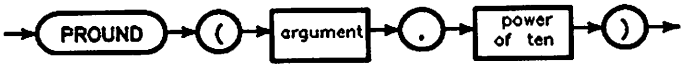
PRIORITY

See the SYSTEM PRIORITY statement.

PROUND

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the value of the argument rounded to the specified power-of-ten.



Item	Description	Range
argument	numeric expression	—
power of ten	numeric expression, rounded to an integer	—

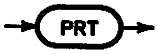
Example Statements

```
Money=PROUND(Result,-2)
PRINT PROUND(Quantity,Decimal_place)
```

PRT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This INTEGER function returns 701, the default (factory set) device selector for an external printer.



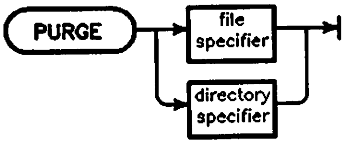
Example Statements

```
PRINTER IS PRT
OUTPUT PRT;A$
```

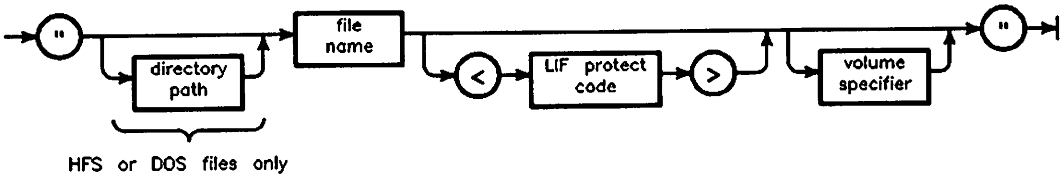
PURGE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

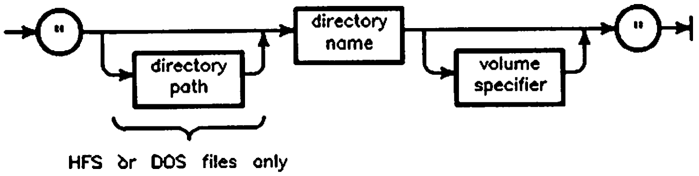
This statement deletes a file from a directory. On hierarchical-directory volumes (such as HFS), PURGE deletes an empty directory from its superior directory.



literal form of file specifier:



literal form of directory specifier:



Item	Description	Range
file specifier	string expression	(see drawing)
directory specifier	string expression	(see drawing)
directory path	literal	(see MASS STORAGE IS)
file name	literal	depends on volume's format (see Glossary)
LIF protect code	literal; first two non-blank characters are significant	> not allowed
volume specifier	literal	(see MASS STORAGE IS)
directory name	literal	depends on volume's format (see Glossary)

Example Statements

```
PURGE File_name$
PURGE "File"
PURGE "George<PC>"
PURGE "Dir1/Dir2/Dir3"
PURGE "Dir1\Dir2\MyFile:DOS,A"
```

Semantics

Once a file is purged, you cannot access the information which was in the file. The records of a purged file are returned to "available space."

An open file must be closed before it can be purged. Any file except a PRINTER IS file, a PLOTTER IS file, or the current working directory can be closed by `ASSIGN TO *` (see ASSIGN). A PRINTER IS file can be closed by executing a PRINTER IS to another device or file. SCRATCH A closes all files.

HFS Files and Directories

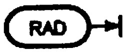
In order to PURGE an HFS directory or file, all of the following conditions must be met:

- It must be closed. The current working directory is closed by an MSI to a different directory. SCRATCH A closes all directories and files.
- It must be empty (directories only). That is, it must not contain any subordinate files or directories.
- You must have the appropriate access capabilities.
  - In order to PURGE a file or directory on an HFS volume, you need to have W (write) and X (search) permission of the immediately superior directory, as well as X (search) permission on all other superior directories. Note that the ability to purge an HFS file is not determined by the file's permissions but rather by the permissions set on the parent directory.

RAD

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement selects radians as the unit of measure for expressing angles.



Semantics

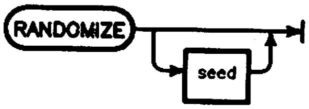
All functions which return an angle will return an angle in radians. All operations with parameters representing angles will interpret the angle in radians. If no angle mode is specified in a program, the default is radians (also see DEG).

A subprogram "inherits" the angle mode of the calling context. If the angle mode is changed in a subprogram, the mode of the calling context is restored when execution returns to the calling context.

RANDOMIZE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement selects a seed for the RND function.



Item	Description	Range
seed	numeric expression, rounded to an integer; Default = pseudo-random	1 thru $2^{31}-2$

Example Statements

```
RANDOMIZE
RANDOMIZE Old_seed*PI
```

Semantics

The seed actually used by the random number generator depends on the absolute value of the seed specified in the RANDOMIZE statement.

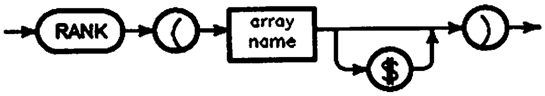
Absolute Value of Seed	Value Used
less than 1	1
1 thru $2^{31}-2$	INT(ABS(seed))
greater than $2^{31}-2$	$2^{31}-2$

The seed is reset to 37 480 660 by power-up, SCRATCH A, SCRATCH, and program prerun.

RANK

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the number of dimensions in an array. The value returned is an INTEGER.



Item	Description	Range
array name	name of an array	any valid name

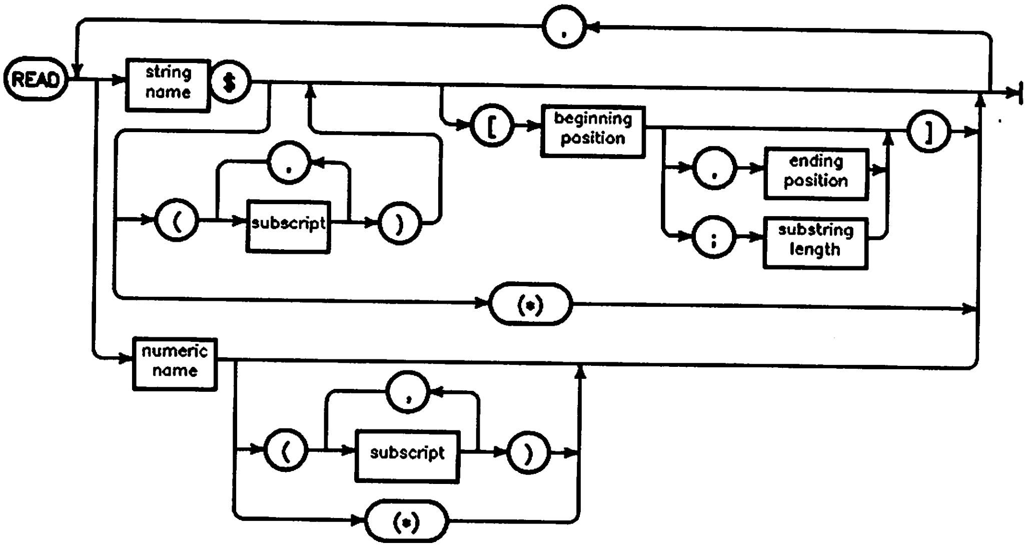
Example Statements

```
IF RANK(A)=2 THEN PRINT "A is a matrix"
R=RANK(Array)
```

READ

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement reads values from DATA statements and assigns them to variables.



Item	Description	Range
numeric name	name of a numeric variable	any valid name
string name	name of a string variable	any valid name
subscript	numeric expression, rounded to an integer	−32 767 thru +32 767 (see "array" in Glossary)
beginning position	numeric expression, rounded to an integer	1 thru 32 767 (see "substring" in Glossary)
ending position	numeric expression, rounded to an integer	0 thru 32 767 (see "substring" in Glossary)
substring length	numeric expression, rounded to an integer	0 thru 32 767 (see "substring" in Glossary)

Example Statements

```
READ Number,String$
READ Array(*)
READ Item(1,1),Item(2,1),Item(3,1)
```

Semantics

The numeric items stored in DATA statements are considered strings by the computer, and are processed with a VAL function to read into numeric variables in a READ statement. If they are not of the correct form, error 32 may result. Real DATA items will be rounded into an INTEGER variable if they are within the INTEGER range (−32 768 through 32 767). A string variable may read numeric items, as long as it is dimensioned large enough to contain the characters.

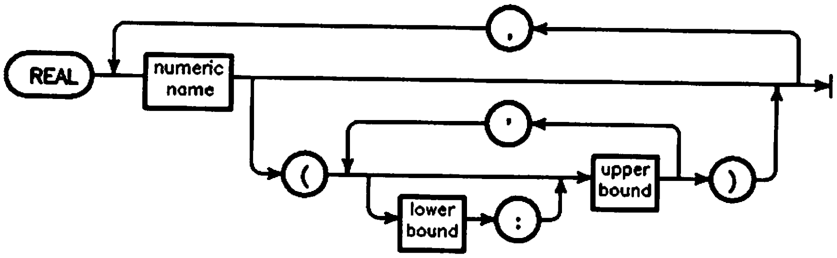
The first READ statement in a context accesses the first item in the first DATA statement in the context unless RESTORE has been used to specify a different DATA statement as the starting point. Successive READ operations access following items, progressing through DATA statements as necessary. Trying to READ past the end of the last DATA statement results in error 36. The order of accessing DATA statements may be altered by using the RESTORE statement.

An entire array can be specified by replacing the subscript list with an asterisk. The array entries are made in row major order (right most subscript varies most rapidly).

REAL

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This statement reserves storage for floating-point variables and arrays. (For information about the REAL function, see the next entry in the keyword dictionary; for information about using REAL as a secondary keyword, see the COM, DEF FN, or SUB statements.)



Item	Description	Range
numeric name	name of a numeric variable	any valid name
lower bound	integer constant; Default = 0	−32 767 thru +32 767 (see "array" in Glossary)
upper bound	integer constant	−32 767 thru +32 767 (see "array" in Glossary)

Example Statements

```
REAL X,Y,Z
REAL Array(-128:127,15)
```

Semantics

Each REAL variable or array element requires eight bytes of number storage. The maximum number of subscripts in an array is six, and no dimension may have more than 32 767 elements.

The total number of REAL variables is limited by the fact that the maximum memory usage for *all* variables—INTEGER, REAL, and string—within any context is $2^{24}-1$, or 16 777 215, bytes (or limited by the amount of available memory, whichever is less).

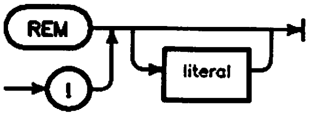
RECOVER

See the ON ... statements.

REM

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This statement allows comments in a program.



Item	Description	Range
literal	string constant composed of characters from the keyboard	—

Example Program Lines

```
100  REM   Program Title
190  !
200  IF BIT(Info,2) THEN Branch  ! Test overrange bit
```

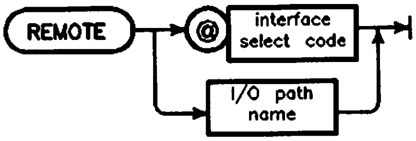
Semantics

REM must be the first keyword on a program line. If you want to add comments to a statement, an exclamation point must be used to mark the beginning of the comment. If the first character in a program line is an exclamation point, the line is treated like a REM statement and is not checked for syntax.

REMOTE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement places HP-IB devices having remote/local capabilities into the remote state.



Item	Description	Range
I/O path name	name assigned to a device or devices	any valid name (see ASSIGN)
device selector	numeric expression, rounded to an integer	(see Glossary)

Example Statements

```
REMOTE 712
REMOTE @Hpib
```

Semantics

If individual devices are not specified, the remote state for all devices on the bus having remote/local capabilities is enabled. The bus configuration is unchanged, and the devices switch to remote if and when they are addressed to listen. If primary addressing is used, only the specified devices are put into the remote state.

When the computer is the system controller and is switched on, reset, or ABORT is executed, bus devices are automatically enabled for the remote state and switch to remote when they are addressed to listen.

The computer must be the system controller to execute this statement, and it must be the active controller to place individual devices in the remote state.

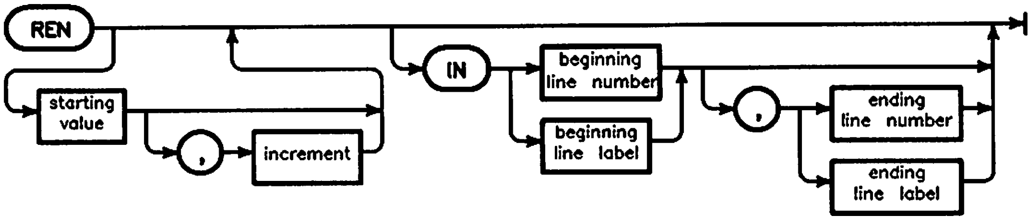
Summary of Bus Actions

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active/Controller	REN $\overline{\mathrm{ATN}}$	REN ATN MTA UNL LAG	Error	Error
Not Active Controller	REN	Error	Error	Error

REN

Keyboard Executable	Yes
Programmable	No
In an IF ... THEN ...	No

This command allows you to renumber all or a portion of the program currently in memory.



Item	Description	Range
starting value	integer constant identifying a program line; Default = 10	1 thru 32 766
increment	integer constant; Default = 10	1 thru 32 767
beginning line number	integer constant identifying program line	1 thru 32 766
beginning line label	name of a program line	any valid name
ending line number	integer constant identifying program line; Default = last program line	1 thru 32 766
ending line label	name of a program line	any valid name

Example Statements

```
REN
REN 1000,5
REN 270,1 IN 260,Label1
```

Semantics

The program segment to be renumbered is delimited by the beginning line number or label (or the first line in the program) and the ending line number or label (or the last line in the program). The first line in the renumbered segment is given the specified starting value, and subsequent line numbers are separated by the increment. If a renumbered line is referenced by a statement (such as GOTO or GOSUB), those references will be updated to reflect the new line numbers. Renumbering a paused program causes it to move to the stopped state.

REN cannot be used to move lines. If renumbering would cause lines to overlap preceding or following lines, an error occurs and no renumbering takes place.

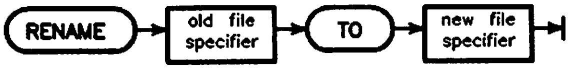
If the highest line number resulting from the REN command exceeds 32 766, an error message is displayed and no renumbering takes place. An error occurs if the beginning line is after the ending line, or if one of line labels specified doesn't exist.

Note This command is only available for host-instruments that implement a command line editor. See your instrument-specific HP Instrument BASIC manual for further information.

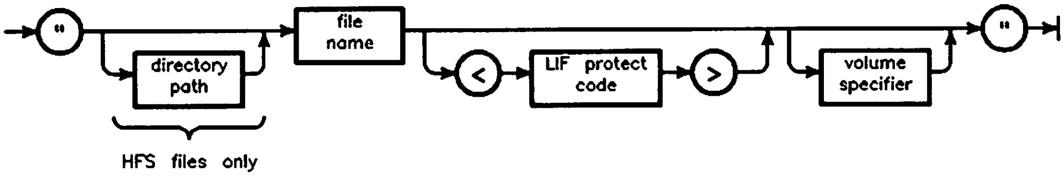
RENAME

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

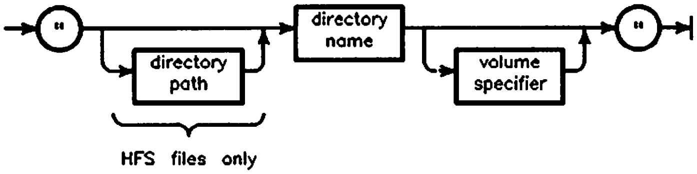
This statement changes a file's or directory's name.



literal form of file specifier:



literal form of directory specifier:



Item	Description	Range
old file specifier	string expression	(see "file specifier" drawing)
new file specifier	string expression	(see "file specifier" drawing)
old directory specifier	string expression	(see "directory specifier" drawing)
new directory specifier	string expression	(see "directory specifier" drawing)
directory path	literal	(see MASS STORAGE IS)
file name	literal	depends on volume's format: 10 characters for LIF; 14 characters for HFS (short file name); 255 characters for HFS (long file name); (see Glossary)
LIF protect code	literal; first two non-blank characters are significant	> not allowed
volume specifier	literal	(see MASS STORAGE IS)
directory name	literal	depends on volume's format: 10 characters for LIF; 14 characters for HFS (short file name); 255 characters for HFS (long file name); (see Glossary)

Example Statements

```
RENAME "Old_name" TO "New_name"
RENAME File_name$&Vol$ TO Temp$
RENAME "TEMP<pc>" TO "FINAL"

RENAME Dir$&File$&Volume$
RENAME "/WORKSTATIONS/AUTOST" TO "old_autost"
```

Semantics

The new file or directory name must not duplicate the name of any other file in the directory.

- Files are closed by ASSIGN ... TO * (explicitly closes an I/O path). A PRINTER IS file can be closed by executing a PRINTER IS to another device or file.
- The current working directory is closed by an MSI to a different directory.

SCRATCH A also closes all files and directories.

If you try to rename an open HFS or LIF file or directory, you will not receive an error.

Because you cannot move a file from one mass storage volume to another with RENAME, an error will be given if a volume specifier is included which is not the current location of the file. (However, RENAME can perform limited file-move operations with HFS files. See details below.)

LIF Protect Codes

A protected file retains its old protect code, which must be included in the old file specifier.

HFS Permissions

In order to RENAME a file or directory on an HFS volume, you need to have W (write) and X (search) permission of the immediately superior directory, as well as X (search) permission on all other superior directories.

Limited File Moves with HFS

With HFS, RENAME can be used to move files within the directory structure. Directories cannot be moved with RENAME. Moving of files must occur within a single volume. If you move a file with RENAME, the original file ("old file specifier") is purged.

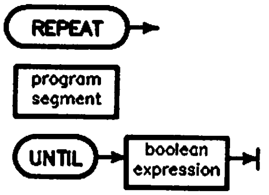
HP-UX Specifics

RENAMEing across HFS volumes is allowed.

REPEAT ... UNTIL

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This construct defines a loop which is repeated until the boolean expression in the UNTIL statement evaluates to be logically true (evaluates to non-zero).



Item	Description	Range
boolean expression	numeric expression; evaluated as true if non-zero and false if zero	—
program segment	any number of contiguous program lines not containing the beginning or end of a main program or subprogram, but which may contain properly nested constructs(s).	—

Example Program Segments

```
530  REPEAT
540    PRINT Factor
550    Factor=Factor*1.1
560  UNTIL Factor>10

680  REPEAT
690    INPUT "Enter a positive number",Number
700  UNTIL Number>=0
```

Semantics

The REPEAT ... UNTIL construct allows program execution dependent on the outcome of a relational test performed at the *end* of the loop. Execution starts with the first program line following the REPEAT statement, and continues to the UNTIL statement where a relational test is performed. If the test is false a branch is made to the first program line following the REPEAT statement.

When the relational test is true, program execution continues with the first program line following the UNTIL statement.

Branching into a REPEAT ... UNTIL construct (via a GOTO) results in normal execution up to the UNTIL statement, where the test is made. Execution will continue as if the construct had been entered normally.

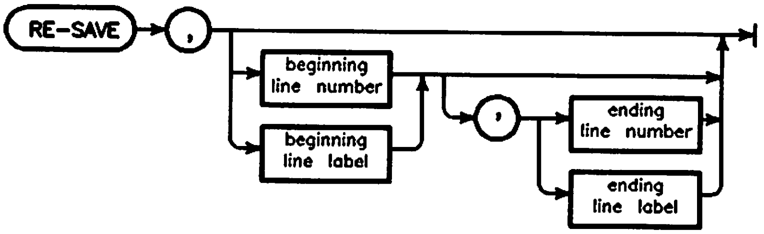
Nesting Constructs Property

REPEAT ... UNTIL constructs may be nested within other constructs provided the inner construct begins and ends before the outer construct can end.

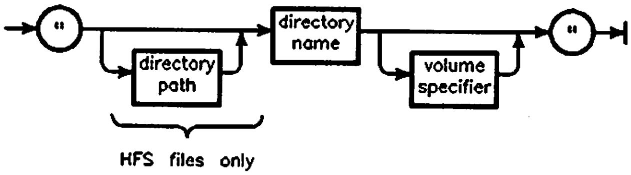
RE-SAVE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement creates a specified ASCII file (or HP-UX file) if it does not exist; otherwise, it re-writes a specified ASCII or HP-UX file by copying program lines as strings into that file.



literal form of directory specifier:



Item	Description	Range
file specifier	string expression	(see drawing)
beginning line number	integer constant identifying program line; Default = first program line	1 thru 32 766
beginning line label	name of a program line	any valid name
ending line number	integer constant identifying a program line; Default = last program line	1 thru 32 766
ending line label	name of a program line	any valid name
directory path	literal	(see MASS STORAGE IS)
file name	literal	depends on volume's format (see Glossary)
volume specifier	literal	(see MASS STORAGE IS)

Example Statements

```
RE-SAVE "NailFile"
RE-SAVE Name$,1,Sort
```

Semantics

An entire program can be saved, or the portion delimited by beginning and (if needed) ending line labels or line numbers. If the file name already exists, the old file entry is removed from the directory before the new file is successfully saved on the mass storage media. Attempting to RE-SAVE any existing file that is not an ASCII or HP-UX text file results in an error. (Note that if you RE-SAVE an existing HP-UX or DOS text file, a new HP-UX or DOS file will be created; the same rule applies to existing ASCII files).

If the file does not already exist, RE-SAVE performs the same action as SAVE.

If a specified line label does not exist, error 3 occurs. If a specified line number does not exist, the program lines with numbers inside the range specified are saved. If the ending line number is less than the beginning line number, error 41 occurs.

HFS Permissions

In order to RE-SAVE a file on an HFS volume, you need to have W (write) permission on the file (if one already exists), W (write) and X (search) permission of the immediately superior directory, as well as X permission on all other superior directories. If a file already exists, its permission bits will be preserved.

HFS File Specifics

If the specified file does not already exist, RE-SAVE will generally create an ASCII type file. However, HP Instrument BASIC will create an HP-UX type file when the program is being RE-SAVEd to an HFS volume.

In order to RE-SAVE a file on an HFS volume, you need to have both R (read) and W (write) permission on the file if one already exists. The rest of the HFS permission requirements are the same as mentioned above.

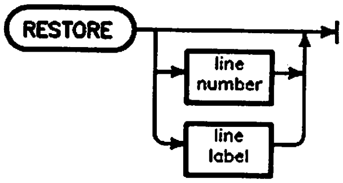
MS-DOS File Specifics

If the specified file does not already exist, RE-SAVE will generally create an ASCII type file. However, HP Instrument BASIC will create a DOS type file when the program is being RE-SAVEd to an DOS volume.

RESTORE

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

RESTORE specifies which DATA statement will be used by the next READ operation.



Item	Description	Range
line label	name of a program line	any valid name
line number	integer constant identifying a program line; Default=first DATA statement in context	1 thru 32 766

Example Statements

```
RESTORE
RESTORE Third_array
```

Semantics

If a line is specified which does not contain a DATA statement, the computer uses the first DATA statement after the specified line. RESTORE can only refer to lines within the current context. An error results if the specified line does not exist.

RETURN

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement returns program execution to the line following the invoking GOSUB. The keyword RETURN is also used in user-defined functions (see DEF FN).

RETURN ...

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement returns a value from a multi-line function.

Item	Description	Range
numeric result	numeric expression	range of REAL
string result	string expression	—

Example Statements

```
IF D THEN RETURN D
RETURN A$&B$
```

Semantics

There may be more than one RETURN statement. The result in the RETURN statement is the value returned to the calling context. The result type, numeric or string, must match the function type (i.e., a numeric function cannot return a string result).

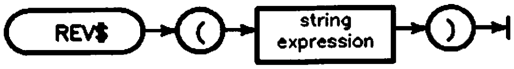
When you exit a multi-line function, the following actions take place:

- local files are closed;
- local variables are deallocated;
- ON ... statements may be affected. See ON ... /OFF ... ;
- some system variables are restored to previous values. See the "Master Reset Table" in the "Useful Tables" section.

REV$

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns a string formed by reversing the sequence of characters in the specified string.



Example Statements

```
Reverse$=REV$("palindrome")
Last_blank=LEN(Sentence$)-POS(REV$(Sentence$)," ")
```

Semantics

The REV$ function is useful when searching for the last occurrence of an item within a string.

RND

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns a pseudo-random number greater than 0 and less than 1.

Example Statements

```
Percent=RND*100
IF RND<.5 THEN Case1
```

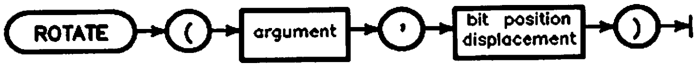
Semantics

The random number returned is based on a seed set to 37 480 660 at power-on, SCRATCH, SCRATCH A, or program prerun. Each succeeding use of RND returns a random number which uses the previous random number as a seed. The seed can be modified with the RANDOMIZE statement.

ROTATE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns an integer which equals the value obtained by shifting the 16-bit binary representation of the argument by the number of bit positions specified. The shift is performed with wrap-around.



Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 thru +32 767
bit position displacement	numeric expression, rounded to an integer	−15 thru +15

Example Statements

```
New_word=ROTATE(Old_word,2)
Q=ROTATE(Q,Places)
```

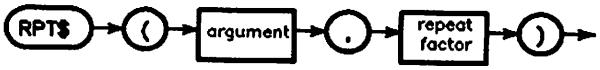
Semantics

The argument is converted into a 16-bit, two's-complement form. If the bit position displacement is positive, the rotation is towards the least-significant bit. If the bit position displacement is negative, the rotation is towards the most-significant bit. The rotation is performed without changing the value of any variable in the argument.

RPT$

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the string repeated a given number of times.



Item	Description	Range
argument	string expression	—
repeat factor	numeric expression, rounded to an integer	0 thru 32 767

Example Statements

```
PRINT RPT$("*",80)
Center$=RPT$(" ",(Right-Left-Length)/2)
```

Semantics

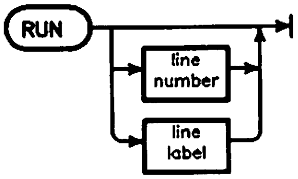
The value of the numeric expression is rounded to an integer. If the numeric expression evaluates to a zero, a null string is returned.

An error will result if the numeric expression evaluates to a negative number or if the string created by RPT$ contains more than 32 767 characters.

RUN

Keyboard Executable	Yes
Programmable	No
In an IF ... THEN ...	No

This command starts program execution at a specified line.



Item	Description	Range
line number	integer constant identifying a program line; Default = first program line	1 thru 32 766
line label	name of a program line	any valid name

Example Statements

```
RUN 10
RUN Part2
```

Semantics

RUN is executed in two phases: prerun initialization and program execution.

The prerun phase consists of:

- Reserving memory space for variables specified in COM statements (both labeled and blank). See COM for a description of when COM areas are initialized.
- Reserving memory space for variables specified by DIM, REAL, INTEGER, or implied in the main program segment. Numeric variables are initialized to 0; string variables are initialized to the null string.
- Checking for syntax errors which require more than one program line to detect. Included in this are errors such as incorrect array references, and mismatched parameter or COM lists.

If an error is detected during prerun phase, prerun halts and an error message is displayed on the CRT.

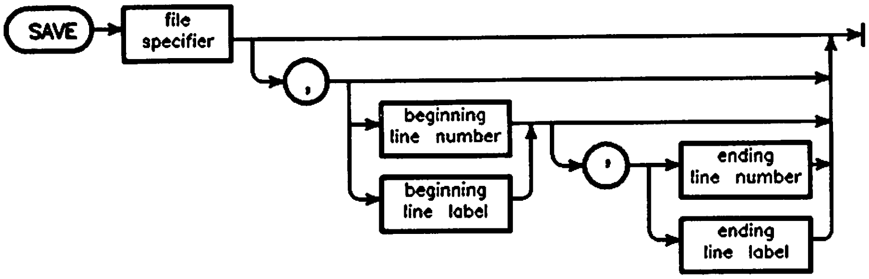
After successful completion of prerun initialization, program execution begins with either the lowest numbered program line or the line specified in the RUN command. If the line number specified does not exist in the main program, execution begins at the next higher-numbered line. An error results if there is no higher-numbered line available within the main program, or if the specified line label cannot be found in the main program.

Note The RUN command may not be available in this form if your host-instrument does not implement the command line editor. See your instrument-specific HP Instrument BASIC manual for details.

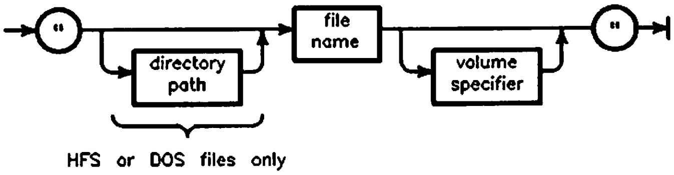
SAVE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement creates an ASCII file (or HP-UX file) and copies program lines as strings into that file.



literal form of directory specifier:



Item	Description	Range
file specifier	string expression	(see drawing)
beginning line number	integer constant identifying a program line; Default = first program line	1 thru 32 766
beginning line label	name of a program line	any valid name
ending line number	integer constant identifying a program line; Default = last program line	1 thru 32 766
ending line label	name of a program line	any valid name
directory path	literal	(see MASS STORAGE IS)
file name	literal	depends on volume's format (see Glossary)
LIF protect code	literal; first two non-blank characters are significant	> not allowed
volume specifier	literal	(see MASS STORAGE IS)

Example Statements

```
SAVE "WHALES"
SAVE "TEMP",1,Sort
SAVE "Dir<SRM_RW_pass>/File"
```

Semantics

An entire program can be saved, or any portion delimited by the beginning and (if needed) ending line numbers or labels. This statement is for creating new files. Attempting to SAVE a file name that already exists causes error 54. If you need to replace an old file, see RE-SAVE.

If a specified line label does not exist, error 3 occurs. If a specified line number does not exist, the program lines with numbers inside the range specified are saved. If the ending line number is less than the beginning line number, error 41 occurs. If no program lines are in the specified range, error 46 occurs.

Lines longer than 256 characters may not be saved correctly. When a GET is performed on a program with such a line, an error will occur.

HFS Permissions and File Headers

In order to SAVE a file on an HFS volume, you need to have W (write) and X (search) permission of the immediately superior directory, as well as X permission on all other superior directories.

When a file is saved on an HFS volume, access permission bits are set to RW-RW-RW-.

All ASCII type files on HFS volumes contain a 512-byte header (at the beginning of the files contents). This header allows the BASIC system to recognize the file as being an ASCII file. (The header is handled automatically by the BASIC system, so you do not have to take any special actions.) HP-UX type files do not have a header.

HP-UX Specifics

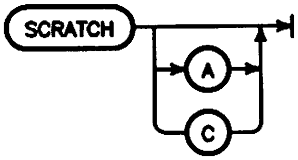
In general, SAVE will create an ASCII type file. However, HP Instrument BASIC will create an HP-UX type file when the program is being SAVEd on an HFS volume. This is done to facilitate the use of other commands and utilities provided by the HP-UX operating system on this file.

SCRATCH

Keyboard Executable	Yes
Programmable	No
In an IF ... THEN ...	No

This command erases all or selected portions of memory.

Note This command may not be available if your host-instrument does not implement the command-line editor. See your instrument-specific HP Instrument BASIC manual for further information.



Item	Description	Range
key number	integer constant	0 thru 23

Example Statements

```
SCRATCH
SCRATCH A
```

Semantics

SCRATCH clears the BASIC program and all variables not in COM. Key definitions are left intact.

SCRATCH C clears all variables, including those in COM. The program and keys are left intact.

SCRATCH A clears the BASIC program memory, all the key definitions, and all variables (including those in COM). Most internal parameters in the computer are reset by this command.

SCRATCH A Effects on HFS Volumes

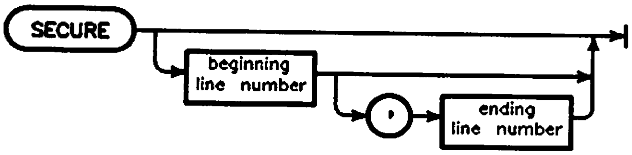
SCRATCH A closes files and directories with HFS volumes.

SECURE

Keyboard Executable	Yes
Programmable	No
In an IF..THEN..	No

This command protects program lines so that they cannot be listed.

Note This command may not be available if your host-instrument does not implement the command-line editor. See your instrument-specific HP Instrument BASIC manual for further information.



Item	Description	Range
beginning line number	integer constant; Default = first line in program	—
ending line number	integer constant; Default = beginning line number if specified, or last line in program	—

Example Statements

```
SECURE
SECURE 45
SECURE 1,100
```

Semantics

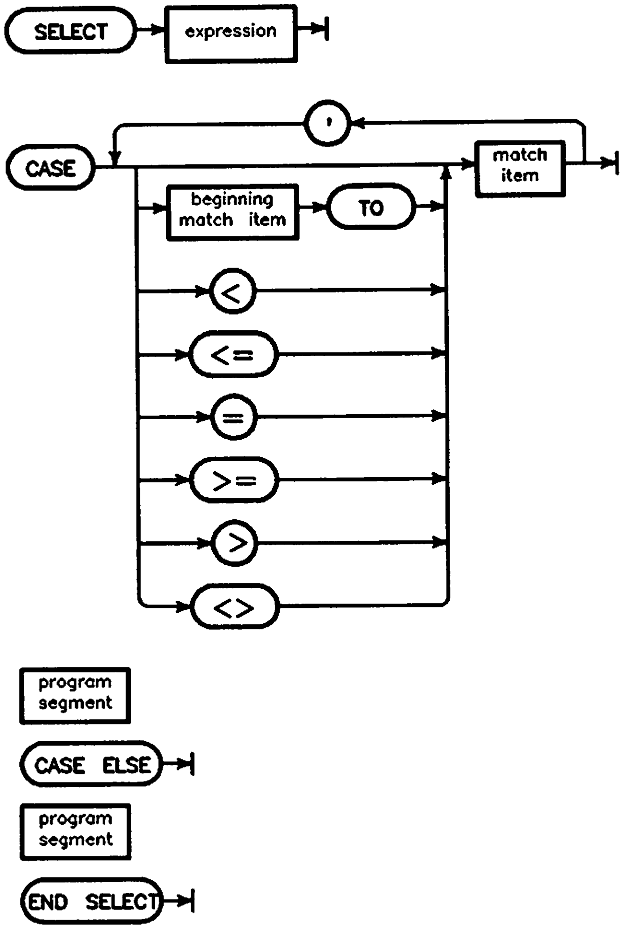
If no lines are specified, the entire program is secured. If one line number is specified, only that line is secured. If two lines are specified, all lines between and including those lines are secured.

Program lines which are secure are listed as an *. Only the line number is listed.

SELECT ... CASE

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This construct provides conditional execution of one of several program segments.



Item	Description	Range
expression	a numeric or string expression	—
match item	a numeric or string expression; must be same type as the SELECT expression	—
program segment	any number of contiguous program lines not containing the beginning or end of a main program or subprogram, but which may contain properly nested construct(s).	—

Example Program Segments

```
650 SELECT Expression
660   CASE <0
670     PRINT "Negative number"
680   CASE ELSE
690     PRINT "Non-negative number"
700 END SELECT

750 SELECT Expression$
760   CASE "A" TO "Z"
770     PRINT "Uppercase alphabetic"
780   CASE ":",";",",","."
790     PRINT "Punctuation"
800 END SELECT
```

Semantics

SELECT ... END SELECT is similar to the IF ... THEN ... ELSE ... END IF construct, but allows *several* conditional program segments to be defined; however, *only one segment will be executed* each time the construct is entered. Each segment starts after a CASE or CASE ELSE statement and ends when the next program line is a CASE, CASE ELSE, or END SELECT statement.

The SELECT statement specifies an expression, whose value is compared to the list of values found in each CASE statement. When a match is found, the corresponding program segment is executed. The remaining segments are skipped and execution continues with the first program line following the END SELECT statement.

All CASE expressions must be of the same type, (either string or numeric) and must agree in type with the corresponding SELECT statement expression.

The optional CASE ELSE statement defines a program segment to be executed when the selected expression's value fails to match any CASE statement's list.

Branching into a SELECT ... END SELECT construct (via GOTO) results in normal execution until a CASE or CASE ELSE statement is encountered. Execution then branches to the first program line following the END SELECT statement.

Errors encountered in evaluating CASE statements will be reported as having occurred in the corresponding SELECT statement.

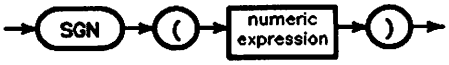
Nesting Constructs Properly

SELECT ... END SELECT constructs may be nested, provided inner construct begins and ends before the outer construct can end.

SGN

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns 1 if the argument is positive, 0 if it equals zero, and −1 if it is negative.



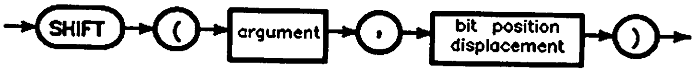
Example Statements

```
Root=SGN(X)*SQR(ABS(X))
Z=2*PI*SGN(Y)
```

SHIFT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns an integer which equals the value obtained by shifting the 16-bit binary representation of the argument by the number of bit positions specified, without wrap-around.



Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 thru +32 767
bit position displacement	numeric expression, rounded to an integer	−15 thru +15

Example Statements

```
New_word=SHIFT(Old_word,-2)
Mask=SHIFT(1,Position)
```

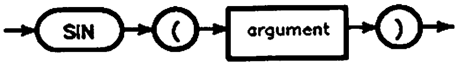
Semantics

If the bit position displacement is positive, the shift is towards the least-significant bit. If the bit position displacement is negative, the shift is towards the most-significant bit. Bits shifted out are lost. Bits shifted in are zeros. The SHIFT operation is performed without changing the value of any variable in the argument.

SIN

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the sine of the angle represented by the argument.



Item	Description	Range Restrictions
argument	numeric expression in current units of angle when arguments are INTEGER or REAL	absolute values less than:1.708 312 781 2 E+10 deg.or 2.981 568 26 E+8 rad.;

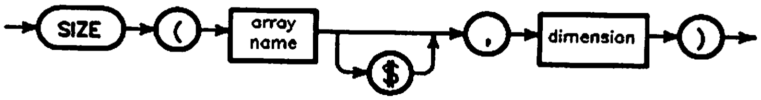
Examples Statements

```
Sine=SIN(Angle)
PRINT "Sine of ";Theta;"=";SIN(Theta)
```

SIZE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the size (number of elements) of a dimension of an array. This INTEGER value represents the difference between the upper bound and the lower bound, plus 1.



Item	Description	Range
array name	name of an array	any valid name
dimension	numeric expression, rounded to an integer	1 thru 6; ≤ the RANK of the array

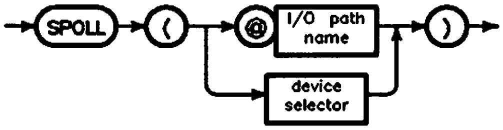
Example Statements

```
Upperbound(2)=BASE(A,2)+SIZE(A,2)-1
Number_words=SIZE(Words$,1)
```

SPOLL

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns an integer containing the serial poll response from the addressed device.



Item	Description	Range
I/O path	name name assigned to a device	any valid name (see ASSIGN)
device selector	numeric expression, rounded to an integer	must include a primary address (see Glossary)

Example Statements

```
Stat=SPOLL(707)
IF SPOLL(@Device) THEN Respond
```

Semantics

The computer must be the active controller to execute this function. Refer to the documentation provided with the device being polled for information concerning the device's status byte.

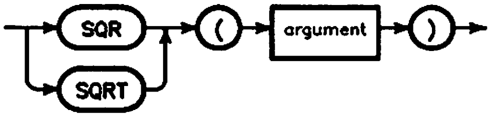
Summary of Bus Actions

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active/Controller	Error	ATN UNL MLA TAD SPE $\overline{\text{ATN}}$	Error	ATN UNL MLA TAD SPE $\overline{\text{ATN}}$ Read Data ATN SPD UNT
Not Active Controller	Error	Error	Error	Error

SQRT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the square root of the argument.



Item	Description/Default	Range Restrictions
argument	numeric expression	any valid INTEGER or REAL value for INTEGER and REAL expressions;

Examples Statements

```
Amps=SQRT(Watts/Ohms)
PRINT "Square root of ";X;"=";SQR(Z)
```

Semantics

If the argument is REAL or INTEGER, the value returned is REAL.

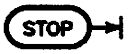
STEP

See the FOR . . . NEXT construct.

STOP

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement terminates execution of the program.



Semantics

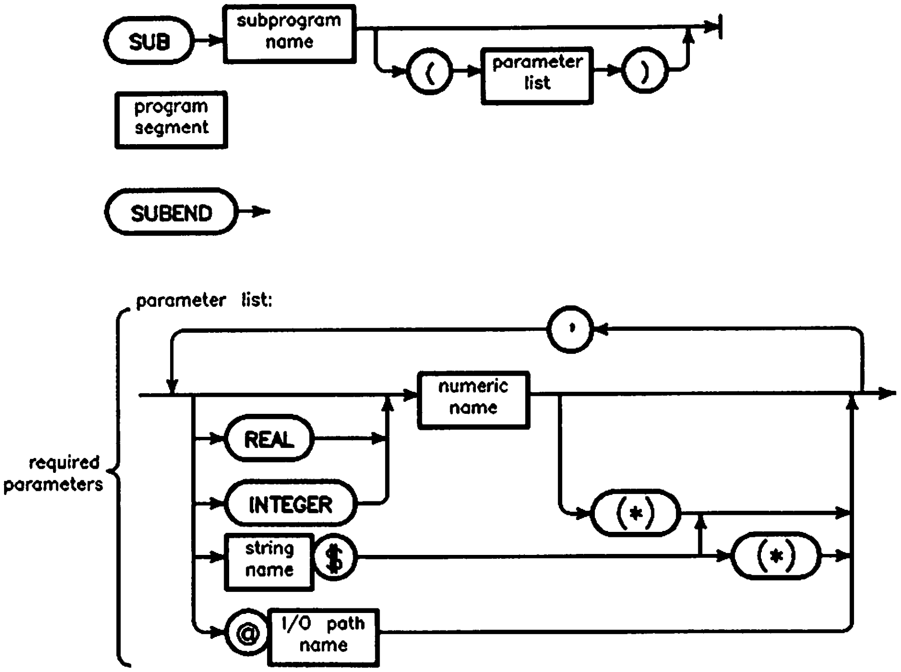
Once a program is stopped, it cannot be continued. The program must be restarted. PAUSE should be used if you intend to continue execution of the program.

A program can have multiple STOP statements. Encountering an END statement key has the same effect as executing STOP. After a STOP, variables that existed in the main context are available for inspection.

SUB

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This is the first statement in a SUB subprogram and can specify the subprogram's formal parameters.



Item	Description	Range
subprogram name	name of the SUB subprogram	any valid name
numeric name	name of a numeric variable	any valid name
string name	name of a string variable	any valid name
I/O path name	name assigned to a device, devices, or mass storage file	any valid name (see ASSIGN)
program segment	any number of contiguous program lines not containing the beginning or end of a main program or subprogram	—

Example Statements

```
SUB Parse(String$)
SUB Transform(@Printer,INTEGER Array(*))
```

Semantics

SUB subprograms must appear after the main program. The first line of the subprogram must be a SUB statement. The last line must be a SUBEND statement. Comments after the SUBEND are considered to be part of the subprogram.

Variables in a subprogram's formal parameter list may not be duplicated in COM or other declaratory statements within the subprogram. A subprogram may not contain any SUB statements, or DEF FN statements. Subprograms can be called recursively and may contain local variables. A unique labeled COM must be used if the local variables are to preserve their values between invocations of the subprogram.

SUBEXIT may be used to leave the subprogram at some point other than the SUBEND. Multiple SUBEXITs are allowed, and SUBEXIT may appear in an IF ... THEN statement. SUBEND is prohibited in IF ... THEN statements, and may only occur once in a subprogram. ERROR SUBEXIT may be used in place of SUBEXIT.

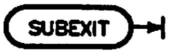
SUBEND

See the SUB statement.

SUBEXIT

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	Yes

This statement may be used to return from a SUB subprogram at some point other than the SUBEND statement. It allows multiple exits from a subprogram.



See also ERROR SUBEXIT

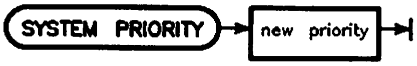
SYSTEM ID

See SYSTEM$

SYSTEM PRIORITY

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement sets System Priority to a specified value.



Item	Description	Range
new priority	numeric expression, rounded to an integer	0 thru 15

Example Statements

```
SYSTEM PRIORITY Old
IF Critical_code THEN SYSTEM PRIORITY 15
```

Semantics

Zero is the lowest user-specifiable priority and 15 is the highest. The END, ERROR, and TIMEOUT events have an effective priority higher than the highest user-specifiable priority. If no SYSTEM PRIORITY has been executed, minimum system priority is 0.

This statement establishes the minimum for system priority. Once the minimum system priority is raised with this statement, any events of equal or lower priority will be logged but not serviced. In order to allow service of lower-priority events, minimum system priority must be explicitly lowered.

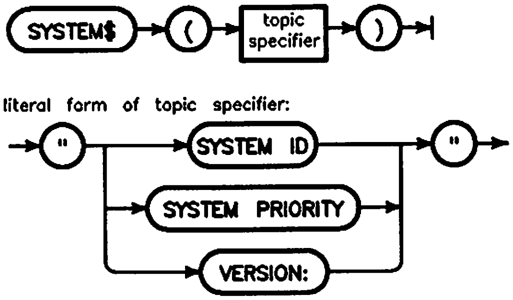
If SYSTEM PRIORITY is used to change the minimum system priority in a subprogram context, the former value is restored when the context is exited.

Error 427 results if SYSTEM PRIORITY is executed in a service routine for an ON ERROR GOSUB or ON ERROR CALL statement.

SYSTEM$

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns a string containing system status and configuration information.



Item	Description	Range
type of information	string expression	—

Example Statements

```
System_prior=VAL(SYSTEM$("SYSTEM PRIORITY"))
```

Semantics

The topic specifier determines what system configuration information SYSTEM$ returns. The following table lists the valid topic specifiers and the information returned for each one.

Topic Specifier	Information Returned
SYSTEM ID	See your instrument-specific HP Instrument BASIC manual for the data returned by this command.
SYSTEM PRIORITY	A string containing numerals which specify the current system priority.
VERSION:	A string containing numerals which specify the revision number HP Instrument BASIC.

TAB

See the PRINT and DISP statements.

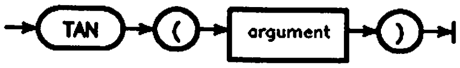
TABXY

See the PRINT statement.

TAN

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the tangent of the angle represented by the argument.



Item	Description/Default	Range Restrictions
argument	numeric expression in the current units of angle when arguments are INTEGER or REAL.	absolute values less than: 8.541 563 906 E+9 deg. or 1.490 784 13 E+8 rad. for INTEGER and REAL arguments;

Examples Statements

```
Tangent=TAN(Angle)
PRINT "Tangent of ";Z;"=";TAN(Z)
```

Semantics

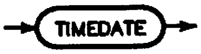
Error 31 is reported for INTEGER and REAL arguments when trying to compute the TAN of an odd multiple of 90 degrees.

If the argument is REAL or INTEGER, the value returned is REAL.

TIMEDATE

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the current value of the real-time clock.



Example Statements

```
Elapsed=TIMEDATE-T0
DISP TIMEDATE MOD 86400
```

Semantics

The value returned by TIMEDATE represents the sum of the last time setting and the number of seconds that have elapsed since that setting was made. The volatile clock value set at power-on is 2.086 629 12 E+11, which represents midnight March 1, 1900. If there is a battery-backed (non-volatile) clock, then the volatile clock is synchronized with it at power-up.

The clock values represent Julian time, expressed in seconds.

The resolution of the TIMEDATE function is .01 seconds. If the clock is properly set, `TIMEDATE MOD` 86400 gives the number of seconds since midnight.

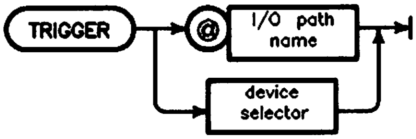
TIMEOUT

See the OFF TIMEOUT and ON TIMEOUT statements.

TRIGGER

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement sends a trigger message to a selected device, or all devices addressed to listen, on the HP-IB.



Item	Description	Range
I/O path name	name assigned to a device or devices	any valid name (see ASSIGN)
device selector	numeric expression, rounded to an integer	(see Glossary)

Example Statements

```
TRIGGER 712
TRIGGER @Hpib
```

Semantics

The computer must be the active controller to execute this statement.

If only the interface select code is specified, all devices on that interface which are addressed to listen are triggered. If a primary address is given, the bus is reconfigured and only the addressed device is triggered.

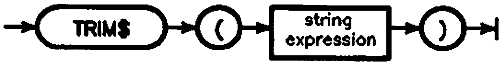
Summary of Bus Actions

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Addressing Specified	Interface Select Code Only	Primary Addressing Specified
Active/Controller	ATN GET	ATN UNL LAG GET	ATN GET	ATN UNL LAG GET
Not Active Controller	Error	Error	Error	Error

TRIM$

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns the string stripped of all leading and trailing ASCII spaces.



Example Statements

```
Unjustify$=TRIM$("   center   ")
Clean$=TRIM$(Input$)
```

Semantics

Only leading and trailing ASCII spaces are removed. Embedded spaces are not affected.

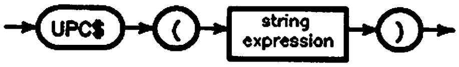
UNTIL

See the REPEAT ... UNTIL construct.

UPC$

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function replaces any lowercase characters with their corresponding uppercase characters.



Example Statements

```
Capital$=UPC$("lower")
IF UPC$(Name$)="TOM" THEN Equal_tom
```

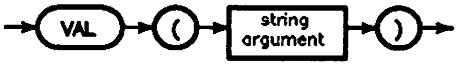
USING

See the DISP, ENTER, LABEL, OUTPUT, and PRINT statements.

VAL

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function converts a string expression into a numeric value.



Item	Description	Range
string argument	string expression	numerals, decimal point, sign and exponent notation

Example Statements

```
Day=VAL(Date$)
IF VAL(Response$)<0 THEN Negative
```

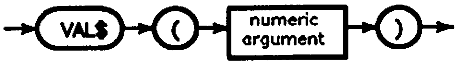
Semantics

The first non-blank character in the string must be a digit, a plus or minus sign, or a decimal point. The remaining characters may be digits, a decimal point, or an E, and must form a valid numeric constant. If an E is present, characters to the left of it must form a valid mantissa, and characters to the right must form a valid exponent. The string expression is evaluated when a non-numeric character is encountered or the characters are exhausted.

VAL$

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This function returns a string representation of the value of the argument. The returned string is in the default print format, except that the first character is not a blank for positive numbers. No trailing blanks are generated.



Item	Description	Range
numeric argument	numeric expression	—

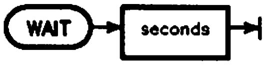
Example Statements

```
PRINT Esc$;VAL$(Cursor-1)
Special$=Text$&VAL$(Number)
```

WAIT

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes

This statement will cause the computer to wait approximately the number of seconds specified before executing the next statement. Numbers less than 0.001 do not generate a WAIT interval.



Item	Description	Range
seconds	numeric expression, rounded to the nearest thousandth	less than 2 147 483.648

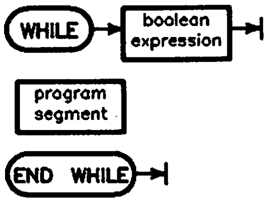
Example Statements

```
WAIT 3
WAIT Old_time/2
```

WHILE

Keyboard Executable	No
Programmable	Yes
In an IF ... THEN ...	No

This construct defines a loop which is executed as long as the boolean expression in the WHILE statement evaluates to true (evaluates to a non-zero value).



Item	Description	Range
boolean expression	numeric expression: evaluated as true if nonzero and false if zero.	—
program segment	any number of contiguous program lines not containing the beginning or end of a main program or subprogram, but which may contain properly nested construct(s).	—

Example Program Segments

```
840   WHILE Value<Min OR Value>Max
850     BEEP
860     INPUT "Out of range; RE-ENTER",Value
870   END WHILE

1220  WHILE P<=LEN(A$)
1230    IF NUM(A$[P])<32 THEN
1240      A$[P]=A$[P+1]  ! Remove control codes
1250    ELSE
1260      P=P+1          ! Go to next character
1270    END IF
1280  END WHILE
```

Semantics

The WHILE ... END WHILE construct allows program execution dependent on the outcome of a relational test performed at the *start* of the loop. If the condition is true, the program segment between the WHILE and END WHILE statements is executed and a branch is made back to the WHILE statement. The program segment will be repeated until the test is false. When the relational test is false, the program segment is skipped and execution continues with the first program line after the END WHILE statement.

Branching into a WHILE ... END WHILE construct (via a GOTO) results in normal execution up to the END WHILE statement, a branch back to the WHILE statement, and then execution as if the construct had been entered normally.

Nesting Constructs Properly

WHILE ... END WHILE constructs may be nested within other constructs, provided the inner construct begins and ends before the outer construct can end.

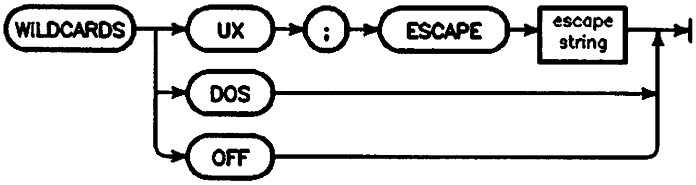
WIDTH

See the PRINTER IS statement.

WILDCARDS

This statement enables and disables wildcard recognition in file related commands.

Keyboard Executable	Yes
Programmable	Yes
In an IF ... THEN ...	Yes



Item	Description	Range
escape string	string expression	any expression that evaluates to "\", "’", or the null string

Example Statements

```
WILDCARDS DOS
WILDCARDS OFF
```

Semantics

Not all implementations of HP Instrument BASIC support WILDCARDS. Refer to your instrument programming manual for details.

Wildcard recognition is disabled at power-up and after SCRATCH A or SCRATCH BIN. To use wildcards, you must explicitly enable them using the WILDCARDS keyword.

Note that you *must* specify an escape string with WILDCARDS UX. The backslash character (“\”) is recommended for HP-UX compatibility.

Definitions for WILDCARDS UX

Wildcard	Meaning
?	Matches any single character. For example, X?? matches file names containing *exactly three* characters that begin with the letter X.
*	Matches any sequence of 0 or more characters. For example, X*Y matches any file names of *two or more* characters that begin with X and end with Y.
[*list*]	Matches any character specified by *list*. The *list* may consist of individual characters or a range of characters. The expression *[aeiou]* matches any filename containing *at least one* lower case vowel. Ranges are specified using a hyphen. For example, *[0-9]* matches any file names containing *at least one* digit.
[!*list*]	Matches any character *not* specified by *list*. Note that the ! must be the first character after [to have the special meaning *not;* otherwise, it is used for matching like any other character. The *list* contains the same types of individual characters and ranges as discussed above. For example, *[!0-9]* matches any file names containing at least one non-digit.

The escape character specified with WILDCARDS UX is used to cancel the special meaning of wildcard characters immediately following it within a file name specification. The escape character itself can be used in a file name by typing it twice.

```
100  WILDCARDS UX ; ESCAPE "\"
110  PURGE "file_*"            deletes files prefixed file_
120  PURGE "file_\*"           deletes file named file_*
130  PURGE "file_\\"           deletes file named file_\
```

Wildcards can be used *only* as the rightmost part of a file specifier.

/user/smith/my_dir*	*allowed*
/user/smith/my_dir1/*file	*allowed*
/user/*/my_dir1/file1	*not allowed*
/user/smith/*/file1	*not allowed*

Wildcards generate matches through file name **expansion** or file name **completion.** Expansion means that more than one file name can match the wildcard specification. Completion means that one and only one file name can match the wildcard specification, or an error is generated.

Commands Allowing Wildcards

File Name Expansion	File Name Completion
CAT	ASSIGN
PURGE	GET
COPY	MSI
LINK	PRINTER IS
	RENAME
	RE-SAVE
	RE-STORE

A

Error Messages

1 Configuration error. Statement recognized but can't be executed.

2 Memory overflow. If you get this error while loading a file, the program is too large for the computer's memory. If the program loads, but you get this error when you press RUN, then the overflow was caused by the variable declarations. Either way, you need to modify the program or add more read/write memory.

3 Line not found in current context. Could be a GOTO or GOSUB that references a non-existent (or deleted) line, or an EDIT command that refers to a non-existent line label.

4 Improper RETURN. Executing a RETURN statement without previously executing an appropriate GOSUB or function call. Also, a RETURN statement in a user-defined function with no value specified.

5 Improper context terminator. You forgot to put an END statement in the program. Also applies to SUBEND and FNEND.

6 Improper FOR ... NEXT matching. Executing a NEXT statement without previously executing the matching FOR statement. Indicates improper nesting or overlapping of the loops.

7 Undefined function or subprogram. Attempt to call a SUB or user-defined function that is not in memory. Look out for program lines that assumed an optional CALL.

8 Improper parameter matching. A type mismatch between a pass parameter and a formal parameter of a subprogram.

9 Improper number of parameters. Passing either too few or too many parameters to a subprogram. Applies only to non-optional parameters.

10 String type required. Attempting to return a numeric from a user-defined string function.

11 Numeric type required. Attempting to return a string from a user-defined numeric function.

12 Attempt to redeclare variable. Including the same variable name twice in declarative statements such as DIM or INTEGER.

13 Array dimensions not specified. Using the (*) symbol after a variable name when that variable has never been declared as an array.

15 Invalid bounds. Attempt to declare an array with more than 32 767 elements or with upper bound less than lower bound.

16 Improper or inconsistent dimensions. Using the wrong number of subscripts when referencing an array element.

17 Subscript out of range. A subscript in an array reference is outside the current bounds of the array.

18 String overflow or substring error. String overflow is an attempt to put too many characters into a string (exceeding dimensioned length). This can happen in an assignment, an ENTER an INPUT, or a READ. A substring error is an attempted violation of the rules for substrings. Watch out for null strings where you weren't expecting them.

19 Improper value or out of range. A value is too large or too small. Applies to items found in a variety of statements. Often occurs when the number builder overflows (or underflows) during an I/O operation.

20 INTEGER overflow. An assignment or result exceeds the range allowed for INTEGER variables. Must be −32 768 thru 32 767.

22 REAL overflow. An assignment or result exceeds the range allowed for REAL variables.

24 Trig argument too large for accurate evaluation. Out-of-range argument for a function such as TAN.

25 Magnitude of ASN or ACS argument is greater than 1. Arguments to these functions must be in the range −1 thru +1.

26 Zero to non-positive power. Exponentiation error.

27 Negative base to non-integer power. Exponentiation error.

28 LOG or LGT of a non-positive number.

29 Illegal floating point number. Does not occur as a result of any calculations, but is possible when a FORMAT OFF I/O operation fills a REAL variable with something other than a REAL number.

30 SQR of a negative number.

31 Division (or MOD) by zero.

32 String does not represent a valid number. Attempt to use "non-numeric" characters as an argument for VAL, data for a READ, or in response to an INPUT statement requesting a number.

33 Improper argument for NUM or RPT$. Null string not allowed.

34 Referenced line not an IMAGE statement. A USING clause contains a line identifier, and the line referred to is not an IMAGE statement.

35 Improper image. See IMAGE or the appropriate keyword in the *HP Instrument BASIC Language Reference.*

36 Out of data in READ. A READ statement is expecting more data than is available in the referenced DATA statements. Check for deleted lines, proper use of RESTORE, or typing errors.

38 TAB or TABXY not allowed here. The tab functions are not allowed in statements that contain a USING clause. TABXY is allowed only in a PRINT statement.

40 Improper attempt to renumber Line numbers must be whole numbers from 1 to 32 766.

41 First line number > second. Attempted to SAVE, REN, DELETE, LIST or SECURE lines with improper line number parameters.

46 Attempting a SAVE when there is no program in memory.

47 COM declarations are inconsistent or incorrect. Includes such things as mismatched dimensions, unspecified dimensions, and blank COM occurring for the first time in a subprogram.

49 Branch destination not found. A statement such as ON ERROR or ON KEY refers to a line that does not exist. Branch destinations must be in the same context as the ON ... statement.

52 Improper mass storage volume specifier. The characters used for a msvs do not form a valid specifier. This could be a missing colon, too many parameters, illegal characters, etc.

53 Improper file name. The file name is too long or has characters that are not allowed. LIF file names are limited to 10 characters; SRM file names to 16 characters; HFS file names to 14 characters. Foreign characters are allowed, but punctuation (in commands, etc.) is not.

54 Duplicate file name. The specified file name already exists in directory. It is illegal to have two files with the same name on one LIF volume or in the same SRM or HFS directory.

55 Directory overflow. Although there may be room on the media for the file, there is no room in the directory for another file name. LIF Discs initialized by HP Instrument BASIC have room for over 100 entries in the directory, but other systems may make a directory of a different size.

56 File name is undefined. The specified file name does not exist in the directory. Check the contents of the disc with a CAT command.

58 Improper file type. Many mass storage operations are limited to certain file types. For example, LOAD is limited to PROG files and ASSIGN is limited to ASCII, BDAT, and HP-UX files.

59 End of file or buffer found. For files: No data left when reading a file, or no space left when writing a file. For buffers: No data left for an ENTER, or no buffer space left for an OUTPUT.

60 End of record found in random mode. Attempt to ENTER or OUTPUT a field that is larger than a defined record.

62 Protect code violation. Failure to specify the protect code of a protected file, or attempting to protect a file of the wrong type.

64 Mass storage media overflow. The disc is full. (There is not enough free space for the specified file size, or not enough contiguous free space on a LIF disc.)

65 Incorrect data type.

66 INITIALIZE failed. Too many bad tracks found. The disc is defective, damaged, or dirty.

67 Illegal mass storage parameter. A mass storage statement contains a parameter that is out of range, such as a negative record number or an out of range number of records.

68 Syntax error occurred during GET. One or more lines in the file could not be stored as valid program lines. The offending lines are usually listed on the system printer. Also occurs if the first line in the file does not start with a valid line number.

72 Disc controller not found or bad controller address. The msus contains an improper device selector, or no external disc is connected.

73 Improper device type in mass storage volume specifier. The msvs has the correct general form, but the characters used for a device type are not recognized.

76 Incorrect unit number in mass storage volume specifier. The msvs contains a unit number that does not exist on the specified device.

77 Operation not allowed on open file. The specified file is assigned to an I/O path name which has not been closed.

78 Invalid mass storage volume label. Usually indicates that the media has not been initialized on a compatible system. Could also be a bad disc.

79 File open on target device. Attempt to copy an entire volume with a file open on the destination disc.

80 Disc changed or not in drive. Either there is no disc in the drive or the drive door was opened while a file was assigned.

81 Mass storage hardware failure. Also occurs when the disc is pinched and not turning. Try reinserting the disc.

82 Mass storage volume not present. Hardware problem or an attempt to access a left-hand drive on the Model 226.

83 Write protected. Attempting to write to a write-protected disc. This includes many operations such as PURGE, INITIALIZE, CREATE, SAVE, OUTPUT, etc.

84 Record not found. Usually indicates that the media has not been initialized.

85 Media not initialized. (Usually not produced by the internal drive.)

87 Record address error. Usually indicates a problem with the media.

88 Read data error. The media is physically or magnetically damaged, and the data cannot be read.

89 Checkread error. An error was detected when reading the data just written. The media is probably damaged.

90 Mass storage system error. Usually a problem with the hardware or the media.

93 Incorrect volume code in msvs. The msvs contains a volume number that does not exist on the specified device.

100 Numeric IMAGE for string item.

101 String IMAGE for numeric item.

102 Numeric field specifier is too large. Specifying more than 256 characters in a numeric field.

103 Item has no corresponding IMAGE. The image specifier has no fields that are used for item processing. Specifiers such as `# X /` are not used to process the data for the item list. Item-processing specifiers include things like `K D B A`.

105 Numeric IMAGE field too small. Not enough characters are specified to represent the number.

106 IMAGE exponent field too small. Not enough exponent characters are specified to represent the number.

107 IMAGE sign specifier missing. Not enough characters are specified to represent the number. Number would fit except for the minus sign.

117 Too many nested structures. The nesting level is too deep for such structures as FOR, SELECT, IF, LOOP, etc.

118 Too many structures in context. Refers to such structures as FOR/NEXT, IF/THEN/ELSE, SELECT/CASE, WHILE, etc.

121 Line not in main program. The run line specified in a LOAD or GET is not in the main context. 122 Program is not continuable. The program is in the stopped state, not the paused state. CONT is allowed only in the paused state.

122 Program is not continuable.

125 Program not running.

126 Quote mark in unquoted string. Quote marks must be used in pairs.

127 Statements which affect the knob mode are out of order.

128 Line too long during GET.

131 Unrecognized non-ASCII keycode. An output to the keyboard contained a CHR$(255) followed by an illegal byte.

134 Improper SCRATCH statement.

135 READIO/WRITEIO to nonexist mem. Attempt to access nonexistent memory location.

136 REAL underflow. The input or result is closer to zero than $10^{/308}$ (approximately).

146 Duplicate line label in context. There cannot be two lines with the same line label in one context.

150 Illegal interface select code or device selector. Value out of range.

153 Insufficient data for ENTER. A statement terminator was received before the variable list was satisfied.

154 String greater than 32 767 bytes in ENTER.

155 Bad interface register number. Attempted to access nonexistent register.

156 Illegal expression type in list. For example, trying to ENTER into a constant.

157 No ENTER terminator found. The variable list has been satisfied, but no statement terminator was received in the next 256 characters. The # specifier allows the statement to terminate when the last item is satisfied.

158 Improper image specifier or nesting images more than 8 deep. The characters used for an image specifier are improper or in an improper order.

159 Numeric data not received. When entering characters for a numeric field, an item terminator was encountered before any "numeric" characters were received.

160 Attempt to enter more than 32 767 digits into one number.

163 Interface not present. The intended interface is not present, set to a different select code, or is malfunctioning.

165 Image specifier greater than dimensioned string length.

167 Interface status error. Exact meaning depends upon the interface type. With HP-IB, this can happen when a non-controller operation by the computer is aborted by the bus.

168 Device timeout occurred and the ON TIMEOUT branch could not be taken.

170 I/O operation not allowed. The I/O statement has the proper form, but its operation is not defined for the specified device. For example, using an HP-IB statement on a non-HP-IB interface or directing a LIST to the keyboard.

171 Illegal I/O addressing sequence. The secondary addressing in a device selector is improper or primary address too large for specified device.

173 Active or system controller required. The HP-IB is not active controller and needs to be for the specified operation.

174 Nested I/O prohibited. An I/O statement contains a user-defined function. Both the original statement and the function are trying to access the same file or device.

177 Undefined I/O path name. Attempting to use an I/O path name that is not assigned to a device or file.

178 Trailing punctuation in ENTER. The trailing comma or semicolon that is sometimes used at the end of OUTPUT statements is not allowed at the end of ENTER statements.

180 HFS disc may be corrupt.

181 No room in HFS buffers.

182 Not supported by HFS.

183 Permission denied. You have insufficient access rights for the specified operation.

185 HFS volumes must be mounted.

186 Cannot open the specified directory.

187 Cannot link across devices.

188 Renaming using ., .., or / not allowed.

189 Too many open files.

190 File size exceeds the maximum allowed.

191 Too many links to a file.

192 Networking error.

193 Resource deadlock would occur.

194 Operation would block.

195 Too many levels of a symbolic link.

196 Target device busy.

197 Incorrect device type in device file.

198 Invalid msvs mapping (e.g., not a directory)

199 Incorrect access to mounted HFS volume

200 Cannot access disk (e.g., uninitialized media)

292 Wildcards not allowed. Attempt to use wild cards with CREATE, INITIALIZE or SAVE.

293 Operations failed on some files. Wildcard operation did not succeed on all files found.

294 Wildcard matches > 1 item. Wildcard name expanded to more than one file name.

295 Improper destination type.

296 Unable to purge file. Unable to purge file during copy operation.

301 Cannot do while connected.

303 Not allowed when trace active.

304 Too many characters without terminator.

308 Illegal character in data.

310 Not connected.

332 Non-existent dimension given. Attempt to specify a non-existent dimension in a SIZE or BASE operation.

345 CASE expression type mismatch. The SELECT statement and its CASE statements must refer to the same general type, numeric or string.

347 Structures improperly matched. There is not a corresponding number of structure beginnings and endings. Usually means that you forgot a statement such as END IF, NEXT, END SELECT, etc.

401 Bad system function argument. An invalid argument was given to a SYSTEM$ function.

427 Priority may not be lowered.

453 File in use—HFS error.

459 Specified file is not a directory—HFS error.

460 Directory not empty—HFS error.

465 Invalid rename across volumes.

466 Duplicate volume entries.

482 Cannot move a directory with a RENAME operation—HFS error.

485 Invalid volume copy—HFS error.

522 Device not present.

609 IVAL or DVAL result too large. Attempt to convert a binary, octal, decimal, or hexadecimal string into a value outside the range of the function.

810 Feature not supported on HP-UX.

816 Invalid opcode in program. Attempted to load a corrupt program.

881 Array is not INTEGER type.

902 Must delete entire context. Attempt to delete a SUB or DEF FN statement without deleting its entire context.

903 No room to renumber. While EDIT mode was renumbering during an insert, all available line numbers were used between insert location and end of program.

906 SUB or DEF FN not allowed here. Attempt to insert a SUB or DEF FN statement into the middle of a context. Subprograms must be appended at the end.

909 May not replace SUB or DEF FN. Similar to deleting a SUB or DEF FN. Attempted to insert lines: between a CSUB statement and the following SUB, DEF FN, or CSUB statement; or after a final CSUB statement at the end of the program.

910 Identifier not found in this context. The keyboard-specified variable does not already exist in the program. Variables cannot be created from the keyboard; they must be created by running a program.

911 Improper I/O list.

920 Numeric constant not allowed.

921 Numeric identifier not allowed.

922 Numeric array element not allowed.

923 Numeric expression not allowed.

924 Quoted string not allowed.

925 String identifier not allowed.

926 String array element not allowed.

927 Substring not allowed.

928 String expression not allowed.

929 I/O path name not allowed.

930 Numeric array not allowed.

931 String array not allowed.

935 Identifier is too long: 15 characters maximum.

936 Unrecognized character. Attempt to store a program line containing an improper name or illegal character.

940 Duplicate formal param name.

942 Invalid I/O path name. The characters after the `@` are not a valid name. Names must start with a letter.

943 Invalid function name. The characters after the `FN` are not a valid name. Names must start with a letter.

946 Dimensions are inconsistent with previous declaration. The references to an array contain a different number of subscripts at different places in the program.

947 Invalid array bounds. Value out of range, or more than 32 767 elements specified.

948 Multiple assignment prohibited. You cannot assign the same value to multiple variables by stating `X=Y=Z=0`. A separate assignment must be made for each variable.

949 Syntax error at cursor. The statement you typed contains elements that don't belong together, are in the wrong order, or are misspelled.

950 Must be a positive integer.

951 Incomplete statement. This keyword must be followed by other items to make a valid statement.

961 CASE expression type mismatch. The CASE line contains items that are not the same general type, numeric or string.

962 Programmable only: cannot be executed from the keyboard.

963 Command only: cannot be stored as a program line.

977 Statement is too complex. Contains too many operators and functions. Break the expression down so that it is performed by two or more program lines.

980 Too many symbols in this context. Symbols include variable names, I/O path names, COM block names, subprogram names, and line identifiers.

982 Too many subscripts: maximum of six dimensions allowed.

983 Wrong type or number of parameters. An improper parameter list for a machine-resident function.

985 Invalid quoted string.

987 Invalid line number: must be a whole number 1 thru 32 766.

999 Internal Error. Hardware failure has occurred.

B

Glossary

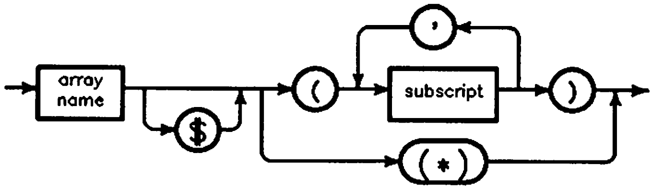
angle mode — The current units used for expressing angles. Either degrees or radians may be specified, using the DEG or RAD statements, respectively. The default at power-on and SCRATCH A is radians.

A subprogram "inherits" the angle mode of the calling context. If the angle mode is changed in a subprogram, the mode of the calling context is restored when execution returns to the calling context.

array — A structured data type that can be of type REAL, INTEGER, or string. Arrays are created with the DIM, REAL, INTEGER, or COM statements. Arrays have 1 to 6 dimensions; each dimension is allowed 32 767 elements. The lower and upper bounds for each dimension must fall in the range −32 767 thru +32 767, and the lower bound must not exceed the upper bound. The default base in every environment is zero.

Each element in a string array is a string whose maximum length is specified in the declaring statement. The declared length of a string must be in the range 1 thru 32 767.

To specify an entire array, the characters (*) are placed after the array name. To specify a single element of an array, subscripts are placed in parentheses after the array name. Each subscript must not be less than the current lower bound or greater than the current upper bound of the corresponding dimension.



If an array is not explicitly dimensioned, it is implicitly given the number of dimensions used in its first occurrence, with an upper bound of 10. Undeclared strings have a default length of 18.

ASCII — This is the acronym for "American Standard Code for Information Interchange". It is a commonly used code for representing letters, numerals, punctuation, special characters, and control characters. A table of the characters in the ASCII set and their code values can be found in the back of this manual.

bit — This term comes from the words "binary digit". A bit is a single digit in base 2 that must be either a 1 or a 0.

byte

A group of eight bits processed as a unit.

command

A statement that can be typed on the input line and executed (see "statement").

context

An instance of an environment. A context consists of a specific instance of all data types and system parameters that may be accessed by a program at a specific point in its execution. Context changes occur when subprograms are invoked or exited.

device selector

A numeric expression used to specify the source or destination of an I/O operation. A device selector can be either an interface select code or a combination of an interface select code and an HP-IB primary address. To construct a device selector with a primary address, multiply the interface select code by 100 and add the primary address. For instance, a device selector that specifies the device at address 1 on interface select code 7 is 701. The device at address 0 on interface select code 14 is 1400. Device selector 1516 selects interface select code 15 and primary address 16.

Secondary addresses may be appended after a primary address by multiplying the device selector by 100 and adding the address. This may be repeated up to 6 times, adding a new secondary address each time. A device selector, once rounded, may contain a maximum of 15 digits. For example, 70502 selects interface 7, primary address 05, and secondary address 02.

directory name

A directory name specifies a directory of files on a hierarchically structured mass storage volume.

The directory name on a Hierarchical File System (HFS) volume consists of 1 to 14 characters, which may include all ASCII characters except "/" and ":" and "<". Spaces are ignored.

dyadic operator

An operator that performs its operation with *two* expressions. It is placed between the two expressions. The following dyadic operators are available:

Dyadic Operator	Operation
+	REAL, or INTEGER addition
-	REAL, or INTEGER subtraction
*	REAL, or INTEGER multiplication
/	REAL division
^	REAL, or INTEGER exponentiation[1]
&	String concatenation
`DIV`	Gives the integer quotient of a division
`MOD`	Gives the remainder of a division
`MODULO`	Gives the remainder of a division, similar to `MOD`
=	Comparison for equality
<>	Comparison for inequality
<	Comparison for less than
>	Comparison for greater than
<=	Comparison for less than or equal to
>=	Comparison for greater than or equal to
`AND`	Logical AND
`OR`	Logical inclusive OR
`EXOR`	Logical exclusive OR

file name

A name used to identify a file. The length and characters allowed in a file name vary according to the format of the volume on which the file resides.

- A file name on a Logical Interchange Format (LIF) volume consists of 1 to 10 characters, which may include uppercase and lowercase letters, digits 0 through 9, the underbar (_) character, and national language characters [CHR$(161) through CHR$(254)]. The first character in a LIF-compatible file name must be a letter. Spaces are ignored. (Note that some LIF implementations do not allow lowercase letters.)
- A file name on a Hierarchical File System (HFS) volume consists of 1 to 14 characters, which may include all ASCII characters except "/" and ":" and "<". Spaces are ignored.
- A file name on an MS-DOS (DOS) volume consists of two parts; a file name and an optional extension. The file name contains from 1 to 8 characters and the extension contains from 1 to 3 characters. A period separates the extension from the file name. All ASCII characters may be used except for the following: ".", "[", "]", "?", "\", "/", "=", ",", "+", "*", ":", ";", " ", "<", ">", "|".

function

A procedural call that returns a value. The call can be to a user-defined-function subprogram (such as FNInvert) or a machine-resident function (such as COS or EXP). The value returned by the function is used in place of the function call when evaluating the expression containing the function call.

hierarchy

When a numeric or string expression contains more than one operation, the order of operations is determined by a precedence system. Operations with the highest precedence are performed first. Multiple operations with the same precedence are performed left to right. The following tables show the hierarchy for numeric and string operations.

Math Hierarchy

Precedence	Operator
Highest	Parentheses: (may be used to force any order of operations)
	Functions: user-defined and machine-resident
	Exponentiation: ^
	Multiplication and division: `*` `/` `MOD` `DIV` `MODULO`
	Addition, subtraction, monadic plus and minus: + −
	Relational operators: = <> < > <= >=
	`NOT`
	`AND`
Lowest	`OR` `EXOR`

String Hierarchy

Precedence	Operator
Highest	Parentheses
	Functions (user-defined and machine-resident) and substring operations
Lowest	Concatenation: &

I/O path

A combination of firmware and hardware that can be used during the transfer of data to and from an HP Instrument BASIC program. Associated with an I/O path is a unique table that describes the I/O path. This association table uses 148 bytes and is referenced when an I/O path name is used. For further details, see the ASSIGN statement.

INTEGER

A numeric data type stored internally in two bytes. Two's-complement representation is used, giving a range of −32 768 thru +32 767. If a numeric variable is not explicitly declared as an INTEGER, it is a REAL.

integer

A number with no fractional part; a whole number.

interface select code

A numeric expression that selects an interface for an I/O operation. Interface select codes 1 thru 7 are generally reserved for internal interfaces. Interface select codes 8 thru 31 are generally used for external interfaces. The internal HP-IB interface with select code 7 can be specified in statements that are restricted to external devices. (Also see "device selector".)

keyword
A group of uppercase ASCII letters that has a predefined meaning to the computer. Keywords may be typed using all lowercase or all uppercase letters.

LIF
This is the acronym for "Logical Interchange Format". This HP standard defines the format of mass storage files and directories. It allows the interchange of data between different machines. See "file name" for file name restrictions.

LIF protect code
A non-listable, two-character code kept with a file description in the directory of a LIF volume. It guards against accidental changes to an individual file. It may be any two characters, but must not contain a ">" since that is used to terminate the protect code. Blanks are trimmed from protect codes. When the result contains more than two characters, only the first two are used as the actual protect code. A protect code that is the null string (or all blanks) is interpreted as no protect code.

literal
A string constant. When quote marks are used to delimit a literal, those quote marks are not part of the literal. To include a quote mark in a literal, type two consecutive quote marks. The drawings showing literal forms of specifiers (such as file specifiers) show the quote marks required to delimit the literal.

monadic operator
An operator that performs its operation with one expression. It is placed in front of the expression. The following monadic operators are available:

Monadic Operator	**Operation**
-	Reverses the sign of an expression
+	Identity operator
NOT	Logical complement

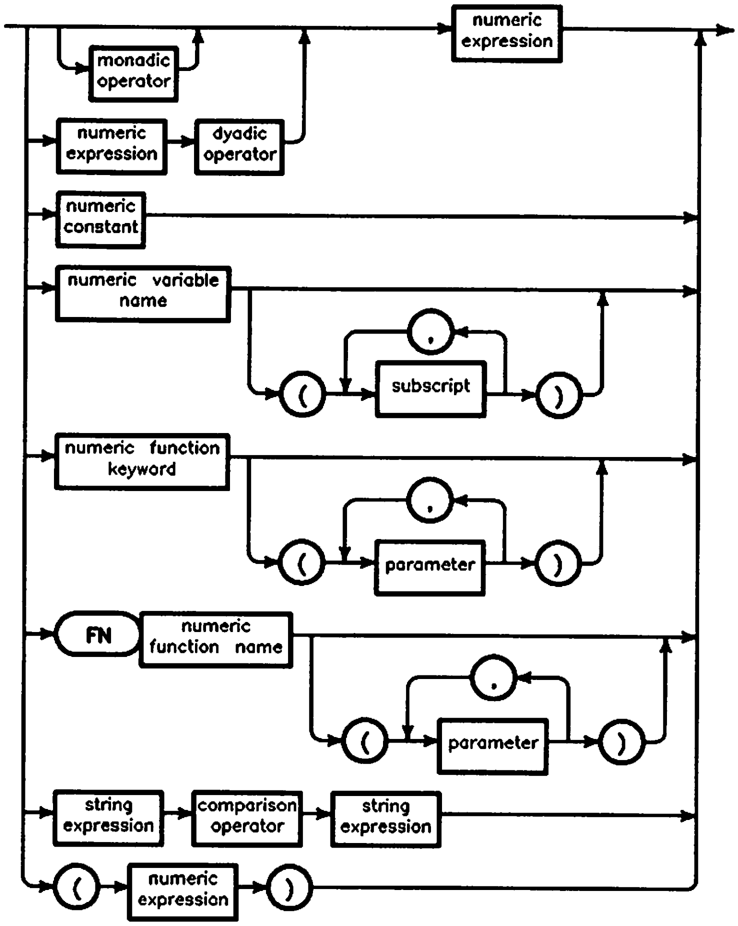
msus
The acronym for "mass storage unit specifier". This archaic term is no longer used, because: it is not descriptive of newer mass storage devices which may have multiple *units* or multiple *volumes;* and it is not an industry-standard term. See the Glossary entry for **volume specifier.**

msvs
The acronym for "mass storage volume specifier". See the Glossary entry for volume specifier.

name
A name identifies one of the following: variable, line label, common block, I/O path, function, or subprogram. A name consists of one to fifteen characters. The first character must be an ASCII letter or one of the characters from CHR$(161) thru CHR$(254). The remaining characters, if any, can be ASCII letters, numerals, the underbar (_), or national language characters CHR$(161) thru CHR$(254). Names may be typed using any combination of uppercase and lowercase letters, unless the name uses the same letters as a

keyword. Conflicts with keywords are resolved by mixing the letter case in the name. (Also see "file name", "directory name", and "volume name".)

numeric expression



Item	Description
monadic operator	An operator that performs its operation on the expression immediately to its right: + - NOT
dyadic operator	An operator that performs its operation on the two expressions it is between: ^ * / MOD DIV + - = <> < > <= >= AND OR EXOR MODULO
numeric constant	A numeric quantity whose value is expressed using numerals, decimal point, and optional exponent notation
numeric variable name	The name of a numeric variable or the name of a numeric array from which an element is extracted using subscripts
subscript	A numeric expression used to select an element of an array (see "array")
numeric function keyword	A keyword that invokes a machine-resident function which returns a numeric value
numeric function name	The name of a user-defined function that returns a numeric value
parameter	A numeric expression, string expression, or I/O path name that is passed to a function
comparison operator	An operator that returns a 1 (true) or a 0 (false) based on the result of a relational test of the operands it separates: > < <= >= = <>

permission

A file-access permission on an HFS volume.

primary address

A numeric expession in the range of 0 thru 31 that specifies an individual device on an interface which is capable of servicing more than one device. The HP-IB interface can service multiple devices. (Also see "device selector".)

program line

A statement that is preceded by a line number (and an optional line label) and stored in a program (see "statement").

protect code

See "LIF protect code".

REAL

A numeric data type that is stored internally in eight bytes using sign-and-magnitude binary representation. One bit is used for the number's sign, 11 bits for a biased exponent (bias = 1023), and 52 bits for a mantissa. On all values except 0, there is an implied "1." preceding the mantissa (this can be thought of as the 53rd bit). The range of REAL numbers is approximately:

−1.797 693 134 862 32 E+308 thru −2.225 073 858 507 2 E−308, 0, and +2.225 073 858 507 2 E−308 thru +1.797 693 134 862 32 E+308.

If a numeric variable is not explicitly declared as INTEGER, it is REAL.

record

The records referred to in the HP Instrument BASIC manuals are *defined* records. Defined records are the smallest unit of storage directly accessible on the mass storage media. The length of a record is different for various types of files. For ASCII files, the record length is the same as the sector size (256, 512, or 1024 bytes). For HP-UX files, defined records are always 1 byte long.

For BDAT files, the defined record length is determined when a BDAT file is created by a CREATE BDAT statement. All records in a file are the same size.

There is another type of record called a “physical record” (or sector) which is the unit of storage handled by the mass storage device and the operating system. Physical records contain 256, 512, or 1024 bytes and are not accessible to the user via standard HP Instrument BASIC statements.

recursive

See “recursive”.

row-major order

The order of accessing an array in which the right-most subscript varies the fastest.

secondary address

A device-dependent command sent on HP-IB. It can be interpreted as a secondary address for the extended talker/listener functions or as part of a command sequence. (Also see “device selector”.)

specifier

A string used to identify a method for handling an I/O operation. A specifier is usually a string expression. For example: *mass storage volume specifier* selects the proper drivers for a mass storage volume, and *plotter specifier* chooses the protocol of a plotting device.

statement

A keyword combined with any additional items that are allowed or required with that keyword. If a statement is placed after a line number and stored, it becomes a program line. If a statement is typed without a line number and executed, it is called a command.

string

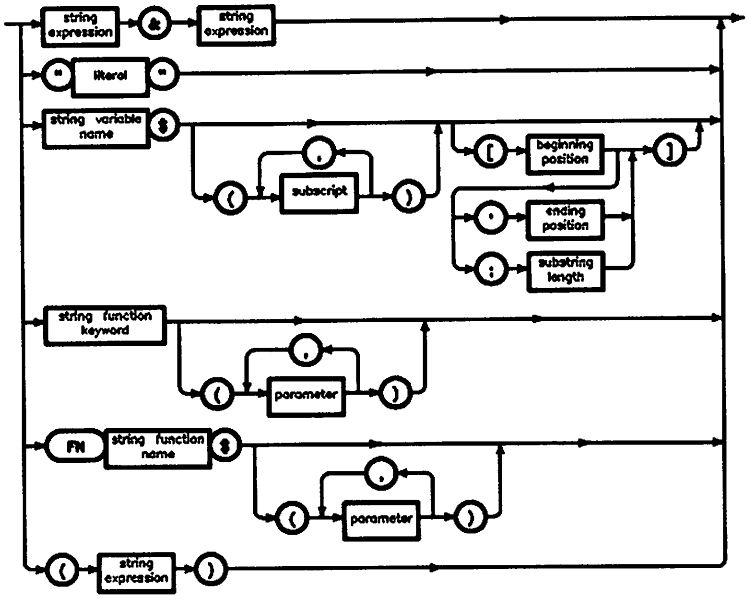
A data type comprised of a contiguous series of characters. Strings require one byte of memory for each character of declared length, plus a two-byte length header. Characters are stored using an extended ASCII character set. The first character in a string is in position 1. The maximum length of a string is 32 767 characters. The current length of a string can never exceed the dimensioned length.

If a string is not explicitly dimensioned, it is implicitly dimensioned to 18 characters. Each element in an implicitly dimensioned string array is dimensioned to 18 characters.

When a string is empty, it has a current length of zero and is called a “null string”. All strings are null strings when they are declared. A null string can be represented as an empty literal (for example: `A$=""`) or as one of three special cases of substring. The substrings that represent the null string are:

1. Beginning position one greater than current length
2. Ending position one less than beginning position
3. Maximum substring length of zero

string expression



Item	Description
literal	A string constant composed of any characters available on the keyboard, including those generated with the ANY CHAR key.
string variable name	The name of a string variable or the name of a string array from which a string is extracted using subscripts
subscript	A numeric expression used to select an element of an array (see "array")
beginning position	A numeric expression specifying the position of the first character in a substring (see "substring")
ending position	A numeric expression specifying the position of the last character in a substring (see "substring")
substring length	A numeric expression specifying the maximum number of characters to be included in a substring (see "substring")
string function keyword	A keyword that invokes a machine-resident function which returns a string value. String function keywords always end with a dollar sign.
string function name	The name of a user-defined function that returns a string value
parameter	A numeric expression, string expression, or I/O path name that is is passed to a function

subprogram Can be a SUB subprogram or a user-defined-function subprogram (DEF FN). The first line in a SUB subprogram is a SUB statement. The last line in a

SUB subprogram (except for comments) is a SUBEND statement. The first line in a function subprogram is a DEF FN statement. The last line in a function (except for comments) is an FNEND statement. Subprograms must follow the END statement of the main program.

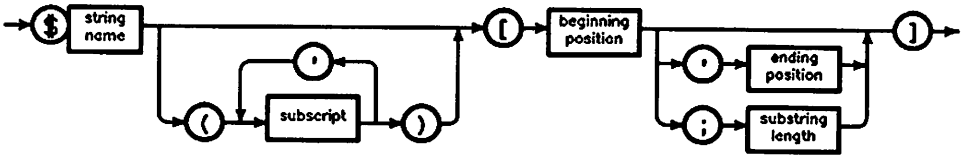
SUB subprograms are invoked by CALL. Function subprograms are invoked by an FN function occurring in an expression. A function subprogram returns a value that replaces the occurrence of the FN function when the expression is evaluated. Subprograms may alter the values of parameters passed by reference or variables in COM. It is recommended that you do not let function subprograms alter values in that way.

Invoking a subprogram establishes a new context. The new context remains in existence until the subprogram is properly exited or program execution is stopped. Subprograms can be recursive.

subroutine

A program segment accessed by a GOSUB statement and ended with a RETURN statement.

substring



A substring is a contiguous series of characters that comprises all or part of a string. Substrings may be accessed by specifying a beginning position, or a beginning position and an ending position, or a beginning position and a maximum substring length.

The beginning position must be at least one and no greater than the current length plus one. When only the beginning position is specified, the substring includes all characters from that position to the current end of the string.

The ending position must be no less than the beginning position minus one and no greater than the dimensioned length of the string. When both beginning and ending positions are specified, the substring includes all characters from the beginning position to the ending position or current end of the string, whichever is less.

The maximum substring length must be at least zero and no greater than one plus the dimensioned length of the string minus the beginning position. When a beginning position and substring length are specified, the substring starts at the beginning position and includes the number of characters specified by the substring length. If there are not enough characters available, the substring includes only the characters from the beginning position to the current end of the string.

volume

A named mass storage media, or portion thereof, which may contain several files. With HP Instrument BASIC, volumes are entities which are recognized by the disc controller. (This is in contrast to Workstation Pascal *logical* volumes, which are handled by the "Unitable" construct in the "TABLE" program; this program partitions a "hard" volume into several "logical" volumes by using byte offsets from sector number zero.)

volume name (or label)

A name used to identify a mass storage volume. The volume name is assigned to the volume at initialization, (and read with CAT).

- LIF volume names consist of 1 to 6 characters which may be any ASCII character except “/”, “:”, “;”, and “<”.
- HFS volume names may contain 1 to 6 characters, which may be any ASCII character except “/” and “:” and “<”. Spaces are ignored.
- DOS volume names may contain 1 to 11 characters, which may be any ASCII character except “.”, “[”, “]”, “?”, “\”, “/”, “=”, “"”, “,”, “+”, “*”, “:”, “;”, “ ”, “<”, “>”, and “|”.

volume specifier

A string of information that identifies a mass storage volume. It consists of a device type (optional), device selector, unit number (optional; default=unit 0), and volume number (optional; default=volume number 0). Here are some examples:

```
:CS80, 700
:, 700
:,802, 0
:,1400,0,0
```

See MASS STORAGE IS for the complete syntax drawing.

HEWLETT
PACKARD

HP Part Number
E2083-90000

Printed in U.S.A. E0790

E2083-90601 Manufacturing Number