Instrument BASIC Users Handbook

Version 2.0

Agilent Technologies

04155-90150

December 2000

Notice

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

Agilent Technologies Inc. shall not be liable for any errors contained in this document. Agilent MAKES NO WARRANTIES OF ANY KIND WITH REGARD TO THIS DOCUMENT, WHETHER EXPRESS OR IMPLIED. Agilent SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Agilent 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 documnet.

Warrenty Information

A copy of the specific warranty terms applicable to your Agilent 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 purpose 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

December 2000 - First 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 that describes the 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.

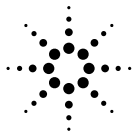
For HP BASIC Programmers

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 specifics about the way HP Instrument BASIC is implemented. If this is your situation, you may want to refer to the following instrument-specific manuals and sections as needed:

- The graphics section of your instrument-specific manual for information on using the display for graphics and text program output.
- Your instrument-specific manual to learn how HP Instrument BASIC interfaces with the host device, (if using an embedded controller) and its external GPIB port.
- 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, consult the manuals on programming and interfacing techniques as needed.

Programming Techniques



Agilent Technologies

December 2000

Contents

1

Manual Organization

Welcome

This purpose of this manual is 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 that came with your instrument that 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 a rudimentary knowledge of programming in BASIC or another language. If you have no programming knowledge, you may 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 an exhaustive 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 program flow and how to 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 techniques 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 tutorial 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, the approach chosen should not present any significant problems. In cases where 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 you 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: END, STOP, and PAUSE.

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

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

Following 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, 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 ... ") that 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	END LOOP	FOR	REPEAT
CASE ELSE	END SELECT	IF	SELECT
ELSE	END WHILE	LOOP	UNTIL
END IF	EXIT IF	NEXT	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:

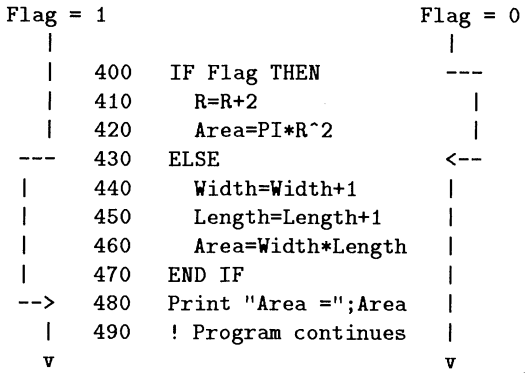
```
100  IF Ph>7.7 THEN
110    PRINT "The value of Ph has been exceeded!"
120    PRINT "Final Ph =";Ph
130    GOSUB Next_tube
140  END IF
150  !  Program continues here
```

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:



HP Instrument BASIC has an IF ... THEN ... ELSE structure that 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 END 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 follows:

```
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 regardless of the number of loop iterations required. The REPEAT ... UNTIL and the WHILE ... END WHILE structures provide this flexibility.

The REPEAT loop and the WHILE loop differ only in their location of the loop exit test. The REPEAT loop has its test at the end of the loop. Therefore, the loop will always be executed once because the condition is not tested until the end of the loop. The WHILE loop has its test at the beginning of the loop, so the test is made before the loop is entered. Therefore, it is possible for a WHILE loop to be skipped entirely.

For example, suppose you wat to print successive powers of two, but want to stop once the value is greater than 1000. Consider the following examples programs:

REPEAT loop

```
10 X=2
20 PRINT X;
30 REPEAT
40   X=X*2
50   PRINT X;
60 UNTIL X>1000
70 END
```

WHILE loop

```
10 X=2
20 PRINT X;
30 WHILE X<1000
40   X=X*2
50   PRINT X;
60 END WHILE
70 END
```

If you ran either of these programs, the results would be:

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

However, if you replace line 10 in each program with

```
10 X=1024
```

then the repeat loop would produce

```
1024  2048
```

whereas the WHILE loop would produce

```
1024
```

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** that 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 that 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 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, 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, 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 mismatched 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 two numeric 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 and Y is INTEGER):

```
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 includes 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 that replaces them in the evaluation of the expression.
- User-defined functions (FNMy_func, FNReturn_val, etc.)—also return a value that 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)
```

To evaluate this expression, it is necessary to have some historical data. We will assume that DEG has been executed earlier, 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 $0 \leq$ variable < 1 then output equals the square root of $(A^2 + B^2)$.
- If variable ≥ 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 that 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 reselected 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), then the specified operation is performed. It is best to restrict bit-oriented binary operations to be 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 find the range of values in a list of variables. 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 that 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 pre-run. 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).

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 decimal 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,

```
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,

```
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 that 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.

4

Numeric Arrays

An array is a multi-dimensioned structure of variables that are given a common name. The array can have one to six dimensions. Each location in an array contains one value, and each value has the characteristics of a single variable, either REAL or INTEGER (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. Arrays are limited to six dimensions, and the subscript range for each dimension must lie between -32767 and 32767. REAL arrays require eight bytes of 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, ALLOCATE, INTEGER or REAL statements. For example:

```
REAL Array_complex(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 (there is one exception to this rule: If you ALLOCATE an array, and then DEALLOCATE it, you can dimension the array again). However, as we explain later in this section, arrays can be REDIMensioned.

When you dimension an array, the system reserves space in internal memory for it. The system also sets up a table which it uses 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 dimensioning a two-dimensional array is as follows:

```
OPTION BASE 0       default numbering of subscripts begins with 0
DIM Array(3,5)      declares elements (0,0) to (3,5)

OPTION BASE 1       default numbering of subscripts begins with 1
Array(2,3)          defines elements (1,1) to (2,3)

OPTION BASE 0           default numbering of subscripts begins with 0
DIM A(1:4,1:4,1:4)      explicitly defines elements (1,1,1) to (4,4,4)
```

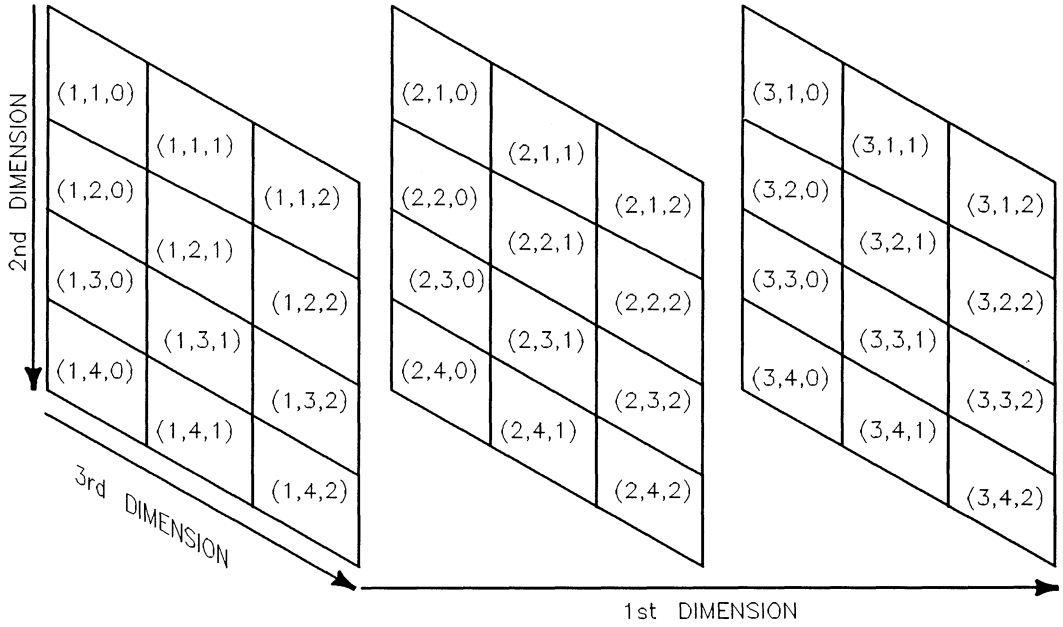
Each context defaults to an option base of 0 (but arrays appearing in COM statements with an (*) keep their original base. However, you can set the option base to 1 using the `OPTION BASE` statement. You can have only one `OPTION BASE` statement in a context, and it must precede all explicit variable declarations.

Some Examples of Arrays

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.

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

```
10   OPTION BASE 1
20   DIM A(3,4,0:2)
```



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

```
10   OPTION BASE 1
20   COM B(1:5,2:6)
```

Two-Dimensional REAL ARRAY

(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
(3,2)	(3,3)	(3,4)	(3,5)	(3,6)
(4,2)	(4,3)	(4,4)	(4,5)	(4,6)
(5,2)	(5,3)	(5,4)	(5,5)	(5,6)

Dimension	Size	Lower Bound	Upper Bound
1st	5	1	5
2nd	5	2	6

```
10   OPTION BASE 1
20   ALLOCATE INTEGER C(2:4,-2:2)
```

A Dynamically Allocated, 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

Note

Throughout this chapter we will be using DIM statements without specifying what the current option base setting is. Unless explicitly specified otherwise, all examples in this chapter use option base 1.

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(5,20,10,100)
```

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.

Problems with Implicit Dimensioning

In any context, an array must have a dimensioned size. It may be explicitly dimensioned through COM, INTEGER, REAL, or ALLOCATE. It can also be implicitly dimensioned through a subscripted reference to it in a program statement *other than* a MAT or a REDIM statement. MAT and REDIM statements cannot be used to implicitly dimension an array.

Finding Out the Dimensions of an Array

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

```
10 OPTION BASE 0
20 DIM F(1,4,-1:2)
30 PRINT RANK (F)
40 END
```

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, indicating this dimension has not been defined as part of F.

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

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

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.

```
A(3,4)=13
```
the element in row 3, column 4, has the value 13

Extracting Single Values From Arrays

As with entering values into arrays, there are a number of ways to extract values as well. To extract the value of a particular element, simply specify the element's subscripts.

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

BASIC automatically converts variable types. For example, if you assign an element from a REAL array to an INTEGER variable, the system will round the REAL to an integer.

Filling Arrays

This section discusses three methods for filling an entire array:

- Assigning every element the same value
- Using READ to fill an entire array
- Copying arrays into other arrays

Assigning Every Element in an Array the Same Value

For some applications, you may want to initialize every element in an array to some particular value. You can do this by assigning a value to the array name. However, you must precede the assignment with the MAT keyword.

```
MAT A= (10)
```

Note that the numeric expression on the right-hand side of the assignment must be enclosed in parentheses and that this expression may be `INTEGER` or `REAL`.

Using the READ Statement to Fill an Entire Array

You can assign values to an array using READ and DATA. DATA allows you to create a stream of data items, and READ enables you to enter the data stream into an array.

```
110   DIM A(3,3)
120   DATA -4,36,2.3,5,89,17,-6,-12,42
130   READ A(*)
140   PRINT USING "3(3DD.DD,3DD.DD,3DD.DD,/)";A(*)
150   END
```

The asterisk in line 140 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 previous program produces the following results:

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

Copying Entire Arrays into Other Arrays

Another way to fill an array is to copy all elements from one array into another (copying sub-sets of arrays is discussed in the subsequent section of "Numeric Arrays" called "Copying Subarrays"). Suppose, for example, that you have the two arrays A and B shown below.

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 5 \\ 8 & 2 \\ 1 & 7 \end{pmatrix}$$

Note that A is a 3×3 array which is filled entirely with 0's, while B is a 3×2 array filled with non-zero values. To copy B to A, we would execute:

```
MAT A= B
```

Again, you must precede the assignment with MAT. The system will automatically redimension the "result array" (the one on the left-hand side of the assignment) so that it is the same size as the "operand array" (the one on the right side of the equation.) There are two restrictions on redimensioning an array.

- The two arrays must have the same rank (e.g., the same number of dimensions.)
- The dimensioned size of the result array must be at least as large as the current size of the operand array.

If BASIC cannot redimension the result array to the proper size, it returns an error.

Automatic redimensioning of an array will not affect the lower bounds, only the upper bounds. So the BASE values of each dimension of the result array will remain the same. Also keep in mind that the size restriction applies to the *dimensioned* size of the result array and the *current* size of the operand array. Suppose we dimension arrays A, B and C to the following sizes:

```
10   OPTION BASE 1
20   DIM A(3,3),B(2,2),C(2,4)
     .
     .
```

We can execute,

```
MAT A= B
```

since A is dimensioned to 9 elements and B is only 4 elements. The copy automatically redimensions A to a 2×2 array. Nevertheless, we can still execute:

```
MAT A= C
```

This works because the nine elements originally reserved for A remain available until the program is scratched. A now becomes a 2×4 matrix. After MAT A= C, we could not execute:

```
MAT B= A
```

or

```
MAT B= C
```

since in each of these cases, we are trying to copy a larger array into a smaller one. But we could execute

```
MAT C= A
```

after the original MAT A= B assignment, since C's dimensioned size (8) is larger than A's current size (4).

Printing Arrays

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(*);
```

would display 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.)

Examples of Formatting Arrays for Display

This section provides two subprograms which have both been given the name Printmat. The first subprogram is used to display a two-dimensional INTEGER array and the second subprogram is used to display a three-dimensional INTEGER array.

To display a two dimensional array, you can use the following subprogram:

```
240   SUB Printmat(INTEGER Array(*))
250   OPTION BASE 1
260   FOR Row=BASE(Array,1) TO SIZE(Array,1)+BASE(Array,1)-1
270     FOR Column=BASE(Array,2) TO SIZE(Array,2)+BASE(Array,2)-1
280       PRINT USING "DDDD,XX,#";Array(Row,Column)
290     NEXT Column
300     PRINT
310   NEXT Row
320   SUBEND
```

Assuming that you intended to display a 5×5 array, your results should look similar to this:

```
11   12   13   14   15
21   22   23   24   25
31   32   33   34   35
41   42   43   44   45
51   52   53   54   55
```

If you were to expand the above subprogram to print three-dimensional INTEGER arrays, your subprogram would be similar to the following:

```
250   SUB Printmat(INTEGER Array(*))
260     OPTION BASE 1
270     FOR Zplane=BASE(Array,3) TO SIZE(Array,3)+BASE(Array,3)-1
280       PRINT TAB(6),"Plane ";Zplane
290       PRINT
300       FOR Yplane=BASE(Array,2) TO SIZE(Array,2)+BASE(Array,2)-1
310         FOR Xplane=BASE(Array,1) TO SIZE(Array,1)+BASE(Array,1)-1
320           PRINT USING "DDDD,XX,#";Array(Zplane,Yplane,Xplane)
330         NEXT Xplane
340         PRINT
350       NEXT Yplane
360       PRINT
370     NEXT Zplane
380   SUBEND
```

If you had a three dimensional array with the following dimensions:

```
DIM Array1(3,3,3)
```

filled with all 3's, the results from executing the above subprogram would be as follows:

```
   Plane 1
3     3     3
3     3     3
3     3     3

   Plane 2
3     3     3
3     3     3
3     3     3

   Plane 3
3     3     3
3     3     3
3     3     3
```

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, we would write:

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

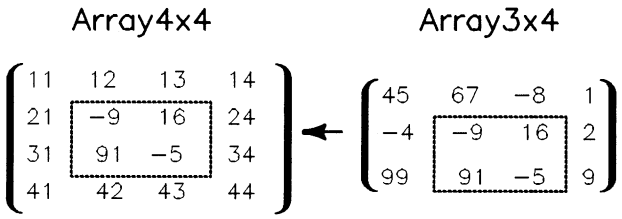
Copying Subarrays

An earlier section discussed copying the contents of an entire array into another entire array.

```
MAT Array55= Array33
```

Each element of `Array33` is copied into the corresponding element of `Array55` which is redimensioned if necessary.

Now suppose you would like to copy a portion of one array and place it in a special location within another array. This process is called copying subarrays.



Copying a Subarray into Another Subarray

Topics discussed in this section are:

- Subarray specifier
- Copying a subarray into an array
- Copying an array into a subarray
- Copying a subarray into a subarray
- Copying a portion of an array into itself
- Rules for copying subarrays

Dimensions for the arrays covered in the above topics will assume an option base of 1 (`OPTION BASE 1`) unless stated differently.

Subarray Specifier

A subarray is a subset of an array (an array within an array). A subarray is indicated after the array name as follows:

```
Array_name(subarray_specifier)

String_array$(subarray_specifier)
```

The above subarray could take on many "sizes" and "shapes" depending on what you used as dimensions for the array and the values used in the *subarray_specifier*. Note that "size" refers

to the number of elements in the subarray and "shape" refers to the number of dimensions and elements in each dimension, respectively [e.g. both of these subscript specifiers have the same shape: `(-2:1,-1:10)` and `(1:4,9:20)`]. Before looking at ways you can express a subarray, let's learn a few terms related to the subarray specifier.

subscript range — is used to specify a set of elements starting with a beginning element position and ending with a final element position. For example, `5:8` represents a range of four elements starting with element 5 and ending at element 8.

subscript expression — is an expression which reduces the **RANK** of the subarray. For example if you wanted to select a one-element subarray from a two-dimensional array which is located in the 2nd row and 3rd column, you would use the following subarray specifier: (2,3:3). The subscript expression in this subarray specifier is 2 which restricts the subarray to row 2 of the array.

default range — is denoted by an asterisk (i.e. `(1,*)`) and represents all of the elements in a dimension from the dimension's lower bound to its upper bound. For example, suppose you wanted to copy the entire first column of a two dimensional array, you would use the following subarray specifier: `(*,1)`, where `*` represents all the rows in the array and `1` represents *only* the first column.

Some examples of subarray specifiers are as follows:

`(1,*)` — a subscript expression and a default range which designate the first row of a two-dimensional array.

`(1:2)` — a given subscript range which represents the first two elements of a one-dimensional array.

`(*,-1:2)` — a default range and subscript range which represents all of the elements in the first four columns of a two-dimensional array (base of 2nd dimension assumed to be -1).

`(3,1:2)` — a subscript expression and subscript range which represent the first two elements in the third row of a two-dimensional array.

`(1,*,*)` — a subscript expression and two default ranges which represent a plane consisting of all the rows and columns of the first plane in the first-dimension.

`(1,1:2,*)` — a subscript expression, subscript range and default range which represent the first two rows in the first plane of the first-dimension.

`(1,2,*)` — two subscript expressions and a default range which represent the entire second row in the first plane of the first-dimension.

`(1:2,3:4)` — two subscript ranges which represent elements located in the third and fourth columns of the first and second rows of a two-dimensional array.

For more information on string arrays, see the "String Manipulation" chapter found in this manual.

Copying an Array into a Subarray

In order to copy a source array into a subarray of a destination array, the destination array's subarray must have the same size and shape as the source array.

A destination and source array are dimensioned as follows:

```
100  OPTION BASE 1
110  DIM Des_array(-3:1,5),Sor_array(2,3)
```

Suppose these arrays contain the following **INTEGER** values:

Des_array

11	12	13	14	15
21	22	23	24	25
31	32	33	34	35
41	42	43	44	45
51	52	53	54	55

Sor_array

11	12	13
21	22	23

you would copy the source array `Sor_array` into a subarray of the destination array `Des_array` by using program line 190 given below:

```
190 MAT Des_array(-1:0,2:4)= Sor_array
```

`Des_array` would have the following values in it as the result of executing the above statement:

Des_array

11	12	13	14	15
21	22	23	24	25
31	11	12	13	35
41	21	22	23	45
51	52	53	54	55

Copying a Subarray into an Array

A subarray can be copied into an array as long as the array can be re-dimensioned to be the size and shape of the subarray specifier.

A destination and source array are dimensioned as follows:

```
100  OPTION BASE 1
110  DIM Des_array(8),Sor_array(-5:4)
```

Suppose both of these one-dimensional arrays contain the following values:

Des_array

-1	14	8	4	98	43	90	-3

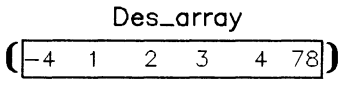
Sor_array

-11	-4	1	2	3	4	78	100	8	18

you would copy a subarray of the source array (`Sor_array`) into a destination array (`Des_array`) by using program line 190 given below:

```
190 MAT Des_array= Sor_array(-4:1)
```

`Des_array` will be re-dimensioned to have six elements with the following values in it as a result of executing the above statement.



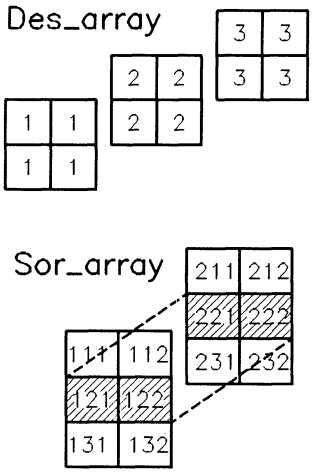
Copying a Subarray into Another Subarray

Subarray specifiers must have the same size and shape when you are copying one subarray into another.

A destination and source array are dimensioned as follows:

```
100   OPTION BASE 1
110   DIM Des_array(3,2,2),Sor_array(2,3,2)
120   .
130   .
```

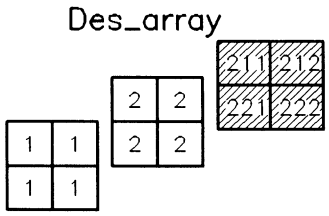
Suppose these three-dimensional arrays contain the following values:



in order to properly copy a source subarray (`Sor_array(*,2,*)`) into a destination subarray using asterisks to represent the ranges of dimensions, you would use line 190 given below:

```
190   MAT Des_array(3,*,*)= Sor_array(*,2,*)
```

A three dimensional array with the following values in it would be the result of executing the above statement.



Copying a Portion of an Array into Itself

If you are going to copy a subarray of an array into another portion of the same array, the two subarray locations should not overlap (e.g., `MAT Array(2:4,1:3)= Array(1:3,2:4)` is an improper assignment). No error message will result from this misuse, but the result is undefined.

A destination and source array are dimensioned as follows:

```
100   OPTION BASE 1
110   DIM Array(4,4)
```

Suppose this two dimensional array contains the following values:

Array

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \\ 41 & 42 & 43 & 44 \end{pmatrix}$$

to copy a slice of this array into another portion of the same array, you would use program line 190 given below:

```
190   MAT Array(3:4,1:2)= Array(1:2,3:4)
```

Array will have the following values in it as a result of executing the above statement.

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 13 & 14 & 33 & 34 \\ 23 & 24 & 43 & 44 \end{pmatrix}$$

Note that you *cannot* copy a subarray into the array it is part of with an implied re-dimensioning of the array. A statement of the form:

```
MAT Array= Array(subarray_specifier)
```

will always generate a run-time error.

Rules for Copying Subarrays

This section should help limit the number of syntax and runtime errors you could make when copying subarrays. A previous section titled "Subarray Specifier" provided you with examples of the correct way of writing subarray specifiers for copying subarrays. In this section, you will be given rules to things you should not do when copying subarrays. The rules are as follows:

- Subarray specifiers *must not* contain all subscript expressions (i.e. `(1,2,3)` is not allowed, it will produce a syntax error). This rule applies to all subscript specifiers.
- Subarray specifiers *must not* contain all asterisks (*) or default ranges (i.e. `(*,*,*)` is not allowed, it will produce a syntax error). This rule applies to all subscript specifiers.
- If two subarrays are given in a `MAT` statement, there *must be* the same number of ranges in each subarray specifier. For example:

  ```
  MAT Des_array1(1:10,2:3)= Sor_array(5:14,*,3)
  ```

  is the correct way of copying a subarray into another subarray provided the default range given in the source array (`Sor_array`) has only two elements in it. Note that the source array is a three-dimensional array. However, it still meets the criteria of having the same number of ranges as the destination array because two of its entries are ranges and one is an expression.
- If two subarrays are given in a `MAT` statement, the subscript ranges in the source array *must be* the same shape as the subscript ranges in the destination array. For example, the following example is *legal*:

  ```
  MAT Des_array(1:5,0:1)= Sor_array(3,1:5,6:7)
  ```

  however, the following example is *not legal*:

  ```
  MAT Des_array(0:1,1:5)= Sor_array(1:5,0:1)
  ```

  because both of its subarray specifiers do not have the same shape (i.e. the rows and columns in the destination array do not match the rows and columns in the source array).

Redimensioning Arrays

In our discussion of copying arrays we saw that the system automatically redimensions an array if necessary. BASIC also allows you to explicitly redimension an array with the `REDIM` statement. As with automatic redimensioning, the following two rules apply to all `REDIM` statements:

- A `REDIM`ed array must maintain the same number of dimensions.
- You cannot `REDIM` an array so that it contains more elements than it was originally dimensioned to hold.

Suppose `A` is the 3×3 array shown below.

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

To redimension it to a 2×4 array, you would execute:

```
REDIM A(2,4)
```

The new array now looks like the figure below:

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

Note that it retains the values of the elements, though not necessarily in the same locations. For instance, A(2,1) in the original array was 4, whereas in the redimensioned array it equals 5. For example, if we REDIMed A again, this time to a 2×2 array, we would get:

```
REDIM A(0:1,0:1)
```

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

We could then initialize all elements to 0:

```
MAT A= (0)
```

$$A = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

It is also important to realize that elements that are out of range in the REDIMed array still retain their values. The fifth through ninth elements in A still equal 5 through 9 even though they are now inaccessible. If we REDIM A back to a 3×3 array, these values will reappear. For example:

```
REDIM A(3,3)
```

results in:

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

One of the major strengths of the REDIM statement is that it allows you to use variables for the subscript ranges: this is not allowed when you originally dimension an array. In effect, this enables you to dynamically dimension arrays. This should not be confused with the ALLOCATE statement which allows you to dynamically reserve memory for arrays. In the example below, for instance, we enter the dimensions from the keyboard.

```
10   OPTION BASE 1
20   INTEGER A(100,100)
30   INPUT "Enter lower and upper bounds of dimensions",
     Low1,Up1,Low2,Up2
40   IF (Up1-Low1+1)*(Up2-Low2+1)>10000 THEN Too_big
50   REDIM A(Low1:Up1,Low2:Up2)
```

Line 40 tests to see whether the new dimensions are too big. If so, program control is passed to a line labelled "Too_big". If line 40 were not present, the REDIM statement would return an error if the dimensions were too large.

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 with a length that exceed 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.
`ALLOCATE Search$[Length]`	Dynamic length allocation.

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"
```

is a true relationship 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 example,

```
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 example,

```
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
```

It's 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 with a value 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 example,

```
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. For example,

```
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. For example,

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

prints

```
* ** ** ** ** ** ** ** ** ** *
```

Trimming a String

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

```
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.

For example,

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

prints

```
1.6734464E+7
170
00000000144
FFFF
```

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 constructed 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, two instances that require the use of CALL when invoking a subprogram.

CALL is required

1. if the subprogram is called after the THEN keyword in an IF statement
2. 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 that 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 that 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 that 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)
```

Here is a general guideline for taking a set of data and analyzing it to generate a single value, then implementing the subprogram as a function. On the other hand, if you actually want to 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 that returns a string. There are two primary differences: the first is that a $ must be added to the name of a function that 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"
```

(Continued)

```
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 that 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 that 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 that 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 that 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 was enclosed in parentheses, it would be passed by value. Notice that if it was 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 name that 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 that are accessible to any context with 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 that 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 that 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 one of the following takes place:
  - SCRATCH A or SCRATCH C is executed
  - a statement declaring a COM block is modified by the user
  - a new program is brought into memory using the GET command that doesn't match the declaration of a given COM block, or that 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 that has 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 that share their own labeled COM blocks, it is wise to collect all the COM declarations together in one subprogram. This makes 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 that overlay each other using GET statements, if you remember a few rules:

1. COM blocks that 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 that are not declared in the new program (the one being brought in with the GET) are destroyed.
3. Any COM blocks that are named identically, but that 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, but by 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 that 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 that is distinct from a main program and all other subprograms. In between the time a CALL statement is executed (or an FN name is used) and the time the first statement in the subprogram is 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 that points to the next item in the current DATA block that 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 that takes place within the subprogram (or any of the subprograms that 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 that 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 reenabling his keys and their associated labels after calling a subprogram that 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 the ON part of an ON ... RECOVER statement (for example a softkey or an external interrupt from the SRQ line on an GPIB), 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.

Loading Subprograms

If you already have subprograms stored in PROG file(s), there are several options to choose from in loading them into memory:

- If you want to load a specific subprogram from a PROG file, you would use something like this:

```
LOADSUB Sub_name FROM "File"
```

- If you want to load all the subprograms from a specific PROG file, you would use the LOADSUB ALL FROM statement.

```
LOADSUB ALL FROM "File"
```

- And, if you wanted to see which subprograms are still missing or load all those still needed, you would use something like this:

```
LOADSUB FROM "File"
```

  (Note that this statement is *not* programmable; that is, it cannot appear in a program line.)

You can also use INMEM to determine if a subprogram is already loaded. For example:

```
IF NOT INMEM ("Mysub")
THEN LOADSUB ALL FROM "MYSUBS"
```

Loading Subprograms One at a Time

Suppose your program has several options to select from, and each one needs many subprograms and much data. All the options, however, are mutually exclusive; that is, whichever option you choose, it does not need anything that the other options use. This means that you can clean up everything you've used when you are finished with that option.

If all of your subprograms can be put into one file, you can selectively retrieve them as needed with this sort of statement:

```
LOADSUB Subprog_1 FROM "SUBFILE"
LOADSUB Subprog_2 FROM "SUBFILE"
LOADSUB FNNumeric_fn FROM "SUBFILE"
LOADSUB FNString_function$ FROM "SUBFILE"
```

Note that only one subprogram per line can be loaded with this form of LOADSUB. If, for any program option, you need so many subprograms that this method would be cumbersome, you could use the following form of the command.

Loading Several Subprograms at Once

For this method, you store *all* the subprograms needed for each option in its own file. Then, when the program's user selects Program Option 1, you could have this line of code execute:

```
LOADSUB ALL FROM "OPT1SUBFL"
```

and if the user selects Option 2,

```
LOADSUB ALL FROM "OP2SUBFL"
```

and so forth.

There is one other form of LOADSUB, but it cannot be used programmatically. This is covered next.

Loading Subprograms Prior to Execution

In the LOADSUB FROM form, neither ALL nor a subprogram name is specified in the command. This is used prior to program execution. It looks through the program in memory, notes which subprograms are needed (referenced) but not loaded, goes to the specified file and attempts to load all such subprograms. If the subprograms are found in the file, they are loaded into memory; if they are not, an error message is displayed and a list of the subprograms still needed but not found in the file is printed.

This can be handy in two ways. The first and obvious way is that subprograms can be loaded quickly. The other way is this: Type a LOADSUB FROM command where the file name is a file in which you *know* there are none of the subprograms you need (perhaps a null PROG file). Of course, no subprograms will be loaded, but *a list of those yet undefined will be printed.*

Any COM blocks declared in subprograms brought into memory with a LOADSUB by a running program must already have been declared. LOADSUB does not allow new COM blocks to be added to the ones already in memory. Furthermore, any COM blocks in the subprograms brought in must match a COM block in memory in both the number and type of the variables. Otherwise, an error occurs.

Note

If a main program is in a file referenced by a LOADSUB, it will *not be loaded*; only subprograms can be loaded with LOADSUB. Main programs are loaded with the LOAD command.

With all this talk of loading subprograms from files, one question arises: How do you get the subprograms *in* the file? Easily: type in the subprograms you want to be in one file, and then STORE them with the desired file name. You must use STORE and not SAVE, because the LOADSUB looks for a PROG-type file. If you can't type in your subprograms error-free the first time (and who can?), you can type in your program with all the subprograms it needs and debug them. *After storing everything in a file for safekeeping*, delete what you do *not* want in the file, and STORE everything else in the subprogram file from which you will later do a LOADSUB. In this way, you know the subprograms will work when you load them.

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 exceptions to this rule are the comment statements "REM" and "!". They 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 $N \times (N-1)!$ where $0!=1$ by definition. Thus, N! is simply the product of all the whole numbers from 1 through N inclusive. A recursive function that 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 Agilent 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 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
```

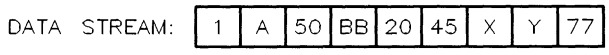
Note that with this type of storage, the values assigned to the corresponding variables are *not* kept with the program when it is stored; they must be entered each time the program is run. This type of data storage can be used when the data are to be checked or modified by the user each time the program 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
```



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 that 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 prerun, 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
```

prints

```
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 following example, 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
```

prints

```
 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 that 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*, that 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.

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 **mass storage unit specifier (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) that 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, the 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"
```

opens 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 following two statements 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 that define the way data is to be read and written. There are two attributes that 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 that provide control of such functions as changing end-of-line (EOL) sequences. (See "ASSIGN" in *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 reenter 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 reassigned. You can reassign 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.

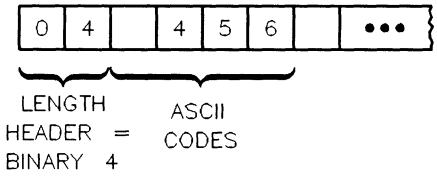
Example of ASCII File I/O

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.

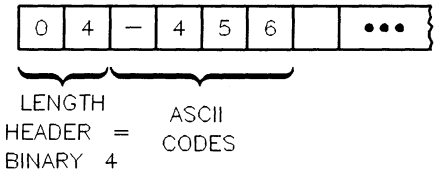
```
100  REAL Real_array1(1:50,1:25),Real_array2(1:50,1:25)
110  INTEGER Integer_var
120  DIM String$[100]
       .
       .
390  ! Specify "default" mass storage device.
400  MASS STORAGE IS ":,700,1"
410  !
420  ! Create ASCII data file with 10 sectors
430  ! on the "default" mass storage device.
440  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

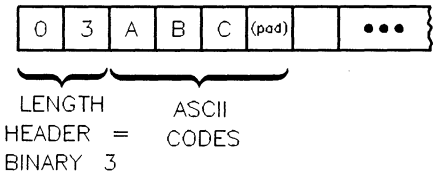
In an ASCII file, every data item, whether string or numeric, is represented by ASCII characters; one byte represents one ASCII character. Each data item is preceded by a two-byte length header that indicates how many ASCII characters are in the item. However, there is no "type" field for each item; 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:



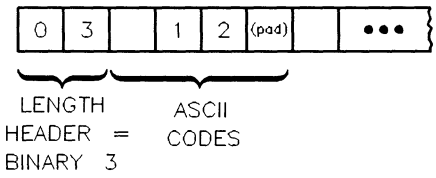
Note that there is a space at the beginning of the data item. This space signifies that the number is positive. If a number is negative, a minus sign precedes the number. For instance, the number −456, would be stored as follows:



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:

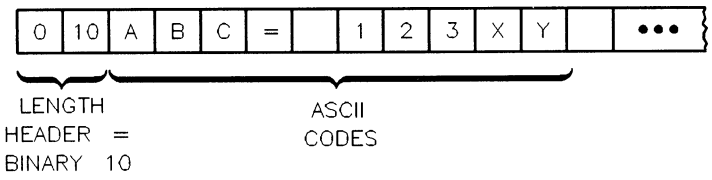


There is often a relatively large amount of overhead for numeric data items. For instance, to store the integer 12 in an ASCII file requires the following six bytes:



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 important. However, ASCII files are interchangeable with many other Agilent 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 that 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, that 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
```

prints

```
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 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 that have the FORMAT OFF attribute by using a 16-bit, two's-complement notation that 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):

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

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

Every character in a string is represented by one byte that 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:

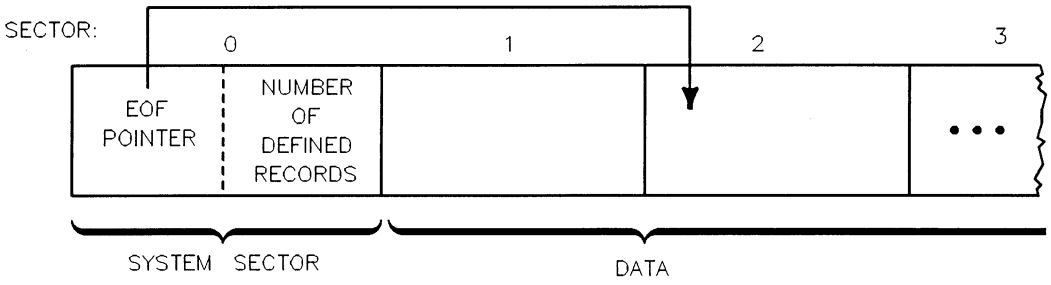
- The internal representation (FORMAT OFF) of strings is slightly different:
  - HP-UX and DOS FORMAT OFF strings have no length header; instead, they are terminated by a null character, CHR$(0).
  - BDAT FORMAT OFF strings have a 4-byte length header;
- HP-UX and DOS files have a *fixed record length 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 on Files, 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

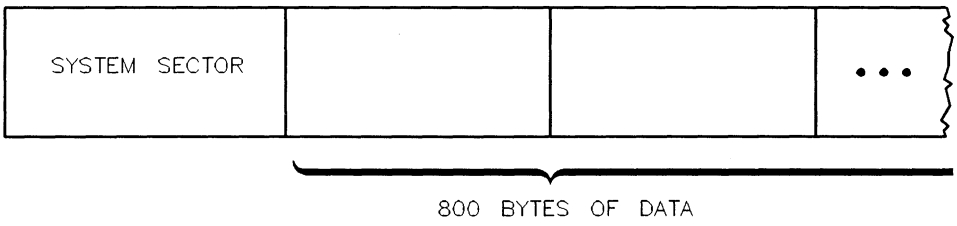
```
CREATE "Odder",3.49,28.3
```

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

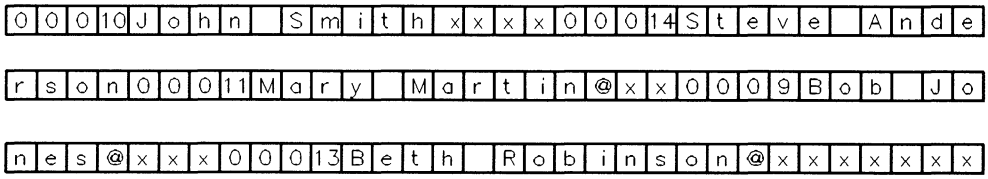
BDAT file. The first, and easiest, is to output each string in random mode. In other words, select a record length that will hold the longest string, then write each string into its own record. Suppose, for example, that you wanted to OUTPUT the following 5 names into a BDAT file and be able to access 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.



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, 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, 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, make sure that access modes agree. If you wrote data serially, you should read it serially; if you wrote it randomly, you should read it randomly. There are a few exceptions to this rule that 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 "Entering Data" in *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
```

prints

```
5 squared equals 25.
```

Note that we re-ASSIGNed the I/O path in line 70. This was done to reposition 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
```

prints

```
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 repositions 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
```

prints

```
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 redirected 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

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 GPIB 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. For some printers, there may 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
```

prints

```
12345678901234567890123456789012345678901234567890
 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
```

prints

```
12345678901234567890123456789012345678901234567890
 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(*);
```

prints

```
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
```

prints

```
12345678901234567890123456789012345
                         -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 reused for the subsequent items.

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

prints

```
12345678901234567890123456789012345
    3.98    5.95   27.50  139.95
```

The image is reused 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 redefining the EOL sequence.
/	Send a carriage return and a line feed.
@	Send a form feed.
+	Send a carriage return as the EOL sequence. (Requires IO)
–	Send a line feed as the EOL sequence. (Requires IO)

For example:

`PRINT USING "@,#"` outputs a form feed.

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

Special Considerations

If nothing prints, see 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 occurrence. One such plan 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 that 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 reattempt 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 that returns the error number that 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 that returns the *line number* of the program line where 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 that 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 that 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 that 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 that was called by the environment that 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*. For example:

```
1140  DISP ERRL(710)

910   IF ERRL(Compute) THEN Fix_compute
```

ERRM$ is a string function that returns the text of the error that 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 reexecute 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 reestablish the original error service routine after correcting any

errors (by reexecuting 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 reexecute 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 setup 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 that 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		
COPYLINES	Copies contiguous program lines from one location to another.	Full support.
DELSUB	Deletes one or more subprograms or user-defined functions from memory.	Full support.
INDENT	Indents program lines in the edit window to reflect the programs structure and nesting.	Full support.
LIST	Lists program lines to system printer.	No support for softkey listing.
MOVELINES	Moves contiguous program lines from one location to another.	Full support.
REM and !	Allows comments on program lines.	Full support.
SECURE	Protects program lines so they cannot be listed.	Full support.
Debugging		
CAUSE ERROR	Simulates the occurrence of an error of the specified number.	Full support.
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 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		
ALLOCATE	Dynamically allocates memory for arrays and string variables during program execution.	No support for COMPLEX.
COM	Dimensions and reserves memory for variables in a common area for access by more than one context.	No support for BUFFER, COMPLEX, LOAD, or subarrays.
DEALLOCATE	Deallocates memory space reserved by the ALLOCATE statement.	No support for COMPLEX.
DIM	Dimensions and reserves memory for REAL numeric arrays and strings.	No support for BUFFER, COMPLEX, or subarrays.
INTEGER	Dimensions and reserves memory for INTEGER variables and arrays.	No support for BUFFER or subarrays.
OPTION BASE	Specifies default lower bound of arrays.	Full support.

Program	HP BASIC Function	HP Instrument BASIC
Memory Allocation (continued)		
REAL	Dimensions and reserves memory for full-precision (REAL) variables and arrays.	No support for BUFFER or subarrays.
REDIM	Changes the subscript range of previously dimensioned arrays.	No support for BUFFER.
SCRATCH	Erases all or portions of memory.	ALL and COM are supported.
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.
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.

Program	HP BASIC Function	HP Instrument BASIC
General Math (continued)		
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 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.
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.

Program	HP BASIC Function	HP Instrument BASIC
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.
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.

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.
MAT	Performs a variety of operations on matrices and other numeric and string arrays.	No support for COMPLEX MAT, SEARCH, MAP, or SORT
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
Mass Storage		
ASSIGN	Assigns an I/O path name and attributes to a file.	No support for BUFFER, BYTE, WORD, CONVERT, RETURN, PARITY, or DELAY. 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 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.	Full support.
CREATE ASCII	Creates an ASCII-type file on the mass storage media.	Full support.
CREATE BDAT	Creates an BDAT-type file on the mass storage media.	Full support.
CREATE DIR	Creates an HFS or MS-DOS-type directory on the mass storage media.	Full support.
DELSUB	Deletes one or more subprograms or user-defined functions from memory.	Full support.
GET	Reads an ASCII file into memory as a program.	Full support.
INITIALIZE	Formats a mass storage media and places a LIF or DOS directory on the media.	No support for EPROM.
LOAD	Loads STOREd programs into memory.	No support for BIN, or KEY.
LOADSUB	Loads HP Instrument BASIC subprograms from a STOREd program into memory.	Full support.
MASS STORAGE IS/ MSI	Specifies the default mass storage device.	No support for DCOMM, BUBBLE, or EPROM.
PURGE	Deletes a file entry from the directory.	Full support.
RENAME	Changes a file's name.	Full support.
SAVE/RE-SAVE	Creates an ASCII file and copies program lines from memory into the file.	Full support.

Program	HP BASIC Function	HP Instrument BASIC
Mass Storage (continued)		
RE-STORE	Writes the current HP Instrument BASIC program to the specified file in a special compace, fast-loading format.	No support for KEY.
STORE	Writes the program currently in memory to a PROG file in a special binary form.	Full support.
Program Control		
CALL	Transfers program execution to a specified subprogram and passes parameters.	Full support.
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 that 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.
ON	Transfers program control to one of several destinations.	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 event-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 card 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, or WRITEIO.
SYSTEM PRIORITY	Sets a minimum level of system priority for event-initiated branches.	Full support.
Graphics Control		
ALPHA ON/OFF	ON shows the alpha window; OFF clears the alpha window	Full support.
AREA	Sets the color used to shade graphical regions subsequently created by various graphics plotting commands.	Full support.
CLIP	Defines, enables, or disables the soft-clip limits for subsequent graphics output.	Full support.
GCLEAR	Clears the graphics area.	No support for external plotter or Multi-Plane displays.

Program	HP BASIC Function	HP Instrument BASIC
Graphics Control (continued)		
GESCAPE	Used for communicating device-dependent graphics information. Type, size, and shape of the arrays must be appropriate for the requested operation.	Full support.
GINIT	Establishes a set of default values for system variables affecting graphics operation.	Full support.
GLOAD	Loads the contents of an INTEGER array into the graphics window.	Full support.
GRAPHICS	Shows or hides the graphics window.	Full support.
GSTORE	Stores the current contents of the graphics window in an integer array.	No support for source devices.
MERGE ALPHA	Performs a no-op which makes it compatible with HP-UX RMB.	Dedicated to RMB-UX.
PLOTTER IS	Determines whether graphics colors operate in the color-mapped or non-color-mapped mode.	No hard-copy device, clip limits, or file support.
RATIO	Returns the ratio of the width (in pixels) to the height (in pixels) of the graph window.	Full support.
SEPARATE ALPHA	Compatible with HP-UX RMB.	Dedicated to RMB-UX.
SET PEN	Assigns a color to graphics pen(s).	Full support.
SHOW	Defines an isotropic current unit-of-measure for graphics operations.	Full support.
VIEWPORT	Defines an area (in GDUs) onto which WINDOW and SHOW statements are mapped.	Full support.
WHERE	Returns the current logical position of the graphics pen.	Full support.
WINDOW	Define an anisotropic current unit-of-measure for graphics operations.	Full support.
Graphics Plotting		
AREA	Sets the color used to shade graphical regions subsequently created by various graphics plotting commands.	Full support.
DRAW	Draws a line to a specified point.	No support for PIVOT.
GLOAD	Loads the contents of an INTEGER array into the graphics window.	Full support.
GSTORE	Stores the current contents of the graphics window in an integer array.	No support for source devices.

Program	HP BASIC Function	HP Instrument BASIC
Graphics Plotting (continued)		
IDRAW	Draws a line from the current position to a position calculated by adding the X and Y displacements to the current pen position.	Full support.
IMOVE	Moves the graphics pen an incremental distance from the current position without drawing a line.	Full support.
IPLOT	Moves the graphics pen an incremental distance from the current position. Plotting action is determined by the current line type and the optional pen control parameter.	Full support.
LINE TYPE	Selects the line type (solid or dashed) for all subsequent lines	Full support.
MOVE	Updates the logical pen position.	No support for PIVOT.
PDIR	Specifies the rotation angle at which the output from IPLOT, RPLOT, POLYGON, POLYLINE, and RECTANGLE is drawn.	Full support.
PEN	Selects the pen number on plotting device.	Full support.
PENUP	Lifts the pen on the current plotting device.	Full support.
PIVOT	Specifies a rotation of coordinates which is applied to all drawn lines, but not to labels or axes.	Full support.
PLOT	Moves the graphics pen from the current position to the specified X and Y coordinates.	Full support.
POLYGON	Draws all or part of a closed, regular polygon.	Full support.
POLYLINE	Draws all or part of an open, regular polygon.	Full support.
RECTANGLE	Draws a rectangle.	Full support.
RPLOT	Moves the pen from the current pen position to the specified relative X and Y position.	Full support.
RPEN	Assigns a color to one or more graphics pens.	Full support.
WHERE	Returns the current logical position of the graphics pen.	Full support.

Program	HP BASIC Function	HP Instrument BASIC
Graphics Axes and Labeling		
AXES	Draws a pair of axes with optional, equally spaced tick marks.	Full support.
CSIZE	Sets the height and aspect ratio (width:height) of the character cell used by LABEL.	Full support.
FRAME	Draws a frame around the current graphics clipping area using the current pen number and line type.	Full support.
GRID	Draws a full grid pattern.	Full support.
LORG	Specifies the relative origin of labels with respect to the current pen position.	Full support.
LABEL	Draws text labels with the graphics pen at the pen's current position.	Full support.
LDIR	Defines the angles at which labels are drawn.	Full support.
GPIB 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.

Program	HP BASIC Function	HP Instrument BASIC
Clock and Calendar		
DATE	Converts a formatted date string into a numeric value in seconds.	Full support.
DATE$	Formats a number of seconds into a string representing the formatted date (DD MMM YYYY).	Full support.
TIME	Converts a formatted time-of-day string into number of seconds past midnight.	Full support.
TIME$	Converts the number of seconds past midnight into a string representing the formatted time of day (HH:MM:SS).	Full support.
SET TIME	Resets the time-of-day given by the real-time clock.	Full support.
SET TIMEDATE	Resets the absolute seconds (time and day) given by the real-time clock.	Full support.
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 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.
DATA	Specifies data accessible via READ statements.	Full support.
DISP	Outputs items to the CRT display line.	No support for COMPLEX.
DUMP	DUMP ALPHA copies the contents of the alphanumeric display to the default printer specified in the Windows Control Panel. DUMP GRAPHICS copies the contents of the graphics display to the default printer specified in the Windows Control Panel.	No support for source or destination devices.
ENTER	Inputs data from a device, file or string to a list of variables.	No support for COMPLEX, BUFFER, TRANSFER, or CRT as source.

Program	HP BASIC Function	HP Instrument BASIC
General Device Input/Output (continued)		
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, or TRANSFER .
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.
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.
Display and Keyboard Control		
ALPHA ON/OFF	ON shows the alpha window; OFF clears the alpha window	Full support.
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.
SET ALPHA MASK	Specifies which plane(s) can be modified by alpha display operations.	Full support

Program	HP BASIC Function	HP Instrument BASIC
Array Operations		
BASE	Returns the lower bound of a dimension of an array.	Full support.
DET	Returns the lower bound of a dimension of an array.	Full support, except COMPLEX.
DOT	Returns the lower bound of a dimension of an array.	Full support, except COMPLEX.
MAT	Performs a variety of operations on matrices and other numeric and string arrays.	Full support, except COMPLEX, MAT SORT, and MAT SEARCH.
MAT REORDER	Reorders elements in an array according to the subscript list in a vector.	Full support.
RANK	Returns the number of dimensions in an array.	Full support.
REDIM	Changes the subscript range of previously dimensioned arrays.	No support for BUFFER
SIZE	Returns the number of elements in a dimension of an array.	Full support.
SUM	Returns the sum of all the elements in a numeric array.	Full support, except COMPLEX.

Index

G

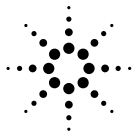
H

I

R

S

Interfacing Techniques



Agilent Technologies

December 2000

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 6 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 review 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 preprogrammed, 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

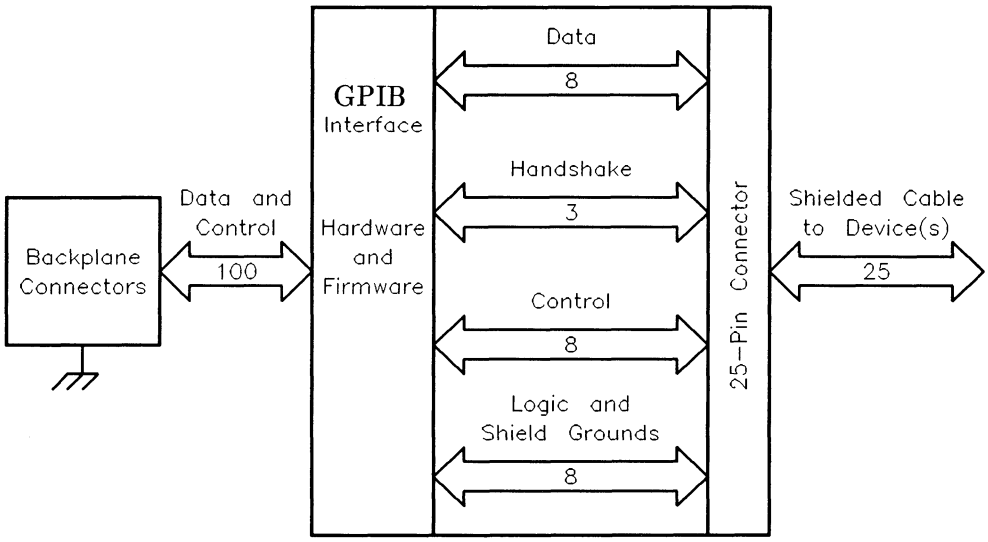
Another powerful feature of some interfaces is to relieve the computer of low-level tasks, such as performing data-transfer handshakes. This distribution of tasks eases some of the computer's burden and also decreases the otherwise-stringent response-time requirements of external devices. The actual 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 GPIB Interface

This interface is Hewlett-Packard's implementation of the IEEE-488 1978 Standard Digital Interface for Programmable Instrumentation. The acronym " GPIB ' comes from Hewlett-Packard Interface Bus, often called the "bus".



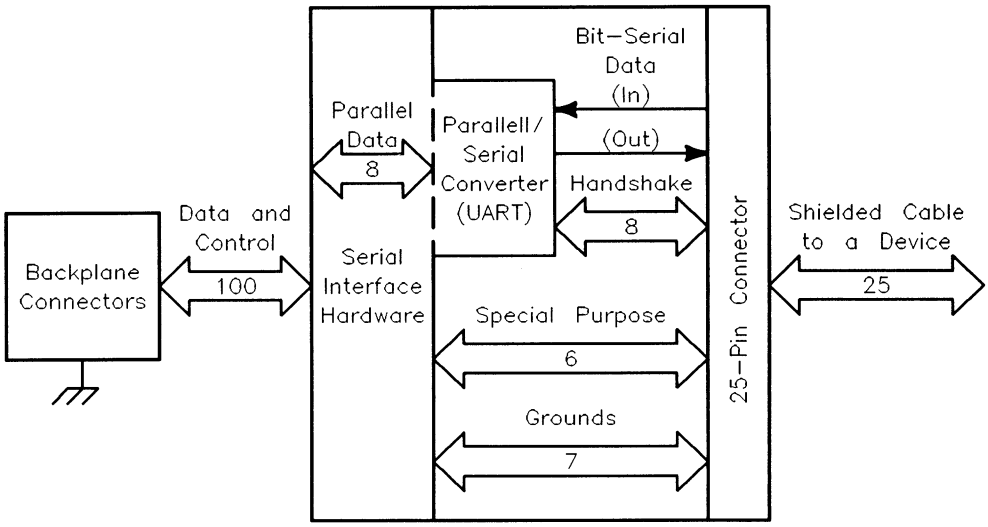
Block Diagram of the GPIB Interface

The GPIB 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 GPIB device and begin programming. All resources connected to the computer through the GPIB 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

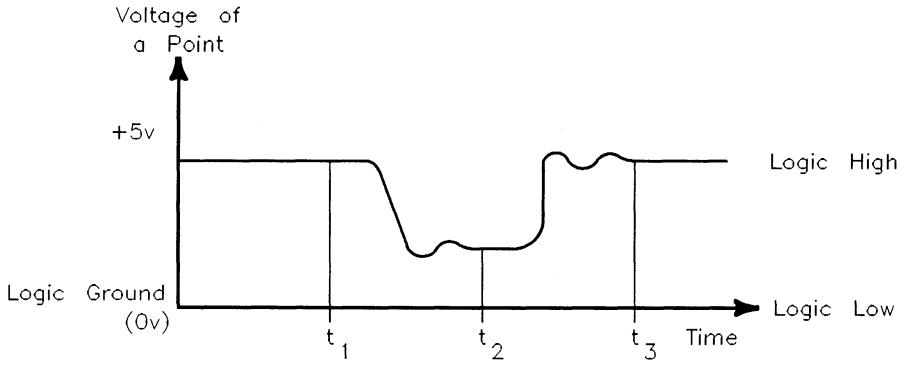
Data is transmitted at several programmable rates using either a simple data handshake or no handshake at all. The main use of this interface is in communicating with simple devices.

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
```

prints

```
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
```

prints

```
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 describes I/O operations regarding 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 that 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 GPIB 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.

GPIB Device Selectors

Each device on the GPIB 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.

GPIB **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 GPIB 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.

ReAssigning I/O Path Names

If an I/O path name already assigned to a resource is to be reassigned 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 GPIB 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 the 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 names 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. These variables 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 assignment is not "restored" upon exiting the subprogram. In this example, "First Main" is sent to the GPIB 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
- redirecting 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).

Redirecting 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 redirecting 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 redirecting 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" that 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 "Data Storage and Retrieval", chapter 7 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

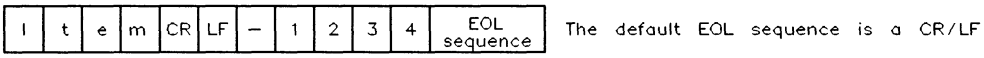
Data items are output one byte at a time, beginning with the left-most item in the source list and continuing until all of the source items have been output. Items *in the list* must be *separated* by either a comma or a semicolon. However, items in the data output may or may not be separated by item 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
```

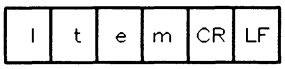


A comma separator specifies that a comma, CHR$(44), terminates numeric items.

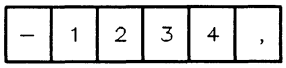
```
OUTPUT Device;-1234,"Item"
```

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",
```

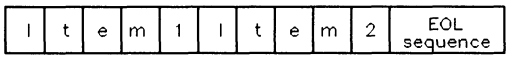



```
OUTPUT Device;-1234,
```

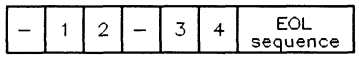


Using a *semicolon separator* suppresses output of the (otherwise automatic) item's terminator.

```
OUTPUT 1;"Item1";"Item2"
```

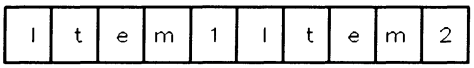



```
OUTPUT 1;-12;-34
```



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

	1	1	,		1	2	,		1	3	,		2	1	,		2	2	,		2	3	EOL sequence
	1	1	,		1	2	,		1	3	,		2	1	,		2	2	,		2	3	,
	1	1		1	2		1	3		2	1		2	2		2	3						
D	O	N	E	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 that 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

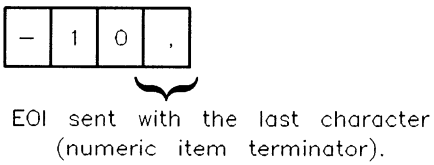
HP Instrument BASIC defines additional action when END is specified in a freefield OUTPUT statement directed to the GPIB interface.

END with GPIB Interfaces

With GPIB 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 is sent from the last source item, EOI is not sent.*

Examples

```
ASSIGN @Device TO 701

OUTPUT @Device;-10,END
```



```
OUTPUT @Device;"AB";END
```


```
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 that 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 example, 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 example, 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.

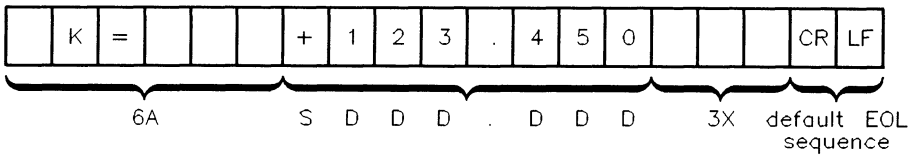
Again, each image list consists of images that each describe the format of data item to be output. The order of images in the list corresponds to the order of data items in the source list. In addition, image specifiers can be added to output (or to suppress the output of) certain characters.

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

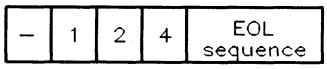
These image specifiers are used to describe the format of numbers.

Sign, Digit, Radix and Exponent Specifiers

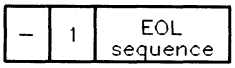
Image Specifier	Meaning
S	Specifies a "+" for positive and a "−" for negative numbers is to be output.
M	Specifies a leading space for positive and a "−" for negative numbers is to be output.
D	Specifies one ASCII digit ("0" through "9") is to to be output. Leading spaces and trailing zeros are used 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

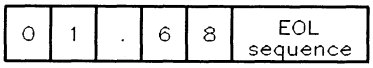
```
OUTPUT @Device USING "DDDD";-123.769
```



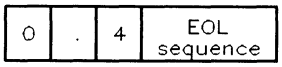

```
OUTPUT @Device USING "4D";-1.2
```



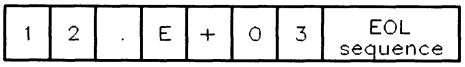

```
OUTPUT @Device USING "ZZ.DD";1.675
```



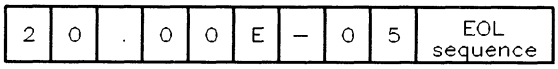

```
OUTPUT @Device USING "Z.D";.35
```



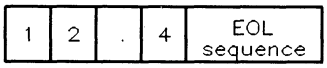

```
OUTPUT @Device USING "DD.E";12345
```



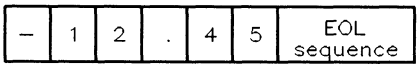

```
OUTPUT @Device USING "2D.DDE";2E-4
```



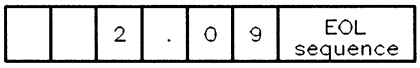

```
OUTPUT @Device USING "K";12.400
```




```
OUTPUT CRT USING "MDD.2D";-12.449
```

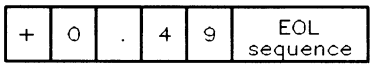


```
OUTPUT CRT USING "MDD.DD";2.09
```

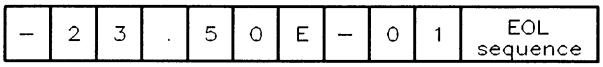



```
OUTPUT 1 USING "SD.D";2.449
```


```
OUTPUT 1 USING "SZ.DD";.49
```

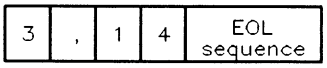



```
OUTPUT CRT USING "SDD.DDE";-2.35
```




```
OUTPUT @Device USING "**.D";2.6
```


```
OUTPUT @Device USING "DRDD";3.1416
```




```
OUTPUT @Device USING "H";3.1416
```

String Images

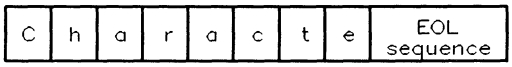
These types of image specifiers are used to specify the format of string data items.

Character Specifiers

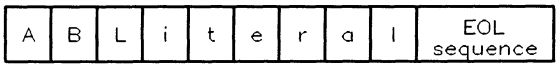
Image Specifier	Meaning
A	Specifies that one character is to be output. Trailing spaces are used as fill characters if the string contains less than the number of characters specified.
"literal"	All characters placed in quotes form a string literal, which is output exactly as is. Literals can be placed 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

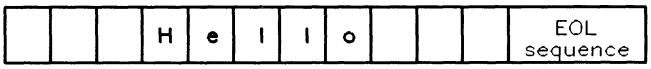
```
OUTPUT @Device USING "8A";"Characters"
```




```
OUTPUT @Device USING "K,""Literal""";"AB"
```




```
OUTPUT @Device USING "K";"  Hello  "
```




```
OUTPUT @Device USING "5A";"  Hello  "
```

Binary Images

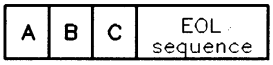
These image specifiers are used to output bytes (8-bit data) and words (16-bit data) to the destination. Typical uses are to output non-ASCII characters or integers in their internal representation.

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.

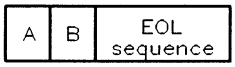
Binary Examples

```
OUTPUT @Device USING "B,B,B";65,66,67
```




```
OUTPUT @Device USING "B";13
```


```
OUTPUT @Device USING "W";256*65+66
```



Special-Character Images

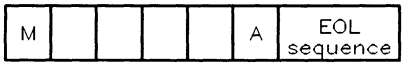
These specifiers require no corresponding data in the source list. They can be used to output spaces, end-of-line sequences, and form-feed characters.

Special-Character Specifiers

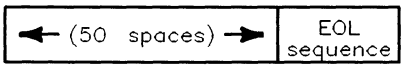
Image Specifier	Meaning
X	Specifies that a space character, CHR$(32), is to be output.
/	Specifies that a carriage-return character, CHR$(13), and a line-feed character, CHR$(10), are to be output.
@	Specifies that a form-feed character, CHR$(12), is to be output.

Special-Character Examples

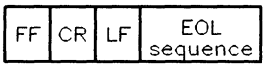
```
OUTPUT @Device USING "A,4X,A";"M","A"
```



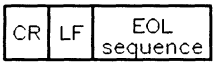

```
OUTPUT @Device USING "50X"
```




```
OUTPUT @Device USING "@,/"
```




```
OUTPUT @Device USING "/"
```



Termination Images

These specifiers are used to output or suppress the end-of-line sequence output after the last data item.

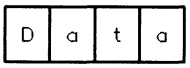
Termination Specifiers

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 redefine 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

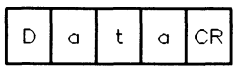
```
OUTPUT @Device USING "4A,L";"Data"
```


```
OUTPUT @Device USING "#,K";"Data"
```

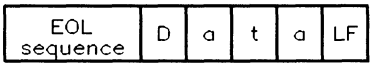



```
OUTPUT @Device USING "#,B";12
```


```
OUTPUT @Device USING "+,K";"Data"
```



```
OUTPUT @Device USING "-,L,K";"Data"
```



Additional Image Features

Several additional features of outputs that use images are available with the computer. Several of these features, which have already been shown, will be explained here in detail.

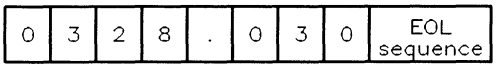
Repeat Factors

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.

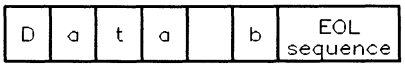
Repeatable Specifiers	Nonrepeatable Specifiers
Z, D, A, X, /, @, L	S, M, ., R, E, K, H, B, W, Y, #, %, +, -

Examples

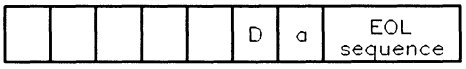
```
OUTPUT @Device USING "4Z.3D";328.03
```



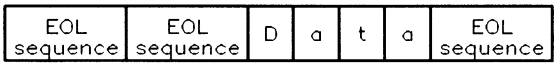

```
OUTPUT @Device USING "6A";"Data bytes"
```



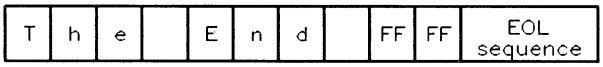

```
OUTPUT @Device USING "5X,2A";"Data"
```



```
OUTPUT @Device USING "2L,4A";"Data"
```




```
OUTPUT @Device USING "8A,2@";"The End"
```




```
OUTPUT @Device USING "2/"
```

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 reuse 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 reuse 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.

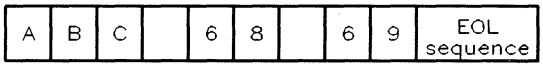
```
110   ASSIGN @Device TO CRT
120   Num_1=1
130   Num_2=2
140   !
150   OUTPUT @Device USING "DD.DD";Num_1,Num_2,"Data_1"
160   END
```

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



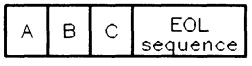
This nesting with parentheses is made with the same hierarchy as with parenthetical nesting within mathematical expressions. Only eight levels of nesting are allowed.

END with OUTPUTs that Use Images

Using the optional secondary keyword END in an OUTPUT statement that uses an image produces results that 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.

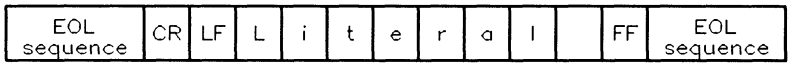
Examples

```
Device=12

OUTPUT Device USING "K";"ABC",END
OUTPUT Device USING "K";"ABC";END
OUTPUT Device USING "K";"ABC" END
```



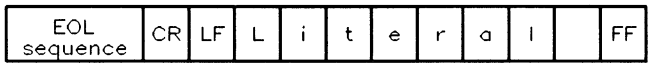
The EOL sequence is not suppressed.

```
OUTPUT Device USING "L,/,""Literal"",X,@"
```



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
```



The EOL sequence is suppressed because no source items were included in the statement; all characters output were the result of specifiers that require no corresponding expression in the source list.

Additional END Definition

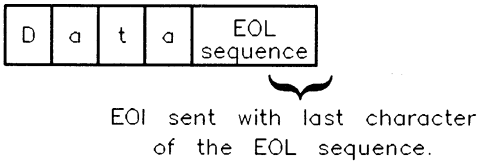
The END secondary keyword has been defined to produce additional action when included in an OUTPUT statement directed to GPIB interfaces.

END with GPIB Interfaces

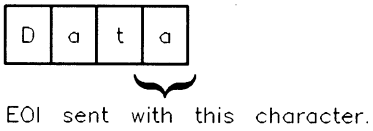
With GPIB interfaces, END has the additional function of sending the End-or-Identify signal (EOI) with the *last character* of either the last source item or the EOL sequence (if sent). As with freefield OUTPUT, *no EOI is sent if no data is sent from the last source item and the EOL sequence is suppressed.*

Examples.

```
ASSIGN @Device TO 701

OUTPUT @Device USING "K";"Data",END
OUTPUT @Device USING "K";"Data","",END
```

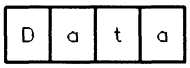



```
OUTPUT @Device USING "#,K";"Data" END
```



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 that was used in the output.

```
OUTPUT @Device USING "#,K";"Data","",END
OUTPUT @Device USING """Data""";END
```



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 that 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.

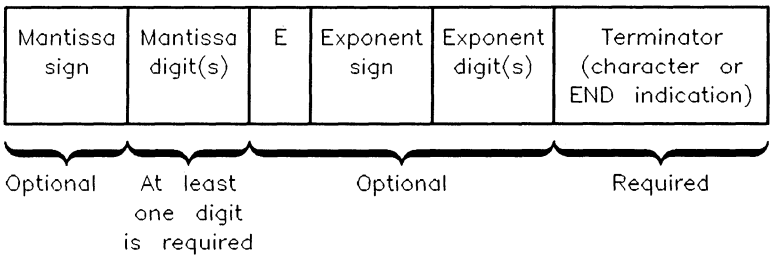
Item terminators must signal the end of each item so that the computer enters data into the proper destination variable. The terminator of the last item may also terminate the ENTER statement (in some cases). The actual character(s) that terminate entry into each type of variable are described in the 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.

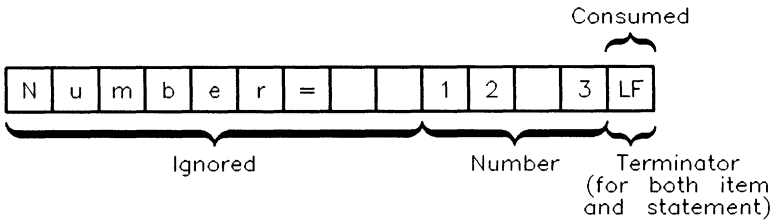


Numeric characters include decimal digits "0" through "9" and the characters ".", "+", "−", "E", and "e". These last five characters must occur in meaningful positions in the data stream to be considered numeric characters; if any of them occurs in a position in which it cannot be considered part of 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.

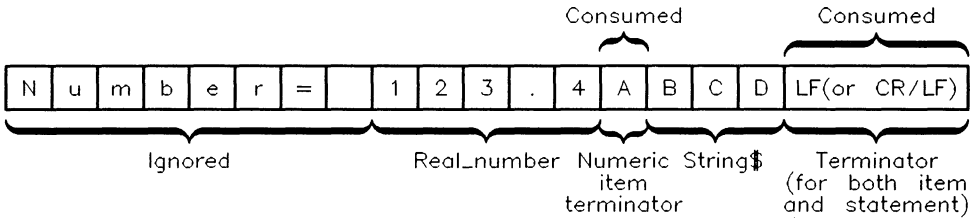
```
100   ASSIGN @Device TO Device_selector
110   ENTER @Device;Number  ! Default is data type REAL.
120   END
```



   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*.

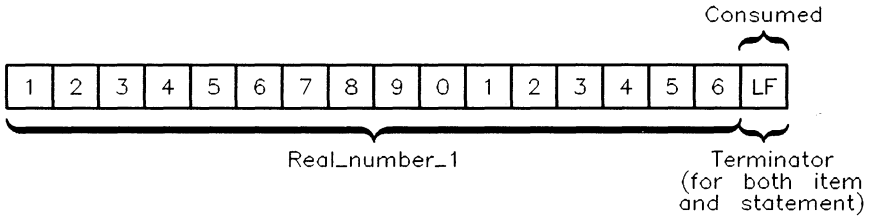
```
ENTER @Device;Real_number,String$
```



   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.

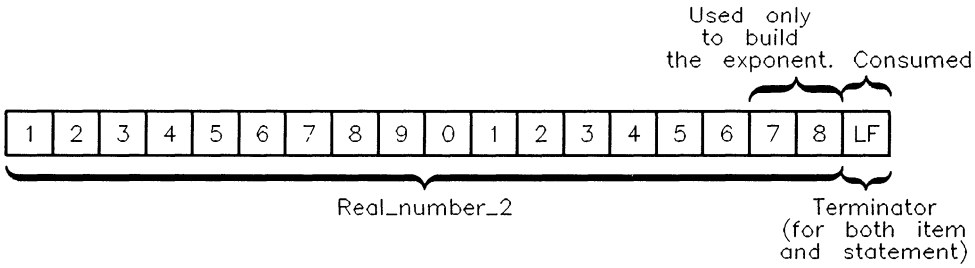
3. If more than 16 digits are received, only the first 16 are used as significant digits. However, all additional digits are treated as trailing zeros so that the exponent is built correctly.

```
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`.

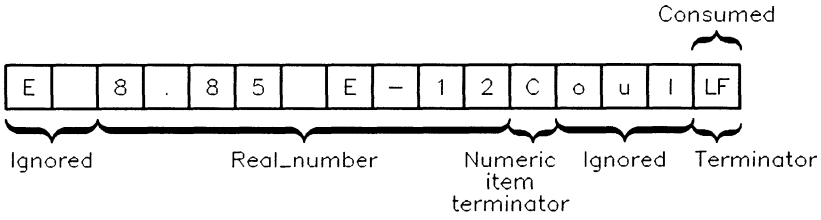
```
ENTER @Device;Real_number_2
```



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.

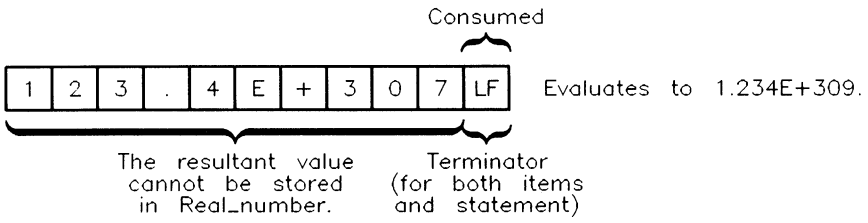
```
ENTER @Device;Real_number
```



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
   ```



   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).

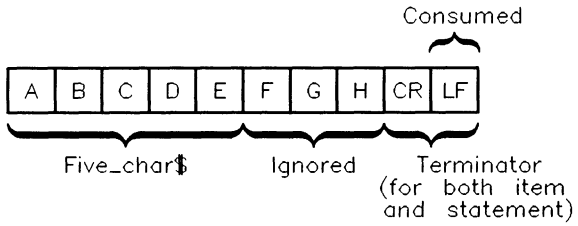
String-item terminator characters are either a line-feed (LF) or a carriage-return followed by a line-feed (CR/LF). As with numeric-item terminators characters, these characters are not entered into the string variable (during free-field enters); they are "lost" when they terminate the entry. The EOI 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.

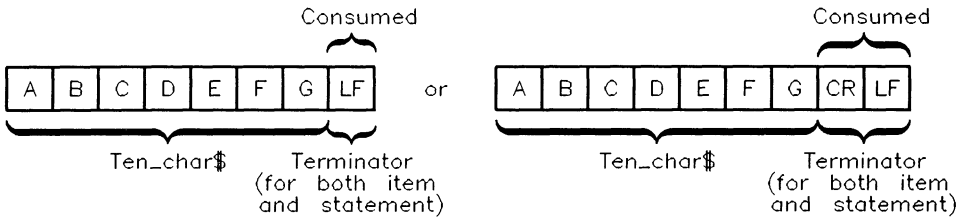
The following statements and resultant variable contents illustrate the first two conditions; the next section describes termination by EOI. Assume that the string variables Five_char$ and Ten_char$ are dimensioned to lengths of 5 and 10 characters, respectively.

```
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.

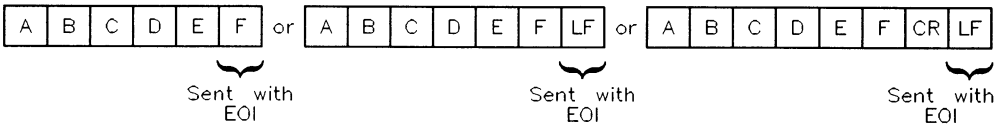
Another case involving numeric data can also occur. (See the example given with "rule 4" describing the number builder.) If a trailing non-numeric character terminates the last item (which is a numeric variable), additional characters will be entered in search of either a line-feed or a character accompanied 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 GPIB 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:

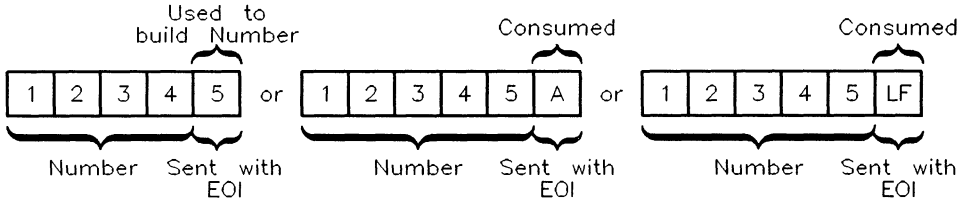
```
ENTER @Device;String$
```



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.

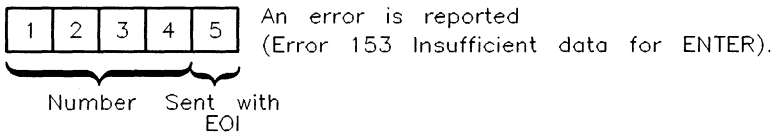
For example:

```
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.

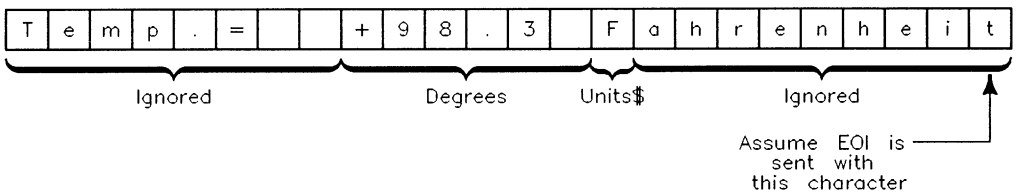
The examples given here treat the image list as a *procedure*.

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.



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 that 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 Redefinition

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

Image Specifier	Meaning
#	Specifies that a statement-termination condition is *not* required; the ENTER statement is automatically terminated as soon as the *last image item* is satisfied.
%	Also specifies that a statement-termination condition is not required. In addition, EOI is redefined to be an *immediate* statement terminator, *allowing early termination* of the ENTER *before all* 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: **Specifier** — **Legal Boundary** K,−K — With any character, since this specifies a variable-width field of characters. S,M,D,E,Z,.,A, X,*literal*,B,W — 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. / — 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.
+	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).
−	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.

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 Reuse

If there are fewer images than items in the destination list, the list will be reused, 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 reused:

```
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.

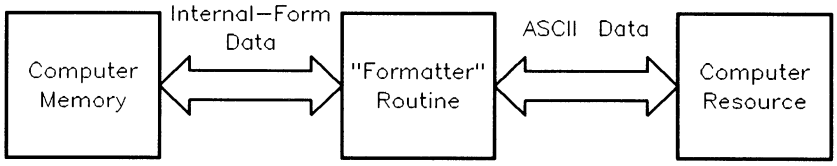
The FORMAT Attributes

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.

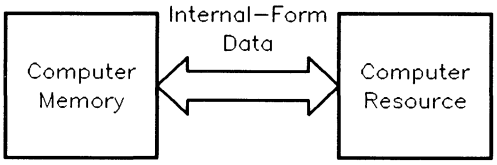
Before getting into how to assign these attributes to I/O paths, let's take a brief look at each one.

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 redefine 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 that 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 GPIB 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   !
```

(Continued)

```
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 that 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. A string may contain both printing and non-printing (control) characters. Printing string contents that contain control characters could interfere with examining the data stream. The preceding subprogram may facilitate viewing this data without viewing such strings.

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.

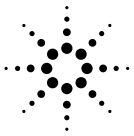
Example

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

Language Reference



Agilent Technologies

December 2000

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.

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

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 that 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.

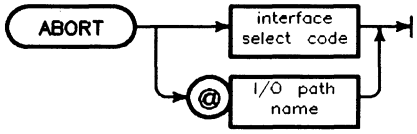
2

Keyword Dictionary

ABORT

ABORT terminates I/O activity on the specified GPIB interface.

Syntax



Item	Description	Range
interface select code	numeric expression, rounded to an integer	5, 7 through 31
I/O path name	name assigned to an GPIB interface	—

Example Statements

```
ABORT 7

ABORT Isc

IF Stop_code THEN ABORT @Source
```

Details

Use ABORT to control only GPIB interfaces. If you specify a select code for any other type of interface, error 150 will result.

If the computer is the system controller, but not currently the active controller, executing ABORT causes the computer to assume active control.

Note that `ABORT` *interface_select* is allowed, but `ABORT` *primary_address* is not. For example:

```
ABORT 7      allowed
ABORT 721    not allowed
```

Summary of Bus Actions

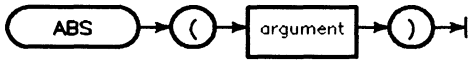
	System Controller	Not System Controller
Active Controller	IFC (duration ≥100 μsec) REN $\overline{\text{ATN}}$	ATN MTA UNL $\overline{\text{ATN}}$
Not Active Controller	IFC (duration ≥100 μsec)[1] REN $\overline{\text{ATN}}$	No Action

[1] The IFC message allows a non-active controller (which is the system controller) to become active.

ABS

ABS returns the absolute value of its argument.

Syntax



Item	Description/Default	Range Restrictions
argument	numeric expression	within valid ranges of INTEGER and REAL data types for INTEGER and REAL arguments.

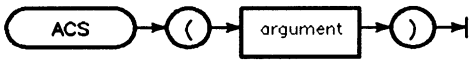
Example Statements

```
PRINT "Value =";ABS(X1)
```

ACS

ACS returns the arccosine of its argument.

Syntax



Item	Description/Default	Range Restrictions
argument	numeric expression	−1 through +1 for INTEGER and REAL arguments.

Example Statements

```
Angle=ACS(Cosine)
```

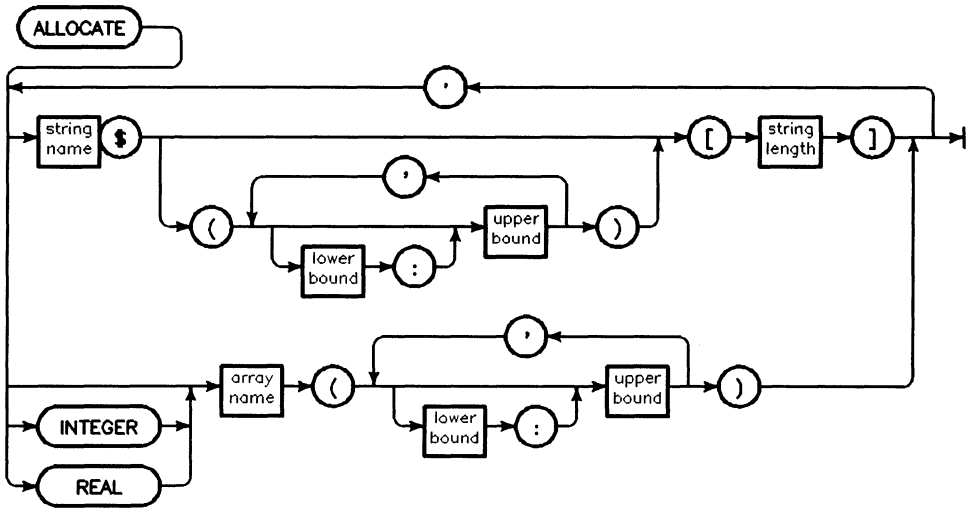
Details

The angle mode (set by RAD or DEG) determines whether the value returned is in degrees or radians. If the current angle mode is DEG, the range of the result is 0 to 180 degrees. 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.

ALLOCATE

ALLOCATE dynamically allocates memory for arrays and string variables during program execution.

Syntax



Item	Description	Range
array name	name of a numeric array	any valid name
lower bound	numeric expression, rounded to an integer; default = OPTION BASE value (0 or 1)	−32 768 through +32 767 (see "array" in Glossary)
upper bound	numeric expression, rounded to an integer	−32 768 through +32 767 (see "array" in Glossary)
string name	name of a string variable	any valid name
string length	numeric expression, rounded to an integer	1 through 32 767

Example Statements

```
ALLOCATE Temp(Low:High)

ALLOCATE INTEGER Array(Index,2,8)

ALLOCATE R$[LEN(A$)+1]

ALLOCATE Text$(Lines)[80]
```

Details

Memory reserved by the ALLOCATE statement can be freed by the DEALLOCATE statement. However, because of the stack discipline used when allocating, the freed memory space does not become available unless all subsequently allocated items are also deallocated. For example, assume that A$ is allocated first, then B$, and finally C$. If a DEALLOCATE A$ statement is executed, the memory space for A$ is not available until B$ and C$ are deallocated.

The variables in an ALLOCATE statement cannot appear in the same context in declaration statements such as COM, DIM, or INTEGER. If variable(s) are to be allocated in a subprogram, the variable(s) cannot have been included in the subprogram's formal parameter list. Implicitly declared variables cannot be allocated. Numeric variables for which a type is not specified are assumed to be REAL. A variable can be reallocated in its program context only if it has been deallocated and its type and number of dimensions remain the same.

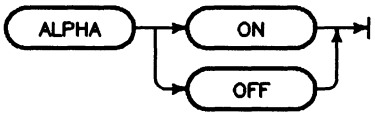
ALLOCATE allows you to dynamically allocate memory for arrays. However, the array dimensions are determined statically. Thus you can change the size of the dimensions, but you cannot change the number of dimensions of an array within a program context.

Exiting a subprogram automatically deallocates any memory space allocated within that program context.

ALPHA ON/OFF

ALPHA ON shows the alpha window; ALPHA OFF hides it.

Syntax



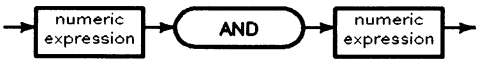
Example Statements

```
ALPHA ON
IF Graph THEN ALPHA OFF
```

AND

AND returns a 1 or a 0 based on the logical AND of its two arguments.

Syntax



Example Statements

```
IF Flag AND Test2 THEN Process
Final=Initial AND Valid
```

Details

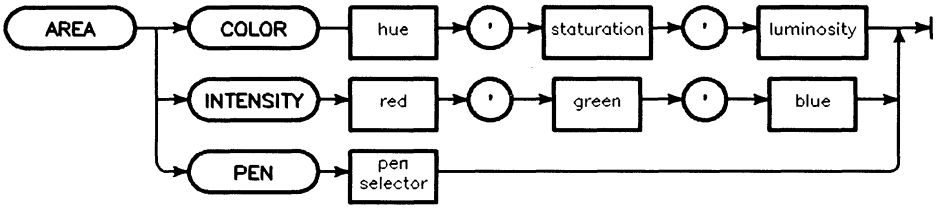
A non-zero value (positive or negative) is treated as a logical 1; only zero is treated as a logical 0.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

AREA

AREA sets the color used to shade in graphical regions subsequently created by various graphics plotting commands.

Syntax



Item	Description	Range
hue	numeric expression	0 through 1
saturation	numeric expression	0 through 1
luminosity	numeric expression	0 through 1
red	numeric expression	0 through 1
green	numeric expression	0 through 1
blue	numeric expression	0 through 1
pen selector	numeric expression, rounded to an integer	−32 768 through +32 767

Example Statements

```
AREA COLOR Hue,Saturation,Luminosity
AREA INTENSITY Red(I),Green(I),Blue(I)
AREA PEN 3
```

Details

The default fill color is the color specified by pen 1. This color is solid white after power-up, SCRATCH A, GINIT.

A fill color remains in effect until the execution of an AREA, GINIT, or SCRATCH A. Other statements that may alter the current fill color (if the data passed to them is an array) are

- PLOT
- RPLOT
- IPLOT

SET PEN affects pen colors, and therefore can also affect fill colors specified with AREA statements.

Specifying color with the SET PEN and AREA PEN statements (resulting in non-dithered color) results in a much more accurate representation of the desired color than the same color requested with an AREA COLOR or AREA INTENSITY statement.

AREA PEN Details ...
AREA COLOR Details ...
Alternate Pen Mode ...

AREA PEN

A fill color specified with AREA PEN is guaranteed to be non-dithered, and the AREA PEN statement executes faster than AREA COLOR or AREA INTENSITY.

The pen numbers have the same effect as described in the PEN statement for line color except that in the alternate pen mode, negative pens erase as in the normal pen mode; they do not complement. Pen 0 in normal pen mode erases; it does not complement.

AREA COLOR

When AREA COLOR executes, the HSL parameters are converted to RGB values. Then, if the color requested is not available in the color map, the computer creates the closest possible color in RGB color space to the one requested by filling the 4 by 4 dither cell with the best combination of colors from the color map.

Alternate Pen Mode Fills

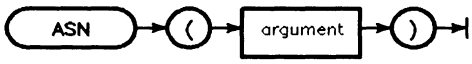
If the alternate drawing mode is in effect when the fill is performed, the area will be filled with non-dominant color. See GESCAPE operation selectors 4 and 5.

In the alternate pen mode, negative pens erase as in the normal pen mode; they do not complement.

ASN

ASN returns the arcsine of its argument.

Syntax



Item	Description/Default	Range Restrictions
argument	numeric expression	−1 through +1 for INTEGER and REAL arguments

Example Statements

```
Angle=ASN(Sine)
```

Details

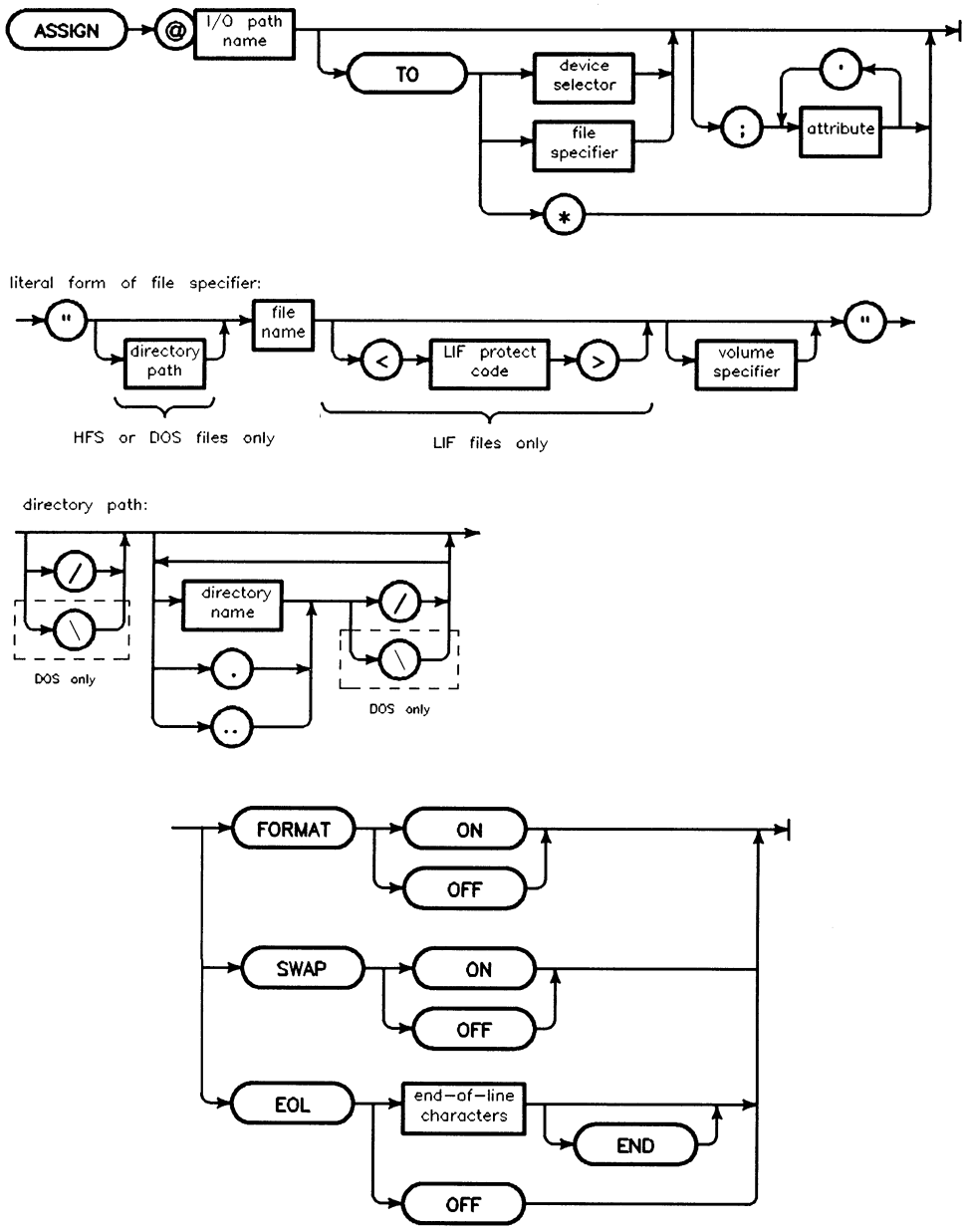
The angle mode (set by RAD or DEG) determines whether the value returned is in degrees or radians. If the current angle mode is DEG, the range of the result is -90 to +90 degrees. If the current angle mode is RAD, the range of the result is $-\pi/2$ to $+\pi/2$ radians. The angle mode is radians unless you specify degrees with the DEG statement.

ASSIGN

ASSIGN assigns an I/O path name and attributes to one of the following:

- a file
- an instrument
- a peripheral device

Syntax for Files



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	(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

These statements assign an I/O path name to a file:

```
ASSIGN @File TO File_name$

ASSIGN @File TO File_name$; FORMAT OFF

ASSIGN @File TO File_name$; FORMAT OFF, SWAP OFF

ASSIGN @File TO * ! Close the file.
```

These statements assign an I/O path name to an instrument:

```
ASSIGN @Hpib_scope TO 724

ASSIGN @Serial_scope TO 9
```

Details

The ASSIGN statement serves a variety of purposes. Its main purpose is to open (create) an I/O path name and assign that name to a resource. ASSIGN can specify attributes that describe how data is shared with the resource. ASSIGN can also close (terminate) an I/O path.

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.

Using ASSIGN with Files ...
Using ASSIGN with Instruments ...

Wildcards

Wildcard file specifiers used with ASSIGN must match one, and only one, file name. You must first enable wildcard recognition using WILDCARDS. Refer to the keyword entry for WILDCARDS for details.

Using FORMAT

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.

If the file was created with the CREATE ASCII statement, the file is always accessed as a LIF ASCII file. LIF ASCII is a file type used by certain HP computers, such as the HP BASIC Workstation. If you specify FORMAT ON or FORMAT OFF for such a file, it is 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
DOS file	FORMAT OFF
HP-UX file	FORMAT OFF

Using Files

Assigning an I/O path name to a file associates the I/O path with the file and opens the file for reading and writing. The file must be a data file. You cannot, for example, ASSIGN an I/O path to a PROG (program) file. The file must already exist; ASSIGN does not do an implied CREATE.

Files have a position pointer that is associated with each I/O path name. The pointer identifies the next byte to be written or read. The 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 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:

- the current end-of-file changes
- END is specified in an OUTPUT statement directed to the file
- a CONTROL statement directed to the I/O path name changes the position of the end-of-file pointer

Using Instruments and Other Devices

I/O path names are assigned to instruments and other devices by placing the device selector after the keyword TO. The statement `ASSIGN @Meter TO 710` creates the I/O path name `@Meter` and assigns it to a device on GPIB.

A device can have more than one I/O path name associated with it. Each I/O path name can have different attributes (such as FORMAT and SWAP), 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.

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.

Using ASSIGN With SWAP

The secondary keyword SWAP sets the order in which bytes are transferred for an I/O path assigned with FORMAT OFF. If the ASSIGN statement does not create the I/O path with FORMAT OFF, any SWAP clause in the statement is ignored. The default behavior of ASSIGN is SWAP OFF.

When SWAP is OFF, data is transferred by sending the least significant byte first. When SWAP is ON, data is transferred by sending the most significant byte first.

SWAP affects only these types of data:

- two-byte integers (such as INTEGER variables)
- eight-byte reals (such as REAL variables)
- strings written to BDAT files with FORMAT OFF

Note that no other type of data is handled properly by SWAP. For example, you cannot properly read a long integer (a four-byte integer) into two INTEGER variables with SWAP ON.

SWAP is useful in these situations:

- You want to transfer integer or real numbers between an instrument and HP Instrument BASIC with FORMAT OFF to increase throughput. Use SWAP ON/OFF as necessary to transfer bytes in the order that is compatible with the instrument.
- You want to share BDAT or DOS data files written with FORMAT OFF between HP Instrument BASIC and other versions HP BASIC. In this case, you must use SWAP ON.

Suppose you want to send commands to an instrument as strings and receive numeric data from the instrument as unformatted bytes. You also want to read string data from the instrument, such as error messages. You could use this approach:

```
100 ASSIGN @Device TO Dev_selector
110 ASSIGN @Device_bin TO Dev_selector;FORMAT OFF,SWAP ON
120 OUTPUT @Device;Cmd$         ! Cmd$ contains an instrument command.
130 ENTER @Device_bin;Real_var ! Real_var is a REAL variable.
140 ENTER @Device;Err_msg$      ! Err_msg$ contains an error message string.
```

Suppose you want to read a BDAT file written on a DOS disk by the HP Measurement Coprocessor. These are the statements that wrote the file:

```
100 INTEGER Int_var
110 CREATE BDAT "MYFILE"
120 ASSIGN @File TO "MYFILE" ! Default for BDAT files is FORMAT OFF.
130 OUTPUT @File;Real_var,Int_var,String$
```

It is important to note that the default formatting for I/O paths assigned to BDAT files is FORMAT OFF. To read MYFILE with HP Instrument BASIC, you must use these statements:

```
100 INTEGER Int_var
110 ASSIGN @File TO "MYFILE";SWAP ON
120 ENTER @File;Real_var,Int_var,String$
```

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 reassigned to a resource, the original I/O path is closed after the new one is opened. The reassignment 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, ERROR SUBEXIT, RETURN..expression, or ON-event..RECOVER.
- when SCRATCH, SCRATCH A, or SCRATCH C is executed, any form of STOP occurs, or an END, LOAD, or GET is executed.

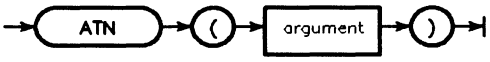
I/O path names that *are* included in a COM statement remain open and valid during a LOAD, GET, STOP, END, or simple SCRATCH. I/O path names in COM are only closed at the following times:

- when they are explicitly closed; for example, `ASSIGN @File TO *`
- when SCRATCH A or SCRATCH C is executed
- when a LOAD, 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

ATN returns the arctangent of its argument.

Syntax



Item	Description/Default	Range Restrictions
argument	numeric expression	within valid ranges of INTEGER or REAL data types for INTEGER and REAL arguments

Example Statements

```
Angle=ATN(Tangent)
```

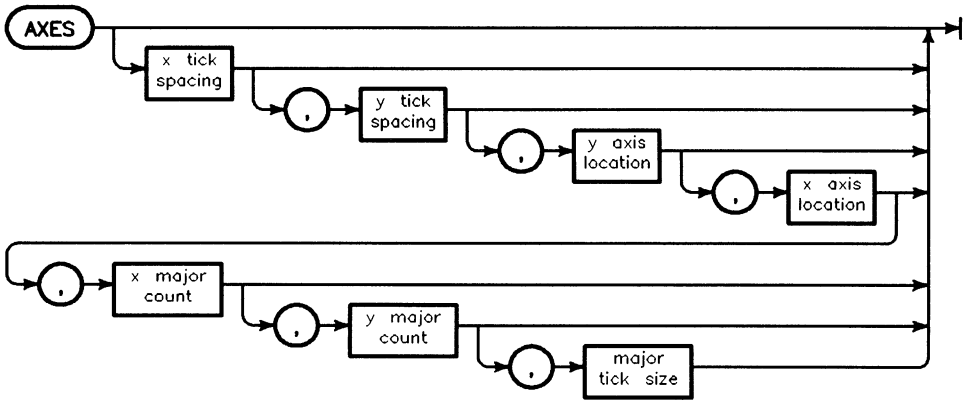
Details

The angle mode (set by RAD or DEG) determines whether the value returned is in degrees or radians. If the current angle mode is DEG, the range of the result is -90 to +90 degrees. 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.

AXES

AXES draws a pair of axes with optional, equally spaced tick marks.

Syntax



Item	Description	Range
x tick spacing	numeric expression in current units; default = 0, no ticks	(see text)
y tick spacing	numeric expression in current units; default = 0, no ticks	(see text)
y axis location	numeric expression specifying the location of the y axis in x-axis units; default = 0	—
x axis location	numeric expression specifying the location of the x axis in y-axis units; default = 0	—
x major count	numeric expression, rounded to an integer, specifying the number of tick intervals between major tickmarks; default = 1 (every tick is major)	1 through 32 767
y major count	numeric expression, rounded to an integer, specifying the number of tick intervals between major tick marks; default = 1 (every tick is major)	1 through 32 767
major tick size	numeric expression in graphic display units; default = 2	—

Example Statements

```
AXES 10,10

AXES Xspace,Yspace

AXES Xspace,Yspace,Xlocy,Ylocx,Xmajor,Ymajor,Majorsize
```

Details

Tick marks are positioned so that a major tick mark coincides with the axis origin, whether or not that intersection is visible. Both axes and tick marks are drawn with the current line type and pen. Minor tick marks are drawn half the size of major tick marks.

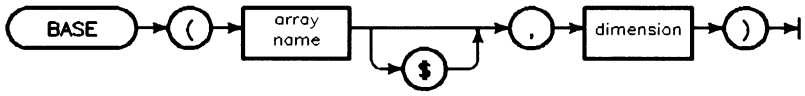
The X and Y tick spacing must not generate more than 32,768 tick marks in the clip area (including the axis), or error 20 will be generated.

The axes and tick marks drawn by AXES are affected by scaling resulting from SHOW and WINDOW. The axes and tick marks are *not* affected by rotations resulting from PIVOT and PDIR.

BASE

BASE returns the lower subscript bound of a dimension of an array.

Syntax



Item	Description	Range
array name	name of an array	any valid name
dimension	numeric expression, rounded to an integer	1 through 6; ≤ the RANK of the array

Example Statements

```
Lowerbound=BASE(Array,Dimension)
Upperbound(2)=BASE(A,2)+SIZE(A,2)-1
```

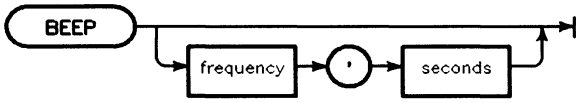
BDAT

See the CREATE BDAT statements.

BEEP

BEEP generates an audible tone.

Syntax



Item	Description	Range
frequency	numeric expression, rounded to the nearest tone; default = 1220.7	81 through 5208
seconds	numeric expression, rounded to the nearest hundredth; default = 0.2	.01 through 2.55

Example Statements

```
BEEP
```

BINAND

BINAND returns the bit-by-bit logical AND of its arguments.

Syntax

Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 through +32 767

Example Statements

```
Low_4_bits=BINAND(Byte,15)
IF BINAND(Stat,3) THEN Bit_set
```

Details

The arguments for BINAND 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.

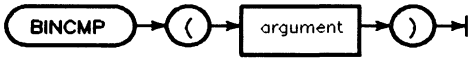
In the following example, the statement `Ctrl_word=BINAND(Ctrl_word,-9)` clears bit 3 of `Ctrl_word` without changing any other bits.

```
12 =  00000000 00001100     old Ctrl_word
-9 =  11111111 11110111     mask to clear bit 3
      ----------------
 4 =  00000000 00000100     new Ctrl_word
```

BINCMP

BINCMP returns the value of the bit-by-bit complement of its argument.

Syntax



Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 through +32 767

Example Statements

```
True=BINCMP(Inverse)
PRINT X,BINCMP(X)
```

Details

The argument for BINCMP 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:

```
-9 =  11111111 11110111    argument
      ----------------
+8 =  00000000 00001000    complement of argument
```

BINEOR

BINEOR returns the bit-by-bit exclusive OR of its arguments.

Syntax

Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 through +32 767

Example Statements

```
Toggle=BINEOR(Toggle,1)
True_byte=BINEOR(Inverse_byte,255)
```

Details

The arguments for BINEOR 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 compose the returned integer.

In the following example, the statement `Ctrl_word=BINEOR(Ctrl_word,4)` inverts bit 2 of Ctrl_word without changing any other bits.

```
12 =  00000000 00001100     old Ctrl_word
 4 =  00000000 00000100     mask to invert bit 2
      ----------------
 8 =  00000000 00001000     new Ctrl_word
```

BINIOR

BINIOR returns the bit-by-bit inclusive OR of its arguments.

Syntax

Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 through +32 767

Example Statements

```
Bits_set=BINIOR(Value1,Value2)

Top_bit_on=BINIOR(All_bits,2^15)
```

Details

The arguments for BINIOR 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.

In the following example, the statement `Ctrl_word=BINIOR(Ctrl_word,6)` sets bits 1 and 2 of Ctrl_word without changing any other bits.

```
19 =  00000000 00010011     old Ctrl_word
 6 =  00000000 00000110     mask to set bits 1 & 2
      -----------------
23 =  00000000 00010111     new Ctrl_word
```

BIT

BIT returns a 1 or 0 representing the value of the specified bit of its argument.

Syntax

Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 through +32 767
bit position	numeric expression, rounded to an integer	0 through 15

Example Statements

```
Flag=BIT(Info,0)
IF BIT(Word,Test) THEN PRINT "Bit #";Test;"is set"
```

Details

The argument for BIT 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.

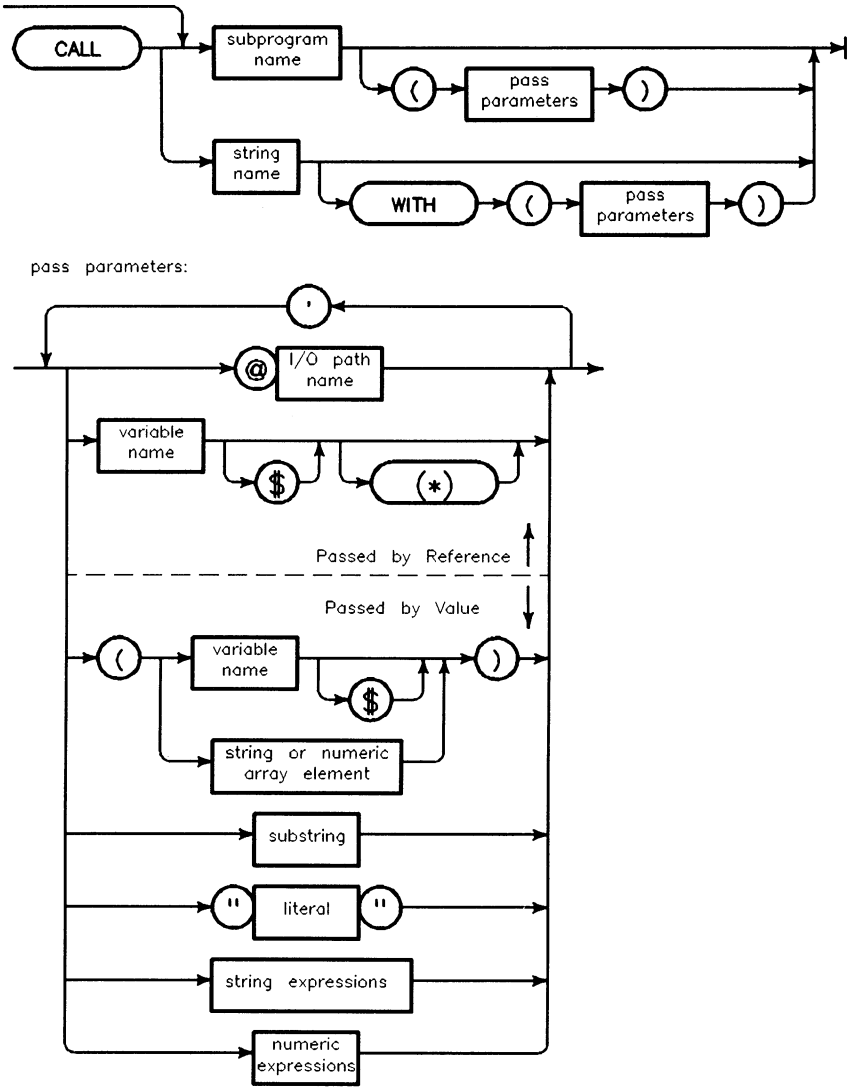
The following example reads the controller status register of the internal GPIB and takes a branch to "Active" if the interface is currently the active controller.

```
100  STATUS 7,3;S                 Reg 3 = control status
110  IF BIT(S,6) THEN Active      Bit 6 = active control
```

CALL

CALL transfers program execution to the specified subprogram and optionally passes parameters to the subprogram.

Syntax



Item	Description	Range
subprogram name	name of the SUB or CSUB subprograms to be called	any valid name
string name	a simple string variable containing the name of a user-defined subprogram	loaded SUBs and CSUBS
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	—

Example Statements

```
CALL Process(Reference,(Value),@Path)

CALL Transform(Array(*))

Transform(Array(*))

CALL MySub$

CALL MySub$ WITH (X,Y,A$)
```

Details

Subprograms may be invoked recursively.

The keyword CALL may be omitted if it is the first word in a program line. However, the keyword CALL is required in all other instances.

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 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, program execution resumes at the point at which the event-initiated branch was permitted.

CALL Using String Names

You can specify the subprogram accessed by CALL using either the subprogram name or a string expression that evaluates to the subprogram name. All of the calls to **Mysub** in the following code segment are legal:

```
100  Name$="Mysub"          using subprogram name with CALL
110  CALL Mysub(1)
120  Mysub(2)               using subprogram name without CALL
120  CALL Name$ WITH (3)    using string name with CALL
130  END
140  !
150  SUB Mysub(I)
160     PRINT "HELLO";I
170  SUBEND
```

Note that the string name must match the subprogram name *exactly*, including upper and lower case letters. Also note that you *must* use the keyword CALL with string subprogram names.

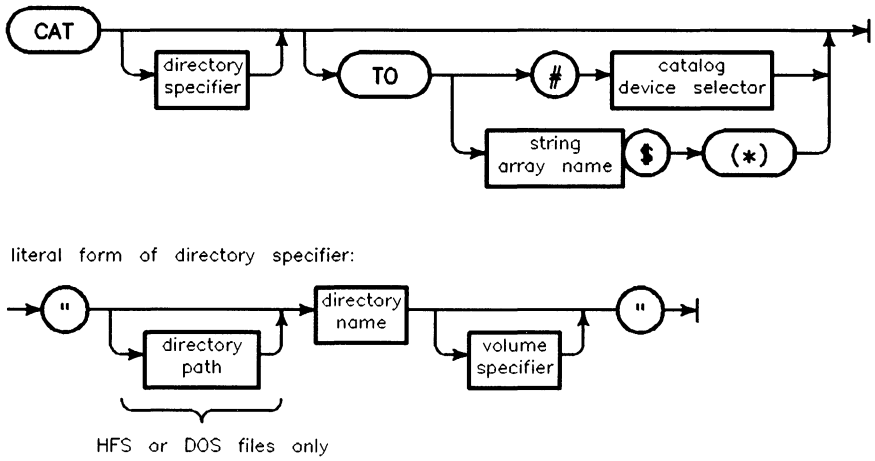
CASE

See SELECT ... CASE.

CAT

CAT lists the contents (files) in a specified directory or mass storage volume.

Syntax



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)
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
media specifier	sting expression specifying media address	any valid media

Example Statements

```
CAT             ! List the contents of the current MSI volume/directory.

CAT TO A$(*)    ! List the contents of the current MSI to a string variable.
CAT "Dir1/Dir2" ! List the contents of a subdirectory.
CAT ":,701,0"   ! Example of media specifier.
```

Details

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.

See the WILDCARDS statement for more on the use of wildcards with CAT.

Note that the format of the catalog listing is different depending on whether the catalog information is sent to the alpha window or a string variable.

```
CAT "A:\DIR1\DIR2"           ! This catalog listing is sent to
                             ! the alpha display in DOS format.

CAT "A:\DIR1\DIR2" TO A$(*) ! This catalog listing is sent to
                             ! the the string variable A$ in
                             ! "to string" format.
```

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.

DOS File System Catalogs

Here is a typical catalog listing of a DOS directory:

```
DIRECTORY: C:\PROJECTS\PROJECT.ONE
LABEL: HARD_DISK_C
FORMAT: DOS
AVAILABLE SPACE:     66776
                FILE     NUM   REC     MODIFIED
FILE NAME       TYPE    RECS   LEN    DATE    TIME PERMISSION
============ ====== ======= ===== ========= ===== ==========
ASCII_1       ASCII      100   256 15-Apr-91 18:06  RW-RW-RW-
BDAT_1        BDAT         5   256 15-Apr-91 18:10  RW-RW-RW-
MEMOS         DIR          0     1 15-Apr-91 14:29  RWXRWXRWX
```

The first line of the catalog shows the path name of the directory to be cataloged (`C:\PROJECTS\PROJECT_ONE` in this example).

The second line gives the volume label of the MS-DOS disk.

The third line gives the format of the mass storage medium, which is "DOS" for any DOS volume.

The forth line lists the number of 256-byte sectors on the disk (66776 in this example).

The fifth and sixth lines label the columns of the catalog.

FILE NAME — Lists the name of the file. The standard MS-DOS file-name conventions are used (up to eight characters followed by an optional period and an extension of up to three characters).

FILE TYPE — Lists the type of the file. DIR specifies a directory. ASCII, BDAT, and PROG specify the standard HP Instrument BASIC data and program file types. DOS specifies an "untyped" MS-DOS file.

NUM RECS — Lists the number of logical records (the number of records allocated to the file when it was created). For a DIR file, NUM RECS is always 0.

REC LEN — The logical record size. The record length is always 256 for an ASCII file, and always 1 for a DOS file. The default record length for a BDAT file is 256, but you can specify a user-defined record length. For a DIR file, REC LEN is always 1.

MODIFIED DATE TIME — The date 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 to. X indicates that the file can be searched (meaningful for directories only).

There are three classes of user permissions for each file:

OWNER (left-most 3 characters). GROUP (center 3 characters). OTHER (right-most 3 characters).

By default, the DFS binary sets the permissions for all new files to "RW-RW-RW-" and for all new directories to "RWXRWXRWX". You can use the PERMIT statement to make a file read-only. However, if you change the

OWNER bits, the GROUP and OTHER bits will also change. Refer to the PERMIT statement for more details.

CAT to a String Array

Refer to CAT listings for details on fields. Note the different DOS and LIF formats.

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 or 45 characters for a PROG file 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 (the SRM directory 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 through 21
File Type	32 through 36
Number of Records	37 through 45
Record Length	46 through 54
Time Stamp	56 through 71
Public Access Capabilities	73 through 75
Open Status	77 through 80

CAUSE ERROR

CAUSE ERROR simulates the occurrence of an error of the specified number, affecting error functions: ERRN, ERRM$, ERRL, and ERRLN.

Syntax

Item	Description	Range
error number	numeric expression, rounded to an integer	1 through 999; 1001 through 1080

Example Statements

```
CAUSE ERROR Err_num

IF Testing THEN CAUSE ERROR 80
```

Details

When the CAUSE ERROR statement is executed, it initiates the normal error-reporting action taken by the system when an error is encountered in a program line.

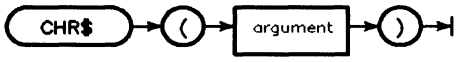
If ON ERROR is in effect and CAUSE ERROR is executed in a program line, the appropriate branch is initiated—just as if an actual error occurred on that line. When executed from a running program, CAUSE ERROR affects the error indications ERRN, ERRM$, ERRL, and ERRLN; each is set to the value appropriate for the specified error number and line number. However, ERRDS is not affected.

If CAUSE ERROR is executed at the keyboard, or if executed in a running program (while ON ERROR is not in effect), HP Instrument BASIC shows the error number and error message in the system message line of the alpha window. (Note that errors caused by executing statements from the command line do not affect the error indications listed in the preceding paragraph.)

CHR$

CHR$ converts a numeric expression into an ASCII character.

Syntax



Item	Description	Range
argument	numeric expression rounded to an integer	0 through 255

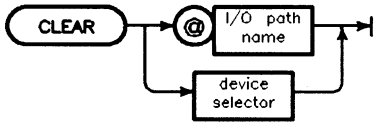
Example Statements

```
Lowercase$=CHR$(NUM(Uppercase$)+32)
A$[Marker;1]=CHR$(Digit+128)
Esc$=CHR$(27)
```

CLEAR

CLEAR clears the specified GPIB interface by sending a Device Clear or Selected Device Clear message.

Syntax



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 Voltmeter

CLEAR @Source
```

Details

GPIB Interfaces

CLEAR places all or only selected GPIB devices into a predefined, device-dependent state. The computer *must* be the active controller to execute this statement. The bus messages sent are the same whether or not the computer is the system controller. When primary addresses are specified, the bus is reconfigured and the SDC (Selected Device Clear) message is sent to all devices that are addressed by the LAG message.

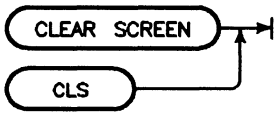
Summary of CLEAR Bus Actions

Interface Select Code Only	Primary Address Specified
ATN DCL	ATN MTA UNL LAG SDC

CLEAR SCREEN

CLEAR SCREEN clears the contents of the alpha window.

Syntax



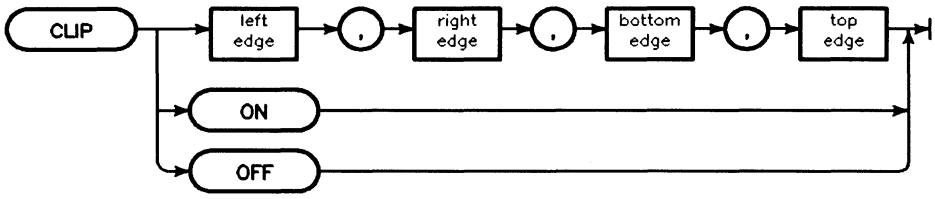
Example Statements

```
CLS

CLEAR SCREEN

IF Loop_count=1 THEN CLEAR SCREEN
```

CLIP

CLIP defines, enables, or disables the **soft clip limits** for subsequent graphics output.

Syntax



Item	Description	Range
left edge	numeric expression in current units	—
right edge	numeric expression in current units	—
bottom edge	numeric expression in current units	—
top edge	numeric expression in current units	—

Example Statements

```
CLIP Left,Right,Bottom,Top

CLIP ON

CLIP OFF
```

Details

Executing CLIP with numeric parameters allows the soft clip area to be set to the specified soft clip limits. If CLIP is not executed, the clipping area is either the entire graph window (if VIEWPORT has not been executed) or the area defined by the most recent VIEWPORT statement. All plotted points, lines, or labels are clipped at this boundary.

The hard clip area is determined by the physical limits of the graphics display area. The soft clip area is specified by the VIEWPORT and CLIP statements. CLIP ON sets the soft clip boundaries to the last specified CLIP or VIEWPORT boundaries, or to the hard clip boundaries if no CLIP or VIEWPORT has been executed. CLIP OFF sets the soft clip boundaries to the hard clip limits.

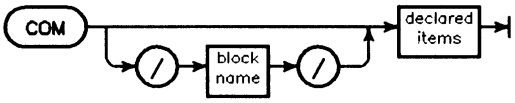
CLS

CLS is identical to CLEAR SCREEN.

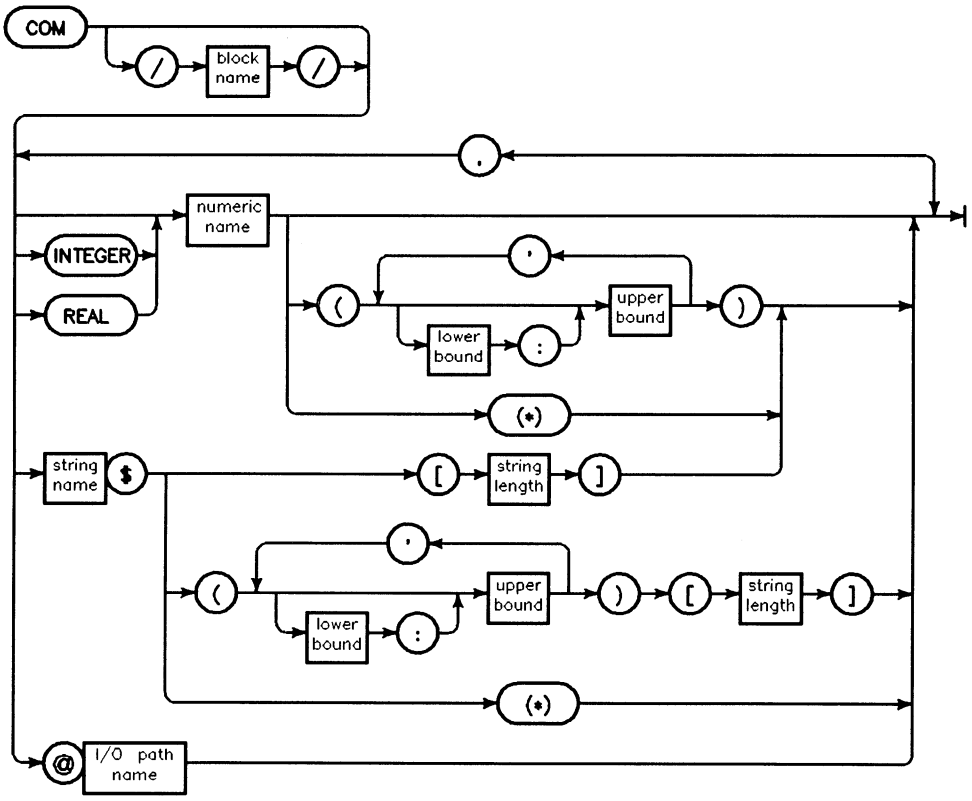
COM

COM dimensions and reserves memory for variables in a special "common" memory area so more than one program **context** can access the variables.

Syntax



Expanded diagram:



Item	Description	Range
block name	name identifying a labeled COM area	any valid name
declared items	list of common variables	see expanded diagram
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 = OPTION BASE value (0 or 1)	−32 767 through +32 767 (see "array" in Glossary)
upper bound	integer constant	−32 767 through +32 767 (see "array" in Glossary)
string length	integer constant	1 through 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 /Block/ Text$,@Path,INTEGER Points(*)

COM INTEGER I,J,REAL Array(-128:127)
```

Details

Storage for COM is allocated at prerun time in an area of memory that is separate from the data storage used for program contexts. This reserved portion of memory remains allocated until SCRATCH A or SCRATCH C is executed.

Changing the definition of the COM space is accomplished by a full program prerun. This can be done by

- pressing the RUN or STEP key when no program is running
- executing a RUN command when no program is running
- executing any GET or LOAD from a program
- executing a GET or LOAD command that tells program execution to begin (such as `LOAD "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, HP Instrument BASIC takes the effect of REDIM into account. If the COM areas could be matched by a REDIM, they are considered to be in agreement. When this happens, the treatment of the arrays in memory depends upon the program state. If the COM matching occurred because of a programmed LOADSUB, the arrays in memory keep their current shape. If the COM matching occurred for any other reason (such as RUN or programmed LOAD), the arrays

in memory are redimensioned to match the 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. If there is an OPTION BASE statement in the context, it must appear before COM statement. 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 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 Specifying a variable type implies that *all* variables that 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 specified otherwise.

COM statements (blank or labeled) in different contexts that 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 specified by the OPTION BASE statement.

Any LOADSUB that attempt to define or change COM areas while a program is running generates error 145.

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.

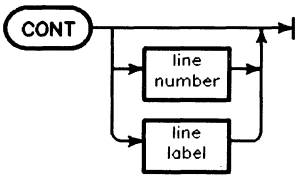
Labeled COM

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 that contains a labeled COM statement with the same label refers to the same labeled COM block.

CONT

CONT resumes execution of a paused program at the specified line. If no line is specified, execution resumes at the next line that would have executed if the program had not PAUSEd.

Syntax



Item	Description	Range
line number	integer constant identifying a program line; default = next program line	1 through 32 766
line label	name identifying a program line	any valid name

Example Commands

```
CONT
CONT 550
CONT Sort
```

Details

CONT can be executed by pressing CONTINUE or by executing a CONT command. Variables retain their current values whenever CONT executes. CONT causes the program to resume execution at the next statement that 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

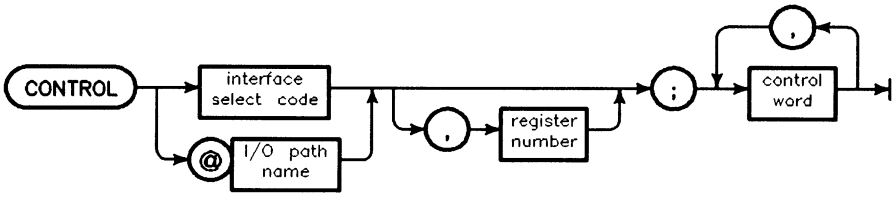
The behavior of this statement will be instrument specific. Refer to the instrument specific manual for more information.

Appendix C contains more information on registers for I/O path names, interfaces, and pseudo-select code 32.

CONTROL writes data to one of the following:

- hardware interface registers
- the internal table associated with an I/O path name

Syntax



Item	Description	Range
interface select code	numeric expression, rounded to an integer	1 through 32 (interface-dependent)
I/O path name	name assigned to a device, devices, mass storage file, or buffer	any valid name (see ASSIGN)
register number	numeric expression, rounded to an integer; default = 0	interface-dependent
control word	numeric expression, rounded to an integer	-2^{31} through $2^{31}-1$ (interface-dependent)

Example Statements

```
CONTROL @Rand_file,7;File_length  ! Write to a file.
CONTROL Interface,Register;Value  ! Write to a hardware interface.
CONTROL @Serial,3;9600            ! Set the baud rate of a serial interface.
```

Writing to File I/O Paths

I/O path names assigned to files have an association table that can be accessed as a set of registers.

CONTROL writes to this table, starting with the specified register and continuing in turn through the remaining registers until all control words are used. The number of control words must not exceed the number of registers available.

Register assignments can be found in the instrument-specific HP Instrument BASIC manual included with your instrument.

Writing to Hardware Interfaces

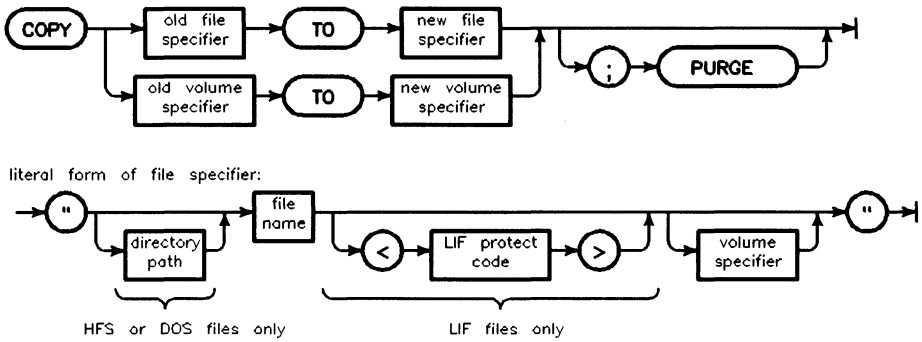
Control words are written to the interface registers, starting with the specified register number and continuing in turn through the remaining registers until all the control words are used. The number of control words must not exceed the number of registers available.

Register assignments can be found in the instrument-specific HP Instrument BASIC manual included with you instrument.

COPY

COPY copies individual files or entire disks. When an entire disk is copied, all old files on the destination disk are destroyed.

Syntax



Item	Description	Range
file specifier	string expression	(see drawing)
directory path	literal	(see MASS STORAGE IS)
file name	literal	depends on volume's format; 10 characters for LIF; 8 characters for DOS (short file name); (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 "new" TO "archive";PURGE

COPY "A:\DIR\FILE1" TO "B:\DIR\FILE2"
```

Details

The contents of the old file are copied into the new file, and a directory entry is created.

HP Instrument BASIC will not replace existing files unless you specify the PURGE option.

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 and the PURGE option is not specified.

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.

Using Wildcards with COPY ...
Using PURGE with COPY ...

Using Wildcards with COPY

If you are using a version of HP Instrument BASIC that supports wildcards, you can use them in file specifiers with COPY. You must first enable wildcard recognition using WILDCARDS. Refer to the keyword entry for WILDCARDS for more details.

You may use wildcards in both the source and destination of the COPY. If the wildcard specification for the source matches more than one file, then the destination must be a directory.

Note that HP Instrument BASIC handles the command

```
COPY "file_name" TO "dir_name"
```

in a different manner when wildcards are enabled than when they are disabled.

When wildcards are enabled, HP Instrument BASIC permits you to copy a file to a directory. It interprets the above command as make a copy of `file_name` and place that copy in a directory called `dir_name`.

When wildcards are disabled, HP Instrument BASIC interprets the above command as make a copy of `file_name` and place it in the *file* called `dir_name`. If a file or a directory already exists that uses the name `dir_name`, HP Instrument BASIC generates ERROR 54, Duplicate file name.

Using the PURGE Option

The PURGE option allows the COPY command to replace existing files.

HP Instrument BASIC interprets the command `COPY "file1" TO "file2"; PURGE` as copy the file `file1` to `file2`, replacing `file2` if it exists.

HP Instrument BASIC interprets the command

```
COPY "file_name" TO "dir_name"; PURGE
```

in different ways depending on whether wildcards are enabled or disabled.

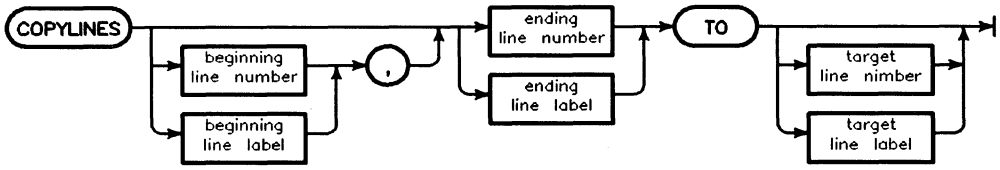
When wildcards are enabled, the preceding statement copies `file_name` into the directory `dir_name`. If a file with the name `file_name` already exists in that directory, COPY will replace it.

When wildcards are disabled, HP Instrument BASIC replaces the directory identified by `dir_name` with the file specified by `file_name`. This works only if `dir_name` is empty.

COPYLINES

COPYLINES copies contiguous program lines from one location to another.

Syntax



Item	Description	Range
beginning line number	integer constant identifying program line	1 to 32 766
beginning line label	name of a program line	any valid name
ending line number	integer constant identifying program line	1 to 32 766
ending line label	name of a program line	any valid name
target line number	integer constant identifying program line	1 to 32 766
target line label	name of a program line	any valid name

Example Commands

```
COPYLINES 1200 TO 3255
COPYLINES 10,120 TO 500
COPYLINES Label1,Label2 TO Label3
```

Details

If the beginning line identifier is not specified, only one line is copied.

The target line identifier will be the line number of the first line of the copied program segment. Copied lines are renumbered if necessary. Any lines that are "pushed down" to make room for the copied lines are renumbered as necessary.

Line number references to the copied code are updated as they would be using REN, with these exceptions: line number references in lines not being copied remain linked to the source lines rather than being renumbered; references to non-existent lines are renumbered as if the lines existed.

If there are any DEF FN or SUB statements in the copied code, the target line number must be greater than any existing line number.

If you try to copy a program segment to a line number *contained* in the segment, an error will be reported and no copying will occur.

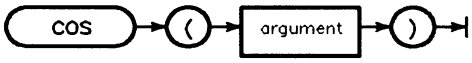
If the starting line number does not exist, the next line is used. If the ending line number does not exist, the previous line is used. If a line label doesn't exist, an error occurs and no copying occurs.

If an error occurs *during* a COPYLINES (for example, a memory overflow), the copy is terminated and the program is left partially modified.

COS

COS returns the cosine of the specified angle.

Syntax



Item	Description/Default	Range Restrictions
argument	numeric expression in current units of angle when INTEGER or REAL argument	absolute values less than 1.708 312 772 2 E+10 deg. or 2.981 568 244 292 04 E+8 rad. for INTEGER and REAL arguments

Example Statements

```
Cosine=COS(Angle)
PRINT COS(X+45)
```

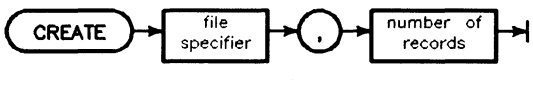
Details

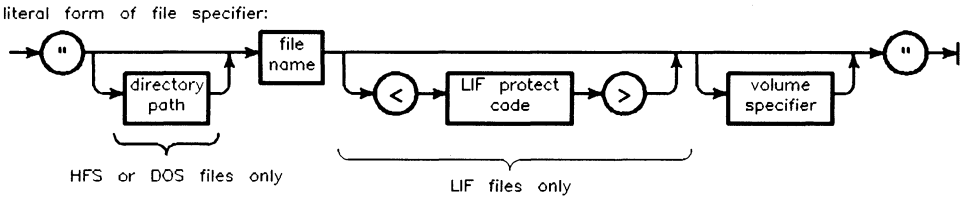
The angle mode set by RAD or DEG determines whether the angle is interpreted in degrees or radians. The angle mode is radians unless you specify degrees using the DEG statement.

CREATE

CREATE creates a DOS file.

Syntax





Item	Description	Range
file specifier	string expression	(see drawing)
number of records	numeric expression, rounded to an integer	1 through $2^{31} - 1$
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

```
CREATE File_spec$,N_records

CREATE "My_file",12
```

Details

The name of the newly created file must be unique within its directory. CREATE does not open the file; that is performed by ASSIGN. If there is 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. Files created with CREATE are extensible; refer to the following explanation of extensible files for details.

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

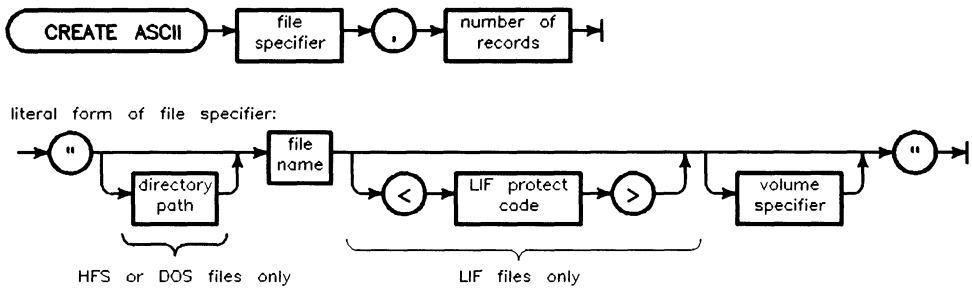
Files created with CREATE are "extensible". This means that the file system automatically allocates additional space for the file as new data is written to it.

CREATE ASCII

CREATE ASCII is supported for backward compatibility with older versions of other HP BASIC products. For new applications, use CREATE instead of CREATE ASCII.

CREATE ASCII creates a file using LIF ASCII format. LIF ASCII is a format used by older HP computers and disk drives.

Syntax



Item	Description	Range
file specifier	string expression	(see drawing)
number of records	numeric expression, rounded to an integer	1 through $(2^{31} - 1)/256$
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

```
CREATE ASCII "TEXT",100

CREATE ASCII "/Dir1/Dir2/AsciiFile",25

CREATE ASCII "C:\MYFILE.ASC",100
```

Details

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.

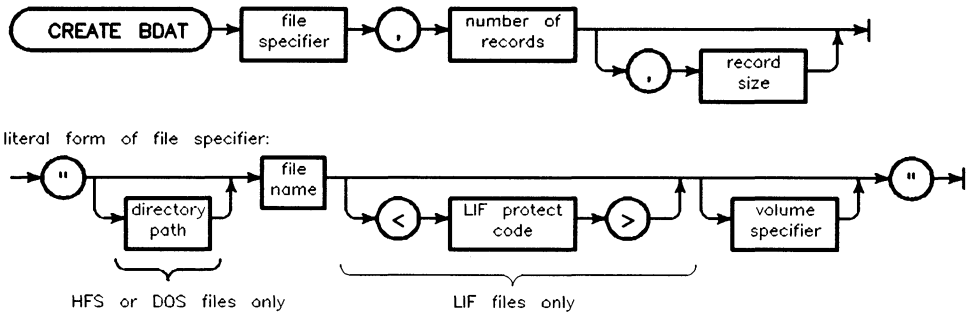
The physical records of an ASCII file have a fixed length of 256 bytes; logical records have variable lengths that are automatically determined when the OUTPUT, SAVE, or RE-SAVE statements are used.

CREATE BDAT

CREATE BDAT is supported for backward compatibility with older versions of other HP BASIC products. For new applications, use CREATE instead of CREATE BDAT.

CREATE BDAT creates a file using LIF BDAT (Binary DATa) format. LIF BDAT is a format used by older HP computers and disk drives.

Syntax



Item	Description	Range
file specifier	string expression	(see drawing)
number of records	numeric expression, rounded to an integer	1 through $(2^{31} - 769)/(\text{record size})$
record size	numeric expression, rounded to next even integer (except 1), which specifies bytes/record; default = 256	1 through 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
```

Details

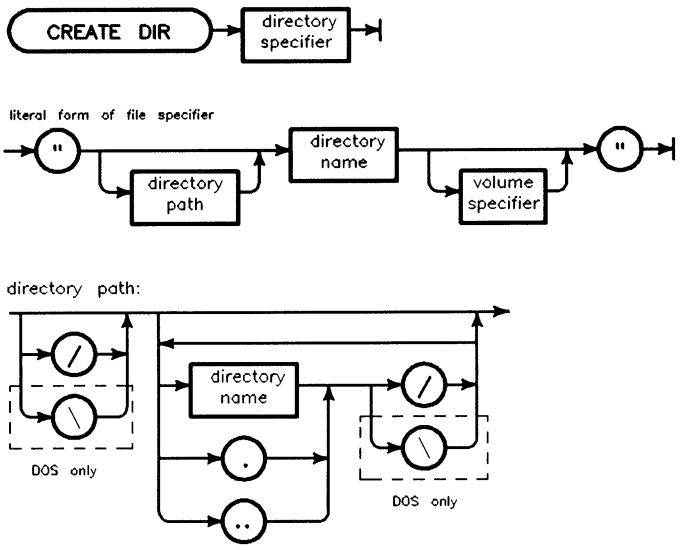
CREATE BDAT creates a new BDAT file and directory entry on the mass storage media. The name of the newly created BDAT file must be unique within its containing directory. CREATE BDAT does not open the file; that is performed by ASSIGN. In the event of an error, no directory entry is made and the file 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.

CREATE DIR

CREATE DIR creates the specified directory.

Syntax



Item	Description	Range
directory specifier	string expression	(see drawing)
directory path	literal	(see drawing)
directory name	literal	depends on volume's format; 8 characters for DOS (short file name); (see Glossary)
volume specifier	literal	(see MASS STORAGE IS)

Example Statements

```
CREATE DIR "WORK_DIR"

CREATE DIR "C:\DIR_1\DIR_2\MY_DIR"

CREATE DIR "Dir3/Dir4:,700"
```

Details

The name of the newly created directory must be unique within its parent directory.

If no directory path is included in the specifier for the new directory, the new directory is created within the current working directory (the directory specified in the latest MASS STORAGE IS statement).

CRT

CRT returns 1, the **device selector** of the CRT.

Syntax

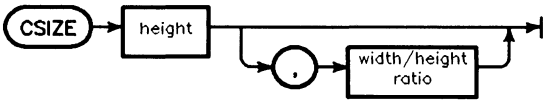
Example Statements

```
PRINTER IS CRT
ENTER CRT;Array$(*)
```

CSIZE

CSIZE sets the height and aspect ratio (width:height) of the character cell used by LABEL.

Syntax



Item	Description	Range
height	numeric expression; default = 5	—
width/height ratio	numeric expression; default = 0.6	—

Example Statements

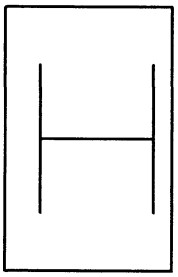
```
CSIZE 10

CSIZE 5,0.6

CSIZE Height,Width/Height
```

Details

At power-on, RESET, and GINIT, the height is 5 graphic-display-units (GDUs), and the aspect ratio is 0.6 (width = 3 GDUs). A negative number for either parameter inverts the character along the associated dimension.

The drawing below shows the relation between the character cell and a character.

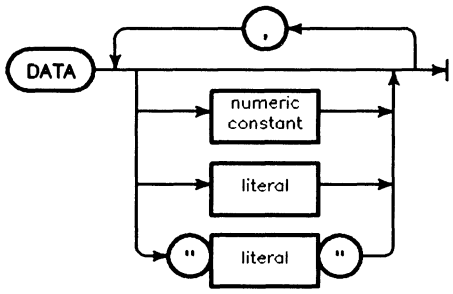
Character in a Character Cell



DATA

DATA statements contain in-line data that is read by READ statements.

Syntax



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(,)"
```

Details

A program or subprogram can contain any number of DATA statements at any location. When a program runs, 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 that 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 function similar to VAL. Error 32 results if the numeric data is not of the proper form (see VAL).

Unquoted literals must not contain quote marks (which delimit strings), commas (which delimit data items), or exclamation marks (which indicate the start of a comment). Leading

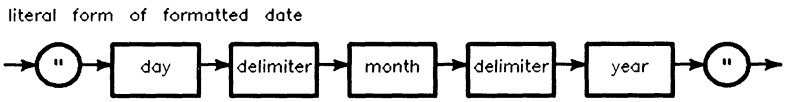
and trailing blanks are deleted from unquoted literals. Enclosing a literal in double quotes enables you to include any punctuation you wish, including embedded double quotes, which are represented by a set of two adjacent double quotes.

```
DATA "Say ""Hello"" to him." ! Embedded quotes.

DATA "Danger, power ON!" ! Space, comma, and ! in the data.
```

DATE

DATE converts a formatted date string into a numeric value (in seconds).

Syntax



Item	Description	Range
formatted date	string expression	(see drawing and text)
day	integer constant	1 through end-of-month
month	literal (lettercase ignored)	JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
year	integer constant	1900 through 2079
delimiter	any non-numeric character except the negative (minus) sign	

Example Statements

```
PRINT DATE("30 MAY 1987")

Days=(DATE(Day1$)-DATE(Day2$)) DIV 86400
```

Details

Specifying an invalid date, such as the thirty-first of February, will result in an error.

Leading blanks or non-numeric characters are ignored. ASCII spaces are recommended as delimiters between the day, month and year. However, any non-alphanumeric character, except the negative sign (minus sign), may be used as the delimiter.

DATE$

DATE$ formats a number of seconds into a string representing the formatted date (DD MMM YYYY).

Syntax

Item	Description	Range
seconds	numeric expression	−4.623 683 256 E+13 through 4.653 426 335 039 9 E+13

Example Statements

```
PRINT DATE$(TIMEDATE)

Day1$=DATE$(Event1)
```

Details

The date returned is in the form: DD MMM YYYY, where DD is the day of the month, MMM is the month mnemonic, and YYYY is the year.

The day is blank filled to two character positions. Single ASCII spaces delimit the day, month, and year.

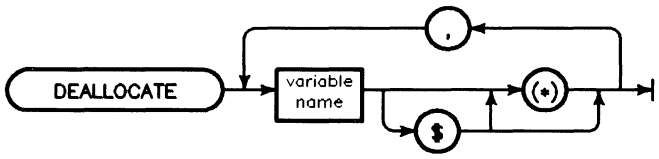
The first letter of the month is capitalized and the rest are lowercase characters.

Years less than the year 0 are expressed as negative years.

DEALLOCATE

DEALLOCATE deallocates memory space reserved by the ALLOCATE statement.

Syntax



Item	Description	Range
variable name	name of an array or string variable any	valid name

Example Statements

```
DEALLOCATE A$,B$,C$

DEALLOCATE Text$(*)

DEALLOCATE Array(*)
```

Details

Memory for ALLOCATEd variables is managed as a stack. As a result, the memory used for a particular variable might not become available as soon as it is DEALLOCATEd. For example, suppose you ALLOCATE memory for A, B, and C (in that order) and then DEALLOCATE B. The memory associated with B is not available until you DEALLOCATE C.

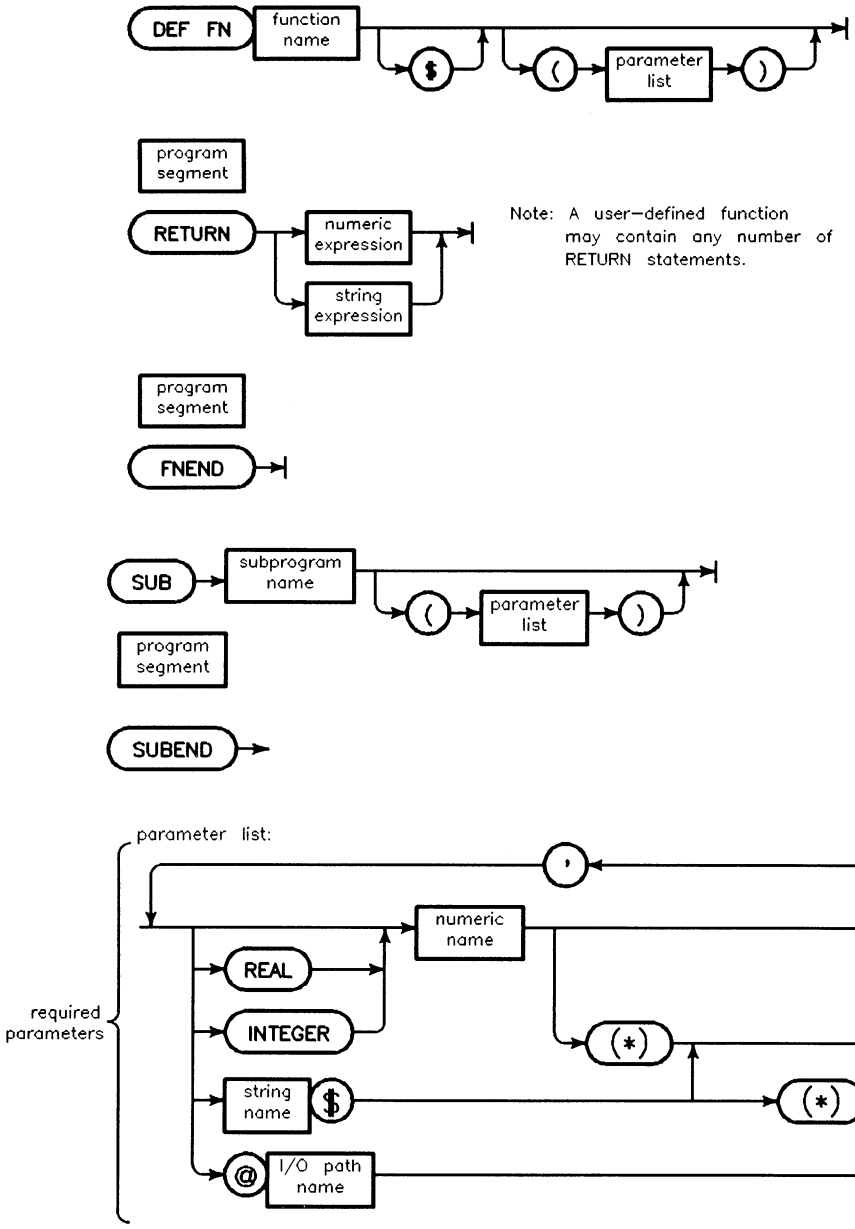
Strings and arrays must be deallocated completely. Deallocation of an array is requested by the (*) specifier.

Attempting to DEALLOCATE a variable that is not allocated in the current context results in an error. When DEALLOCATE is executed from the keyboard, deallocation occurs within the current context.

DEF FN

DEF FN indicates the beginning of a function subprogram. It also indicates whether the function is string or numeric and defines the formal parameter list.

Syntax



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

```
970  ! main program here.
980  END
990  !
1000 DEF FNNew$(String$)
1010 ! Additional statements
1020  RETURN Result$
1030  FNEND
```

Details

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 function's formal parameter list may not be declared in COM or other declaratory statements within the function. 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 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. If RETURN specifies a numeric expression, the value returned by the function is always a real number, never an integer.

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

DEG selects degrees as the current angle mode (unit of measure for angles).

Syntax

Example Statements

```
DEG
```

Details

Unless you set the angle mode to degrees, it is radians. HP Instrument BASIC sets the angle mode to radians when you do one of the following:

- start HP Instrument BASIC
- load or enter a new program
- execute SCRATCH, SCRATCH A, or SCRATCH C

Executing DEG sets the current angle mode to degrees. All functions that return an angle will return an angle in degrees. All operations with parameters representing angles will interpret the angle in degrees.

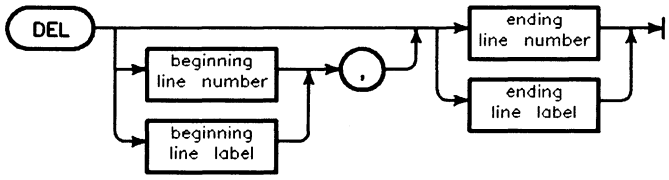
To set the angle mode to radians, use RAD.

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.

DEL

DEL deletes one or more program lines from memory.

Syntax



Item	Description	Range
beginning line number	integer constant identifying a program line	1 through 32 766
beginning line label	name of a program line	any valid name
ending line number	integer constant identifying a program line	1 through 32 766
ending line label	name of a program line	any valid name

Example Commands

```
DEL 15
DEL Start,Last
DEL Sort,32000
```

Details

DEL cannot be executed while a program is running. If DEL is executed while a program is paused, HP Instrument BASIC changes to the stopped state.

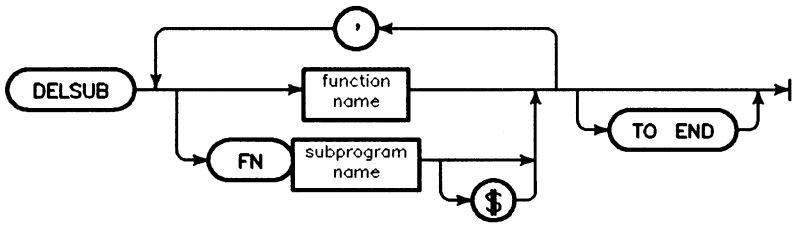
When a line is specified by a line label, the computer uses the lowest numbered line that has the label. If the label does not exist, error 3 is generated. An attempt to delete a non-existent program line is ignored when the line is specified by a line number. An error results if the ending line number is less than 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.

DELSUB

DELSUB deletes one or more subprograms or user-defined functions from memory.

Syntax



Item	Description	Range
function name	name of a user-defined function	any valid name
subprogram name	name of a SUB or CSUB subprogram	any valid name

Example Statements

```
DELSUB FNTrim$

DELSUB Mysub

DELSUB Process TO END

DELSUB Special1,Special3
```

Details

Subprograms being deleted do not need to be contiguous in memory. The order of the names in the deletion list does not have to agree with the order of the subprograms in memory. If there are subprograms with the same name, the one occurring first (lowest line number) is deleted.

The lines deleted begin with the line delimiting the beginning of the subprogram (SUB or DEF FN) and include the comments following the line delimiting the end of the subprogram (SUBEND or FNEND). If TO END is included, all subprograms following the specified subprogram are also deleted.

You cannot delete the following:

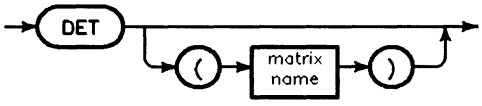
- busy subprograms (ones being executed)
- subprograms that are referenced by active ON-event CALL statements

If an error occurs while attempting to delete a subprogram, the subprogram is not deleted, and neither are any subprograms listed to the right of that subprogram in the DELSUB statement.

DET

DET returns the determinant of a matrix.

Syntax



Item	Description	Range
matrix name	name of a square, two-dimensional numeric array; default = (see text)	any valid name

Example Statements

```
Last_det=DET

PRINT DET(Matrix)
```

Details

If you do not specify a matrix, DET returns the determinant of the most recently inverted matrix. This value is not affected by context switching. If no matrix has been inverted since power-on, pre-run, SCRATCH or SCRATCH A, 0 is returned.

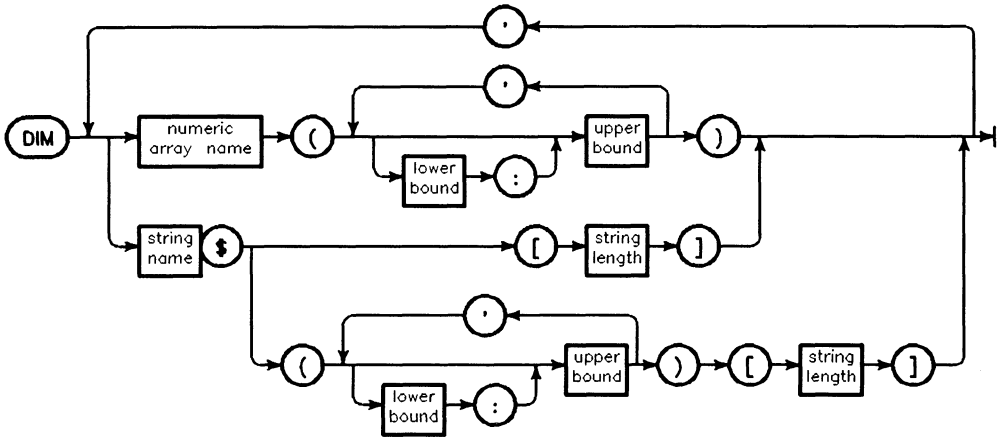
The determinant provides an indication of whether an inverse is valid. If the determinant of a matrix equals 0, then the matrix has no inverse. If the determinant is very small compared with the elements of its matrix, then the inverse may be invalid and should be checked.

DIM

DIM dimensions and reserves memory for

- REAL numeric arrays
- strings
- string arrays

Syntax



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 = OPTION BASE value (0 or 1)	−32 767 through + 32 767 (see "array" in Glossary)
upper bound	integer constant	−32 767 through +32 767 (see "array" in Glossary)
string length	integer constant	1 through 32 767

Example Statements

```
DIM String$[100],Name$(12)[32]

DIM Param(48,8,8,2,2,2)

DIM Array(-128:127,16)
```

Details

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, as long as they do not precede an OPTION BASE statement. Dimensioning occurs at pre-run or subprogram entry time. Dynamic run-time allocation of memory is provided by the ALLOCATE statement.

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—numeric and string—within any context is 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 the OPTION BASE value.

DISABLE

DISABLE disables all event-initiated branches currently defined, except for branches defined by these statements:

- ON ERROR
- ON TIMEOUT

Syntax

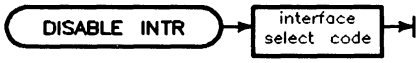
Example Statements

```
DISABLE
```

DISABLE INTR

DISABLE INTR disables interrupts from an interface by turning off the interrupt-generating mechanism on the interface.

Syntax



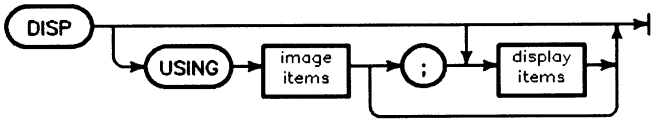
Example Statements

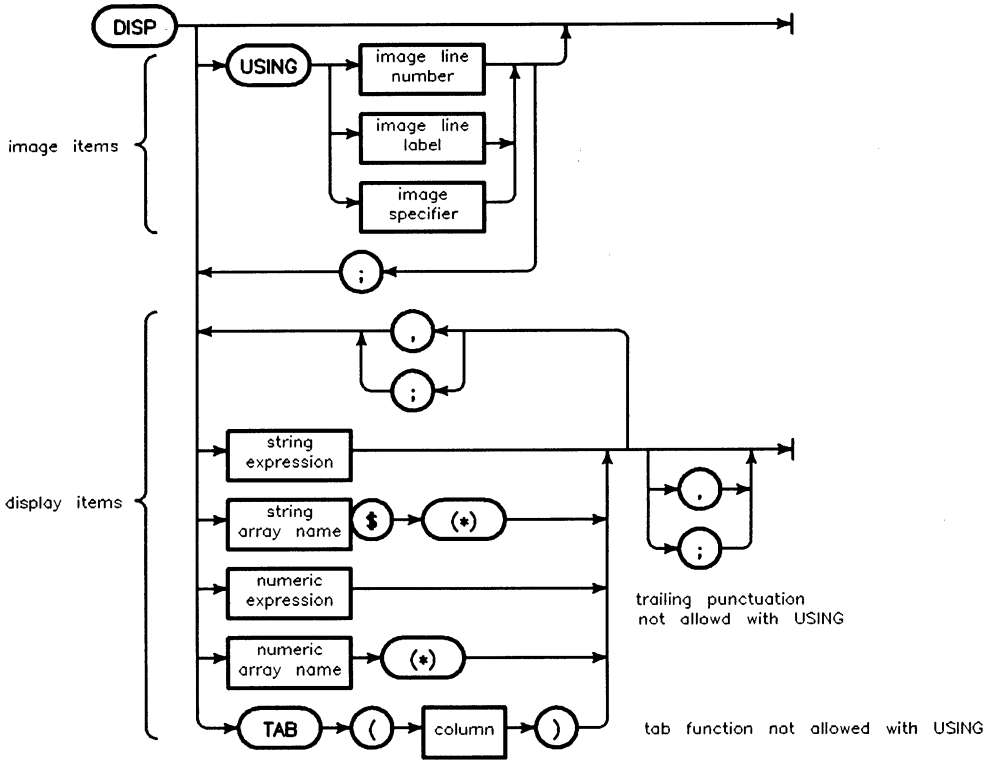
```
DISABLE INTR 7
DISABLE INTR Isc
```

DISP

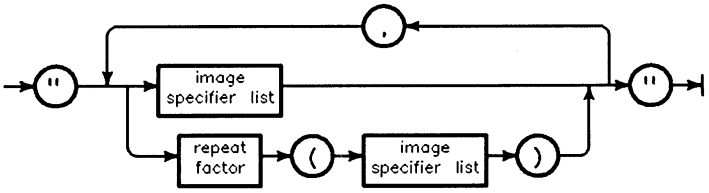
DISP prints the specified items on the display line. The display line is a single line near the bottom of the alpha window.

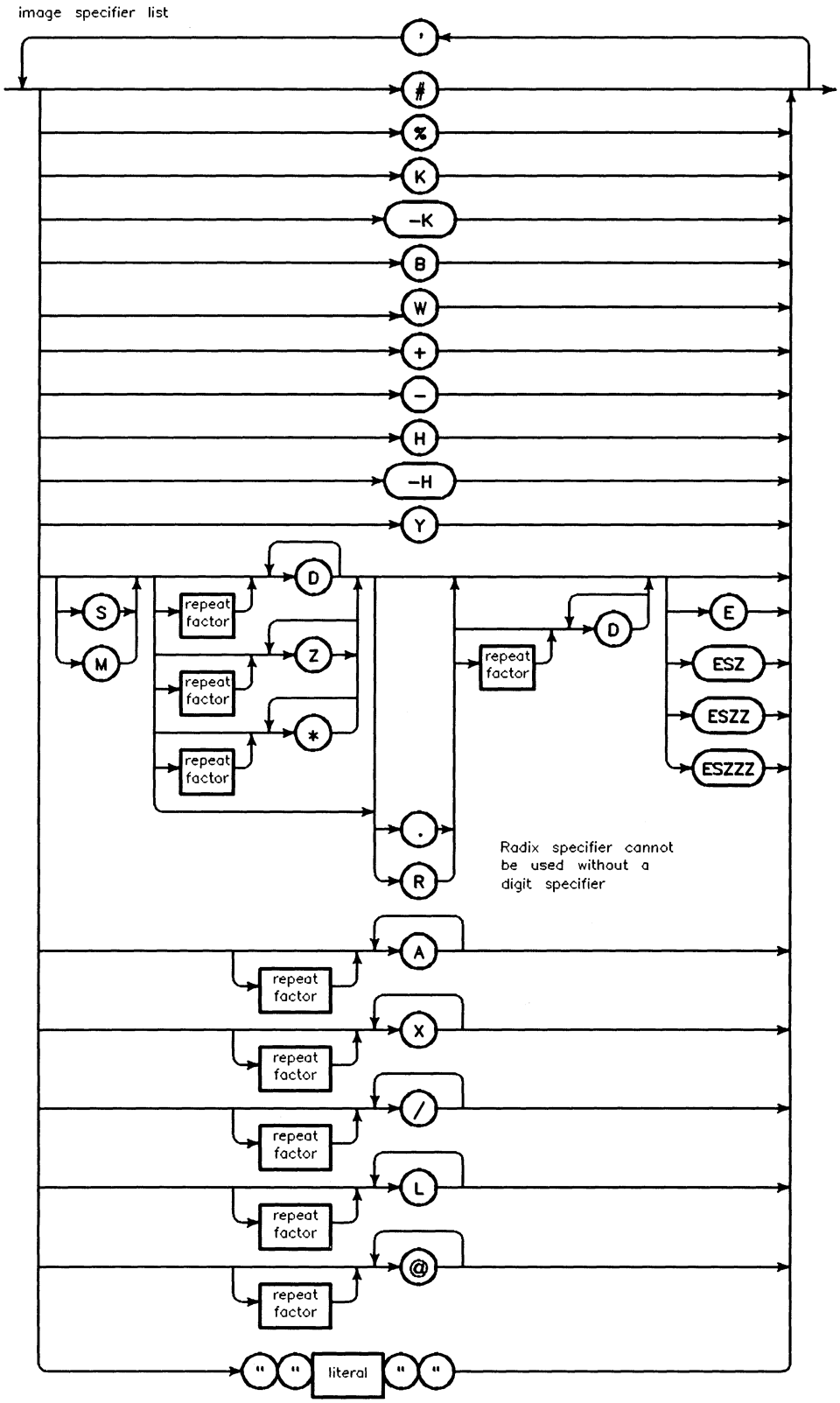
Syntax





literal form of image specifier





Item	Description	Range
image line number	integer constant identifying an IMAGE statement	1 through 32 766
image line label	name identifying an IMAGE statement	any valid name
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 through screenwidth
image specifier list	literal	(see diagram)
repeat factor	integer constant	1 through 32 767
literal	string constant composed of characters entered from the keyboard	quote mark not allowed

Example Statements

```
DISP Prompt$;
DISP TAB(5),First,TAB(20),Second
DISP
DISP Name$,Id;Code
DISP USING Form3;Item(1),Item(2)
DISP USING "5Z.DD";Money
```

Details

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). (Requires IO.)
−H	Same as H. (Requires IO.)
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. (Requires IO.)
.	Displays a decimal-point radix indicator.
R	Displays a comma radix indicator (European radix). (Requires IO.)
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.

Image Specifier	Meaning
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. (Requires IO.)
#	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. (Requires IO.)
−	Changes the EOL automatic sequence that normally follows the last display item to a single line-feed. (Requires IO.)
/	Sends a carriage-return and a line-feed to the display line.
L	Same as /.
@	Sends a form-feed to the display line.

DIV

DIV returns the integer portion of the quotient of the dividend and the divisor.

Syntax

Item	Description	Range
dividend	numeric expression	—
divisor	numeric expression	not equal to 0

Example Statements

```
Quotient=Dividend DIV Divisor
PRINT "Hours =";Minutes DIV 60
```

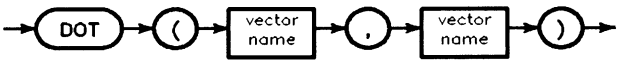
Details

DIV returns a REAL value unless both arguments are INTEGER. In the latter example, the returned value is INTEGER.

DOT

DOT returns the inner (dot) product of two numeric vectors.

Syntax



Item	Description	Range
vector name	name of a one-dimensional numeric array	any valid name

Example Statements

```
Res=DOT(Vec1,Vec2)
PRINT DOT(A,B)
```

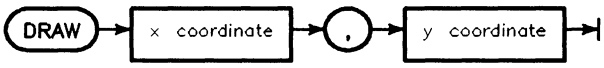
Details

The dot product is calculated by multiplying corresponding elements of the two vectors and then summing the products. The two vectors must be the same current size. If both vectors are INTEGER, the product will be an INTEGER. Otherwise, the product will be of type REAL.

DRAW

DRAW 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.

Syntax



Item	Description	Range
x coordinate	numeric expression, in current units	
y coordinate	numeric expression, in current units	

Example Statements

```
DRAW 10,90
DRAW Next_x,Next_y
```

Details

The X and Y coordinate information is interpreted according to the current unit-of-measure.

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. 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.

Graphics Transformations

The output of DRAW is affected by *only* these graphics transformations:

- scaling specified by WINDOW
- scaling specified by SHOW
- rotations specified by PIVOT

DROUND

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.

Syntax

Item	Description	Range
argument	numeric expression	—
number of digits	numeric expression, rounded to an integer	—

Example Statements

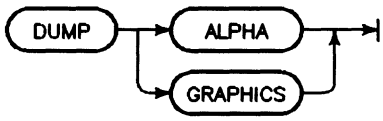
```
Test_real=DROUND(True_real,12)
PRINT "Approx. Volts =";DROUND(Volts,3)
```

DUMP

DUMP ALPHA copies the contents of the alphanumeric display to the default printer specified in the Windows Control Panel.

DUMP GRAPHICS copies the contents of the graphics display to the default printer specified in the Windows Control Panel. DUMP GRAPHICS will work with any printer that supports Windows graphics output.

Syntax



Item	Description	Range
source device selector	numeric expression, rounded to an integer; default = last CRT plotting device	(see Glossary)
destination device selector	numeric expression, rounded to an integer; default = DUMP DEVICE IS device	external interfaces and windows only (see Glossary)

Example Statements

```
DUMP ALPHA

DUMP GRAPHICS
```

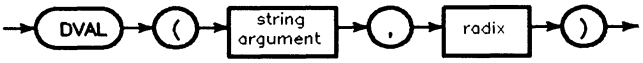
Details

To set the size of the output produced by DUMP GRAPHICS, use GESCAPE 39.

DVAL

DVAL converts a binary, octal, decimal, or hexadecimal character representation to a numeric value.

Syntax



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

```
Number=DVAL(String$,Radix)
PRINT DVAL("FF5900",16)
```

Details

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.

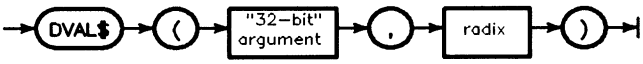
Radix	Base	String Range	String Length
2	binary	0 through 11111111111111111111111111111111	1 to 32 characters
8	octal	0 through 37777777777	1 to 11 characters
10	decimal	−2 147 483 648 through 2 147 483 647	1 to 11 characters
16	hexadecimal	0 through FFFFFFFF	1 to 8 characters

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$

DVAL$ converts a numeric value to a string of binary, octal, decimal, or hexadecimal digits.

Syntax



Item	Description	Range
"32-bit" argument	numeric expression, rounded to an integer	-2^{31} through $2^{31}-1$
radix	numeric expression, rounded to an integer	2, 8, 10, or 16

Example Statements

```
String$=DVAL$(Number,Radix)
PRINT DVAL$(Count MOD 256,2)
```

Details

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 through F.

Radix	Base	Range of Returned String	String Length
2	binary	00000000000000000000000000000000 through 11111111111111111111111111111111	32 characters
8	octal	00000000000 through 37777777777	11 characters
10	decimal	−2 147 483 648 through 2 147 483 647	11 characters
16	hexadecimal	00000000 through FFFFFFFF	8 characters

EDGE

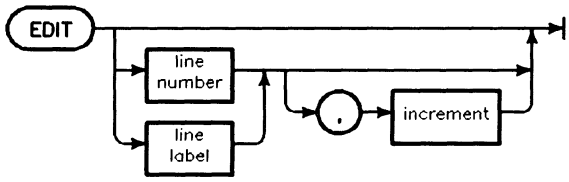
EDGE is a secondary keyword used to draw a border around the regions specified by these graphics keywords:

- IPLOT
- PLOT
- POLYGON
- RECTANGLE
- RPLOT

EDIT

EDIT activates the edit window, allowing you to enter a new program or modify a program already in memory.

Syntax



Item	Description	Range
line number	integer constant identifying program line; default (see Details)	1 through 32 766
line label	name of a program line	any valid name
increment	integer constant; default = 10	1 through 32 766

Example Statements

```
EDIT
EDIT Label2
EDIT 1000,5
```

Details

If the program was changed while paused, pressing CONTINUE will cause an error. Modifying a program moves it to the stopped state, making it impossible to continue.

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 LOAD 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. The increment will determine the interval between line numbers. If a program is in the computer, any increment provided is not used until lines are added to 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 IF ... THEN.

ENABLE

ENABLE reenables all event-initiated branches that were suspended by DISABLE.

Syntax

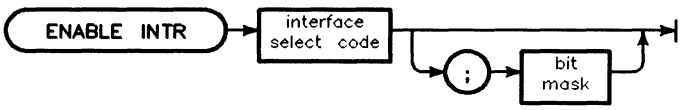
Example Statements

```
ENABLE
```

ENABLE INTR

ENABLE INTR enables the specified interface to generate an interrupt which can cause event-initiated branches.

Syntax



Item	Description	Range
interface select code	numeric expression, rounded to an integer	5, and 7 through 31
bit mask	numeric expression, rounded to an integer	−32 768 through +32 767

Example Statments

```
ENABLE INTR 7

ENABLE INTR Isc;Mask
```

Details

If a bit mask is specified, its value is stored in the interface's interrupt-enable register.

If no bit mask is specified, the previous bit mask for the select code is restored. A bit mask of all zeros is used when there is no previous bit mask.

END

END marks the end of the main program. Subprograms (if any) follow the END statement.

Example Statements

```
END
```

Syntax

Details

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 be located before the END statement. Subprograms and user-defined functions must be located after the END statement.

END IF

See IF ... THEN.

END LOOP

See LOOP.

END SELECT

See SELECT ... CASE.

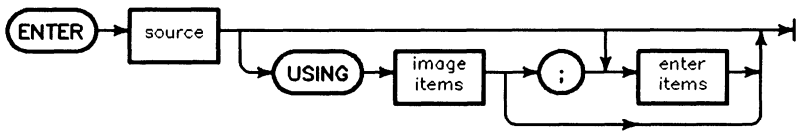
END WHILE

See WHILE.

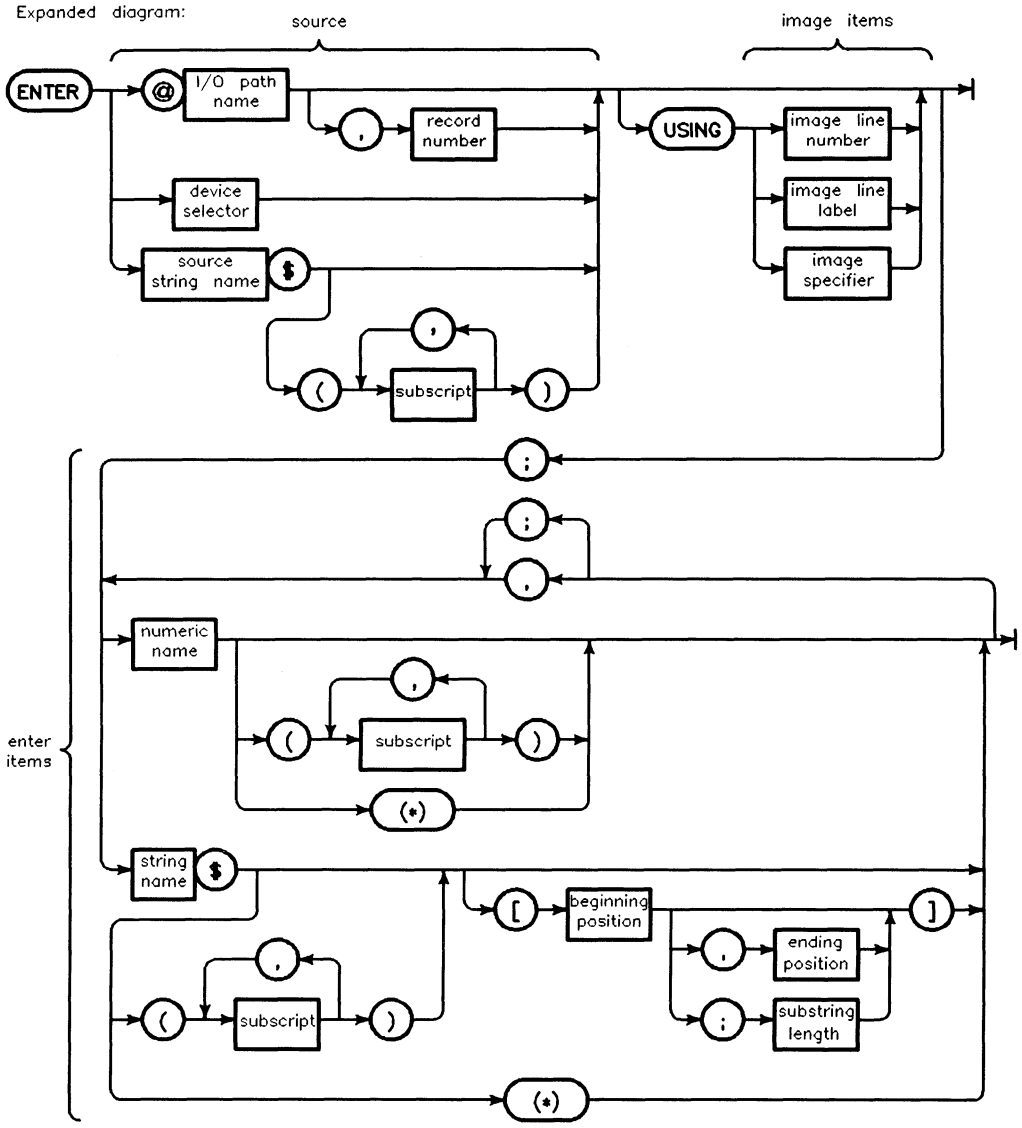
ENTER

ENTER is used to input data from a specified source and assign the values entered to variables.

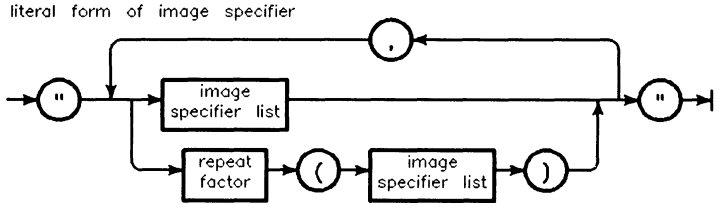
Syntax

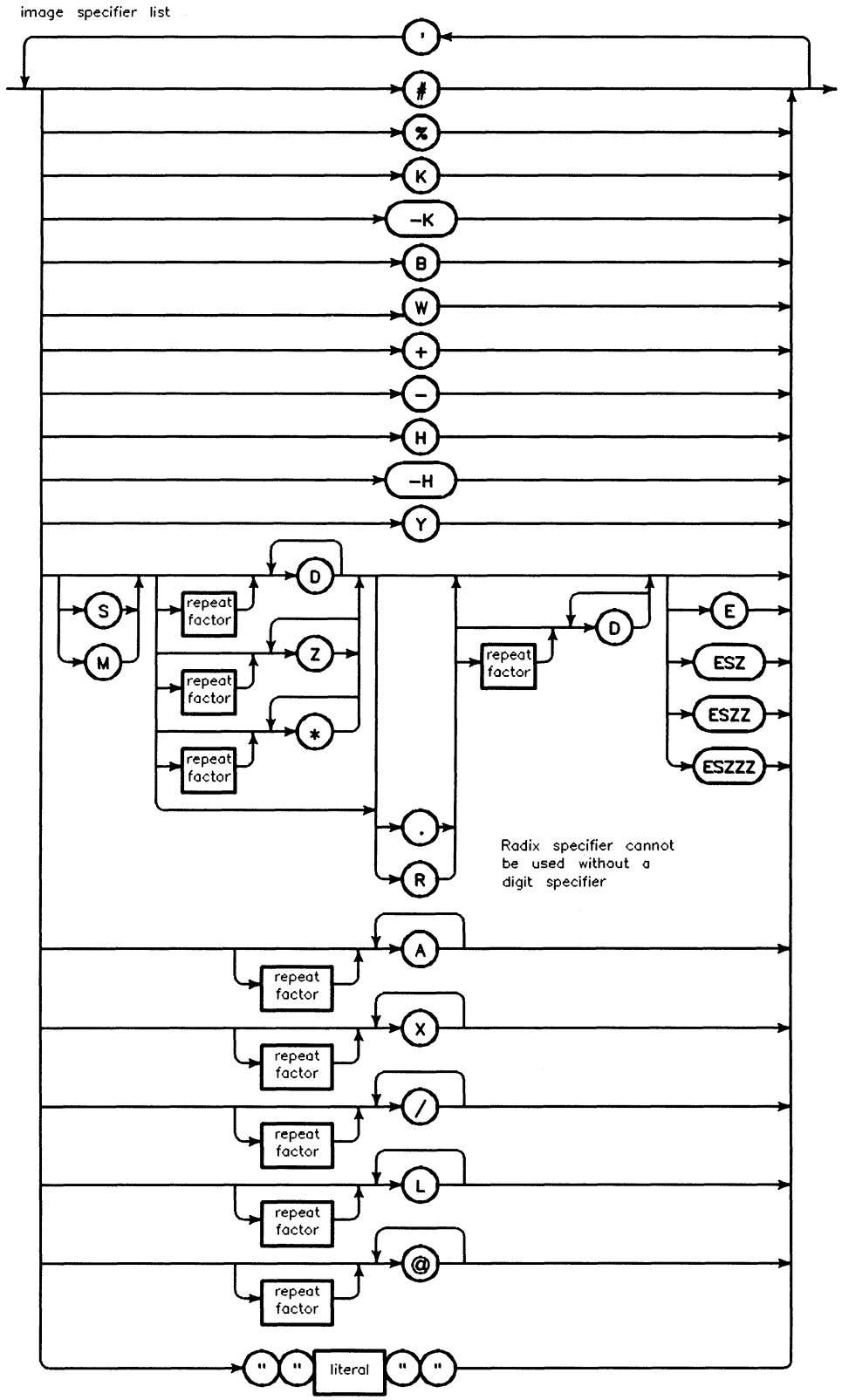


Expanded diagram:



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 through $2^{31}-1$
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 through +32 767 (see "array" in Glossary)
image line number	integer constant identifying an IMAGE statement	1 through 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 through 32 767 (see "substring" in Glossary)
ending position	numeric expression, rounded to an integer	0 through 32 767 (see "substring" in Glossary)
substring length	numeric expression, rounded to an integer	0 through 32 767 (see "substring" in Glossary)
image specifier list	literal	(see next drawing)
repeat factor	integer constant	1 through 32 767
literal	string constant composed of characters from the keyboard, including those generated using the ANY CHAR key	quote mark not allowed

Example Statements

```
ENTER 705;Number,String$

ENTER Device;X;Y;Z

ENTER Command$;Parameter

ENTER @File;Array(*)

ENTER @Random,Record USING 20;Text$(Line)

ENTER @Source USING Fmt5;Item(1),Item(2),Item(3)
```

Details

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 through 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 attributes specified in the ASSIGN statement are used only if the file is a BDAT or DOS file. Data read from a LIF ASCII file (a file created using CREATE ASCII) 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 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 DOS files. Random access is available for BDAT 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.)

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 and eight bytes are read 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.

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 LIF ASCII files (files create by CREATE ASCII. 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. (Requires IO.)
−H	Same as −K for strings; same as H for numbers. (Requires IO.)
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. (Requires IO.)
.	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. (Requires IO.)
E	Same action as 4D.
ESZ	Same action as 3D.

Image Specifier	Meaning
ESZZ	Same action as 4D.
ESZZZ	Same action as 5D.
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). (Requires IO.)
#	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. (Requires IO.)
−	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. (Requires IO)
/	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.

When entering FORMAT ON text into string variables, care should be taken to avoid unexpected interactions between terminating on dimensioned string length and termination on line feeds in the text. It is recommended that the string variable be dimensioned at least two characters longer than the text if it will be terminated by a carriage return/line feed.

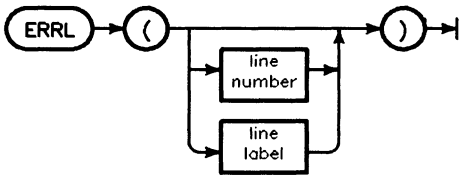
EOL

See ASSIGN and PRINTER IS statements.

ERRL

ERRL returns a value of 1 if the most recent error occurred in the specified line; otherwise, it returns 0.

Syntax



Item	Description	Range
line number	integer constant	1 through 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
```

Details

The specified line must be in the same context as the ERRL function, or an error will occur.

Data Communications

This function returns 0 for all data communications errors.

ERRLN

ERRLN returns the number of the program line on which the most recent error occurred. If no error has occurred, this function returns a value of 0.

Syntax

Example Statements

```
Bad_line=ERRLN
IF ERRLN=240 THEN GOSUB Fix_240
```

Details

ERRLN will return 0 if no error has occurred since one of these events:

- power-on
- prerun
- SCRATCH
- SCRATCH A
- LOAD
- GET

ERRM$

ERRM$ returns the text of the error message associated with the most recent program execution error.

Syntax

Example Statements

```
PRINT ERRM$

Em$=ERRM$

ENTER Em$;Error_num,Error_line
```

Details

The line number and error number returned in the ERRM$ string are the same as those used by ERRN and ERRL.

ERRM$ will return the null string if no error has occurred since one of these events:

- power-on
- prerun
- SCRATCH
- SCRATCH A
- LOAD
- GET

ERRN

ERRN returns the number of the most recent program execution error. If no error has occurred, a value of 0 is returned.

Syntax

Example Statements

```
IF ERRN=80 THEN Disc_out
DISP "Error Number = ";ERRN
```

Details

CLEAR ERROR resets ERRN to 0.

ERROR

See OFF ERROR and ON ERROR.

EXIT IF

See LOOP.

EXOR

EXOR returns a 1 or a 0 based on the logical exclusive OR of its arguments.

Syntax

Example Statements

```
Ok=First_pass EXOR Old_data
IF A EXOR Flag THEN Exit
```

Details

A non-zero value (positive or negative) is treated as a logical 1; only a zero is treated as a logical 0.

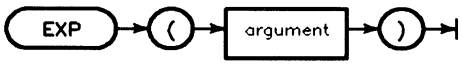
The EXOR function is summarized in this table.

A	B	A EXOR B
0	0	0
0	1	1
1	0	1
1	1	0

EXP

EXP raises e to the power of the argument. Internally, Napierian e=2.718 281 828 459 05.

Syntax



Item	Description/Default	Range Restrictions
argument	numeric expression	−708.396 418 532 264 through +709.782 712 893 383 8 for INTEGER and REAL arguments

Example Statements

```
Y=EXP(-X^2/2)

PRINT "e to the";Z;"=";EXP(Z)
```

FILL

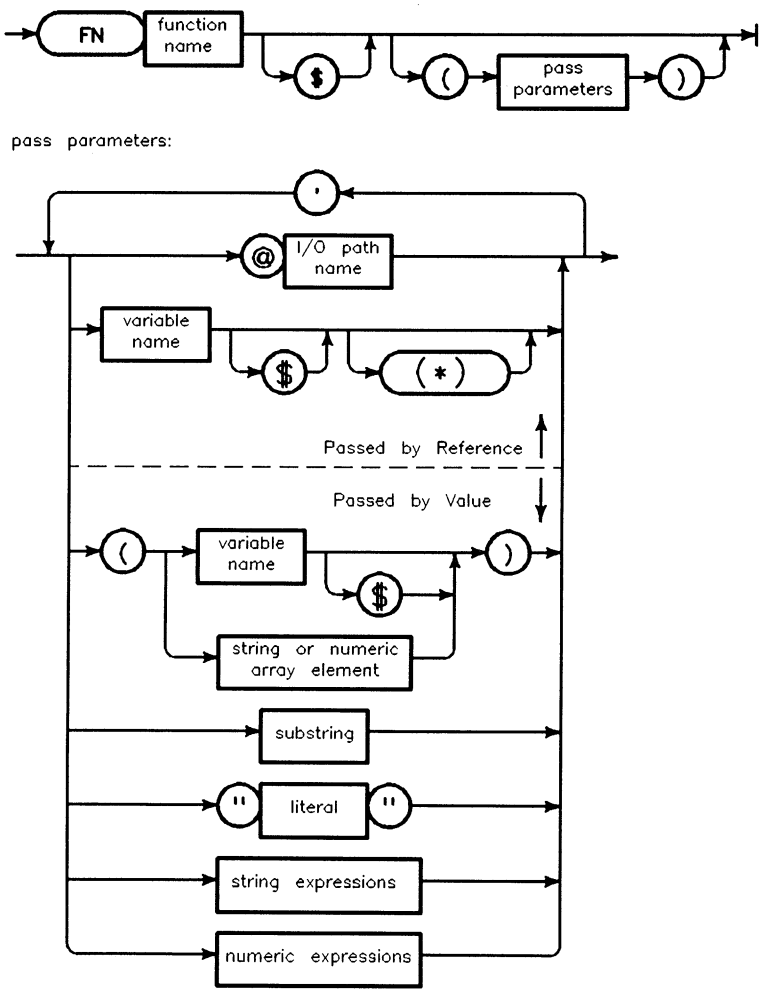
FILL is a secondary keyword used to create shading with these graphics keywords:

- IPLOT
- PLOT
- POLYGON
- RECTANGLE
- RPLOT

FN

FN 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.

Syntax



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, including those generated using the ANY CHAR key	—

Example Statements

```
PRINT X;FNChange(X)

Final$=FNStrip$(First$)

Parameter=FNProcess(Reference,(Value),@Path)

R=FNTrans(Item(Start+Offset),Lookup(*))
```

Details

A user-defined function may be invoked as part of a stored program line or as part of a statement executed from the keyboard. If you type the function name on the command line and then press ENTER or Return, the value returned by the function is displayed. The dollar sign suffix indicates that the returned value will be a string. User-defined functions are created with the DEF FN statement.

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

FNEND is the last statement of a function subprogram. Control is actually transferred back to calling context by a RETURN statement.

Example Statement

```
FNEND
```

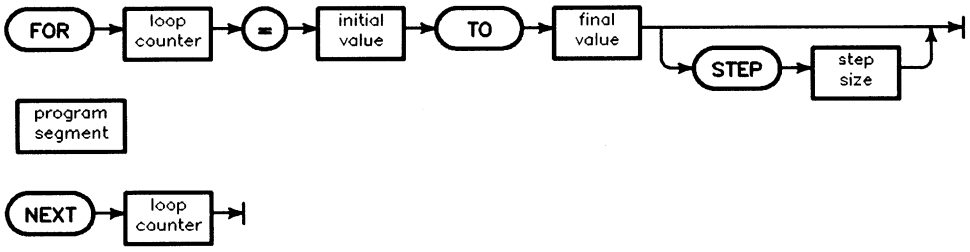
FORMAT

See the ASSIGN statement.

FOR ... NEXT

FOR ... NEXT defines a loop that is repeated until the loop counter passes a specified value.

Syntax



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 Statements

```
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
```

Details

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 you use a variable or expression for any of these values, you may change its value 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.

Note

Avoid using fractional values in a FOR ... NEXT statement. Remember that some REAL fractional numbers cannot be represented exactly by the computer. For example, if you use a step size of 0.1, the loop may execute a different number of times than you expect. Also, if the step size evaluates to 0, the loop may repeat infinitely. In either case no error message is given.

Refer to the "Numeric Computation" chapter of the *HP BASIC 6.2 Programming Guide* for detailed information on the effects of the computer's internal numeric representation.

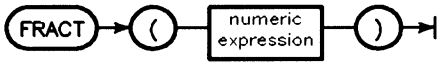
Nesting Constructs Properly

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.

FRACT

FRACT returns the "fractional part" of its argument. For REAL X, X=INT(X)+FRACT(X).

Syntax



Example Statements

```
PRINT FRACT(17/3)
Right_part=FRACT(Both_parts)
```

FRAME

FRAME draws a frame around the current graphics clipping area using the current pen number and line type. After drawing the frame, the current pen position coincides with the lower left corner of the frame, and the pen is down.

Syntax

Example Statement

```
FRAME
```

GCLEAR

GCLEAR clears the graphics display.

Syntax

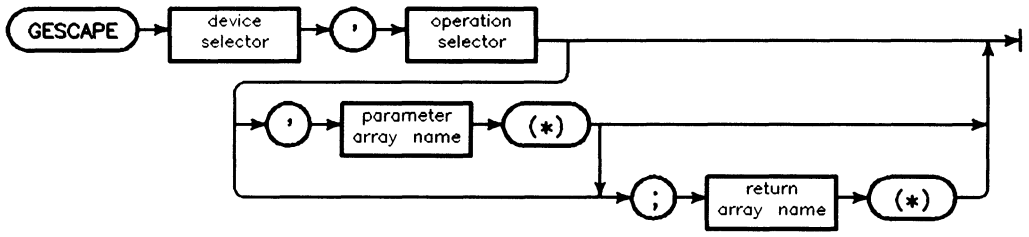
Example Statements

```
GCLEAR
```

GESCAPE

GESCAPE is used for communicating device-dependent graphics information. The type, size, and shape of the arrays must be appropriate for the requested operation.

Syntax



Item	Description	Range
device selector	numeric expression, rounded to an integer	(see Glossary)
operation selector	numeric expression, rounded to an integer	(device dependent, see Details)
parameter array name	name of array that has a specific rank and size, containing parameters necessary for executing request	any valid name
return array name	name of array that has a specific rank and size into which the returned parameters are placed	any valid name

Example Statements

```
GESCAPE Dev_select,operation
GESCAPE Dev_select,operation,array_in(*)
GESCAPE Dev_select,operation;array_out(*)
GESCAPE Dev_select,operation,array_in(*);array_out(*)
GESCAPE CRT,35 ! Bring the graphics output window to the top.
GESCAPE CRT,45 ! Bring the text output window to the top.
```

Details

Color Map Information

The number of entries in the color map can be determined with a GESCAPE operation selector of 1. The return array must be one-dimensional with at least one element.

The RGB values of the pens in the color map can be obtained through GESCAPE operation selector 2. The return array must be a two-dimensional three-column array with at least one row. The values returned are in the range of 0 to 1 and are multiples of 1/255 (one two-hundred fifty fifth) for color displays. The first row in the array always contains the values for PEN 0; if you want PEN 12, you must have at least thirteen rows in the array. Array filling occurs until either the array or the color map is exhausted.

Determining Hard Clip Limits and GSTORE Array Size

The hard clip limits of the current plotting device can be obtained by executing a GESCAPE with operation selector 3. The return array must be a one-dimensional INTEGER array with at least four elements. Values will be returned in the smallest resolvable units for that device. For the graph window, the units are pixels.

Operation selector 3 also returns information useful for GSTORE and GLOAD files. The fifth and sixth elements returned give the two array dimensions to use (in conjunction with the ALLOCATE statement) to GSTORE the contents of the specified display. This allows you to programmatically determine the size of the integer array to allocate for storing an image and thus avoid hardware-dependent code.

Drawing Mode Dominance

The normal drawing mode and the alternate drawing mode can be entered by using GESCAPE operation selectors 4 and 5, respectively.

For this discussion, the color of the area formed by the intersection of a newly plotted item and an existing item is the color X. In the normal drawing mode, X is the same color as the pen used to draw the newly plotted items. This is the color most recently set by PEN (for lines) or AREA (for area filling). In the alternate drawing mode, HP Instrument BASIC assigns the color X based on the color of the newly plotted item and the color of the existing item. The color assigned to X is neither the color of the newly plotted item nor the existing item; the color X is visible with either item as a background. Note that the exact color assigned to X varies depending on the video drivers and hardware used by your computer.

Drawing mode dominance affects the entire display. Thus in a windowing environment, all windows have the same drawing mode.

Summary of GESCAPE Operations

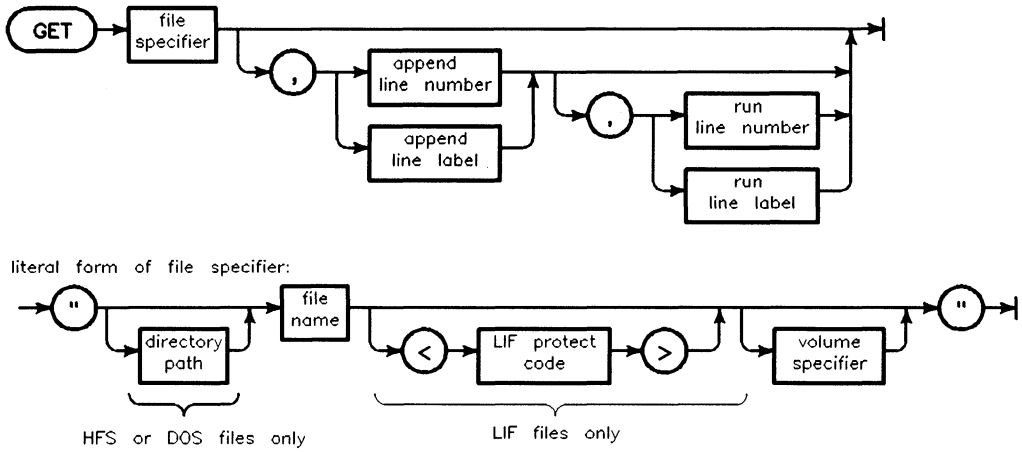
Functions Available Through GESCAPE

Operation Selector	Return Array (R) or Parameter Array (P)
1	(R) A(0): Number of entries in the color map
2	(R) A(0,0): Pen 0 red color map value A(0,1): Pen 0 green color map value A(0,2): Pen 0 blue color map value : A(15,0): Pen 15 red color map value A(15,1): Pen 15 green color map value A(15,2): Pen 15 blue color map value
3	(R) A(0): X minimum hard clip value A(1): Y minimum hard clip value A(2): X maximum hard clip value A(3): Y maximum hard clip value A(4): Rows required for GSTORE integer array A(5): Columns required for GSTORE integer array
4	Set normal drawing mode.
5	Set alternate drawing mode.

GET

GET reads the specified file that contains a program saved in text format. Typically, this file is created using SAVE or RE-SAVE.

Syntax



Item	Description	Range
file specifier	string expression	(see drawing)
append line number	integer constant identifying a program line	1 through 32 766
append line label	name of a program line	any valid name
run line number	integer constant identifying a program line	1 through 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)
LIF protect code	literal; first two non-blank characters are significant	> not allowed
volume specifier	literal	(see MASS STORAGE IS)

Example Statements

```
GET "George"

GET Progname$,Appndline,Runline

GET "C:\PROGS\MYPROG"
```

Details

In general, the file read by GET contains program lines written using SAVE. The file can be of any origin (such as a generic text editor) as long as it is of the same form as a SAVEd program. This means that the file contains ASCII characters representing program lines, each consisting of a line number followed by blank space and a valid program statement.

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 read, 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 that contain syntax errors are listed on the PRINTER IS device. Those erroneous lines that 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 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. 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.

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.

If you are using a version of HP Instrument BASIC that supports wildcards, you can use them in file specifiers with GET. You must first enable wildcard recognition using WILDCARDS. Refer to the keyword entry for WILDCARDS for details. Wildcard file specifiers used with GET must match one, and only one, file name.

GINIT

GINIT establishes a set of default values for system variables affecting graphics operations.

Syntax

Example Statements

```
GINIT
```

Details

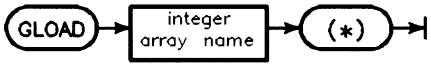
The following operations are performed when GINIT is executed:

```
AREA PEN 1
CLIP OFF
CSIZE 5,0.6
LDIR 0
LINE TYPE 1,5
LORG 1
MOVE 0,0
PDIR 0
PEN 1
PIVOT 0
GESCAPE CRT,4                    PEN MODE NORMAL
VIEWPORT 0,RATIO*100,0,100
WINDOW 0,RATIO*100,0,100
```

GLOAD

GLOAD loads the contents of an INTEGER array into the graphics window. The integer array is filled with the contents of a .BMP format bitmap file or previous graphics window contents stored using GSTORE.

Syntax



Item	Description	Range
integer array name	name of an INTEGER array.	any valid name

Example Statements

```
GLOAD Picture(*)

IF Flag THEN GLOAD Array(*)
```

Details

The integer array used to store the graphics window contents must be at least large enough to hold the entire contents, but larger arrays are acceptable. The minimum size for the array can be determined using GESCAPE. The following program segment illustrates the typical procedure for storing and loading the graphics window.

```
10    ! Insert statements here that draw in the graphics window.
20    ! Determine the size of the array required by GSTORE.
30    INTEGER Limits(0:5)
40    GESCAPE CRT,3;Limits(*)
50    ! ALLOCATE an integer array of the size required by GSTORE.
60    ALLOCATE INTEGER Array(Limits(4),Limits(5))
70    ! Store the contents of the graphics window.
80    GSTORE Array(*)
90    ! Insert statements here that alter the graphics window.
100   ! Load the previously stored image.
110   GLOAD Array(*)
120   ! Free the memory used to store the image.
130   DEALLOCATE Array(*)
```

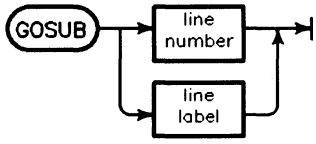
Note that the format of the data in the integer array is the same as a .BMP bitmap file. You can write the integer array to a file so that the GSTOREd image can be used by other programs. The following program segment illustrates the procedure for saving the graphics image in a file.

```
10    ! Insert statements here that draw in the graphics window.
20    ! Determine the size of the array required by GSTORE.
30    INTEGER Limits(0:5)
40    GESCAPE CRT,3;Limits(*)
50    ! ALLOCATE an integer array of the size required by GSTORE.
60    ALLOCATE INTEGER Array(Limits(4),Limits(5))
70    ! Store the contents of the graphics window.
80    GSTORE Array(*)
90    ! Create a file to hold the image.
90    CREATE "MYBMP.BMP",1
100   ASSIGN @File TO "MYBMP.BMP"
110   ! Write the array to the file.
120   OUTPUT @File;Array(*)
130   ! Close the file.
140   ASSIGN @File TO *
150   ! Free the memory used to store the image.
160   DEALLOCATE Array(*)
```

GOSUB

GOSUB 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.

Syntax



Item	Description	Range
line number	integer constant identifying a program line	1 through 32 766
line label	name of a program line	any valid name

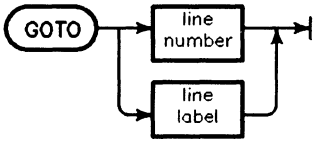
Example Statements

```
GOSUB 120

IF Numbers THEN GOSUB Process
```

GOTO

GOTO transfers program execution to the specified line. The specified line must be in the current context.

Syntax



Item	Description	Range
line number	integer constant identifying a program line	1 through 32 766
line label	name of a program line	any valid name

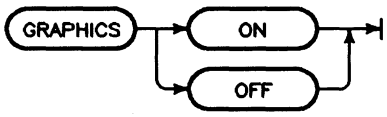
Example Statements

```
GOTO 550

GOTO Loop_start

IF Full THEN GOTO Exit
```

GRAPHICS

GRAPHICS shows or hides the graphics window.

Syntax



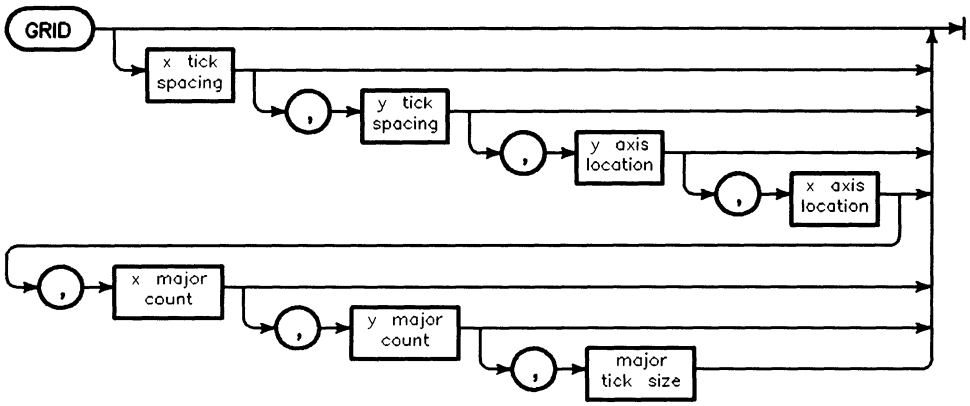
Example Statements

```
GRAPHICS ON
IF Flag THEN GRAPHICS OFF
```

GRID

GRID draws a full grid pattern. The pen is left at the intersection of the X and Y axes.

Syntax



Item	Description	Range
x tick spacing	numeric expression in current units; default = 0, no ticks	(see text)
y tick spacing	numeric expression in current units; default = 0, no ticks	(see text)
y axis location	numeric expression specifying the location of the y axis in x-axis units; default = 0	—
x axis location	numeric expression specifying the location of the x axis in y-axis units; default = 0	—
x major count	numeric expression, rounded to an integer, specifying the number of tick intervals between major tick marks; default = 1 (every tick is major)	1 through 32 767
y major count	numeric expression, rounded to an integer, specifying the number of tick intervals between major tick marks; default = 1 (every tick is major)	1 through 32 767
major tick size	numeric expression in graphic display units; default = 2	

Example Statements

```
GRID 10,10
GRID Xspace,Yspace,Xlocy,Ylocx,Xcount,Ycount,Major_size
```

Details

Grids are drawn with the current line type and pen number. Major tick marks are drawn as lines across the entire soft clipping area. A cross tick is drawn at the intersection of minor tick marks.

The X and Y tick spacing must not generate more than 32 768 grid marks in the clip area, or error 20 will be generated. Only the grid marks within the current clip area are drawn.

Applicable Graphics Transformations

GRID output is affected by only these graphics transformations:

- scaling specified by WINDOW
- scaling specified by SHOW plot scaling

GSTORE

GSTORE stores the current contents of the graphics window in an integer array. The integer array can be subsequently loaded into the graphics window using GLOAD.

Syntax

Item	Description	Range
integer array name	name of an INTEGER array	any valid name

Example Statements

```
GSTORE Picture(*)
IF Final THEN GSTORE A(*)
```

Details

The integer array used to store the graphics window contents must be at least large enough to hold the entire contents, but larger arrays are acceptable. The minimum size for the array can be determined using GESCAPE. The following program segment illustrates the typical procedure for store and loading the graphics window.

```
10    ! Insert statements here that draw in the graphics window.
20    ! Determine the size of the array required by GSTORE.
30    INTEGER Limits(0:5)
40    GESCAPE CRT,3;Limits(*)
50    ! ALLOCATE an integer array of the size required by GSTORE.
60    ALLOCATE INTEGER Array(Limits(4),Limits(5))
70    ! Store the contents of the graphics window.
80    GSTORE Array(*)
90    ! Insert statements here that alter the graphics window.
100   ! Load the previously stored image.
110   GLOAD Array(*)
120   ! Free the memory used to store the image.
130   DEALLOCATE Array(*)
```

Note that the format of the data in the integer array is the same as a .BMP bitmap file. You can write the integer array to a file so that the GSTOREd image can be used by other programs.

The following program segment illustrates the procedure for saving the graphics image in a file.

```
10    ! Insert statements here that draw in the graphics window.
20    ! Determine the size of the array required by GSTORE.
30    INTEGER Limits(0:5)
40    GESCAPE CRT,3;Limits(*)
50    ! ALLOCATE an integer array of the size required by GSTORE.
60    ALLOCATE INTEGER Array(Limits(4),Limits(5))
70    ! Store the contents of the graphics window.
80    GSTORE Array(*)
90    ! Create a file to hold the image.
90    CREATE "MYBMP.BMP",1
100   ASSIGN @File TO "MYBMP.BMP"
110   ! Write the array to the file.
120   OUTPUT @File;Array(*)
130   ! Close the file.
140   ASSIGN @File TO *
150   ! Free the memory used to store the image.
160   DEALLOCATE Array(*)
```

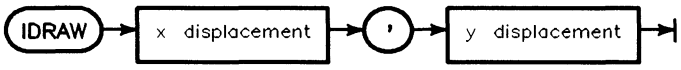
IDN

See MAT.

IDRAW

IDRAW draws a line from the current pen position to a position calculated by adding the X and Y displacements to the current pen position.

Syntax



Item	Description	Range
x displacement	numeric expression in current units	—
y displacement	numeric expression in current units	—

Example Statements

```
IDRAW X+50,0

IDRAW Delta_x,Delta_y
```

Details

The X and Y displacement information is interpreted according to the current unit-of-measure.

The line drawn by IDRAW is clipped at the current clipping boundary.

IDRAW updates the logical pen position at the completion of the IDRAW statement and leaves the pen down.

If none of the line is inside the current clipping limits, the pen is not moved, but the logical pen position is updated.

Graphics Transformations

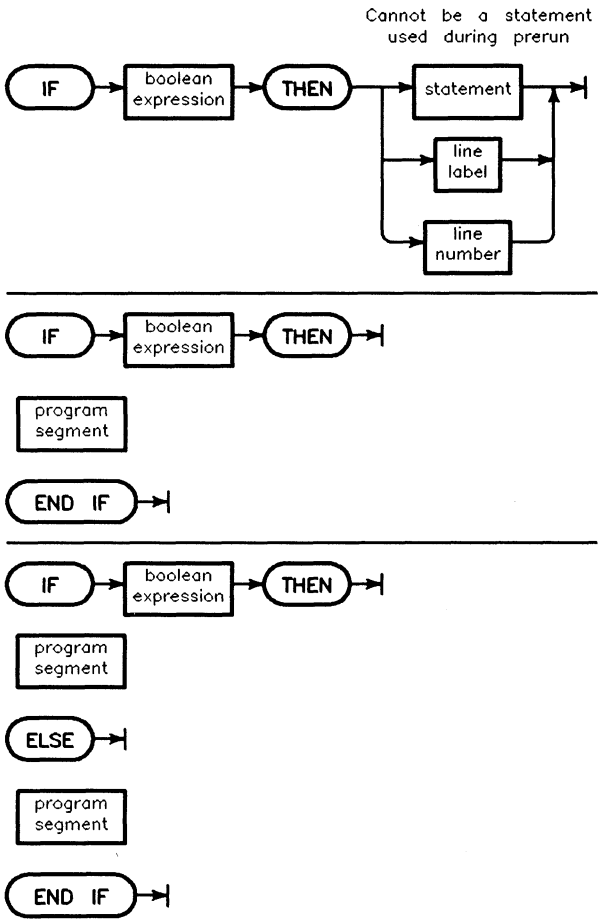
The output of IDRAW is affected by only these graphics transformations:

- scaling specified by WINDOW
- scaling specified by SHOW
- rotations specified by PIVOT

IF . . . THEN

IF . . . THEN provides conditional branching.

Syntax



Item	Description	Range
Boolean expression	numeric or string expression; evaluated as true if non-zero and false if zero	—
statement	a programmable statement	(see following list)
line label	name of a program line	any valid name
line number	integer constant identifying a program line	1 through 32 766
program segment	any number of contiguous program lines not containing the beginning or end of a main program or subprogram.	—

Example Statements

```
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
```

Details

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	END	IF	REM
CASE ELSE	END IF	IMAGE	REPEAT
COM	END LOOP	INTEGER	SELECT
COMPLEX	END SELECT	LOOP	SUB
DATA	END WHILE	NEXT	SUBEND
DEF FN	EXIT IF	OPTION BASE	UNTIL
DIM	FNEND	REAL	WHILE
ELSE	FOR		

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.

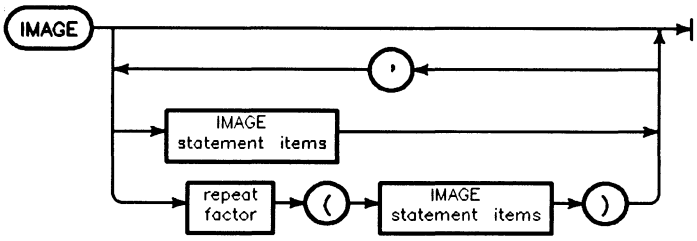
Branching into an IF ... THEN construct (such as with a GOTO) results in a branch to the program line following the END IF when the ELSE statement is executed.

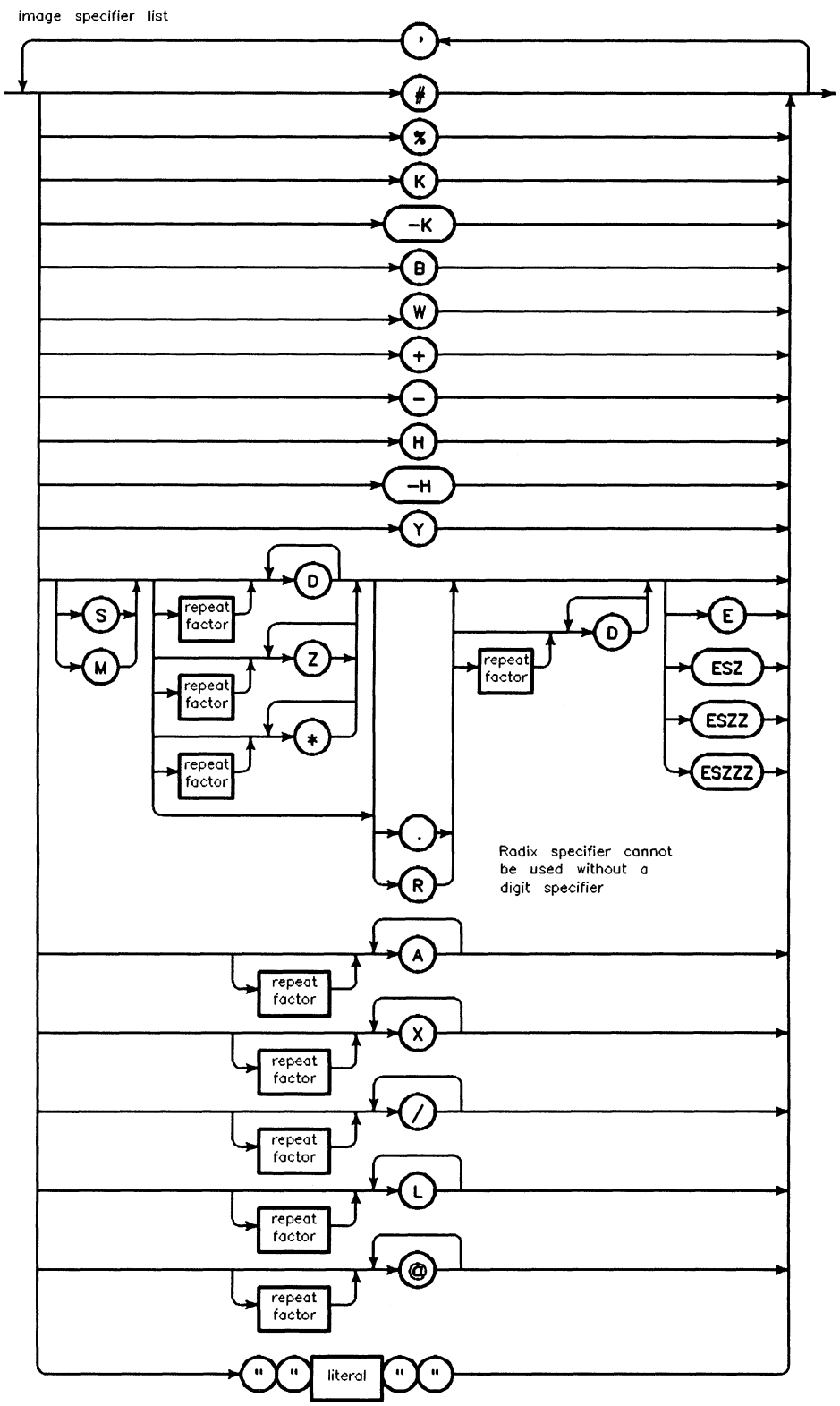
The prohibited statements listed above are allowed in multiple-line IF ... THEN constructs. However, these statements are not executed conditionally. The exceptions are other IF ... THEN statements or constructs such as FOR ... NEXT, REPEAT ... UNTIL, etc. These are executed conditionally, but need to be properly nested. To be properly nested, the entire construct must be contained in one program segment.

IMAGE

IMAGE statements specify special codes (image specifiers) for formatting data for use with various I/O statements. These image specifiers can also be included after the secondary keyword USING within the I/O statements.

Syntax





Item	Description	Range
IMAGE statement items	literal	(see drawing)
repeat factor	integer constant	1 through 32 767
literal	string composed of characters from the keyboard, including those generated using the ANY CHAR key.	quote mark not allowed

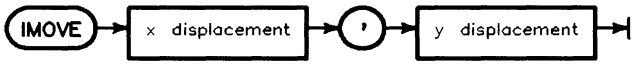
Example Statements

```
IMAGE 4Z.DD,3X,K,/

IMAGE "Result = ",SDDDE,3(XX,ZZ)

IMAGE #,B
```

IMOVE

IMOVE moves the graphics pen an incremental distance from the current position without drawing a line.

Syntax



Item	Description	Range
x displacement	numeric expression in current units	—
y displacement	numeric expression in current units	—

Example Statements

```
IMOVE X+50,0

IMOVE Delta_x,Delta_y
```

Details

IMOVE updates the logical position of the graphics pen, by adding the X and Y displacements to the current logical pen position. The pen is raised before it moves and remains up after the move. The X and Y displacements are interpreted according to the current unit-of-measure.

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 is moved the specified displacement.

Graphics Transformations

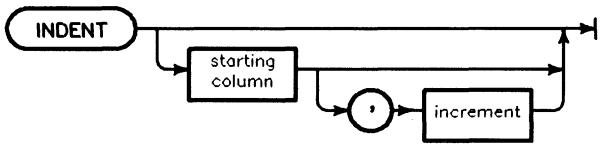
The output of IMOVE is affected by *only* these graphics transformations:

- scaling specified by WINDOW
- scaling specified by SHOW
- rotations specified by PIVOT

INDENT

INDENT indents program lines in the edit window to reflect the program's structure and nesting.

Syntax



Item	Description	Range
starting column	integer constant; default = 7	0 through Screen Width−8
increment	integer constant; default = 2	0 through Screen Width−8

Example Commands

```
INDENT
INDENT 8,4
```

Details

The starting column specifies the column in which the first character of the first statement of each context appears. The increment specifies the number of spaces that the beginning of the lines move to the left or right when the nesting level of the program changes. Note that a line label may override the indentation computed for a particular line. The INDENT command does not move comments which start with an exclamation point, but it does move comments starting with REM. However, if a program line is moved to the right a comment after it may have to be moved to make room for it. In both of these cases (line labels and comments), the text moves only as far as is necessary; no extra blanks are generated.

Indenting a program may cause the length of some of its lines to become longer than the machine can list. This condition is indicated by the presence of an asterisk after the line numbers of the lines which are overlength. If this occurs, the program will run properly, STORE properly and LOAD properly. If the total length of a line exceeds 256 characters, you cannot do a SAVE, then a GET. Doing an INDENT with smaller values will alleviate this problem.

Indentation occurs after the following statements:

FOR	REPEAT
LOOP	WHILE
SUB	SELECT
IF ... THEN[1]	DEF FN

[1]This is only true for IF..THEN statements where the THEN is followed by an end-of-line or an exclamation point.

The following statements cause a one-line indentation reversal; that is, indentation is reversed for these statements but reindented immediately after them:

CASE	EXIT IF
CASE ELSE	FNEND
ELSE	SUBEND

Indentation is reversed before the following statements:

END IF	END WHILE
END LOOP	NEXT
END SELECT	UNTIL

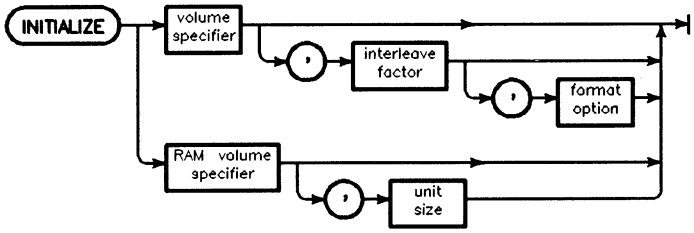
Indentation remains the same from line to line for all other statements.

Improperly matched nesting will cause improper indentation. Deeply nested constructs may cause indentation to exceed the width of the edit window.

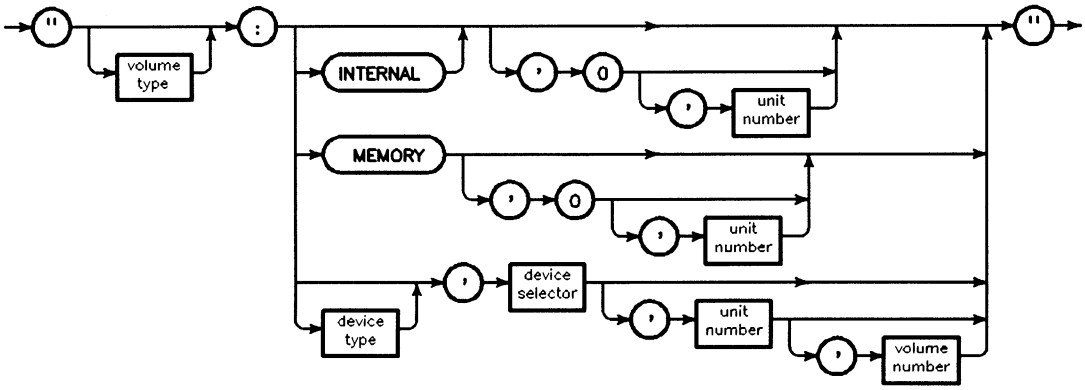
INITIALIZE

This statement prepares ("formats") mass storage media and places a LIF (Logical Interchange Format) directory on the media. When INITIALIZE is executed, *any existing files on the media are destroyed.*

Syntax



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
RAM volume specifier	string expression	(see drawing)
unit size	numeric expression, rounded to an integer; specifies number of 256-byte sectors;	4 thru 32 767 memory-dependent
volume type	selects media format; default = LIF	DOS or LIF
unit number	integer constant; default = 0	0 through 255 (device-dependent)
device type	literal	See instrument-specific HP Instrument BASIC for the device type supported in this instrument.
device selector	integer constant	(see Glossary)

Example Statements

```
INITIALIZE ":INTERNAL"
INITIALIZE Disc$,2
INITIALIZE ":,700",0,4
INITIALIZE ":MEMORY,0",Sectors
```

Details

Any media used by the computer must be initialized before its *first* use. Initialization creates a new LIF directory, 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, it *cannot* alone be used to format an HFS disc; it will still, however, scan the volume for bad sectors.

To format an HFS volume on an HP BASIC Workstation, use the System Disc Utility (DISC_UTIL, which calls the "Mkhfs" compiled subprogram to place an HFS-format directory on the disc volume). See the "BASIC Utilities Library" chapter of *Installing, and Maintaining the BASIC System* for instructions on using this utility.

On HP-UX, use the command **`newfs`** command. See the *HP-UX Reference*, newfs(1m) entry.

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`

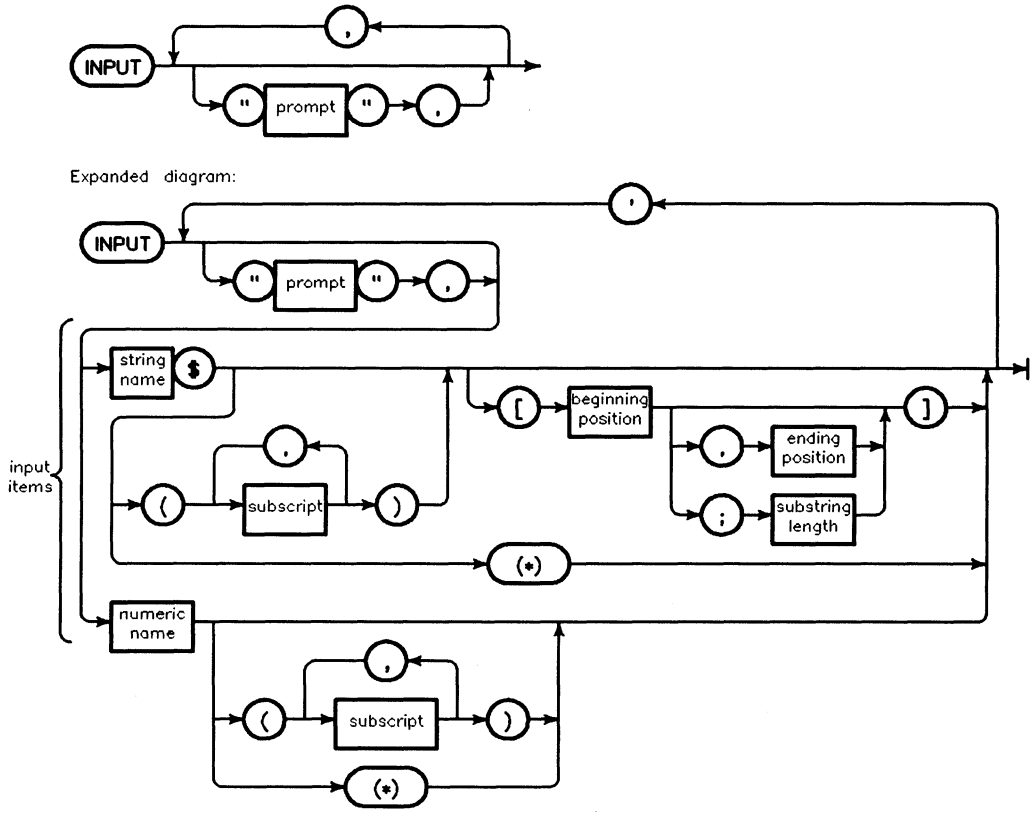
Initializing the volume to 0 sectors removes it from memory.

Memory volumes are allocated in a mark and release stack. What this means is, you get the memory back only when other subsequently created memory volumes have been reclaimed. You can re-initialize a removed memory volume in its original space provided the newly allocated space is no larger than the original space that was allocated. Otherwise, new space will be allocated for it.

INPUT

INPUT is used to assign keyboard input to program variables.

Syntax



Item	Description/Default	Range Restrictions
prompt	a literal composed of characters from the keyboard, including those generated using the ANY CHAR key;default = question mark	—
string name	name of a string variable	any valid name
subscript	numeric expression, rounded to an integer	−32 768 through +32 767 (see "array" in Glossary)
beginning position	numeric expression, rounded to an integer	1 through +32 767 (see "substring" in Glossary)
ending position	numeric expression, rounded to an integer	0 through +32 767 (see "substring" in Glossary)
substring length	numeric expression, rounded to an integer	0 through +32 767 (see "substring" in Glossary)
numeric name	name of a numeric variable	any valid name

Example Statements

```
INPUT "Name?",N$,"ID Number?",Id
INPUT "Enter 3 numbers",V(1),V(2),V(3)
INPUT "",String$[1;10]
INPUT Array(*)
```

Details

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 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 typing the final data item. 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. Entering no data items and pressing CONTINUE, ENTER, EXECUTE, Return, or STEP retains the old values for all remaining variables in the list.

If you click on Cont in the control pad or press ENTER or Return to end the data input, program execution continues at the next program line. If you click on [STEP] in the control pad, 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 you click on Cont in the control pad or press ENTER or Return.)

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).

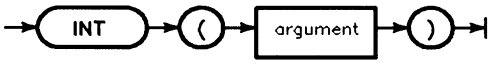
Live keyboard operations are not allowed while an INPUT is awaiting data entry.

Keyboard-initiated events are deactivated during an INPUT statement. Errors do not cause an ON ERROR branch. If an input response results in an error, reentry begins with the variable which would have received the erroneous response.

INT

INT returns the greatest integer which is less than or equal to its argument. The result will be of the same numeric type (REAL or INTEGER) as the argument.

Syntax



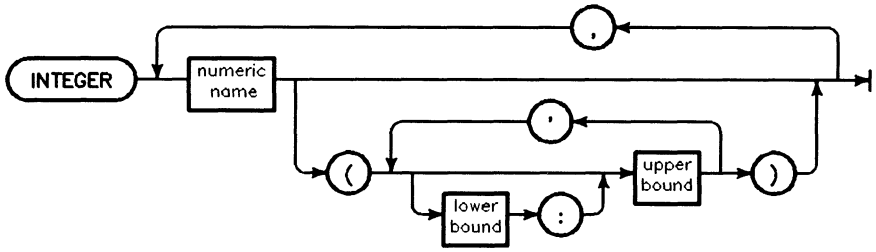
Example Statements

```
Whole=INT(Number)
PRINT "Integer portion =";INT(X)
```

INTEGER

INTEGER declares integer variables, dimensions integer arrays, and allocates memory for the variables and arrays.

Syntax



Item	Description	Range
numeric name	name of a numeric variable	any valid name
lower bound	integer constant; default = OPTION BASE value (0 or 1)	−32 767 through +32 767 (see "array" in Glossary)
upper bound	integer constant	−32 767 through +32 767 (see "array" in Glossary)

Example Statements

```
INTEGER I,J,K
INTEGER Array(-128:255,4)
```

Details

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—numeric and string—within any context is 16 777 215 bytes (or limited by the amount of available memory, whichever is less).

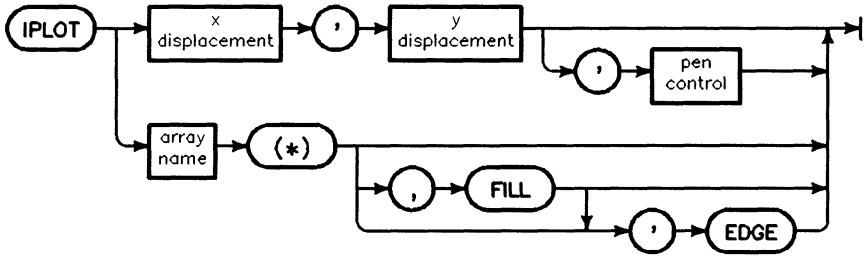
INTR

See the OFF INTR and ON INTR statements.

IPLOT

IPLOT moves the graphics pen an incremental distance from the current position. Plotting action is determined by the current line type and the optional pen control parameter.

Syntax



Item	Description	Range
x displacement	numeric expression, in current units	—
y displacement	numeric expression, in current units	—
pen control	numeric expression, rounded to an integer; default=1 (down after move)	−32 768 through +32 767
array name	name of two-dimensional, two-column or three-column numeric array. Requires GRAPHX.	any valid name

Example Statements

```
IPLOT 0,5
IPLOT Delta_x,Delta_y,Pen_control
IPLOT Array(*)
IPLOT Shape(*),FILL,EDGE
```

Details

Non-Array Parameters

The specified X and Y displacement information is interpreted according to the current unit-of-measure. Lines are drawn using the current pen color and line type.

The line is clipped at the current clipping boundary. If none of the line is inside the current clip limits, the pen is not moved, but the logical pen position is updated.

Graphics Transformations

The output of IPLOT is affected by *only* these graphics transformations:

- scaling specified by WINDOW
- scaling specified by SHOW
- rotations specified by PIVOT
- rotations specified by PDIR

The optional pen control parameter specifies the following plotting actions; the default value is +1 (down after move).

Pen Control Parameter

Pen Control	Resultant Action
−Even	Pen up before move
−Odd	Pen down before move
+Even	Pen up after move
+Odd	Pen down after move

Zero is considered positive.

Summary of Array Parameter Effects

When using an IPLOT statement with an array, the following table of operation selectors applies. An operation selector is the value in the third column of a row of the array to be plotted. The array must be a two-dimensional, two-column or three-column array. If the third column exists, it will contain operation selectors which instruct the computer to carry out certain operations. Polygons may be defined, edged (using the current pen), filled (using the current fill color), pen and line type may be selected, and so forth.

IPLOT Array Parameter Effects

Column 1	Column 2	Operation Selector	Meaning
X	Y	−2	Pen up before moving
X	Y	−1	Pen down before moving
X	Y	0	Pen up after moving (Same as +2)
X	Y	1	Pen down after moving
X	Y	2	Pen up after moving
pen number	ignored	3	Select pen
line type	repeat value	4	Select line type
color	ignored	5	Color value
ignored	ignored	6	Start polygon mode with FILL
ignored	ignored	7	End polygon mode
ignored	ignored	8	End of data for array
ignored	ignored	9	NOP (no operation)
ignored	ignored	10	Start polygon mode with EDGE
ignored	ignored	11	Start polygon mode with FILL and EDGE
ignored	ignored	12	Draw a FRAME
pen number	ignored	13	Area pen value
red value	green value	14	Color
blue value	ignored	15	Value
ignored	ignored	>15	Ignored

FILL and EDGE

When FILL or EDGE is specified, each sequence of two or more lines forms a polygon. The polygon begins at the first point on the sequence, includes each successive point, and the final point is connected or closed back to the first point. A polygon is closed when the end of the array is reached, or when the value in the third column is an even number less than three, or in the range 5 to 8 or 10 to 15.

If FILL and/or EDGE are specified on the IPLOT statement itself, it causes the polygons defined within it to be filled with the current fill color and/or edged with the current pen color. If polygon mode is entered from within the array, and the FILL/EDGE directive for that series of polygons differs from the FILL/EDGE directive on the IPLOT statement itself, the directive in the array replaces the directive on the statement. In other words, if a "start polygon mode" operation selector (a 6, 10, or 11) is encountered, any current FILL/EDGE directive (whether specified by a keyword or an operation selector) is replaced by the new FILL/EDGE directive.

If FILL and EDGE are both declared on the IPLOT statement, FILL must occur first. If neither one is specified, simple line drawing mode is assumed; that is, polygon closure does not take place.

Moving and Drawing

If the operation selector is less than or equal to two, it is interpreted in exactly the same manner as the third parameter in a non-array IPLOT statement. As mentioned above, even means lift the pen up, odd means put the pen down, positive means act after pen motion, negative means act before pen motion. Zero is considered positive.

Selecting Pens

The operation selector of 3 is used to select pens. The value in column one is the pen number desired. The value in column two is ignored.

Selecting Line Types

The operation selector of 4 is used to select line types. The line type (column one) selects the pattern, and the repeat value (column two) is the length in GDUs that the line extends before a single occurrence of the pattern is finished and it starts over. On the CRT, the repeat value is evaluated and rounded *down* to the next multiple of 5, with 5 as the minimum.

Selecting a Fill Color

Operation selector 13 selects a pen from the color map with which to do area fills. This works identically to the AREA PEN statement. Column one contains the pen number.

Defining a Fill Color

Operation Selector 14 is used in conjunction with Operation Selector 15. Red and green are specified in columns one and two, respectively, and column three has the value 14. Following this row in the array (not necessarily immediately), is a row whose operation selector in column three has the value of 15. The first column in that row contains the blue value. These numbers range from 0 to 32 767, where 0 is no color and 32 767 is full intensity. Operation selectors 14 and 15 together comprise the equivalent of an AREA INTENSITY statement.

Operation Selector 15 actually puts the area intensity into effect, but only if an operation selector 14 has already been received.

Operation selector 5 is another way to select a fill color. The color selection is through a Red-Green-Blue (RGB) color model. The first column is encoded in the following manner. There are three groups of five bits right-justified in the word; that is, the most significant bit in the word is ignored. Each group of five bits contains a number which determines the intensity of the corresponding color component, which ranges from zero to sixteen. The value in each field will be sixteen minus the intensity of the color component. For example, if the value in the first column of the array is zero, then all three five-bit values would be zero. Sixteen minus zero in all three cases would turn on all three color components to full intensity, and the resultant color would be a bright white.

Assuming you have the desired intensities for red, green, and blue ranging from zero to one in the variables `R`, `G`, and `B`, respectively, the value for the first column in the array could be defined thus:

```
Array(Row,1)=SHIFT(16*(1-B),-10)+SHIFT(16*(1-G),-5)+16*(1-R)
```

If there is a pen color in the color map identical to that which you request here, that non-dithered color will be used. If there is not a similar color, you will get a dithered pattern.

If you are using a gray scale display, Operation selector 5 uses the five bit values of the RGB color specified to calculate luminosity. The resulting gray luminosity is then used as the area fill. For detailed information on gray scale calculations, see the chapter "More About Color Displays" in the *HP BASIC 6.2 Advanced Programming Techniques* manual.

Polygons

A six, ten, or eleven in the third column of the array begins a "polygon mode." If the operation selector is 6, the polygon will be filled with the current fill color. If the operation selector is 10, the polygon will be edged with the current pen number and line type. If the operation selector is 11, the polygon will be both filled and edged. Many individual polygons (series of draws separated by moves) can be filled without terminating the mode with an operation selector 7. The first and second columns are ignored and, consequently, should not contain the X and Y values of the first point of a polygon.

Operation selector 7 in the third column of a plotted array terminates definition of a polygon to be edged and/or filled and also terminates the polygon mode (entered by operation selectors 6, 10, or 11). The values in the first and second columns are ignored, and the X and Y values of the last data point should not be in them. Edging and/or filling will begin immediately upon encountering this operation selector.

Drawing a FRAME

Operation selector 12 does a FRAME around the current soft-clip limits. Soft clip limits cannot be changed from within the IPLOT statement, so one probably would not have more than one operation selector 12 in an array to IPLOT, since the last FRAME will overwrite all the previous ones.

Premature Termination

Operation selector 8 causes the IPLOT statement to be terminated. The IPLOT statement will successfully terminate if the actual end of the array has been reached, so the use of operation selector 8 is optional.

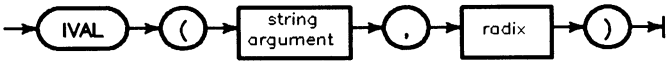
Ignoring Selected Rows in the Array

Operation selector 9 causes the row of the array it is in to be ignored. Any operation selector greater than fifteen is also ignored, but operation selector 9 is retained for compatibility reasons. *Operation selectors less than -2 are not ignored.* If the value in the third column is less than zero, only evenness/oddness is considered.

IVAL

IVAL converts a binary, octal, decimal, or hexadecimal character representation into an INTEGER.

Syntax



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(String$,Radix)
PRINT IVAL("FE56",16)
```

Details

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.

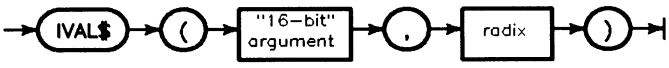
Radix	Base	String Range	String Length
2	binary	0 through 1111111111111111	1 to 16 characters
8	octal	0 through 177777	1 to 6 characters
10	decimal	−32 768 through +32 768	1 to 6 characters
16	hexadecimal	0 through FFFF	1 to 4 characters

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$

IVAL$ converts an integer value into a binary, octal, decimal, or hexadecimal string representation.

Syntax



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

```
String$=IVAL$(Number,Radix)
PRINT IVAL$(Count MOD 256,2)
```

Details

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 through F.

Radix	Base	Range of Returned String	String Length
2	binary	0000000000000000 thru 1111111111111111	16 characters
8	octal	000000 through 177777	6 characters
10	decimal	−32 768 through +32 768	6 characters
16	hexadecimal	0000 through FFFF	4 characters

KBD

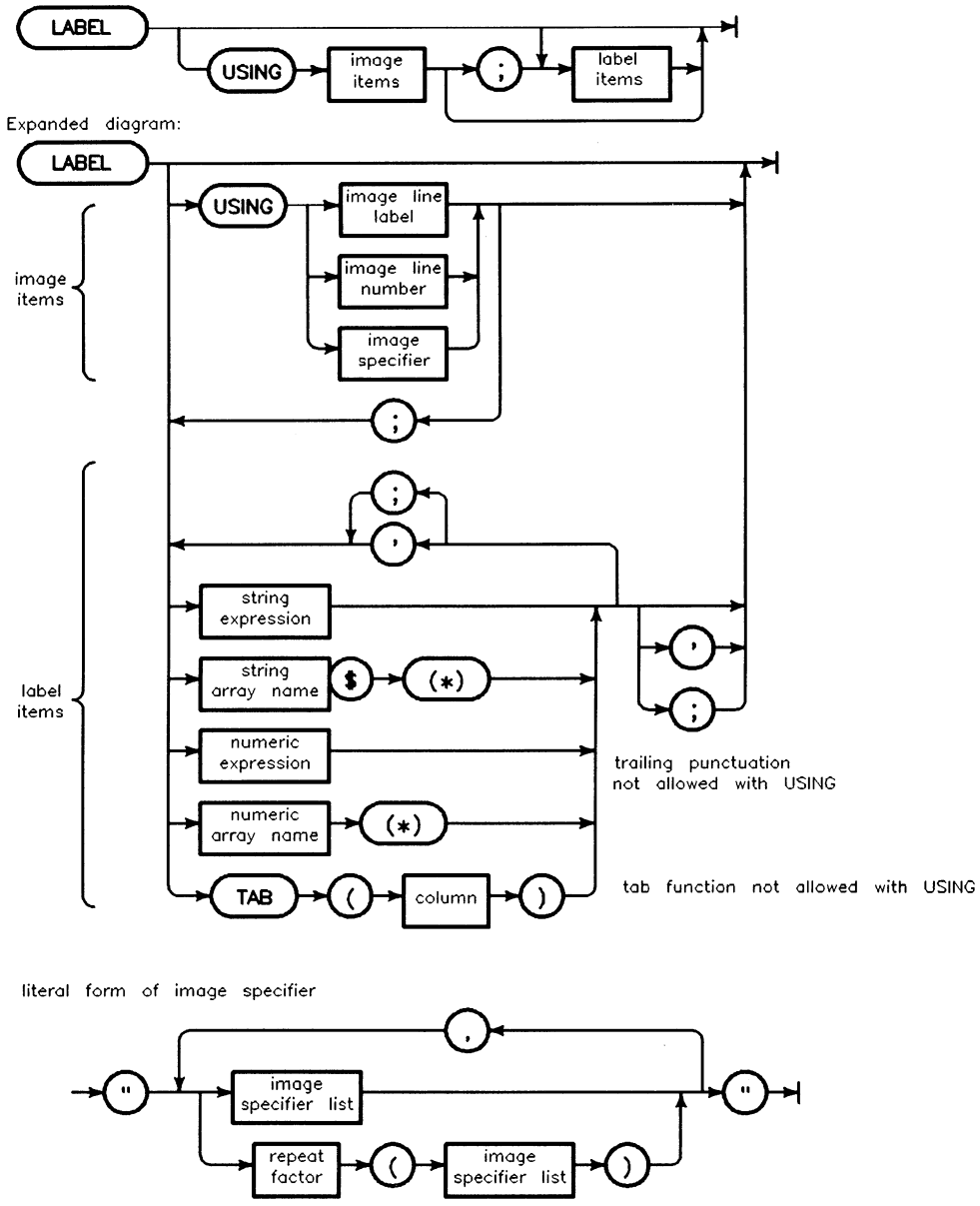
KBD returns 2, the device selector of the keyboard.

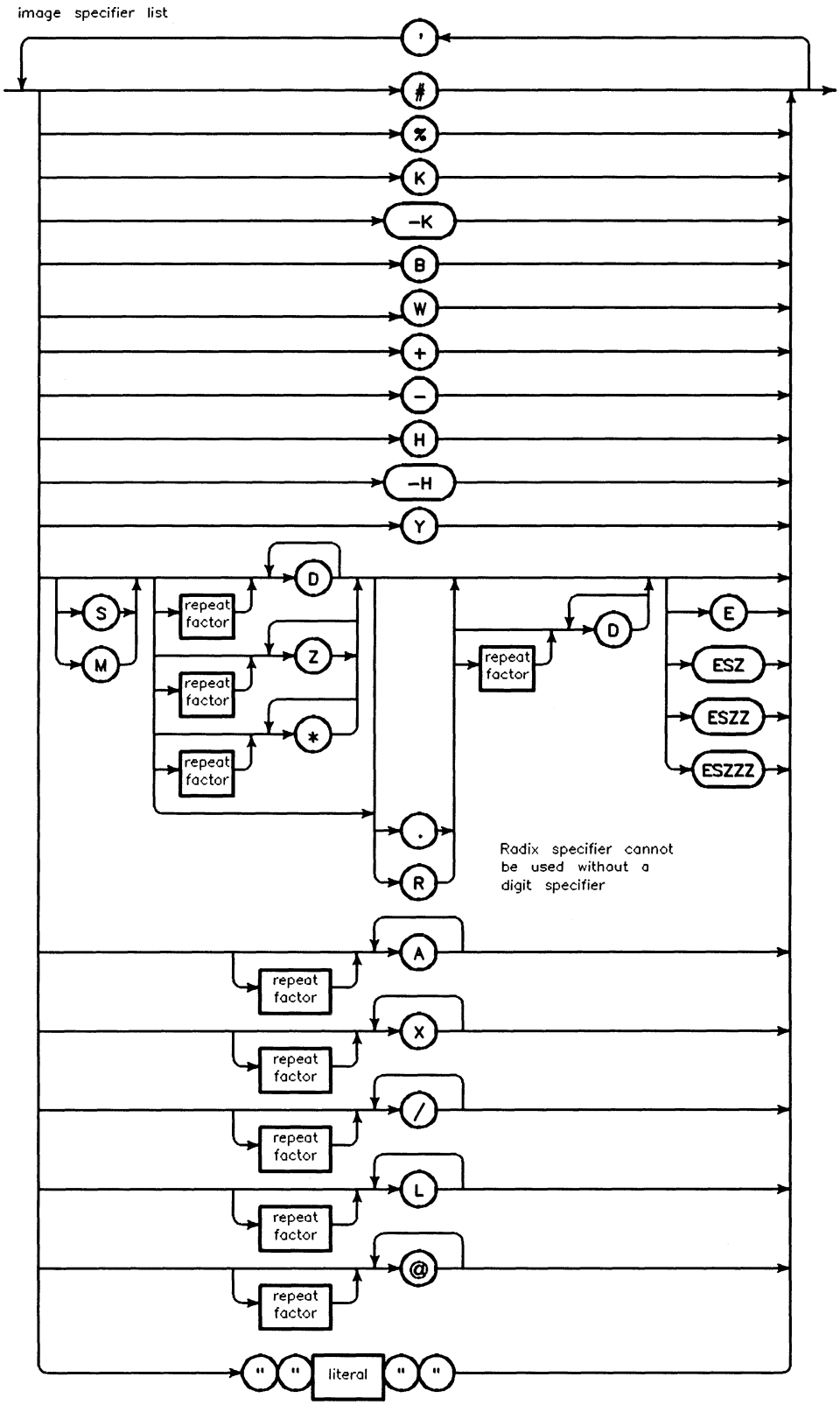
Syntax

LABEL

LABEL draws text labels with the graphics pen at the pen's current position.

Syntax





Item	Description	Range
image line label	name identifying an IMAGE statement	any valid name
image line number	integer constant identifying an IMAGE statement	1 through 32 766
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 diagram)
repeat factor	integer constant	1 through 32 767
literal	string composed of characters entered from the keyboard	quote mark not allowed

Example Statements

```
LABEL Number,String$
LABEL Array(*)
LABEL USING 160;X,Y,Z
LABEL USING "5Z.DD";Money
```

Details

The label begins at the current logical pen position, with the current pen. Labels are clipped at the current clip boundary. The current pen position is updated at the end of the label operation.

The color and line type used for the label are determined by PEN and LINE TYPE respectively. The size (width and height) of the label is set by CSIZE.

Graphics Transformations

The output of LABEL is affected by *only* these graphics transformations:

- origins specified by LORG
- rotations specified by LDIR

Standard Numeric Format

The standard numeric format depends on the value of the number being output. 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 label list is exhausted, an End-of-Line (EOL) sequence is sent to the logical pen, unless it is suppressed by trailing punctuation or a pound-sign image specifier. The EOL sequence is also sent after every 256 characters. This "plot buffer exceeded" EOL is not suppressed by trailing punctuation, but is suppressed by the pound-sign specifier.

Control Codes

Character	Keystroke	Name	Action
CHR$(8)	CTRL-H	backspace	Back up the width of one character cell.
CHR$(10)	CTRL-J	line-feed	Move down the height of one character cell.
CHR$(13)	CTRL-M	carriage-return	Move back the length of the label just completed.

Any control character that the LABEL statement does not recognize is treated as an ASCII blank [CHR$(32)].

Arrays

Entire arrays may be output by using the asterisk specifier. Each element in an array is treated as an item by the LABEL 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).

LABEL without USING

If LABEL is used without USING, the punctuation following an item determines the width of the item's label field; a semicolon selects the compact field, and a comma selects the default label field. When the label item is an array with the asterisk array specifier, each array element is considered a separate label item. Any trailing punctation will suppress the automatic EOL sequence, in addition to selecting the label field to be used for the label item preceding it.

The compact field is slightly different for numeric and string items. Numeric items are output with one trailing blank. String items are output with no leading or trailing blanks.

The default label field labels items with trailing blanks to fill to the beginning of the next 10-character field.

Numeric data is output 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.

LABEL with USING

When the computer executes a LABEL 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 label items, the field specifier is acted upon without accessing the label list. When the field specifier requires characters, it accesses the next item in the label 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 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 LABEL statement are shown in the following table:

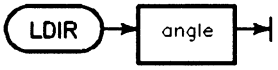
Image Specifier	Meaning
K	Compact field. Outputs a number or string as a label in standard form with no leading or trailing blanks.
−K	Same as K.
H	Similar to K, except the number is output using the European number format (comma radix). (Requires IO.)
−H	Same as H. (Requires IO.)
S	Outputs the number's sign (+ or −) as a label.
M	Outputs the number's sign as a label if negative, a blank if positive.
D	Outputs one-digit character as a label. 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.
*	Same as Z, except that asterisks are output instead of leading zeros. (Requires IO.)
.	Outputs a decimal-point radix indicator as a label.
R	Outputs a comma radix indicator as a label (European radix). (Requires IO.)
E	Outputs as a label: an E, a sign, and a two-digit exponent.
ESZ	Outputs as a label: an E, a sign, and a one-digit exponent.
ESZZ	Same as E.
ESZZZ	Outputs as a label: an E, a sign, and a three-digit exponent.

Image Specifier	Meaning
A	Outputs a string character as a label. 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 as a label.
literal	Outputs as a label the characters contained in the literal.
B	Outputs as a label 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	Outputs as a label 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. (Requires IO.)
#	Suppresses the automatic output of the EOL (End-Of-Line) sequence following the last label item.
%	Ignored in LABEL images.
+	Changes the automatic EOL sequence that normally follows the last label item to a single carriage-return. (Requires IO.)
−	Changes the automatic EOL sequence that normally follows follows the last label item to a single line-feed. (Requires IO.)
/	Sends a carriage-return and a line-feed to the PLOTTER IS device.
L	Same as /.
@	Sends a form-feed to the PLOTTER IS device; produces a blank.

LDIR

LDIR defines the angle at which labels are drawn. The angle is measured counter-clockwise from the X axis using the current angle mode.

Syntax



Item	Description	Range
angle	numeric expression in current units of angle; default = 0	(same as COS)

Example Statements

```
LDIR 90

LDIR ACS(Side)
```

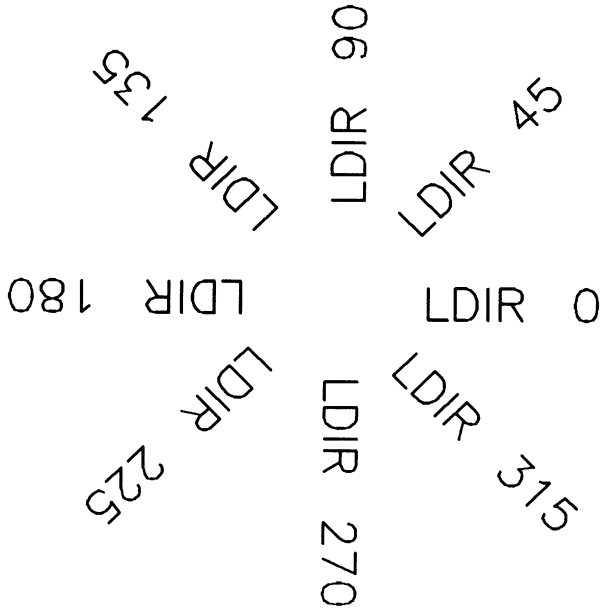
Details

LDIR affects the appearance of LABEL output only; it does not affect the output of other graphics commands.

The rotation angle specified with LDIR is measured counterclockwise from the positive X axis. Thus, the positive X axis is at zero degrees, the positive Y axis is at 90 degrees, and so on. The unit of measure for angles can be set using DEG and RAD.

The angle is interpreted as shown in the following figure.

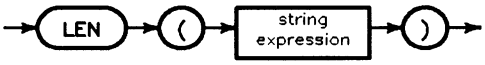
LDIR EXAMPLES (in Degrees)



LEN

LEN returns the current number of characters in the specified string.

Syntax



Example Statements

```
Last=LEN(String$)
IF NOT LEN(A$) THEN Empty
```

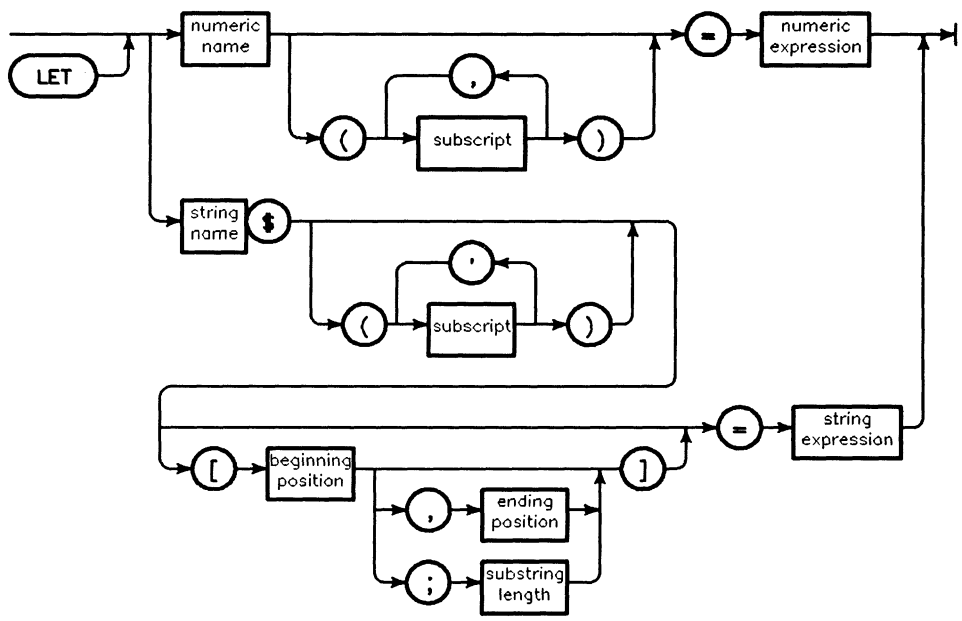
Details

The length of the null string ("") is 0.

LET

LET assigns values to variables. The keyword LET is optional.

Syntax



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 through +32 767 (see "array" in Glossary)
beginning position	numeric expression, rounded to an integer	1 through 32 767 (see "substring" in Glossary)
ending position	numeric expression, rounded to an integer	0 through 32 767 (see "substring" in Glossary)
substring length	numeric expression, rounded to an integer	0 through 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"
```

Details

The assignment is done to the variable that 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, as in this statement:

```
A=A+(B=1)+5
```

A variable can occur on both sides of the assignment operator, as in these statements:

```
I=I+1

Source$=Old$
```

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.

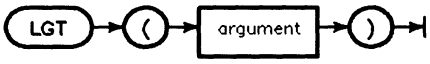
The length of the string expression must be less than or equal to the dimensioned length of the string it is being assigned to. Assignments may be made into substrings, using the normal rules for substring definition. The string expression will be truncated or blank-filled on the right (if necessary) 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. Otherwise, a live keyboard execution is attempted and fails, even though the line number is present.

LGT

LGT returns the common logarithm (base 10) of its argument.

Syntax



Item	Description/Default	Range Restrictions
argument	numeric expression	> 0 for INTEGER and REAL arguments

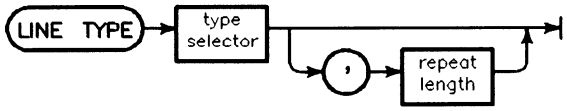
Example Statements

```
Decibel=20*LGT(Volts)
PRINT "Log of";X;"=";LGT(X)
```

LINE TYPE

LINE TYPE selects a line type (solid or dashed) for all subsequent lines drawn with the graphics pen.

Syntax



Item	Description	Range
type selector	numeric expression, rounded to an integer; default = 1	1 through 10
repeat length	numeric expression, rounded to an integer; default = 5	greater than 0

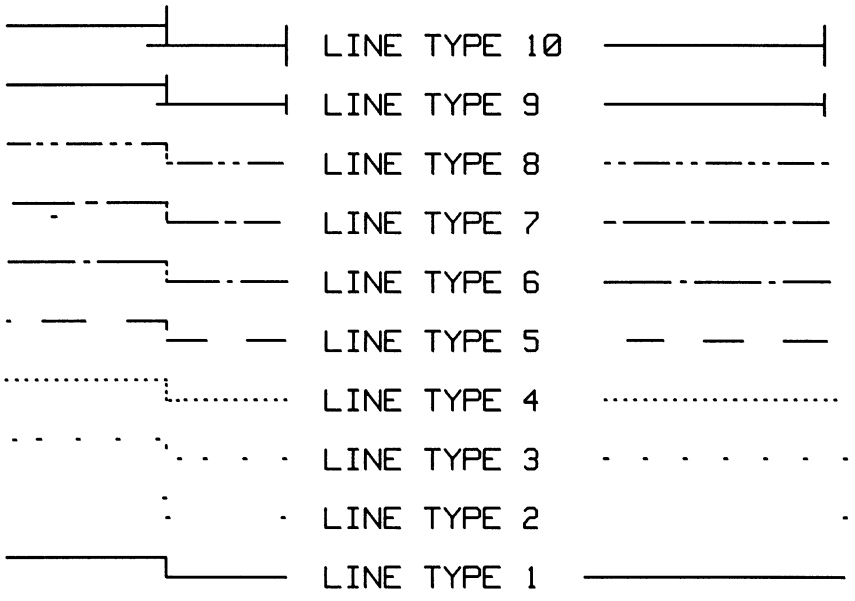
Example Statements

```
LINE TYPE 1
LINE TYPE Style
```

Details

At power-up the default line type is a solid line (type 1).

The available CRT line types are shown here.

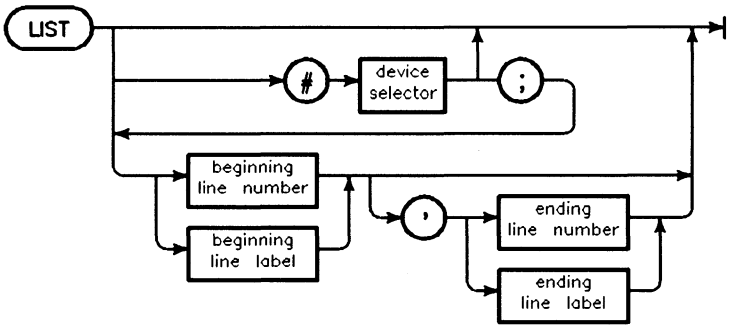


Note that the solid and dashed line patterns are different in versions of HP BASIC other than HP Instrument BASIC. However, in all versions of HP BASIC, line types 3-8 are always dashed and line type 1 is always solid.

LIST

LIST lists the program currently in memory.

Syntax



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 through 32 766
beginning line label	name of a program line	any valid name
ending line number	integer constant identifying program line	1 through 32 766
ending line label	name of a program line	any valid name

Example Statements

```
LIST
LIST 110,250
```

Details

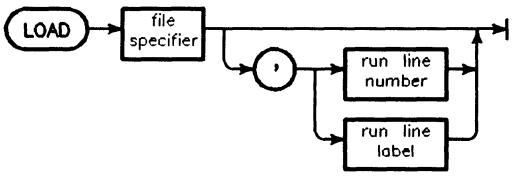
When a label is used as a line identifier, the lowest-numbered line in memory having that label is used. When a number is used as a line identifier, the lowest-numbered line in memory having a number equal to or greater than the specified line is used. An error occurs if the ending line identifier occurs before the beginning line identifier or if a specified line label does not exist in the program.

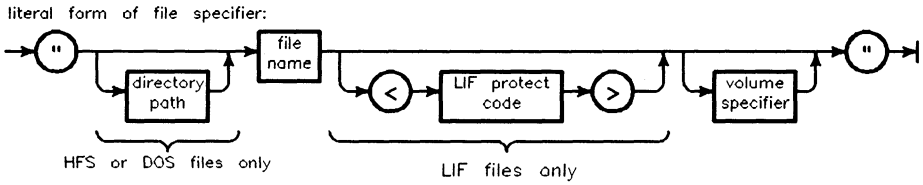
Executing a LIST from the keyboard while a program is running causes the program to pause at the end of the current line. The listing is sent to the selected device and program execution resumes. Note that the default width of the PRINTER IS device is 80 characters, which means that a carriage-return (CR) and line-feed (LF) character will be sent after 80 characters are printed on any one line.

LOAD

LOAD loads STOREd programs into memory.

Syntax





Item	Description	Range
file specifier	string expression	(see drawing)
run line label	name of a program line	any valid name
run line number	integer constant identifying program line	1 through 32 765
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

```
LOAD Filename$
LOAD "UTIL",Run_line
```

Details

The HP Instrument BASIC program currently in memory and all variables not in COM are lost when a LOAD is executed. Every COM block in the newly-loaded program is compared with the COM blocks of the program in memory. If a COM area of the newly-loaded program does not match an existent COM area, the values in the old COM area are lost. Thus, some COM areas may be retained while others are lost.

LOAD is allowed from the keyboard if a program is not running. If no run line is specified, you must press (Run) in the control pad to initiate program execution, and execution will begin on the first line in the program. If a run line is specified, prerun initialization (see RUN) is performed, and program execution begins at the line specified. The line on which execution begins must exist in the main program context of the newly-loaded program. If you specify a line *number* and it doesn't exist, execution begins with the next higher-numbered line, provided that line is in the main program context.

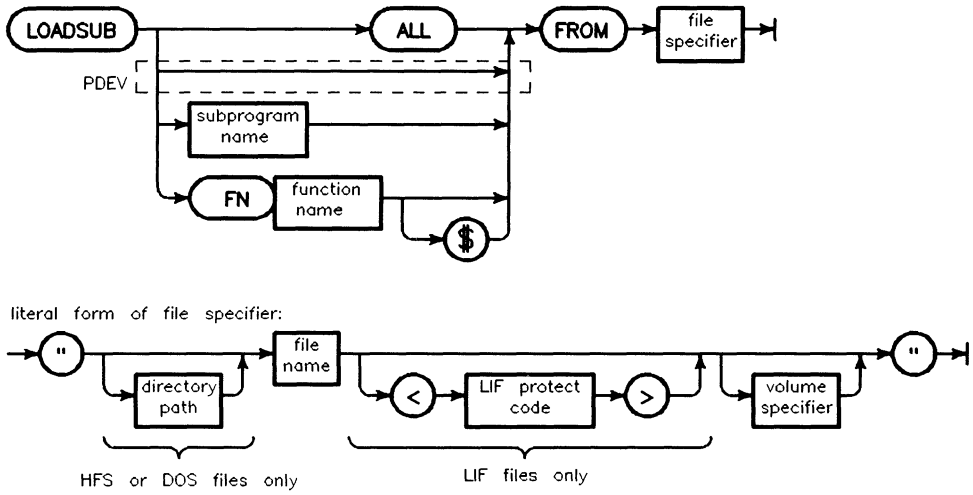
Executing LOAD from a program causes the new program file to be loaded, prerun, and program execution to resume. Execution begins at the line specified, or the lowest-numbered program line if a run line is not specified.

You can use wildcards to match a file specifiers with LOAD. You must first enable wildcard recognition using WILDCARDS. Refer to the keyword entry for WILDCARDS for details. Wildcard file specifiers used with LOAD must match one and only one file name.

LOADSUB

LOADSUB loads HP Instrument BASIC subprograms from a STOREd file into memory.

Syntax



Item	Description	Range
file specifier	string expression	(see drawing)
subprogram name	name of a SUB or CSUB subprogram	any valid name
function name	name of a user-defined function	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

```
LOADSUB FNReplace$ FROM "Subs_file"

LOADSUB ALL FROM Subfile$

LOADSUB ALL FROM "/Dir1/Dir2/Prog23"
```

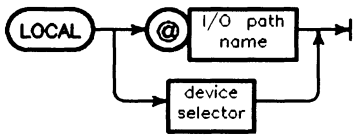
LOADSUB with ALL or Subprogram Name

When LOADSUB is used with a subprogram name or ALL, LOADSUB loads the matching subprogram(s) from the specified file. This form of LOADSUB *is programmable*. If either the file name or the subprogram name specified is not found, or the file name is not a PROG file, an error will occur. As the subprogram is loaded, it will be renumbered to fit at the end of the program. LOADSUB does not cause the program or any data currently in memory to be lost.

LOCAL

LOCAL returns all specified devices to their local state.

Syntax



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 728

LOCAL Isc
```

Details

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.

Bus Actions

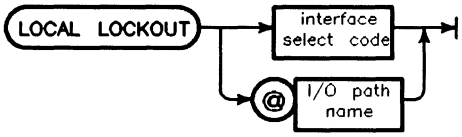
Summary of Bus Actions

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Address Specified	Interface Select Code Only	Primary Address Specified
Active Controller	$\overline{\text{REN}}$	ATN MTA UNL LAG GTL	ATN GTL	ATN MTA UNL LAG GTL
Not Active Controller	$\overline{\text{REN}}$	Error	Error	Error

LOCAL LOCKOUT

LOCAL LOCKOUT sends the LLO (local lockout) message, which prevents local (front panel) control of devices in the remote state.

Syntax



Item	Description	Range
interface select code	numeric expression, rounded to an integer	7 through 31
I/O path name	name assigned to an interface select code	any valid name (see ASSIGN)

Example Statements

```
LOCAL LOCKOUT 7

LOCAL LOCKOUT Isc

LOCAL LOCKOUT @Hpib
```

Details

The following conditions must be met to use LOCAL LOCKOUT without error:

- The computer sending LOCAL LOCKOUT *must* be Active Controller
- Only an interface select code may be specified, not a primary address

Either System Controllers or Non-system Controllers may send 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.

Bus Actions

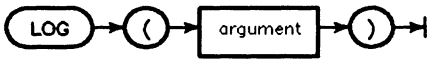
Summary of LOCAL LOCKOUT Bus Actions

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Address Specified	Interface Select Code Only	Primary Address Specified
Active Controller	ATN LLO	Error	ATN LLO	Error
Not Active Controller	Error	Error	Error	Error

LOG

LOG returns the natural logarithm (base e) of the argument.

Syntax



Item	Description/Default	Range Restrictions
argument	numeric expression	> 0 for INTEGER and REAL arguments

Example Statements

```
Time=-1*Rc*LOG(Volts/Emf)
PRINT "Natural log of";Y;"=";LOG(Y)
```

LOOP

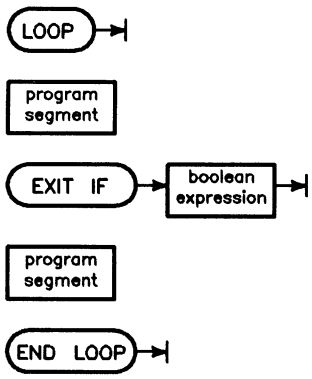
LOOP defines a loop that is repeated until the expression in an EXIT IF statement is evaluated as true (non-zero).

Example Program Segment

```
100  ! Reading to the end of a file
110  ASSIGN @File TO "MYFILE"
110  ON EOF GOSUB Done
120  LOOP
130     ENTER @File;Text$
140  END LOOP
150  Done: !
```

```
100  ! Process control loop
110  LOOP
120     Level=FNGet_level
130     EXIT IF Level>High_level
140     Pressure=FNSet_press(Decrease)
150     EXIT IF Pressure<Low_press
160  END LOOP
```

Syntax



Item	Description	Range
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).	—
Boolean expression	numeric expression; evaluated as true if nonzero and false if 0	—

Details

The LOOP ... END LOOP construct allows continuous looping with conditional exits. These exits 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 that 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.

LORG

LORG specifies the relative origin of labels with respect to the current pen position.

Syntax

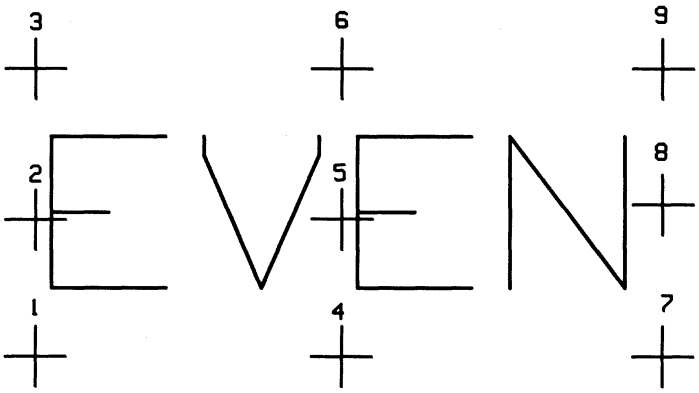
Item	Description	Range
label origin position	numeric expression, rounded to an integer; default = 1	1 through 9

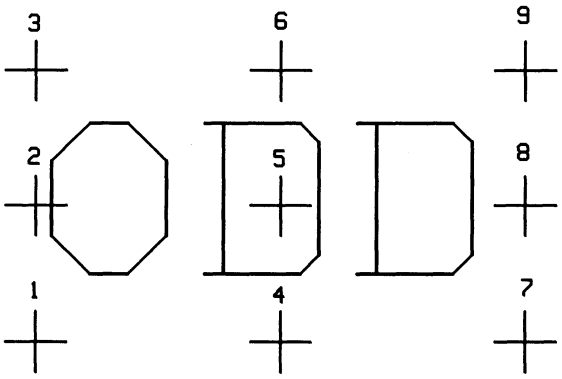
Example Statements

```
LORG New_lorg

IF Y>Limit THEN LORG 3
```

Details

The following drawings show the relationship between a label and the logical pen position. The pen position before the label is drawn is represented by a cross marked with the appropriate LORG number.

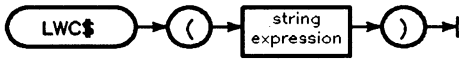




LWC$

LWC$ returns a string formed by replacing any uppercase characters with their corresponding lower-case character.

Syntax



Example Statements

```
Lower$=LWC$(Mixed$)
IF LWC$(Answer$)="y" THEN True_test
```

Details

The LWC$ function converts only uppercase alphabetic characters to their corresponding lowercase characters and will not alter numerals or special characters.

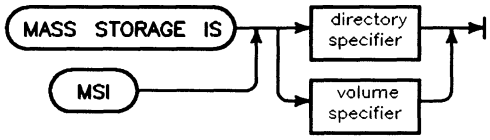
The corresponding characters for the Roman Extension alphabetic characters are determined by the current lexical order. When the lexical order is a user-defined table, the correspondence is determined by the STANDARD lexical order.

MASS STORAGE IS

MASS STORAGE IS specifies the system mass storage device, which is the path name for a specific disk drive and directory. The MASS STORAGE IS device is used as the implied source or destination for all file-related related operations that do not specify an explicit source or destination.

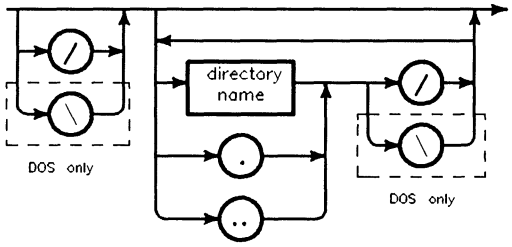
MASS STORAGE IS may be abbreviated as MSI when executed from directly from the keyboard.

Syntax

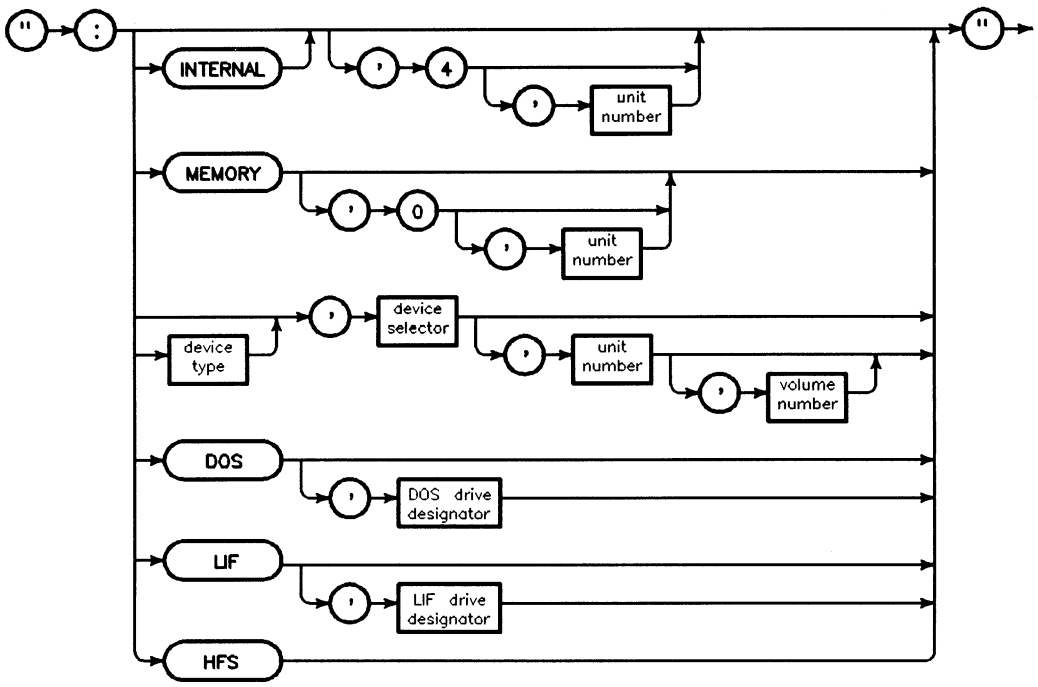


literal form of directory specifier (HFS and DOS volumes only):

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 (8 characters for DOS (short file name); (see Glossary)
unit number	integer constant; default = 0	0 through 255 (device-dependent)
device type	literal	See instrument-specific HP Instrument BASIC for the device type supported in this instrument.
device selector	integer constant	(see Glossary)
volume number	integer constant; default = 0	(device-dependent)
DOS drive designator	literal	any valid DOS drive designator in the range A through Z (or a through z)

Example Statements

```
MASS STORAGE IS Vol_specifier$

MASS STORAGE IS Dir_path$&Vol_specifier$

MASS STORAGE IS "A:"

MASS STORAGE IS "C:\TEST1\MONDAY"

MASS STORAGE IS "MYDIR"
```

Details

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.

If you are using a version of HP Instrument BASIC that supports wildcards, you can use them in volume specifiers with MSI. You must first enable wildcard recognition using WILDCARDS. Refer to the keyword entry for WILDCARDS for details. Wildcard file specifiers used with MSI must match one and only one volume name.

External disk drives must be on-line.

MAT

MAT performs a variety of operations on matrices and other numeric and string arrays.

Example Statements

```
MAT Array= A*(Ref+1/3)
MAT String$= (RPT$(" ",80))
MAT Clone= Parent
MAT A= Array1Array2
MAT Vector= CSUM(Matrix)
MAT Vector= RSUM(Matrix)
MAT Transposition= TRN(Matrix)
MAT Identity= IDN
MAT Inverse= INV(Matrix)
MAT Des_array(-1:0,2:4)= Sor_array
MAT Array_1= Array_2(-4:1)
MAT Destination(3,*,*)= Source(*,2,*)
```

Details

The MAT statement allows you to

- copy a string expression into a string array or copy the contents of one string array into another string array
- copy a numeric expression into an array
- copy the contents of one array or subarray into another array or subarray
- add an array and a numeric expression, or two arrays
- subtract a numeric expression from an array, an array from a numeric expression, or an array from an array
- multiply an array by a numeric expression or another array
- divide a numeric expression by an array, an array by a numeric expression, or an array by an array
- compare an array to a numeric expression or to another array
- calculate the Identity, Inverse, Transpose, Sum of rows, and Sum of columns of a matrix
- calculate the absolute value of a numeric array

Note

If an error occurs during the calculations involved in a MAT assignment the result array will contain only a partial result. Since you will have no idea which entries are valid, the contents of the array should be considered invalid.

Numeric Operations

In the case of operators, the specified operation is generally performed on every array element, and the results are placed in corresponding locations of the result array (the exception is the * operator, which is discussed under Matrix Multiplication, below.) This means that the result array must have the same "size" and "shape" (though not necessarily the same subscript ranges) as the operand array(s). Note that "size" refers to the number of elements in the array and "shape" refers to the same number of dimensions and elements in each dimension, respectively (e.g. both of these subscript specifiers have the same shape: (-2:1,-1:10) and (1:4,9:20)). If necessary, the system will redimension the result array to make it the proper size. The redimensioning can only take place, however, if the dimensioned size of the result array has at least as many elements as the current size of the operand array(s).

When two arrays are operated on, they must be exactly the same size and shape. If not, the computer returns an error. The specified operation is performed on corresponding elements in each operand array and the result is placed in the corresponding location of the result array. Multiplication of the elements of two arrays is performed with a period rather than an asterisk. The asterisk is reserved for matrix multiplication described below.

Relational Operators

Relational operations are performed on each element of the operand array(s). If the relation is TRUE, a 1 is placed in the corresponding location of the result array. If the relation is FALSE, a 0 is recorded. The result array, therefore, consists of all 0's and 1's.

Matrix Multiplication

The asterisk is used for two operations. If it is between an array and a numeric expression, each element in the array is multiplied by the numeric expression. If it is between two matrices, it results in matrix multiplication. If A and B are the two operand matrices, and C is the result matrix, the matrix multiplication is defined by:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

where n is the number of elements in a column in the matrix A. (This formula assumes that the array subscripts run from 1 through N; in actuality, the computer only requires that the two arrays be the correct size and shape, the actual values of the subscripts are unimportant.)

Note that the subscript values of the result array correspond to the rows of the first operand matrix and the columns of the second operand matrix. Note also that the column subscript of the first operand array is equal to the row subscript of the second operand array. We can summarize these observations in two general rules:

- The result matrix will have the same number of rows as the first operand matrix and the same number of columns as the second operand matrix.
- Matrix multiplication is legal if, and only if, the column size of the first operand matrix is equal to the row size of the second operand matrix.

A third rule of matrix multiplication is:

- The result matrix cannot be the same as either operand matrix.

If one or both operands is REAL, the calculation is done in REAL math. If both operands are INTEGER, the computation is also INTEGER. If the result matrix and the operand matrixes are different types (i.e., one is REAL and the others are INTEGER), the computer makes the conversion necessary for the assignment. However, the conversion is made *after* the multiplication is calculated, so even if the matrix receiving the result is REAL, the multiplication can generate an INTEGER overflow when the operands are INTEGER matrixes.

The computer allows you to do matrix multiplication on vectors by treating the vectors as if they were matrices. If the first operand is a vector, it is treated as a 1-by-N matrix. If the second operand is a vector, it is treated as an N-by-1 matrix.

Copying Subarrays

A subarray is a subset of an array (an array within an array). A subarray is indicated by a specifier after the array name as follows:

```
Array_name(subarray_specifier)
Array_name$(subarray_specifier)
```

For example, to specify the entire second column of a two-dimensional array, use the following subarray:

```
Array_name(*,2)
```

Copying subarrays is useful in these situations:

- copying a subarray into an array
- copying an array into a subarray
- copying a subarray into a subarray
- copying a portion of an array into itself

Before discussing the rules for subarrays the concept of range needs to be understood as it appears in this text. The two ranges related to subarrays are the subscript range and default range. The subscript range specifies a set of elements starting with a beginning element position and ending with a final element position. For example, 5:8 specifies a range of four elements starting with element 5 and ending at element 8. The default range is denoted by an asterisk (*) and it represents all of the elements in a dimension. For example, suppose you wanted to copy the entire first column of a two dimensional array, you would use the following subarray specifier: (*,1), where * represents all the rows in the array and 1 represents *only* the first column.

Follow these rules when copying subarrays:

- Subarray specifiers *must not* contain all subscript expressions (i.e., (1,2,3) is not allowed, it will produce a syntax error). This rule applies to all subarray specifiers.
- Subarray specifiers *must not* contain all asterisks (*) or default ranges (i.e. (*,*,*) is not allowed, it will produce a syntax error). This rule applies to all subarray specifiers.
- If two subarrays are given in a MAT statement, there *must be* the same number of ranges in each subarray specifier. For example,

  ```
  MAT Des_array1(1:10,2:3)= Sor_array(5:14,*,3)
  ```

  is the correct way of copying a subarray into another subarray provided the default range given in the source array (Sor_array) has only two elements in it. Note that the source array is a three-dimensional array. However, it still meets the criteria of having the same number of ranges as the destination array because two of its entries are ranges and one is an expression.
- If two subarrays are given in a MAT statement, the subscript ranges in the source subarray *must be* the same shape as the subscript ranges in the destination subarray. For example, the following is *legal*:

  ```
  MAT Des_array(1:5,0:1)= Sor_array(3,1:5,6:7)
  ```

  however, the one below is *not* legal:

  ```
  MAT Des_array(0:1,1:5)= Sor_array(1:5,0:1)
  ```

  because both of its subarray specifiers do not have the same shape (i.e. the rows and columns in the destination subarray do not match the rows and columns in the source subarray).

CSUM

The secondary keyword CSUM computes the sum of each column in a matrix and places the results in a vector. The result vector must have at least as many elements as the matrix has columns. If the vector is too large or its current size is too small (and there are enough elements in its original declaration to allow redimensioning), the computer redimensions it. If the result vector and the argument array are different types (i.e., one is REAL and the other is INTEGER), the computer makes the necessary conversion. However, the conversion is made *after* the column sums are calculated, so even if the vector receiving the result is REAL, CSUM can generate an INTEGER overflow when the argument is an INTEGER array.

IDN

The secondary keyword IDN turns a square matrix into an identity matrix. An identity matrix has 1s along the main diagonal and 0s everywhere else. The matrix *must* be square.

INV

The secondary keyword INV finds the inverse of a square matrix. A matrix multiplied by its inverse produces an identity matrix. The inverse is found by using the pivot-point method. If the value of the determinant (see DET) is 0 after an INV, then the matrix has no inverse—whatever inverse the computer came up with is invalid. If the value of the determinant is very small compared with the elements in the argument matrix, then the inverse may be invalid and should be checked.

If the result matrix is not the same size and shape as the argument matrix, the computer will attempt to redimension it. If it is too large, or its current size is too small (and there are enough elements in its original declaration to allow redimensioning) the computer redimensions it. An error is returned if the computer cannot redimension the result array.

RSUM

The secondary keyword RSUM computes the sum of each row in a matrix and places the values in a vector. The result vector must be large enough to hold the sums of each row. If it is too large, or its current size is too small (and there are enough elements in its original declaration to allow redimensioning) the computer redimensions it. If the result vector and the argument array are different types (i.e., one is REAL and the other is INTEGER), the computer makes the necessary conversion. However, the conversion is made *after* the row sums are calculated, so even if the vector receiving the result is REAL, RSUM can generate an INTEGER overflow when the argument is an INTEGER array.

TRN

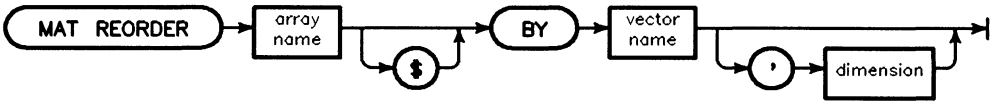
The secondary keyword TRN produces the transpose of a matrix. The transpose is produced by exchanging rows for columns and columns for rows. The result matrix must be dimensioned to be at least as large as the current size of the argument matrix. If it's the wrong shape, the computer redimensions it. The result and argument matrices cannot be the same.

The transpose of an N-by-M matrix is an M-by-N matrix, and each element is defined by switching the subscripts. That is, A(m,n) in the argument matrix equals B(n,m) in the result matrix. (This description assumes that the array subscripts run from 1 through M and 1 through N; in actuality, the computer only requires that the array be the correct size and shape, the actual values of the subscripts are unimportant.)

MAT REORDER

MAT REORDER reorders elements in an array according to the subscript list in a vector.

Syntax



Item	Description	Range
array name	name of an array	any valid name
vector name	name of a one-dimensional numeric array	any valid name
dimension	numeric expression, rounded to an integer; default=1	1 through 6; ≤ the RANK of the array

Example Statements

```
MAT REORDER Array BY Vector,Dimension
MAT REORDER Lines$ BY New_order
```

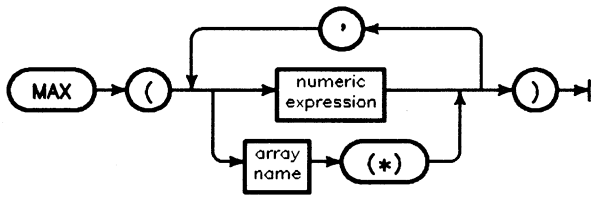
Details

The dimension parameter is used to specify which dimension in a multidimensional array is to be reordered. If no dimension is specified, the computer defaults to dimension 1. The vector must be the same size as the specified dimension and it should contain integers corresponding to the subscript range of that dimension (no duplicate numbers, or numbers out of range).

MAX

MAX returns a value equal to the greatest value in the list of arguments. Each element of an array is considered a separate value.

Syntax



Item	Description	Range
array name	name of a numeric array	any valid name

Example Statements

```
Biggest=MAX(Elements(*))
PRINT MAX(Item1,17,Total/3)
Result=MAX(Floor,MIN(Ceiling,Argument))
```

MAXREAL

MAXREAL returns the largest positive REAL number available in the range of the computer. Its value is approximately 1.797 693 134 862 32E+308.

Syntax

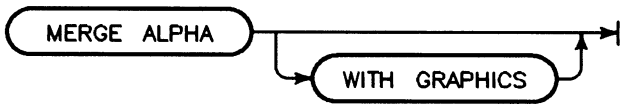
Example Statements

```
IF X<=LGT(MAXREAL) THEN Y=10^X

Half_max=MAXREAL/2
```

MERGE ALPHA

MERGE ALPHA is included for compatibility with RMB-UX. It has no effect except in RMB Workstation.

Syntax



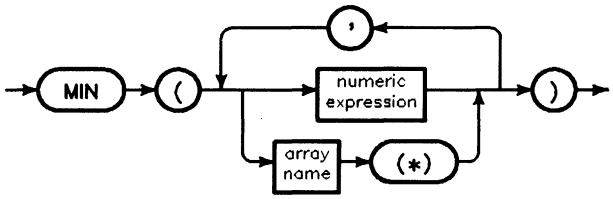
Example Statements

```
MERGE ALPHA

IF Done THEN MERGE ALPHA WITH GRAPHICS
```

MIN

MIN returns a value equal to the least value in the list of arguments. Each element of an array is considered a separate value.

Syntax



Item	Description	Range
array name	name of a numeric array	any valid name

Example Statements

```
Smallest=MIN(Elements(*))
PRINT MIN(Item1,17,Total/3)
```

MINREAL

MINREAL returns the smallest positive REAL number available in the range of the computer. Its value is approximately 2.225 073 858 507 24E-308.

Syntax

Example Statements

```
IF X>=LOG(MINREAL) THEN Y=EXP(X)
```

MOD

MOD returns the remainder of a division.

Syntax

Item	Description	Range
dividend	numeric expression	—
divisor	numeric expression	not equal to 0

Example Statements

```
Remainder=Dividend MOD Divisor
PRINT "Seconds =";Time MOD 60
```

Details

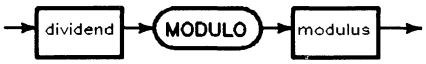
MOD returns an INTEGER value if both arguments are INTEGER. Otherwise the returned value is REAL.

MODULO

MODULO returns the remainder of a division just like MOD, only with one additional constraint—the result satisfies:

```
0<=(X MODULO Y)<Y if Y>0
Y<(X MODULO Y)<=0 if Y<0
```

Syntax



Item	Description	Range
dividend	numeric expression	range of REAL
modulus	numeric expression	range of REAL, $\neq 0$

Example Statements

```
Remainder=Dividend MODULO Divisor
PRINT "Seconds =";Time MODULO 60
```

Details

X MODULO Y is equivalent to $X-Y\times INT(X/Y)$.

The result satisfies:

```
0 <= (X MODULO Y) < Y if Y>0
Y < (X MODULO Y) <= 0 if Y&<0
```

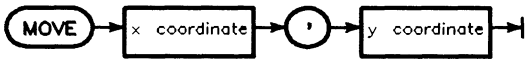
The type of the result is the higher of the types of the two operands. If the modulus is zero, then error 31 occurs.

MODULO returns the remainder of a division.

MOVE

MOVE moves the physical and logical position of the graphics pen to specified X and Y coordinates without drawing a line.

Syntax



Item	Description	Range
x coordinate	numeric expression in current units	—
y coordinate	numeric expression in current units	—

Example Statements

```
MOVE 10,75
MOVE Next_x,Next_y
```

Details

The pen is raised before the move and remains up after the move. The X and Y coordinates are interpreted according to the current unit-of-measure.

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.

Graphics Transformations

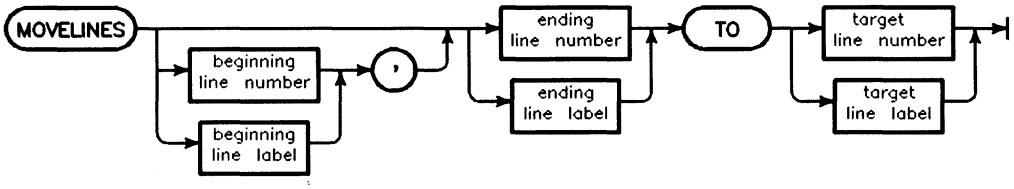
The output of MOVE is affected by *only* these graphics transformations:

- scaling specified by WINDOW
- scaling specified by SHOW
- rotations specified by PIVOT

MOVELINES

MOVELINES moves contiguous program lines from one location to another. If only one line identifier is specified, only that line is moved.

Syntax



Item	Description	Range
beginning line number	integer constant identifying program line	1 to 32 766
beginning line label	name of a program line	any valid name
ending line number	integer constant identifying program line	1 to 32 766
ending line label	name of a program line	any valid name
target line number	integer constant identifying program line	1 to 32 766
target line label	name of a program line	any valid name

Example Commands

```
MOVELINES 1200 TO 3250
MOVELINES 10,440 TO 540
MOVELINES Label1,Label2 TO Label3
```

Details

If the ending line identifier is not specified, only one line is moved.

The target line identifier will be the line number of the first line of the moved program segment. Moved lines are renumbered if necessary. The code (if any) that is "pushed down" to make room for the moved code is renumbered if necessary.

Line number references to the moved code are updated as they would be by a REN command (except external references to non-existent lines are renumbered).

If there are any DEF FN or SUB statements in the moved code, the target line number must be greater than any existing line number.

If you try to move a program segment to a line number *contained* in the segment, an error will result and no moving will occur.

If the starting line number does not exist, the next line is used. If the ending line number does not exist, the previous line is used. If a line label doesn't exist, an error occurs and no moving takes place.

If an error occurs *during* a MOVELINES (for example, a memory overflow), the move is terminated and the program is left partially modified.

MSI

MSI is identical to MASS STORAGE IS.

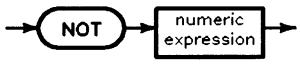
NEXT

See FOR ... NEXT.

NOT

NOT returns 1 if its argument equals 0. Otherwise, 0 is returned.

Syntax



Example Statements

```
Invert_flag=NOT Std_device
IF NOT My_job THEN Sleep
```

Details

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.

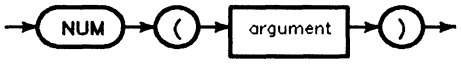
The logical complement is shown below:

A	NOT A
0	1
1	0

NUM

NUM returns the decimal value of the ASCII code of the first character in the specified string. The range of returned values is 0 through 255.

Syntax



Item	Description	Range
argument	string expression	not a null string

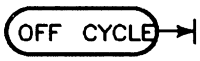
Example Statements

```
Ascii_val=NUM(String$)
A$[I;1]=CHR$(NUM(A$[I])+32)
```

OFF CYCLE

OFF CYCLE cancels event-initiated branches previously defined and enabled by an ON CYCLE statement.

Syntax



Example Statements

```
OFF CYCLE

IF Stop_timer THEN OFF CYCLE
```

Details

OFF CYCLE destroys the log of any CYCLE event that has already occurred but that 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

OFF ERROR cancels event-initiated branches previously defined and enabled by an ON ERROR statement. Subsequent errors are reported to the user in the usual fashion.

Syntax

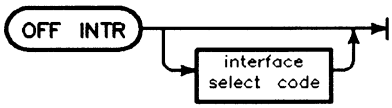
Example Statement

```
OFF ERROR
```

OFF INTR

OFF INTR cancels event-initiated branches previously defined by an ON INTR statement.

Syntax



Example Statements

```
OFF INTR

OFF INTR 12

OFF INTR Hpib
```

Details

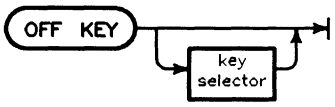
Not specifying an interface select code disables the event-initiated branches for all interfaces. Specifying an interface select code causes the OFF INTR to apply to the event-initiated log entry for the specified interface only.

Any pending ON INTR branches for the affected interfaces are lost and further interrupts are ignored.

OFF KEY

This statement cancels event-initiated branches previously defined and enabled by an ON KEY statement.

Syntax



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
```

Details

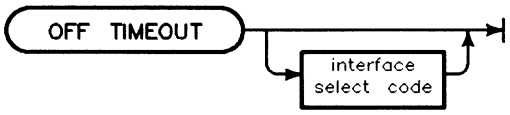
Not specifying a softkey number disables the event-initiated branches for all softkeys. Specifying a softkey number causes the OFF KEY to apply to the specified softkey only. If OFF KEY is executed in a subprogram and cancels an ON KEY in the context that called the subprogram, the ON KEY definitions are restored when the calling context is restored.

Any pending ON KEY branches for the affected softkeys are lost.

OFF TIMEOUT

OFF TIMEOUT cancels event-initiated branches previously defined and enabled by an ON TIMEOUT statement.

Syntax



Example Statements

```
OFF TIMEOUT

OFF TIMEOUT 12

OFF TIMEOUT Hpib
```

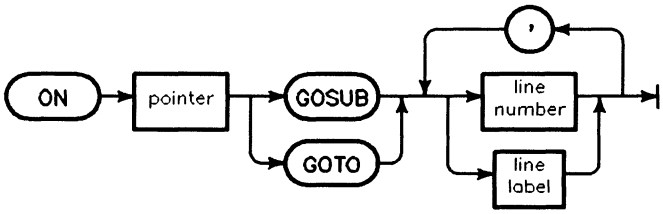
Details

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 affected interfaces.

ON

ON transfers program execution to one of several destinations selected by the value of the pointer.

Syntax



Item	Description	Range
pointer	numeric expression, rounded to an integer	1 through 74
line number	integer contstant identifying a program line	1 through 32 766
line label	name of program line	any valid name

Example Statements

```
ON X1 GOTO 100,150,170
```

```
IF Point THEN ON Point GOSUB First,Second,Third,Last
```

Details

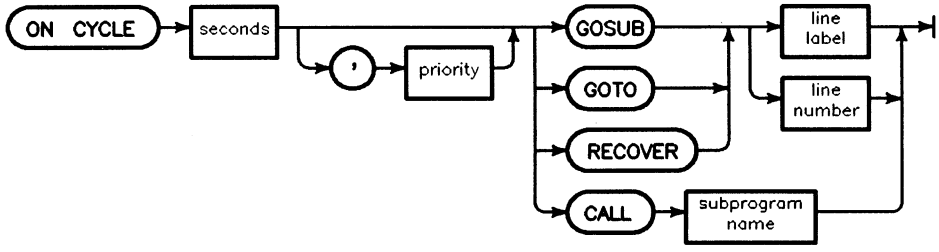
If the pointer evaluates to 1, the first line number or label is used. If the pointer evaluates to 2, the second line number or label is used, and so on. If GOSUB is used, the RETURN is to the line following the ON ... GOSUB statement.

If the pointer evaluates to less than 1 or greater than the number of lines listed, error 19 results. The specified line numbers or labels must be in the same context as the ON statement.

ON CYCLE

ON CYCLE defines and enables an event-initiated branch to be taken each time the specified number of seconds has elapsed.

Syntax



Item	Description	Range
seconds	numeric expression, rounded to the nearest 0.02 second	0.01 through 167 772.16
priority	numeric expression, rounded to an integer; default=1	1 through 15
line label	name of a program line	any valid name
line number	integer constant identifying a program line	1 through 32 766
subprogram name	name of a SUB or CSUB subprogram	any valid name

Example Statements

```
ON CYCLE Seconds,Priority CALL Sub_name

ON CYCLE Max_time RECOVER Backup

ON CYCLE 3600,3 GOTO 1200
```

Details

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 that 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.

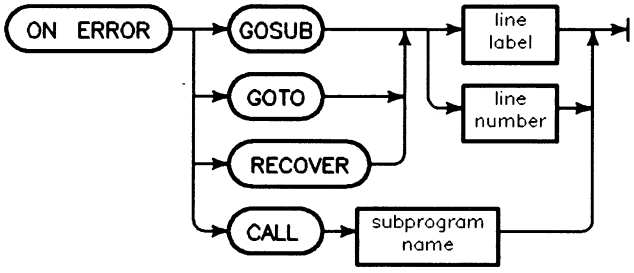
CALL and RECOVER remain active when the context changes to a subprogram, unless the change in context is caused by a keyboard-originated call. GOSUB and GOTO remain active when the context changes to a subprogram, but the branch cannot be taken until the calling context is restored.

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

ON ERROR defines and enables an event-initiated branch, which results from a trappable error. This allows you to write your own error-handling routines.

Syntax



Item	Description	Range
line label	name of a program line	any valid name
line number	integer constant identifying a program line	1 through 32 766
subprogram name	name of a SUB or CSUB subprogram	any valid name

Example Statements

```
ON ERROR GOTO 1200

ON ERROR RECOVER Crash

ON ERROR CALL Report
```

Details

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 via RETURN, SUBEXIT, or SUBEND 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 that 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, unless the change in context is caused by a keyboard-originated call. In this case, the error is reported to the user, as if ON ERROR had not been executed.

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.

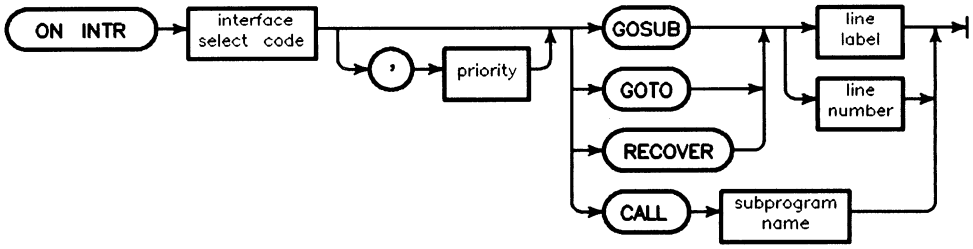
If an ON ERROR routine cannot be serviced because inadequate memory is available for the computer, the original error is reported and program execution pauses at that point.

ON ERROR is deactivated by OFF ERROR. DISABLE does not affect ON ERROR.

ON INTR

ON INTR defines an event-initiated branch to be taken when an interface card generates an interrupt. The interrupts must be explicitly enabled using ENABLE INTR.

Syntax



Item	Description	Range
interface select code	numeric expression, rounded to an integer	5, 7 through 31
priority	numeric expression, rounded to an integer; default=1	1 through 15
line label	name of a program line	any valid name
line number	integer constant identifying a program line	1 through 32 766
subprogram name	name of a SUB or CSUB subprogram	any valid name

Example Statements

```
ON INTR 7 GOTO 500

ON INTR Hpib,4 GOSUB Service

ON INTR Isc,Priority CALL Sub_name
```

Details

The occurrence of an interrupt performs an implicit DISABLE INTR for the interface. An ENABLE INTR must be performed to reenable 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 that 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 that 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.

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.

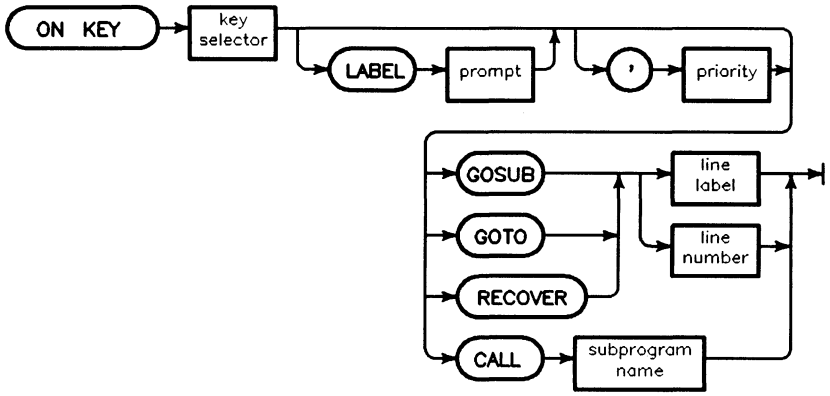
ON INTR is disabled by DISABLE INTR or DISABLE and deactivated by OFF INTR.

ON INTR and OFF INTR statements may be executed for *any* I/O card in the machine. It is not necessary to have a driver for the card.

ON KEY

This statement defines and enables an event-initiated branch to be taken when a softkey is pressed.

Syntax



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
```

Details

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 that 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.

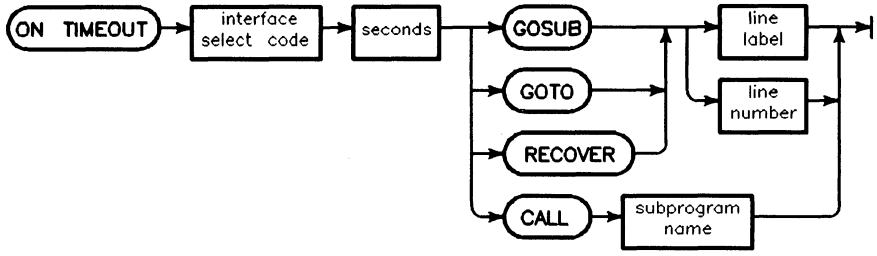
ON TIMEOUT

ON TIMEOUT defines and enables an event-initiated branch to be taken when an I/O timeout occurs on the specified interface.

Example Statements

```
ON TIMEOUT 7,4 GOTO 770
ON TIMEOUT Printer,Time GOSUB Message
```

Syntax



Item	Description	Range
interface select code	numeric expression rounded to an integer	7 through 31
seconds	numeric expression, rounded to the nearest 0.001 second for BASIC/WS and 0.020 second for BASIC/UX	
line label	name of program line	any valid name
line number	integer contstant identifying a program line	1 through 32 766
subprogram name	name of a SUB or CSUB subprogram	any valid name

Details

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 CONTROL, STATUS, READIO, WRITEIO, 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.

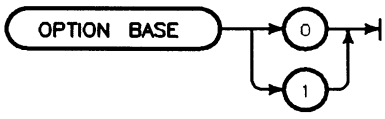
CALL and RECOVER remain active when the context changes to a subprogram, unless the change in context is caused by a keyboard-originated call. GOSUB and GOTO do not remain active when the context changes to a subprogram. The TIMEOUT event does remain active. Unlike other ON events, TIMEOUTs are never 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.

OPTION BASE

OPTION BASE specifies the default lower bound of arrays. Zero is the default lower bound unless OPTION BASE 1 statement is executed.

Syntax



Example Statements

```
OPTION BASE 0

OPTION BASE 1
```

Details

OPTION BASE determines the default lower bound for arrays that you declare without specifying a lower bound. If you specify an explicit upper and lower bound, it takes precedence over the OPTION BASE specification. The following code segment illustrates this behavior:

```
100 OPTION BASE 1
110 DIM Var(5:10) ! The lower bound of Var is 5.
```

OPTION BASE can occur only once in each context. If used, OPTION BASE must precede any explicit variable declarations in a context. Since arrays are passed to subprograms by reference, they maintain their original lower bound, even if the new context has a different OPTION BASE. Any context that does not contain an OPTION BASE statement assumes default lower bounds of zero.

The OPTION BASE value is determined at prerun, and is used with all arrays declared without explicit lower bounds in COM, DIM, INTEGER, and REAL statements as well as with all implicitly dimensioned arrays. OPTION BASE is also used at run time for any arrays declared without lower bounds in ALLOCATE.

OR

OR returns a 1 or a 0 based on the logical inclusive OR of the arguments.

Syntax

Example Statements

```
X=Y OR Z
```

```
IF File_type OR Device THEN Process
```

Details

An expression that evaluates to a non-zero value is treated as a logical 1. An expression must evaluate to zero to be treated as a logical 0.

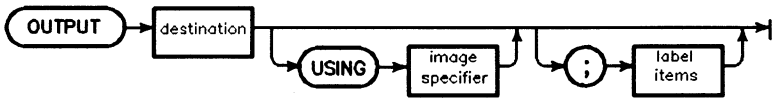
The truth table is

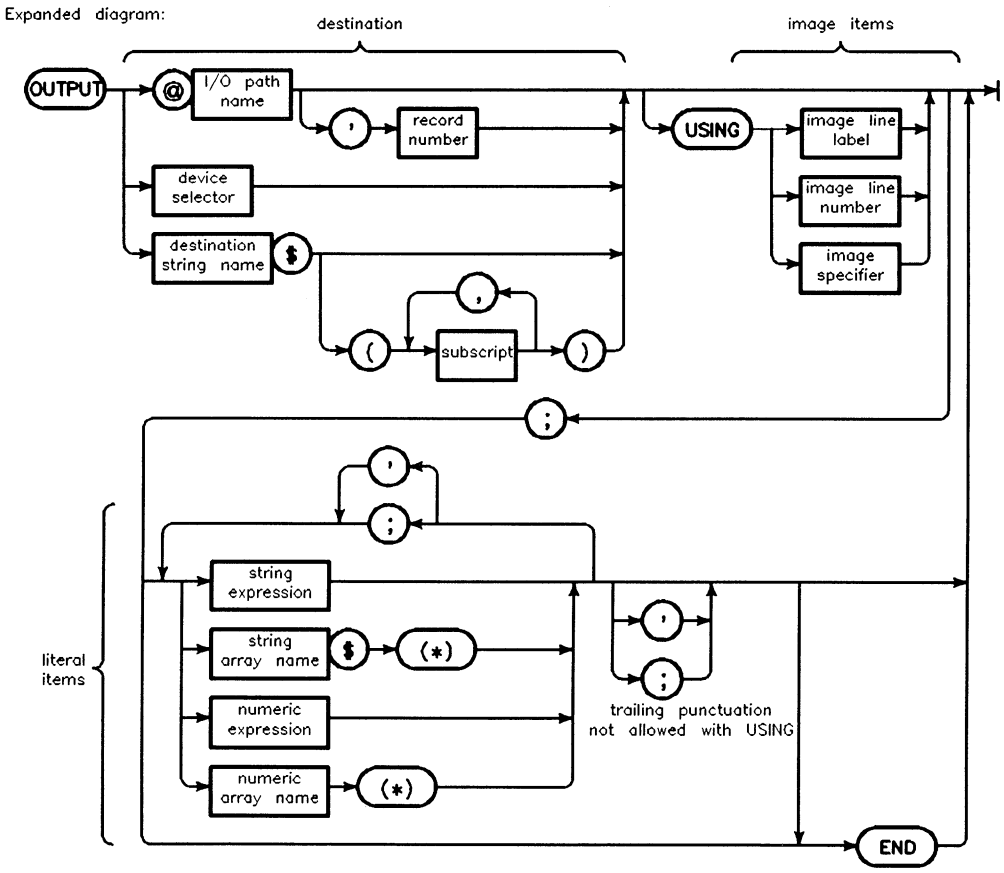
A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

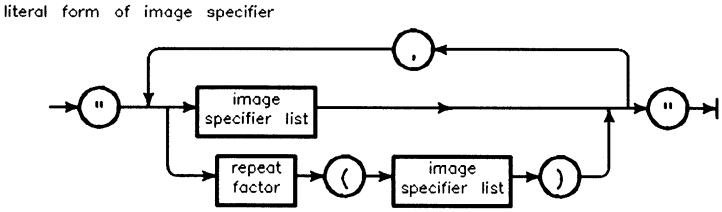
OUTPUT

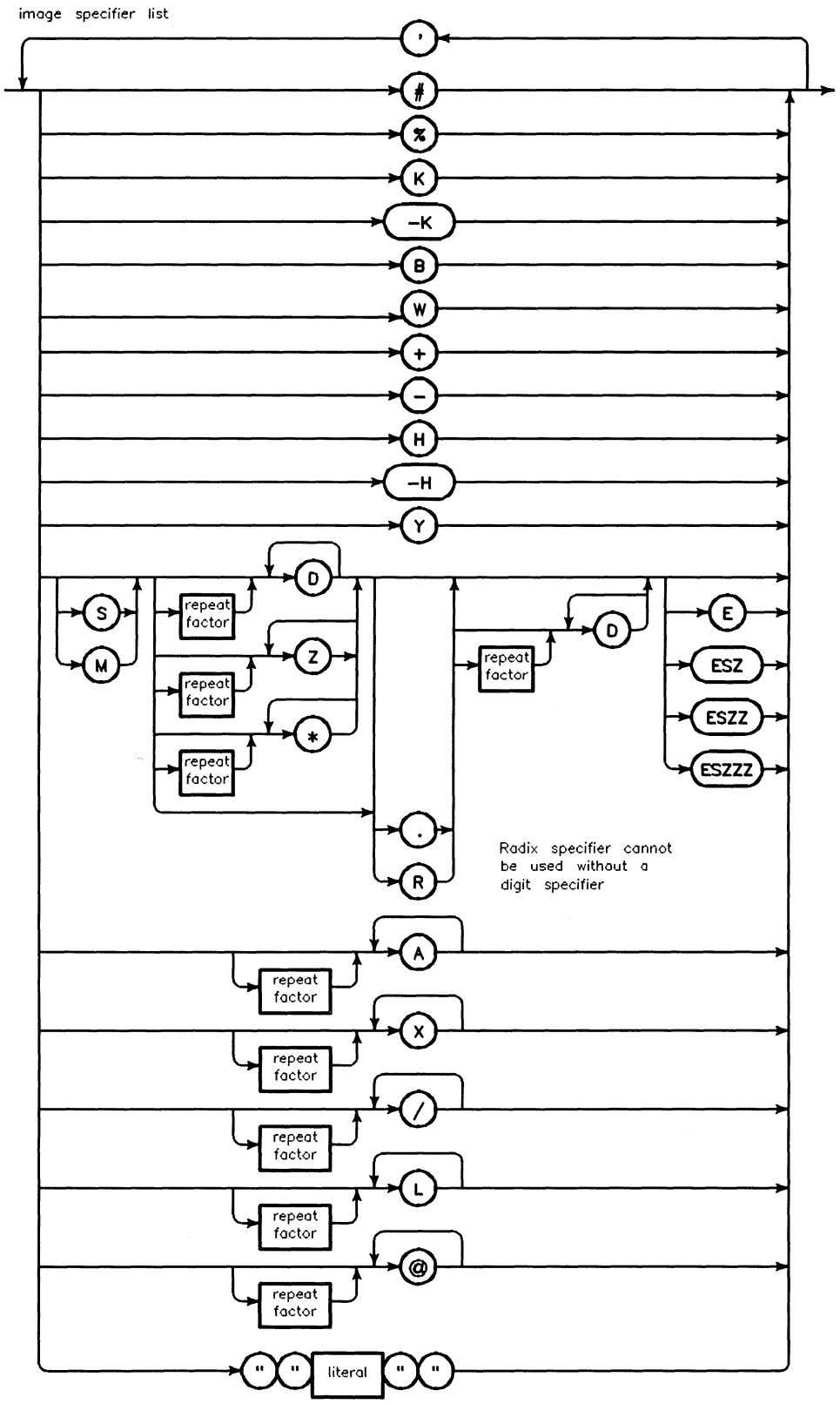
OUTPUT outputs items to the specified destination.

Syntax









Item	Description	Range
I/O path name	name assigned to a device, devices, mass storage file, buffer, or pipe	any valid name
record number	numeric expression, rounded to an integer	1 through $2^{31}-1$
device selector	numeric expression, rounded to an integer	(see Glossary)
destination string name	name of a string variable	any valid name
image line number	integer constant identifying an IMAGE statement	1 through 32 766
subscript	numeric expression, rounded to an integer	−32 767 through +32 767 (see "array" in Glossary)
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 through 32 767
literal	string constant composed of characters from the keyboard	quote mark not allowed

Example Statements

```
OUTPUT 701;Number,String$;
OUTPUT @File;Array(*),END
OUTPUT @Random,Record USING Fmt1;Item(5)
OUTPUT 12 USING "#,6A";B$[2;6]
OUTPUT Dest$ USING 110;A/1000,VAL$(Res)
OUTPUT @Printer;Rank;Id;Name$
```

Details

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 with OUTPUT statements. The file must be an ASCII, BDAT, or DOS file. The attributes specified in the ASSIGN statement are used if the file is a BDAT or DOS file. LIF ASCII files (files created by CREATE ASCII are always assigned a special case of the FORMAT ON attribute.

Serial access is available for ASCII, BDAT, and DOS files. Random access is available for BDAT and DOS 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.
- When a CONTROL statement directed to the I/O path name changes the position of the EOF.

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 a LIF ASCII file (a file created with CREATE ASCII), each item is sent as an ASCII representation with a 2-byte length header. You cannot use OUTPUT with USING to LIF ASCII files; see the following section, "OUTPUT with USING" for details.

Data sent to a BDAT, DFS, 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.)

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 and eight bytes are sent 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.

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 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 LIF ASCII files (files created by CREATE ASCII). Instead, direct the OUTPUT with USING to a string variable, and then OUTPUT this variable to the file.

```
100 OUTPUT String$ USING "5A,X,6D.D";Chars$,Number
110 OUTPUT @File;String$
```

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). (Requires IO.)
−H	Same as H. (Requires IO.)
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. (Requires IO.)
.	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.

Image Specifier	Meaning
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. Use `AA` or `2A` for two-byte globalization characters.
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.
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 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 output; the most-significant byte is sent 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 output in a single operation. If an I/O path name with the WORD attribute is used to access a 16-bit interface, a null pad byte is output whenever necessary to achieve alignment on a word boundary. If the destination is a BDAT file, string variable, or buffer, the BYTE or WORD attribute is ignored and all data are sent as bytes; however, pad byte(s) will be output when necessary to achieve alignment on a word boundary. The pad character may be changed by using the CONVERT attribute; see the ASSIGN statement for further information.
Y	Like W, except that no pad bytes are output to achieve word alignment. If an I/O path with the BYTE attribute is used to access a 16-bit interface, the BYTE attribute is not overridden (as with the W specifier above). (Requires IO.)
#	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. (Requires IO.)
−	Changes the automatic EOL sequence that normally follows the last output item to a single line-feed. (Requires IO.)
/	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 redefining 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.

Note

Some localized versions of HP Instrument BASIC, such as Japanese localized HP Instrument BASIC, support two-byte characters. When using this localized language remember that the IMAGE, ENTER USING, OUTPUT USING, and PRINT USING statements define a one-byte ASCII character image with `A`. Use the image `AA` to designate a two-byte character.

END with OUTPUT ... USING

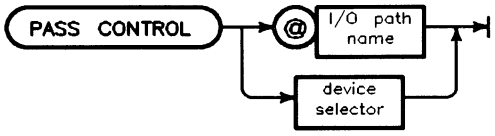
Using the optional secondary keyword END in an OUTPUT ... USING statement produces results that differ from those in an OUTPUT statement without USING. Instead of always suppressing the EOL sequence, the END keyword only suppresses the EOL sequence when no data is output from the last output item. Thus, 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

PASS CONTROL passes Active Controller capability to a specified GPIB device.

Syntax



Item	Description	Range
I/O path name	name assigned to an GPIB 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 @Device
```

Details

PASS CONTROL 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. The controller may be either a System or Non-system controller.

Summary of Bus Actions

	System Controller		Not System Controller	
	Interface Select Code Only	Primary Address Specified	Interface Select Code Only	Primary Address Specified
Active Controller	Error	ATN UNL TAD TCT $\overline{\text{ATN}}$	Error	ATN UNL TAD TCT $\overline{\text{ATN}}$
Not Active Controller	Error	Error	Error	Error

PAUSE

PAUSE temporarily suspends program execution; execution can be continued by pressing CONTINUE.

Syntax

Example Statement

```
PAUSE
```

Details

PAUSE suspends program execution until you click on Cont in the control pad or execute CONT from the command line. If the program is modified while paused, RUN must be used to restart program execution.

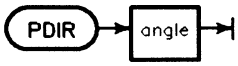
When program execution resumes, the computer attempts to service any ON INTR events that occurred while the program was paused. ON ERROR and ON TIMEOUT events generate errors if they occur while the program is paused.

Clicking on Pause in the control pad or executing PAUSE from the command line suspends program execution at the end of the line currently being executed.

PDIR

PDIR specifies the rotation angle at which the output from IPLOT, RPLOT, POLYGON, POLYLINE, and RECTANGLE is drawn. The angle is measured counterclockwise from the X axis using the current angle mode (DEG or RAD).

Syntax



Item	Description	Range
angle	numeric expression in current units of angle; default = 0	—

Example Statements

```
PDIR 30

IF Done THEN PDIR Old_angle
```

Details

The rotation is about the local origin of the RPLOT, POLYGON, POLYLINE or RECTANGLE. The local origins are defined as follows:

- RPLOT - pen position before execution of RPLOT
- POLYGON - center
- POLYLINE - center
- RECTANGLE - bottom-left corner

The rotation angle specified with PDIR is measured counterclockwise from the positive X axis. Thus, the positive X axis is at zero degrees, the positive Y axis is at 90 degrees, and so on.

PEN

PEN specifies the color of the graphics pen used for drawing lines and labels.

Syntax

Item	Description	Range
pen selector	numeric expression, rounded to an integer	−32 768 through +32 767 (device dependent)

Example Statements

```
PEN 2

PEN -1

PEN Pen
```

Details

This statement specifies the line color or physical pen to be used for all subsequent lines and labels until one of the following executes:

- another PEN statement
- a PLOT, IPLOT, or RPLOT statement with an array argument which changes the pen color (see Operation Selector 3 of these statements)
- a GINIT statement

The PEN statement can be used to specify that the current drawing mode is to erase lines on all devices which support such an operation. This is specified with a negative pen number. An alternate mode of operation which allows non-dominant and complementing drawing may be accessed through the GESCAPE statement. "Complement" means to change the state of pixels; that is, to draw lines where there are none, and to erase where lines already exist. When the PEN statement is executed, the pen used is mapped into the appropriate range, retaining the sign. The formulae used are as follows:

For color displays *not* in COLOR MAP mode:

If pen selector > 0 then use PEN (pen selector − 1) MOD 7 + 1
If pen selector = 0 then use PEN 0 (complement)
If pen selector < 0 then use PEN − ((ABS(pen selector) − 1) MOD 7 + 1)

For displays in COLOR MAP mode:

If pen selector>0 then use PEN (pen selector-1) MOD MaxPen + 1
If pen selector=0 then use PEN 0
If pen selector<0 then use PEN − ((ABS(pen selector) - minus; 1) MOD MaxPen + 1)

Where MaxPen is the highest pen number (the lowest is 0). Four planes: MaxPen=15; six planes: MaxPen=63; eight planes: MaxPen=255.

For an HPGL Plotter:

Use PEN pen selector

Where MaxPen is the highest available pen number (the lowest is 0). MaxPen is 15 or 31, depending on the video hardware and driver used by your computer.

Non-Color Map Mode

The value written into the frame buffer depends not only on what pen is being used, but whether or not the computer is in color map mode. The colors or gray levels for the default (non-color map) mode are given because the color map cannot be changed in this mode.

The meanings of the different pen values are shown in the table below. The pen value can cause either a 1 (draw), a 0 (erase), no change, or invert the value of each location in the frame buffer.

Non-Color Map Mode

Pen	Color	Plane 1 (Red)	Plane 2 (Green)	Plane 3 (Blue)
1	White	1	1	1
2	Red	1	0	0
3	Yellow	1	1	0
4	Green	0	1	0
5	Cyan	0	1	1
6	Blue	0	0	1
7	Magenta	1	0	1

Drawing with the pen numbers indicated in the above table results in the frame buffer planes being set to the indicated values. Drawing with the negatives of the pen numbers while in *normal pen mode* causes the bits to be cleared where there are 1s in the table. Drawing with the negatives of the pen numbers while in *alternate pen mode* causes the bits to be inverted where there are 1s in the table. In either case, no change will take place where there are 0s in the table. Although complementing lines can be drawn, complementing area fills cannot be executed.

Positive pen numbers in alternate drawing mode allows non-dominant drawing. (Non-dominant drawing causes the values in the frame buffer to be inclusively ORed with the value of the pen.) Pen 0 in normal mode complements. Pen 0 in alternate mode draws in the background color. Since the table represents the computer in non-color map mode, table entries for any additional frame buffer planes are all zeros.

Default Colors

The default pen colors while in color map mode are shown in the following tables. These can be changed by the SET PEN statement.

Pen	Color	Red	Green	Blue
0	Black	0	0	0
1	White	1	1	1
2	Red	1	0	0
3	Yellow	1	1	0
4	Green	0	1	0
5	Cyan	0	1	1
6	Blue	0	0	1
7	Magenta	1	0	1
8	Black	0	0	0
9	Olive Green	.80	.73	.20
10	Aqua	.20	.67	.47
11	Royal Blue	.53	.40	.67
12	Maroon	.80	.27	.40
13	Brick Red	1.00	.40	.20
14	Orange	1.00	.47	0.00
15	Brown	.87	.53	.27

PENUP

PENUP lifts the pen on the current plotting device.

Syntax

Example Statement

```
PENUP
```

PI

PI returns 3.14159265358979, which is an approximate value for pi.

Syntax

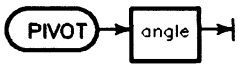
Example Statements

```
Area=PI*Radius^2
PRINT X,X*2*PI
```

PIVOT

PIVOT specifies a rotation of coordinates which is applied to all drawn lines, but not to labels or axes.

Syntax



Item	Description	Range
angle	numeric expression in current units of angle	(same as COS)

Example Statements

```
PIVOT 30

IF Special THEN PIVOT Radians
```

Details

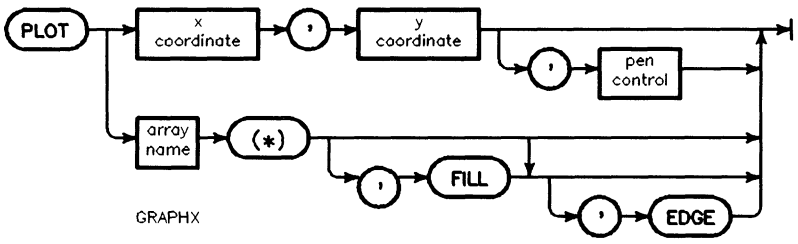
The specified angle is interpreted according to the current angle mode (RAD or DEG).

The specified angular rotation is performed about the logical pen's position at the time the PIVOT is executed. This rotation is applied only to lines drawn subsequent to the PIVOT; logical pen movement is *not* affected by PIVOT. Consequently, PIVOT generally causes the logical and physical pens to be left at different positions.

PLOT

PLOT moves the graphics pen from the current pen position to the specified X and Y coordinates.

Syntax



Item	Description	Range
x coordinate	numeric expression, in current units	—
y coordinate	numeric expression, in current units	—
pen control	numeric expression, rounded to an integer; default = 1 (down after move)	−32 768 through +32 767
array name	name of two-dimensional, two-column or three-column numeric array. (Requires GRAPHX)	any valid name

Example Statements

```
PLOT 20,90
PLOT Next_x,Next_y,Pen_control
```

Details

Non-Array Parameters

The specified X and Y position information is interpreted according to the current unit-of-measure. Lines are drawn using the current pen color and line type.

PLOT is affected by the PIVOT transformation.

The line is clipped at the current clipping boundary. If none of the line is inside the current clip limits, the pen is not moved, but the logical pen position is updated.

The optional pen control parameter specifies the following plotting actions; the default value is +1 (down after move).

Pen Control	Resultant Action
−Even	Pen up before move
−Odd	Pen down before move
+Even	Pen up after move
+Odd	Pen down after move

Graphics Transformations

The output of PLOT is affected by *only* these graphics transformations:

- scaling specified by WINDOW
- scaling specified by SHOW
- rotations specified by PIVOT

Array Parameters

When using the PLOT statement with an array, either a two-column or a three-column array may be used. If a two-column array is used, the third parameter is assumed to be +1; pen down after move.

FILL and EDGE

When FILL or EDGE is specified, each sequence of two or more lines forms a polygon. The polygon begins at the first point on the sequence, includes each successive point, and the final point is connected or closed back to the first point. A polygon is closed when the end of the array is reached, or when the value in the third column is an even number less than three, or in the range 5 to 8 or 10 to 15.

If FILL and/or EDGE are specified on the PLOT statement itself, it causes the polygons defined within it to be filled with the current fill color and/or edged with the current pen color. If polygon mode is entered from within the array, and the FILL/EDGE directive for that series of polygons differs from the FILL/EDGE directive on the PLOT statement itself, the directive in the array replaces the directive on the statement. In other words, if a "start polygon mode" operation selector (a 6, 10, or 11) is encountered, any current FILL/EDGE directive (whether specified by a keyword or an operation selector) is replaced by the new FILL/EDGE directive.

If FILL and EDGE are both declared on the PLOT statement, FILL occurs first. If neither one is specified, simple line drawing mode is assumed; that is, polygon closure does not take place.

When using a PLOT statement with an array, the following table of **operation selectors** applies. An operation selector is the value in the third column of a row of the array to be plotted. The array must be a two-dimensional, two-column or three-column array. If the third column exists, it will contain operation selectors which instruct the computer to carry out certain operations. Polygons may be defined, edged (using the current pen), filled (using the current fill color), pen and line type may be selected, and so forth.

Column 1	Column 2	Operation Selector	Meaning
X	Y	−2	Pen up before moving
X	Y	−1	Pen down before moving
X	Y	0	Pen up after moving (Same as +2)
X	Y	1	Pen down after moving
X	Y	2	Pen up after moving
pen number	ignored	3	Select pen
line type	repeat value	4	Select line type
color	ignored	5	Color value
ignored	ignored	6	Start polygon mode with FILL
ignored	ignored	7	End polygon mode
ignored	ignored	8	End of data for array
ignored	ignored	9	NOP (no operation)
ignored	ignored	10	Start polygon mode with EDGE
ignored	ignored	11	Start polygon mode with FILL and EDGE
ignored	ignored	12	Draw a FRAME
pen number	ignored	13	Area pen value
red value	green value	14	Color
blue value	ignored	15	Value
ignored	ignored	>15	Ignored

Moving and Drawing

If the operation selector is less than or equal to two, it is interpreted in exactly the same manner as the third parameter in a non-array PLOT statement. Even is up, odd is down, positive is after pen motion, negative is before pen motion. Zero is considered positive.

Selecting Pens

An operation selector of 3 selects a pen. The value in column one is the pen number desired. The value in column two is ignored.

Selecting Line Types

An operation selector of 4 selects a line type. The line type (column one) selects the pattern, and the repeat value (column two) is the length in GDUs that the line extends before a single occurrence of the pattern is finished and it starts over. On the CRT, the repeat value is evaluated and rounded *down* to the next multiple of 5, with 5 as the minimum.

Selecting a Fill Color

Operation selector 13 selects a pen from the color map with which to do area fills. This works identically to the AREA PEN statement. Column one contains the pen number.

Defining a Fill Color

Operation selector 14 is used in conjunction with operation selector 15. Red and green are specified in columns one and two, respectively, and column three has the value 14. Following this row in the array (not necessarily immediately), is a row whose operation selector in column three has the value of 15. The first column in that row contains the blue value. These numbers range from 0 to 32 767, where 0 is no color and 32 767 is full intensity. Operation selectors 14 and 15 together comprise the equivalent of an AREA INTENSITY statement.

Operation selector 15 actually puts the area intensity into effect, but only if an operation selector 14 has already been received.

Operation selector 5 is another way to select a fill color. The color selection is through a Red-Green-Blue (RGB) color model. The first column is encoded in the following manner. There are three groups of five bits right-justified in the word; that is, the most significant bit in the word is ignored. Each group of five bits contains a number which determines the intensity of the corresponding color component, which ranges from zero to sixteen. The value in each field will be sixteen minus the intensity of the color component. For example, if the value in the first column of the array is zero, all three five-bit values would thus be zero. Sixteen minus zero in all three cases would turn on all three color components to full intensity, and the resultant color would be a bright white.

Assuming you have the desired intensities (which range from 0 thru 1) for red, green, and blue in the variables `R`, `G`, and `B`, respectively, the value for the first column in the array could be defined thus:

```
Array(Row,1)=SHIFT(16*(1-B),-10)+SHIFT(16*(1-G),-5)+16*(1-R)
```

If there is a pen color in the color map similar to that which you request here, that non-dithered color will be used. If there is not a similar color, you will get a dithered pattern.

If you are using a gray scale display, Operation selector 5 uses the five bit values of the RGB color specified to calculate luminosity. The resulting gray luminosity is then used as the area fill.

Polygons

A six, ten, or eleven in the third column of the array begins a "polygon mode." If the operation selector is 6, the polygon will be filled with the current fill color. If the operation selector is 10, the polygon will be edged with the current pen number and line type. If the operation selector is 11, the polygon will be both filled and edged. Many individual polygons can be filled without terminating the mode with an operation selector 7. This can be done by specifying several series of draws separated by moves. The first and second columns are ignored and should not contain the X and Y values of the first point of a polygon.

Operation selector 7 in the third column of a plotted array terminates definition of a polygon to be edged and/or filled and also terminates the polygon mode (entered by operation selectors 6, 10, or 11). The values in the first and second columns are ignored, and the X and Y values of the last data point should not be in them. Edging and/or filling of the most recent polygon will begin immediately upon encountering this operation selector.

Doing a FRAME

Operation selector 12 does a FRAME around the current soft-clip limits. Soft clip limits cannot be changed from within the PLOT statement, so one probably would not have more than one operation selector 12 in an array to PLOT, since the last FRAME will overwrite all the previous ones.

Premature Termination

Operation selector 8 causes the PLOT statement to be terminated. The PLOT statement will successfully terminate if the actual end of the array has been reached, so the use of operation selector 8 is optional.

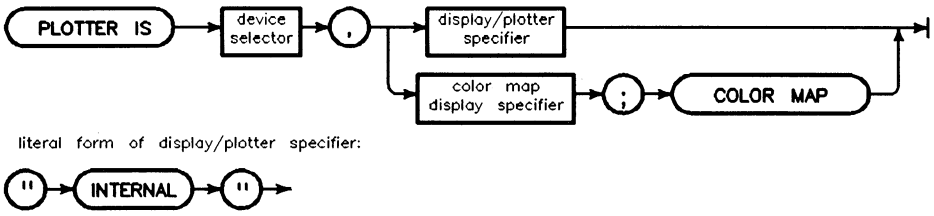
Ignoring Selected Rows in the Array

Operation selector 9 causes the row of the array it is in to be ignored. Any operation selector greater that fifteen is also ignored, but operation selector 9 is retained for compatibility reasons. *Operation selectors less than -2 are not ignored*. If the value in the third column is less than zero, only evenness/oddness is considered.

PLOTTER IS

PLOTTER IS determines whether graphics colors operate in the color mapped or non-color mapped mode. This behavior is determined by the use of the secondary keywords COLOR MAP; the default mode of operation is color-mapped.

Syntax



Item	Description	Range
device selector	numeric expression, rounded to an integer	(see Glossary)
display/plotter specifier	string expression	(see drawing)
color map display specifier	string expression	INTERNAL or WINDOW

Example Statements

```
PLOTTER IS CRT,"INTERNAL" ! Use display in non-color mapped mode.

PLOTTER IS CRT,"INTERNAL";COLOR MAP ! Use display
in color mapped mode.
```

Details

In HP Instrument BASIC, PLOTTER IS does not allow graphics output to be sent to any device selector other than 1 (the graph window). Graphics output is always sent to the graph window. The only use for PLOTTER IS is to switch between color-mapped and non-color-mapped mode using the secondary keywords COLOR MAP.

In non-color-mapped mode, you can use only the default colors for pens described in dictionary entry for PEN. In color-mapped mode, you can define your own pen colors using SET PEN.

Displays

Non-Color Map Mode

Executing a PLOTTER IS statement *without* the COLOR MAP keyword causes the color map to be defined as follows, where 0 is zero intensity and 1 is full intensity.

Pen	Color	Red	Green	Blue
0	Complement	0	0	0
1	White	1	1	1
2	Red	1	0	0
3	Yellow	1	1	0
4	Green	0	1	0
5	Cyan	0	1	1
6	Blue	0	0	1
7	Magenta	1	0	1

Default Pen Colors

The PLOTTER IS statement defines the color map to default values. These values are different depending on whether or not the COLOR MAP option was selected.

Note that the color assignments for pens 16-31 depend on the video hardware and drivers used by your computer. If your video hardware and drivers support 256 colors or more, HP Instrument BASIC will assign the colors listed in the following table to pens 16-31. If your video hardware and drivers do not support at least 256 colors, x HP Instrument BASIC will assign the same colors to pens 16-31 that are used for pens 1-15. Thus, on a video display that supports fewer than 256 colors (such as VGA), pens 1 and 16 are the same color, pens 2 and 17 are the same color, and so on.

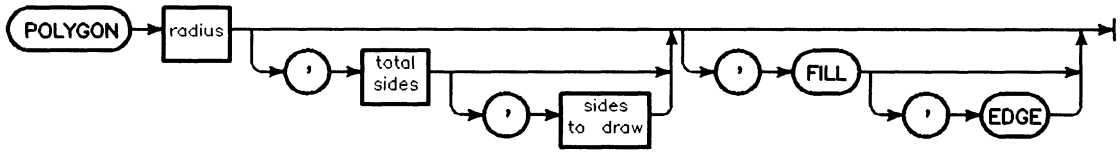
Color Map Default Color Definitions (RGB)

Pen	Color	Red	Green	Blue
0	Black	0	0	0
1	White	1	1	1
2	Red	1	0	0
3	Yellow	1	1	0
4	Green	0	1	0
5	Cyan	0	1	1
6	Blue	0	0	1
7	Magenta	1	0	1
8	Black	0	0	0
9	Olive Green	.80	.73	.20
10	Aqua	.20	.67	.47
11	Royal Blue	.53	.40	.67
12	Maroon	.80	.27	.40
13	Brick Red	1.00	.40	.20
14	Orange	1.00	.47	0.00
15	Brown	.87	.53	.27

POLYGON

POLYGON draws all or part of a closed regular polygon.

Syntax



Item	Description	Range
radius	numeric expression, in current units	—
total sides	numeric expression, rounded to an integer. default = 60	3 through 32 767
sides to draw	numeric expression, rounded to an integer. default = all sides	1 through 32 767

Example Statements

```
POLYGON Radius,Total_sides,Drawn_sides
POLYGON -Size,5,FILL,EDGE
```

Details

The radius is the distance that the vertices of the polygon will be from the logical pen position. The first vertex will be at a distance specified by "radius" in the direction of the positive X-axis. Specifying a negative radius results in the figure being rotated 180 degrees.

The total sides and the number of sides drawn need not be the same. Thus,

```
POLYGON 1.5,8,5
```

will start to drawn an octagon whose vertices are 1.5 units from the current pen position, but will only draw five sides of it before closing the polygon at the first point. If the number of sides to draw is greater than the specified total sides, sides to draw is treated as if it were equal to total sides.

POLYGON forces polygon closure, that is, the first vertex is connected to the last vertex, so there is always an inside and an outside area. This is true even for the degenerate case of drawing only one side of a polygon, in which case a single line results. This is actually two lines, from the first point to the last point, and back to the first point.

Graphics Transformations

The output of POLYGON is affected by *only* these graphics transformations:

- scaling specified by WINDOW
- scaling specified by SHOW
- rotations specified by PIVOT
- rotations specified by PDIR

Polygon Shape

The shape of the polygon is affected by the viewing transformation specified by SHOW or WINDOW. Therefore, anisotropic scaling causes the polygon to be distorted; stretched or compressed along the axes. If a rotation transformation is in effect, the polygon will be rotated first, then stretched or compressed along the unrotated axes.

The pen status also affects the final shape of a polygon if sides to draw is less than total sides. If the pen is up at the time POLYGON is specified, the first vertex specified is connected to the last vertex specified, *not* including the center of the polygon, which is the current pen position. If the pen is down, however, the center of the polygon is also included in it. If sides to draw is less than total sides, piece-of-pie shaped polygon segments are created.

FILL and EDGE

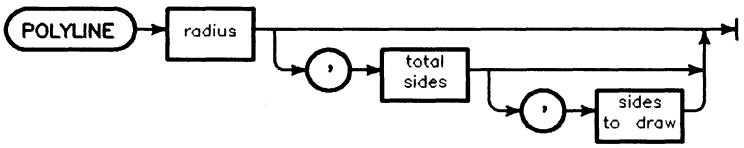
FILL causes the interior of the polygon or polygon segment to be filled with the current fill color as defined by AREA PEN, AREA COLOR, or AREA INTENSITY. EDGE causes the edges of the polygon to be drawn using the current pen and line type. If both FILL and EDGE are specified, the interior will be filled, then the edge will be drawn. If neither FILL nor EDGE is specified, EDGE is assumed.

After POLYGON has executed, the pen is in the same position it was before the statement was executed, and the pen is up. The polygon is clipped at the current clip limits.

POLYLINE

POLYLINE draws all or part of an open regular polygon.

Syntax



Item	Description	Range
radius	numeric expression, in current units	—
total sides	numeric expression, rounded to an integer. default = 60	3 through 32 767
sides to draw	numeric expression, rounded to an integer. default = all sides	1 through 32 767

Example Statements

```
POLYLINE Radius,Total_sides,Drawn_sides
POLYLINE -Size,5
```

Details

The radius is the distance that the vertices of the polygon will be from the current pen position. The first vertex will be at a distance specified by "radius" in the direction of the positive X-axis. Specifying a negative radius results in the figure being rotated 180 degrees.

The total sides and the number of sides drawn need not be the same. Thus,

```
POLYLINE 1.5,8,5
```

will start to drawn an octagon whose vertices are 1.5 units from the current pen position, but will only draw five sides of it. If the number of sides to draw is greater than the total sides specified, it is treated as if it were equal to the total sides.

Graphics Transformations

The output of POLYLINE is affected by *only* these graphics transformations:

- scaling specified by WINDOW
- scaling specified by SHOW
- rotations specified by PIVOT
- rotations specified by PDIR

Shape of Perimeter

POLYLINE does not force polygon closure, that is, if sides to draw is less than total sides, the first vertex is not connected to the last vertex, so there is no "inside" or "outside" area.

The shape of the polygon is affected by the viewing transformation specified by SHOW or WINDOW. Therefore, anisotropic scaling causes the perimeter to be distorted; stretched or compressed along the axes. If a rotation transformation is in effect, the polygon will be rotated first, then stretched or compressed along the unrotated axes.

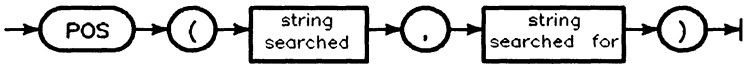
The pen status also affects the way a POLYLINE statement works. If the pen is up at the time POLYLINE is specified, the first vertex is on the perimeter. If the pen is down, the first point is the current pen position, which is connected to the first point on the perimeter.

After POLYLINE has executed, the current pen position is in the same position it was before the statement was executed, and the pen is up. The polygon is clipped at the current clip limits.

POS

POS returns the first position of a **substring** within a string.

Syntax



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
```

Details

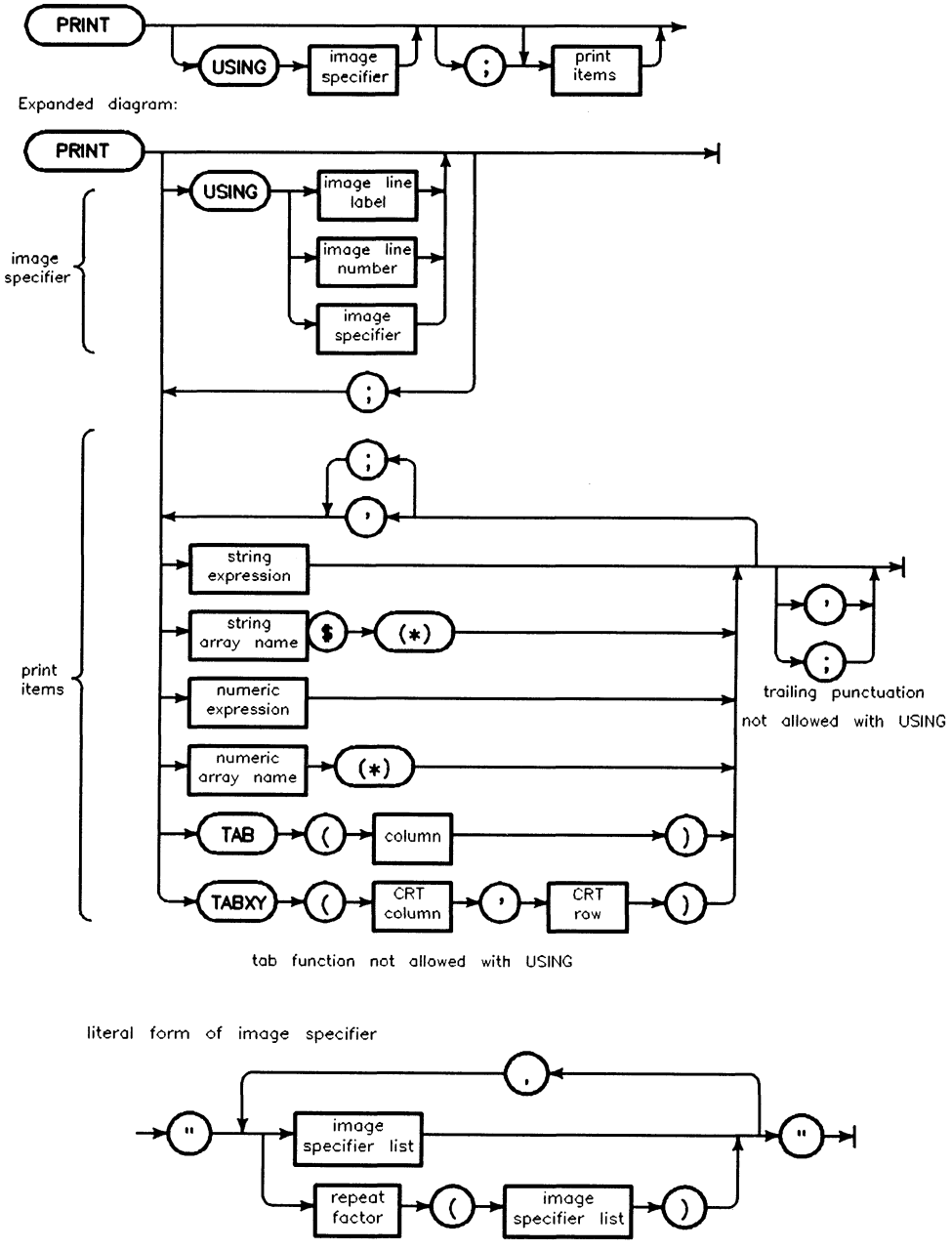
If the value returned is greater than 0, it is the position of the first character of the string being searched for in the string being searched. If the value returned is 0, the string being searched for cannot be found (or the string searched for is the null string).

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.

PRINT

PRINT sends items to the PRINTER IS device. The default PRINTER IS device is the alphanumeric display.

Syntax



Item	Description	Range
image line number	integer constant identifying an IMAGE statement	1 through 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 through screen width
CRT row	numeric expression, rounded to an integer	1 through alpha height
image specifier list	literal	(see next drawing)
repeat factor	integer constant	1 through 32 767
print items	string constant composed of characters from the keyboard, including those generated using the ANY CHAR key	quote mark not allowed

Example Statements

```
PRINT "LINE";Number
PRINT Array(*)
PRINT TABXY(1,1),Head$,TABXY(Col,3),Msg$
PRINT String$[1,8],TAB(12),Result
PRINT USING 125;X,Y,Z
PRINT USING "5Z.DD";Money
PRINT USING Fmt3;Id,Item$,Kilograms/2.2
```

Details

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 (50, 80 or 128 characters) 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.

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

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:

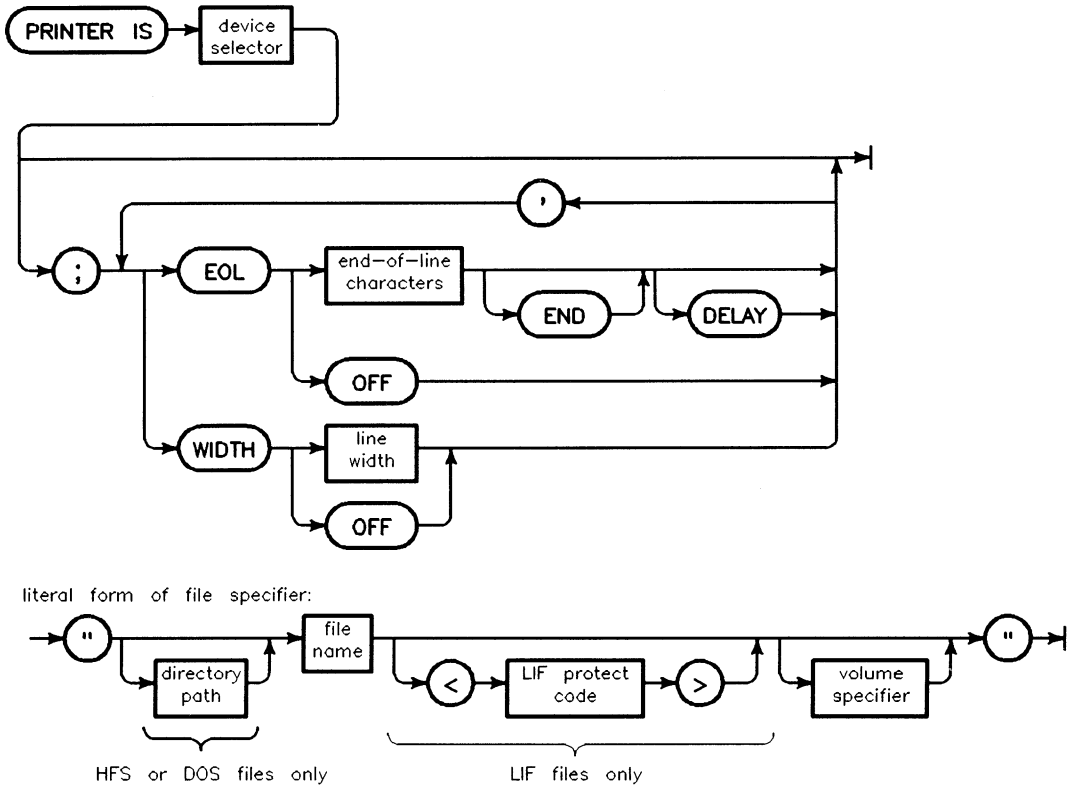
Image Specifier	Meaning
K	Compact field. Prints a number or string in standard form with no leading or trailing blanks.
−K	Same as K.
H	Similar to K, except the number is printed using the European number format (comma radix).
−H	Same as H.
S	Prints the number's sign (+ or −).
M	Prints the number's sign 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.
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.

Image Specifier	Meaning
#	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

PRINTER IS specifies the default destination for the output of various statements that send output to a "printer". The PRINTER IS device is set to 1 (the alpha window) at power-on and after SCRATCH A.

Syntax



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 through 8 characters
line width	numeric expression, rounded to an integer; default = (see text)	1 through 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 CRT

PRINTER IS "myfile"

PRINTER IS PRT;WIDTH 80
```

Details

The system printing device or file receives all data sent by the following statements in which the destination is not explicitly specified:

- PRINT
- CAT
- LIST

The default printing device is the alphanumeric display (select code 1) at power-on and after executing SCRATCH A.

Using PRINTER IS with WIDTH

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. If a USING clause is included in the PRINT statement, WIDTH is ignored. The default WIDTH for files is OFF.

Using PRINTER IS with Files

The file must be a BDAT or DOS file (a file created by CREATE).

The PRINTER IS file statement positions the file pointer to the beginning of the file you specify. Thus, PRINTER IS overwrites the file if it already exists.

You can read the file with ENTER if it is created using ASSIGN with FORMAT ON.

You may close the file by executing another PRINTER IS statement or SCRATCH A.

PROUND

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

Syntax

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

PRT returns 701, the default (factory set) device selector for an external printer.

Syntax

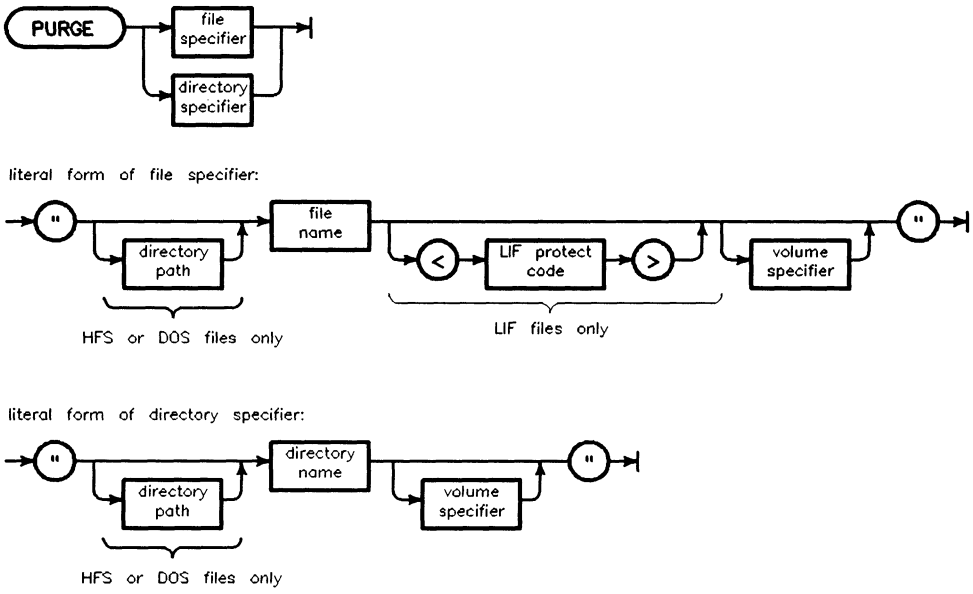
Example Statements

```
OUTPUT PRT;Text$
```

PURGE

PURGE deletes a file or directory.

Syntax



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 "C:\TEMP\*.*"

PURGE "Dir1/Dir2/Dir3"

PURGE "*"

PURGE "Monday_?"
```

Details

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 opened by ASSIGN can be closed by `ASSIGN TO *` (see ASSIGN). All files except those opened with the PRINTER IS statement are closed by clicking on the Stop button in the control pad. A PRINTER IS file can be closed by executing a PRINTER IS to another device or file. SCRATCH A closes all files.

If you are using a version of HP Instrument BASIC that supports wildcards, you can use them in file specifiers with PURGE. You must first enable wildcard recognition using WILDCARDS. Refer to the keyword entry for WILDCARDS for details.

Purging Files and Directories

To PURGE a 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.
- If it is a directory, it must be empty. That is, it must not contain any subordinate files or directories.

RAD

RAD selects radians as the unit of measure for angles.

Syntax

Example Statements

```
RAD
```

Details

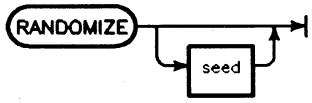
All functions which return an angle will return an angle in radians. All operations with parameters representing angles will interpret the angle in radians. If no angle mode is specified in a program, the default is radians (also see DEG).

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

RANDOMIZE selects a seed for the RND function.

Syntax



Item	Description	Range
seed	numeric expression, rounded to an integer; default = pseudo-random	1 through $2^{31}-2$

Example Statements

```
RANDOMIZE
RANDOMIZE Old_seed*PI
```

Details

The seed actually used by the random number generator depends on the absolute value of the seed specified in the RANDOMIZE statement.

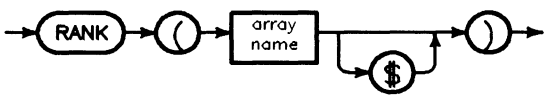
Absolute Value of Seed	Value Used
less than 1	1
1 through $2^{31}-2$	INT(ABS(seed))
greater than $2^{31}-2$	$2^{31}-2$

The seed is reset to 37 480 660 by power-up, SCRATCH A, SCRATCH, and program prerun.

RANK

RANK returns the number of dimensions in an array.

Syntax



Item	Description	Range
array name	name of an array	any valid name

Example Statements

```
Dimensions=RANK(Array$)
IF RANK(A)=2 THEN PRINT "A is a matrix"
```

RATIO

RATIO returns the ratio of the width (in pixels) to the height (in pixels) of the graph window.

Syntax

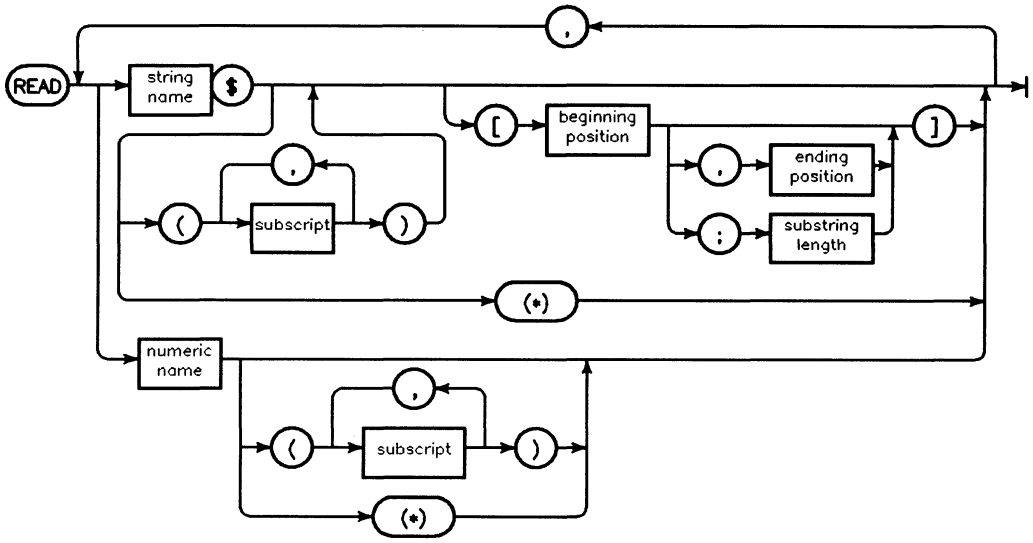
Example Statements

```
WINDOW 0,10*RATIO,-10,10
X_gdu_max=100*MAX(1,RATIO)
Y_gdu_max=100*MAX(1,1/RATIO)
```

READ

READ reads values from DATA statements and assigns them to variables.

Syntax



Item	Description	Range
string name	name of a string variable	any valid name
subscript	numeric expression, rounded to an integer	−32 767 through +32 767 (see "array" in Glossary)
beginning position	numeric expression, rounded to an integer	1 through 32 767 (see "substring" in Glossary)
ending position	numeric expression, rounded to an integer	0 through 32 767 (see "substring" in Glossary)
substring length	numeric expression, rounded to an integer	0 through 32 767 (see "substring" in Glossary)
numeric name	name of a numeric variable	any valid name

Example Statements

```
READ Number,String$
READ Array(*)
READ Field$[5,15]
READ Item(1,1),Item(2,1),Item(3,1)
```

Details

The numeric items stored in DATA statements are considered strings by the computer, and if they are READ into a numeric variable, they are first processed by a number builder to convert them to numbers. The number builder for READ recognizes the usual combinations of signs, digits, decimal points, and signed exponents. If the characters are not a valid representation of a number, an error results. If a number in a DATA statement contains a fractional part and is read into an INTEGER variable, it will be rounded up or down (not truncated) appropriately. A string variable may read numeric items, as long as it is dimensioned large enough to contain the characters.

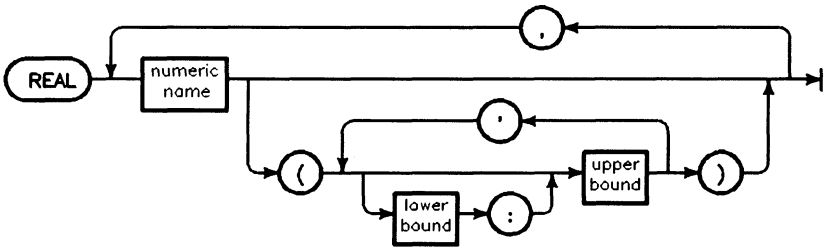
The first READ statement in a context accesses the first item in the first DATA statement in the context unless RESTORE has been used to specify a different DATA statement as the starting point. Successive READ operations access following items, progressing through DATA statements as necessary. Trying 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

REAL reserves storage for floating point variables and arrays.

Syntax



Item	Description	Range
numeric name	name of a numeric variable	any valid name
lower bound	integer constant; default = OPTION BASE value (0 or 1)	−32 767 through +32 767 (see "array" in Glossary)
upper bound	integer constant	−32 767 through +32 767 (see "array" in Glossary)

Example Statements

```
REAL X,Y,Z

REAL Array(-128:127,15)
```

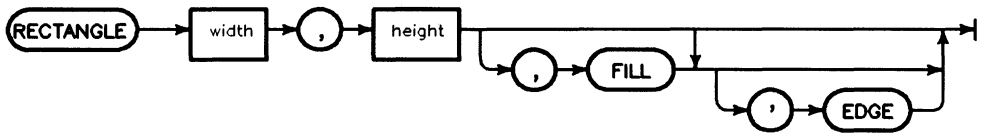
Details

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.

RECTANGLE

RECTANGLE draws a rectangle.

Syntax



Item	Description	Range
width	numeric expression	—
height	numeric expression	—

Example Statements

```
RECTANGLE Width,Height
RECTANGLE 4,-6,FILL,EDGE
```

Details

The rectangle is drawn with dimensions specified as displacements from the current pen position. Thus, both the width and the height may be negative.

Which corner of the rectangle is at the pen position at the end of the statement depends upon the signs of the parameters:

Sign of X	Sign of Y	Corner of Rectangle at Pen Position
+	+	Lower left
+	−	Upper left
−	+	Lower right
−	−	Upper right

Graphics Transformations

The output of RECTANGLE is affected by *only* these graphics transformations:

- scaling specified by WINDOW
- scaling specified by SHOW
- rotations specified by PIVOT
- rotations specified by PDIR

A rectangle's shape is affected by the current viewing transformation. If isotropic units are in effect, the rectangle will be the expected shape, but if anisotropic units (set by WINDOW) are in effect, the rectangle will be distorted; it will be stretched or compressed along the axes.

If a rotation transformation *and* anisotropic units (set by WINDOW) are in effect, the rectangle is rotated first, then stretched or compressed along the unrotated axes.

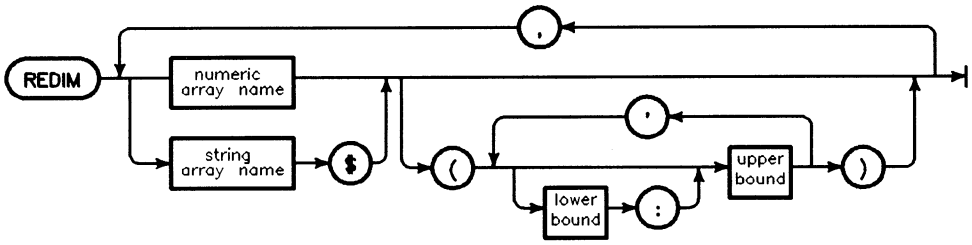
FILL and EDGE

FILL causes the rectangle to be filled with the current fill color, and EDGE causes the perimeter to be drawn with the current pen color and line type. If both FILL and EDGE are specified, the interior will be filled, then the edge will be drawn. If neither FILL nor EDGE is specified, EDGE is assumed.

REDIM

REDIM changes the subscript range of previously dimensioned arrays.

Syntax



Item	Description	Range
array name	name of an array	any valid name
lower bound	numeric expression, rounded to an integer; default=OPTION BASE value (0 or 1)	−32 768 through +32 767 (see "array" in glossary)
upper bound	numeric expression, rounded to an integer	−32 768 through +32 767 (see "array" in glossary)

Example Statements

```
REDIM Array(New_lower:New_upper)
REDIM String$(A,B,C)
```

Details

The following rules must be followed when redimensioning an array:

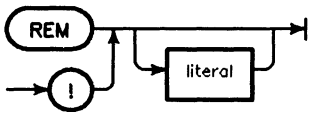
- The array to be redimensioned must have a currently dimensioned size known to the context (i.e., it must have been implicitly or explicitly dimensioned, or be currently allocated, or it must have been passed into the context.)
- You must retain the same number of dimensions as specified in the original dimension statement.
- The redimensioned array cannot have more elements than the array was originally dimensioned to hold.
- You cannot change the maximum string length of string arrays.

REDIM does not change any values in the array, although their locations will probably be different. The REDIM is performed left-to-right and if an error occurs, arrays to the left of the array the error occurs in will be redimensioned while those to the right will not be. If an array appears more than once in the REDIM, the right-most dimensions will be in effect after the REDIM.

REM

REM specifies that the remainder of a program line is a comment, not a program statement or label.

Syntax



Item	Description	Range
literal	string constant composed of characters from the keyboard	—

Example Statements

```
100 REM  Program Title
190 !
200 Info=0  ! Clear flag byte
```

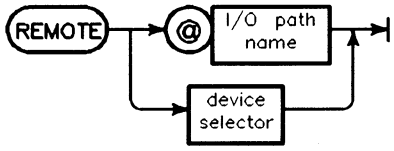
Details

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

REMOTE places GPIB devices having remote/local capabilities into the remote state.

Syntax



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 Device

REMOTE @Hpib
```

Details

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.

Bus Actions

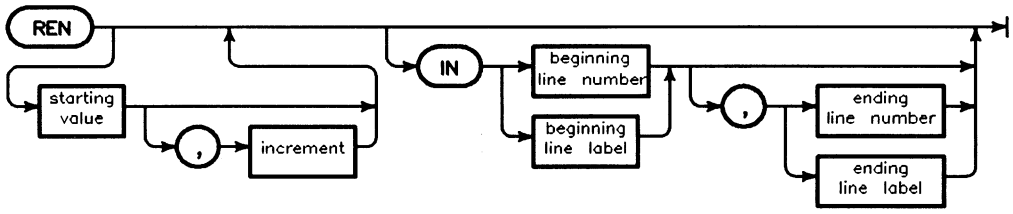
Summary of Bus Actions

	Interface Select Code Only	Primary Address Specified
Active Controller	REN $\overline{\text{ATN}}$	REN ATN MTA UNL LAG
Not Active Controller	REN	Error

REN

REN renumbers the lines in all or part of a program.

Syntax



Item	Description	Range
starting value	integer constant identifying a program line; default = 10	1 through 32 766
increment	integer constant; default = 10	1 through 32 767
beginning line number	integer constant identifying program line	1 through 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 through 32 766
ending line label	name of a program line	any valid name

Example Commands

```
REN       ! Renumbers the entire program by 10s.
REN 1000  ! Renumbers starting at line 1000 by 10s.
REN 100,2 ! Renumbers starting at line 100 by 2s.

REN 261,1 IN 260,Label2 ! Renumbers the range 260-Label2
                        ! starting with 261 by 1s.
```

Details

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.

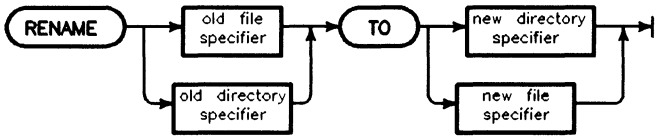
REN cannot be used to move lines. If renumbering would cause lines to overlap preceding or following lines, an error occurs and no renumbering takes place.

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 the line labels specified doesn't exist.

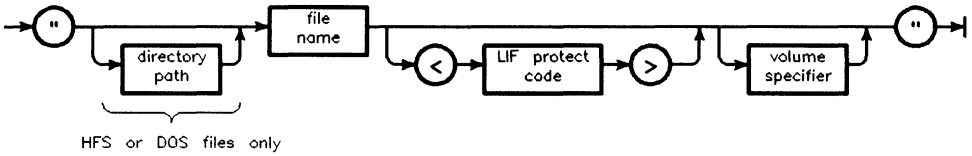
RENAME

RENAME changes a file's or directory's name.

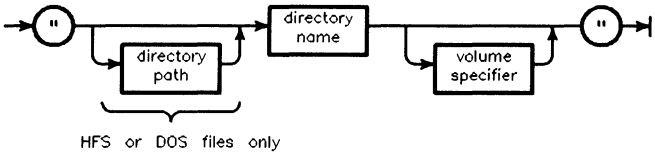
Syntax



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; 8 characters for DOS (short file name); (see Glossary)
LIF protect code	literal; first two non-blank characters are significant	> not allowed
volume specifier	literal	(see MASS STORAGE IS)

Example Statements

```
RENAME "Old_name" TO "New_name"

RENAME Name$ TO Temp$

RENAME "Dir1/file" TO "Dir2/file"
```

Details

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). All files except those opened with the PRINTER IS statement are also closed by clicking on [Stop] in the control pad. 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 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 between directories.

If you are using a version of HP Instrument BASIC that supports wildcards, you can use them in file specifiers with RENAME. You must first enable wildcard recognition using WILDCARDS. Refer to the keyword entry for WILDCARDS for details. Wildcard file specifiers used with RENAME must match one and only one file name.

LIF Protect Codes

A protected file retains its old protect code, which must be included in the old file specifier.

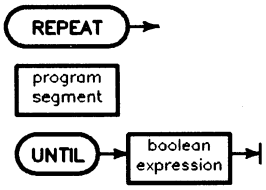
Limited File Moves and Directories

RENAME can be used to move files within and among directories. 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 is purged.

REPEAT ... UNTIL

REPEAT ... UNTIL defines a loop which is repeated until the expression in the UNTIL statement is evaluated as true (non-zero).

Syntax



Item	Description	Range
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).	—
Boolean expression	numeric expression; evaluated as true if non-zero and false if zero	—

Example Statements

```
770  REPEAT
780    CALL Process(Param)
790    Param=Param*Scaling
800  UNTIL Param>Maximum
```

Details

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.

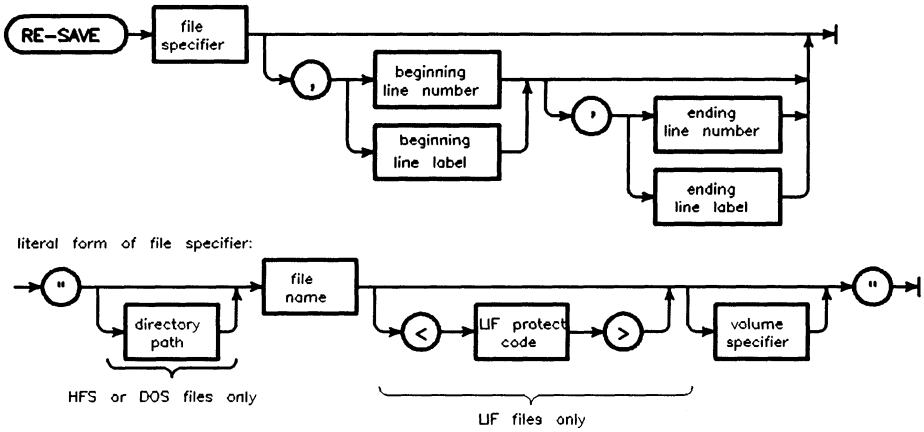
Nesting Constructs Property

REPEAT ... UNTIL constructs may be nested within other constructs provided the inner construct begins and ends before the outer construct can end.

RE-SAVE

RE-SAVE writes the current HP Instrument BASIC program to the specified file in a human-readable ASCII format. If the specified file already exists, the old entry is purged after the new file is written.

Syntax



Item	Description	Range
file specifier	string expression	(see drawing)
beginning line number	integer constant identifying program line; default = first program line	1 through 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 through 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

```
RE-SAVE "NailFile"
RE-SAVE Name$,1,Sort
```

Details

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 after the new file is successfully saved on the mass storage media.

If the file does not already exist, RE-SAVE performs the same action as SAVE (a new file is created).

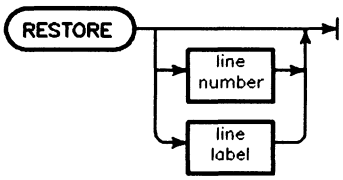
If a specified line label does not exist, error 3 occurs. If a specified line number does not exist, the program lines with numbers inside the range specified are saved. If the ending line number is less than the beginning line number, error 41 occurs.

If you are using a version of HP Instrument BASIC that supports wildcards, you can use them in file specifiers with RE-SAVE. You must first enable wildcard recognition using WILDCARDS. Refer to the keyword entry for WILDCARDS for details. Wildcard file specifiers used with RE-SAVE must match one and only one file name.

RESTORE

RESTORE specifies which DATA statement will be used by the next READ operation.

Syntax



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 through 32 766

Example Statements

```
RESTORE

RESTORE Third_array
```

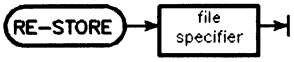
Details

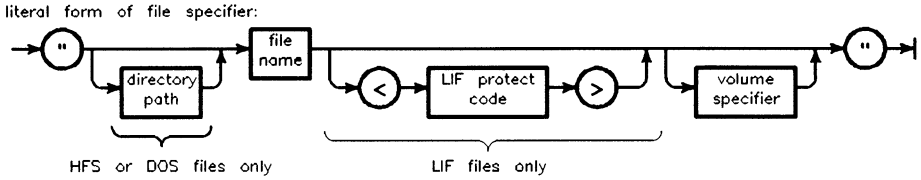
If a line is specified which does not contain a DATA statement, the computer uses the first DATA statement after the specified line. RESTORE can only refer to lines within the current context. An error results if the specified line does not exist.

RE-STORE

RE-STORE writes the current HP Instrument BASIC program to the specified file in a special compact, fast-loading format. If the specified file already exists, the old entry is purged after the new file is successfully stored.

Syntax





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)

Example Statements

```
RE-STORE Filename$&Volume$

RE-STORE "Prog_a"
```

Details

If the specified file already exists, the old file is removed from the directory after the new file is successfully stored in the current mass storage device. If an old file does not exist, a new one is created as if this were the STORE statement.

If you are using a version of HP Instrument BASIC that supports wildcards, you can use them in file specifiers with RE-STORE. You must first enable wildcard recognition using WILDCARDS. Refer to the keyword entry for WILDCARDS for details. Wildcard file specifiers used with RE-STORE must match one and only one file name.

RETURN

RETURN returns program execution to the line following the invoking GOSUB. The keyword RETURN is also used in user-defined functions.

Syntax

Example Statements

To return from a GOSUB subroutine:

```
RETURN
```

To return a value from a user-defined function:

```
RETURN Value
RETURN 13774
RETURN SIN(X)-4*EXP(SIN(PI/Q))
RETURN File$
```

Details

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).

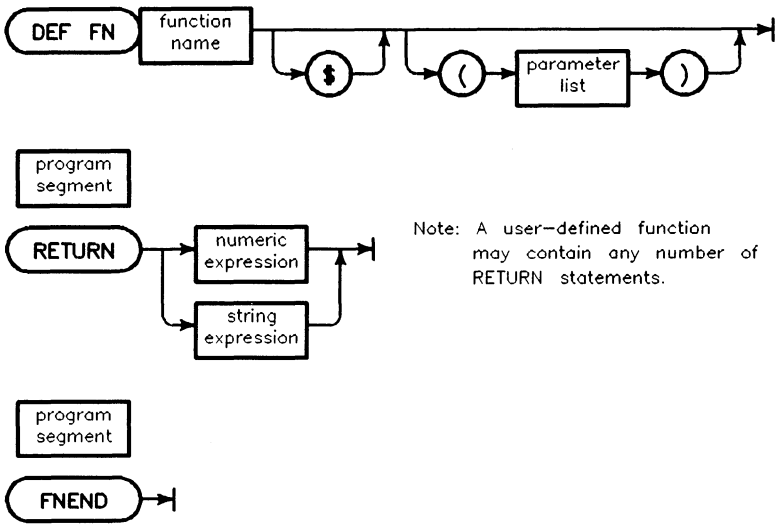
When you exit a multi-line function, the following actions take place:

- local files are closed
- local variables are deallocated
- ON ... statements may be affected

RETURN ...

This statement returns a value from a multi-line function.

Syntax



Item	Description	Range
function name	name of the user-defined function	any valid name
program segment	any number of contriguous program lines not containing the beginning or end of a main program or subprogram	—
numeric expression	numeric expression	range of REAL
string expression	string expression	—

Example Statements

```
IF D THEN RETURN D
RETURN A$&B$
```

Details

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).

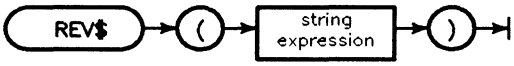
When you exit a multi-line function, the following actions take place:

- local files are closed;
- local variables are deallocated;
- ON ... statements may be affected. See ON ... /OFF ...

REV$

REV$ returns a string formed by reversing the sequence of characters in the argument.

Syntax



Example Statements

```
Reverse$=REV$(Forward$)
Last_blank=LEN(A$)-POS(REV$(A$)," ")
```

Details

The REV$ function is useful when searching for the last occurrence of an item within a string.

RND

RND returns a pseudo-random number greater than 0 and less than 1.

Syntax

Example Statements

```
Percent=RND*100
IF RND<.5 THEN Case1
```

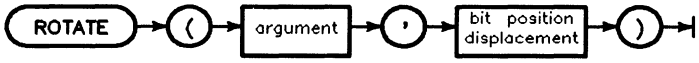
Details

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

ROTATE returns an integer which equals the value obtained by shifting the 16-bit binary representation of the argument the number of bit positions specified. The shift is performed with wraparound.

Syntax



Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 through +32 767
bit position displacement	numeric expression, rounded to an integer	−15 through +15

Example Statements

```
New_word=ROTATE(Old_word,2)
```

```
Q=ROTATE(Q,Places)
```

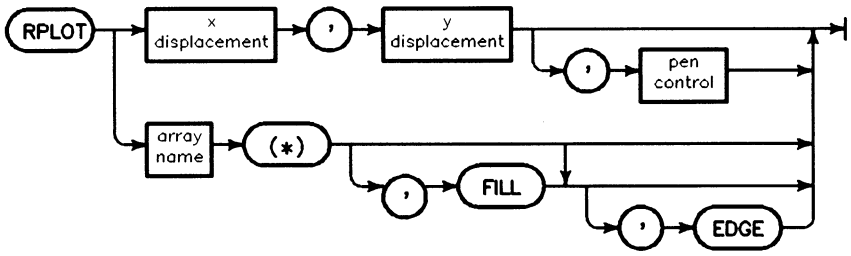
Details

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.

RPLOT

RPLOT moves the pen from the current pen position to the specified relative X and Y position.

Syntax



Item	Description	Range
x displacement	numeric expression in current units	—
y displacement	numeric expression in current units	—
pen control	numeric expression, rounded to an integer; default = 1	−32 768 through +32 767
array name	name of two-dimensional, two-column or three-column numeric array. Requires GRAPHX	any valid name

Example Statements

```
RPLOT 10,12

RPLOT Rel_x,Rel_y,Pen_control

RPLOT Array(*)

RPLOT Shape(*),FILL,EDGE
```

Details

RPLOT moves the pen to the specified X and Y coordinates relative to the local coordinate origin. Both moves and draws may be generated, depending on the pen control parameter. Lines are drawn using the current pen color set by PEN and the current line type set by LINE TYPE.

The local coordinate origin is *not* changed by the RPLOT statement.

Lines drawn by RPLOT are clipped at the current clipping boundary. If none of the lines are inside the current clip limits, the pen is not moved, but the logical pen position is updated.

Graphics Transformations

The output of RPLOT is affected by only these graphics transformations:

- scaling specified by WINDOW
- scaling specified by SHOW
- rotations specified by PIVOT
- rotations specified by PDIR

Non-Array Parameters

The specified X and Y displacements information is interpreted according to the current unit-of-measure. Lines are drawn using the current pen color and line type.

If none of the line is inside the current clip limits, the pen is not moved, but the logical pen position is updated.

The optional pen control parameter specifies the following plotting actions; the default value is +1 (down after move).

Pen Control Parameter

Pen Control	Resultant Action
−Even	Pen up before move
−Odd	Pen down before move
+Even	Pen up after move
+Odd	Pen down after move

Array Parameters

When using the RPLOT statement with an array, either a two-column or a three-column array may be used. If a two-column array is used, the third parameter is assumed to be +1; pen down after move.

FILL and EDGE

When FILL or EDGE is specified, each sequence of two or more lines forms a polygon. The polygon begins at the first point on the sequence, includes each successive point, and the final point is connected or closed back to the first point. A polygon is closed when the end of the array is reached, or when the value in the third column is an even number less than three, or in the range 5 to 8 or 10 to 15.

If FILL and/or EDGE are specified on the RPLOT statement itself, it causes the polygons defined within it to be filled with the current fill color and/or edged with the current pen color. If polygon mode is entered from within the array, and the FILL/EDGE directive for that series of polygons differs from the FILL/EDGE directive on the RPLOT statement itself, the directive in the array replaces the directive on the statement. In other words, if a "start polygon mode" operation selector (a 6, 10, or 11) is encountered, any current FILL/EDGE

directive (whether specified by a keyword or an operation selector) is replaced by the new FILL/EDGE directive.

If FILL and EDGE are both declared on the RPLOT statement, FILL occurs first. If neither one is specified, simple line drawing mode is assumed; that is, polygon closure does not take place.

When using an RPLOT statement with an array, the following table of **operation selectors** applies. An operation selector is the value in the third column of a row of the array to be plotted. The array must be a two-dimensional, two-column or three-column array. If the third column exists, it will contain operation selectors which instruct the computer to carry out certain operations. Polygons may be defined, edged (using the current pen), filled (using the current fill color), pen and line type may be selected, and so forth.

Column 1	Column 2	Operation Selector	Meaning
X	Y	−2	Pen up before moving
X	Y	−1	Pen down before moving
X	Y	0	Pen up after moving (Same as +2)
X	Y	1	Pen down after moving
X	Y	2	Pen up after moving
pen number	ignored	3	Select pen
line type	repeat value	4	Select line type
color	ignored	5	Color value
ignored	ignored	6	Start polygon mode with FILL
ignored	ignored	7	End polygon mode
ignored	ignored	8	End of data for array
ignored	ignored	9	NOP (no operation)
ignored	ignored	10	Start polygon mode with EDGE
ignored	ignored	11	Start polygon mode with FILL and EDGE
ignored	ignored	12	Draw a FRAME
pen number	ignored	13	Area pen value
red value	green value	14	Color
blue value	ignored	15	Value
ignored	ignored	>15	Ignored

Moving and Drawing

If the operation selector is less than or equal to two, it is interpreted in exactly the same manner as the third parameter in a non-array RPLOT statement. Even is up, odd is down, positive is after pen motion, negative is before pen motion. Zero is considered positive.

Selecting Pens

An operation selector of 3 selects a pen. The value in column one is the pen number desired. The value in column two is ignored.

Selecting Line Types

An operation selector of 4 selects a line type. The line type (column one) selects the pattern, and the repeat value (column two) is the length in GDUs that the line extends before a single occurrence of the pattern is finished and it starts over. On the CRT, the repeat value is evaluated and rounded *down* to the next multiple of 5, with 5 as the minimum.

Selecting a Fill Color

Operation selector 13 selects a pen from the color map with which to do area fills. This works identically to the AREA PEN statement. Column one contains the pen number.

Defining a Fill Color

Operation selector 14 is used in conjunction with operation selector 15. Red and green are specified in columns one and two, respectively, and column three has the value 14. Following this row in the array (not necessarily immediately), is a row whose operation selector in column three has the value of 15. The first column in that row contains the blue value. These numbers range from 0 to 32 767, where 0 is no color and 32 767 is full intensity. Operation selectors 14 and 15 together comprise the equivalent of an AREA INTENSITY statement.

Operation selector 15 actually puts the area intensity into effect, but only if an operation selector 14 has already been received.

Operation selector 5 is another way to select a fill color. The color selection is through a Red-Green-Blue (RGB) color model. The first column is encoded in the following manner. There are three groups of five bits right-justified in the word; that is, the most significant bit in the word is ignored. Each group of five bits contains a number which determines the intensity of the corresponding color component, which ranges from zero to sixteen. The value in each field will be sixteen minus the intensity of the color component. For example, if the value in the first column of the array is zero, all three five-bit values would thus be zero. Sixteen minus zero in all three cases would turn on all three color components to full intensity, and the resultant color would be a bright white.

Assuming you have the desired intensities (which range from 0 thru 1) for red, green, and blue in the variables R, G, and B, respectively, the value for the first column in the array could be defined thus:

```
Array(Row,1)=SHIFT(16*(1-B),-10)+SHIFT(16*(1-G),-5)+16*(1-R)
```

If there is a pen color in the color map similar to that which you request here, that non-dithered color will be used. If there is not a similar color, you will get a dithered pattern.

If you are using a gray scale display, Operation selector 5 uses the five bit values of the RGB color specified to calculate luminosity. The resulting gray luminosity is then used as the area fill.

Polygons

A six, ten, or eleven in the third column of the array begins a "polygon mode". If the operation selector is 6, the polygon will be filled with the current fill color. If the operation selector is 10, the polygon will be edged with the current pen number and line type. If the operation selector is 11, the polygon will be both filled and edged. Many individual polygons can be filled without terminating the mode with an operation selector 7. This can be done by specifying several series of draws separated by moves. The first and second columns are ignored and should not contain the X and Y values of the first point of a polygon.

Operation selector 7 in the third column of a plotted array terminates definition of a polygon to be edged and/or filled and also terminates the polygon mode (entered by operation selectors 6, 10, or 11). The values in the first and second columns are ignored, and the X and Y values of the last data point should not be in them. Edging and/or filling of the most recent polygon will begin immediately upon encountering this operation selector.

Doing a FRAME

Operation selector 12 does a FRAME around the current soft-clip limits. Soft clip limits cannot be changed from within the RPLOT statement, so one probably would not have more than one operation selector 12 in an array to RPLOT, since the last FRAME will overwrite all the previous ones.

Premature Termination

Operation selector 8 causes the RPLOT statement to be terminated. The RPLOT statement will successfully terminate if the actual end of the array has been reached, so the use of operation selector 8 is optional.

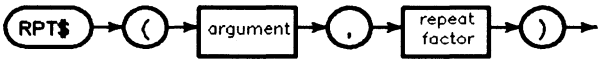
Ignoring Selected Rows in the Array

Operation selector 9 causes the row of the array it is in to be ignored. Any operation selector greater than fifteen is also ignored, but operation selector 9 is retained for compatibility reasons. Operation selectors less than -2 are not ignored. If the value in the third column is less than zero, only evenness/oddness is considered.

RPT$

RPT$ returns a string formed by repeating the argument a specified number of times.

Syntax



Item	Description	Range
argument	string expression	—
repeat factor	numeric expression, rounded to an integer	0 through 32 767

Example Statements

```
PRINT RPT$("*",80)
Center$=RPT$(" ",(Right-Left-Length)/2)
```

Details

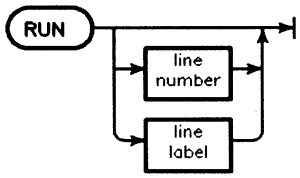
The value of the numeric expression is rounded to an integer. If the numeric expression evaluates to a zero, a null string is returned.

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

RUN starts program execution at the specified line. If no parameter is specified, the program starts at the beginning.

Syntax



Item	Description	Range
line number	integer constant identifying a program line; default = first program line	1 through 32 766
line label	name of a program line	any valid name

Example Commands

```
RUN
RUN 10
RUN Part2
```

Details

Clicking on RUN in the control pad is the same as executing RUN with no label or line number.

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. This does not include variables used with ALLOCATE, which is done at run-time. 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 HP Instrument BASIC detects an error during prerun, prerun halts and an error message is displayed.

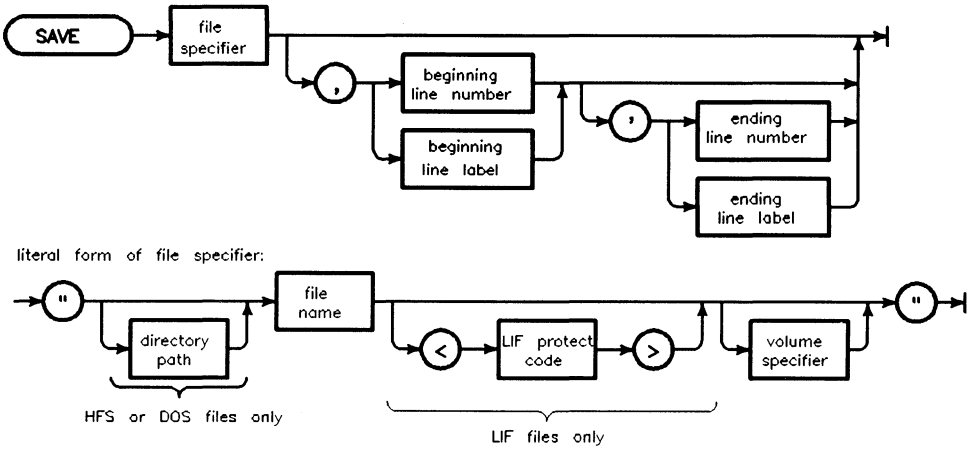
After successful completion of prerun initialization, program execution begins with either the lowest numbered program line or the line specified in the RUN command. If the line number specified does not exist in the main program, execution begins at the next higher-numbered

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.

SAVE

SAVE writes all or part of the program currently in memory to a text file in a human-readable format.

Syntax



Item	Description	Range
file specifier	string expression	(see drawing)
beginning line number	integer constant identifying a program line; default = first program line	1 through 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 through 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 File$,First_line,Last_line

SAVE "TEMP",1,Sort
```

Details

SAVE writes all or part of the program in memory to the specified text file. The format of a SAVEd file is different from the PROG format used by STORE. The PROG format uses less disk space and loads more quickly into HP Instrument BASIC. The ASCII format is more portable (it can be read by other versions of HP BASIC) and it is human-readable.

SAVE creates a new file; to replace an existing file use RE-SAVE. Attempting to SAVE a program to a file name that already exists causes error 54.

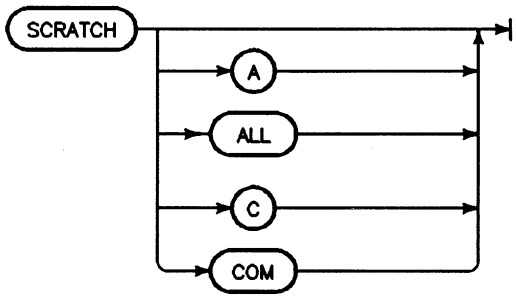
If a specified line label does not exist, error 3 results. If a specified line number does not exist, the program lines with numbers inside the range specified are saved. If the ending line number is less than the beginning line number, error 41 occurs. If no program lines are in the specified range, error 46 occurs.

Lines longer than 256 characters will not be saved correctly. When a GET is performed on a program with such a line, an error will occur. However, a program containing lines exceeding this length can be successfully STOREd and LOADed.

SCRATCH

SCRATCH erases all or selected portions of memory.

Syntax



Details

SCRATCH clears the HP Instrument BASIC program and all variables not in COM.

SCRATCH A clears the HP Instrument BASIC program memory and all variables (including those in COM). Most internal parameters in the computer are reset by this command. The clock is not reset and the recall buffer is not cleared.

SCRATCH C clears all variables, including those in COM. The program is left intact.

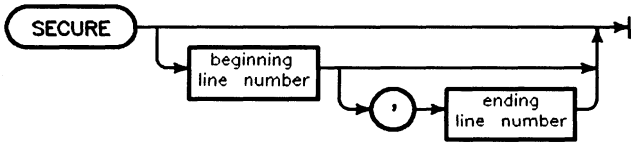
Example Commands

```
SCRATCH
SCRATCH A
SCRATCH ALL
SCRATCH COM
```

SECURE

SECURE protects programs lines so they cannot be listed.

Syntax



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 Commands

```
SECURE
SECURE Check_password
SECURE Routine1,Routine2
```

Details

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.

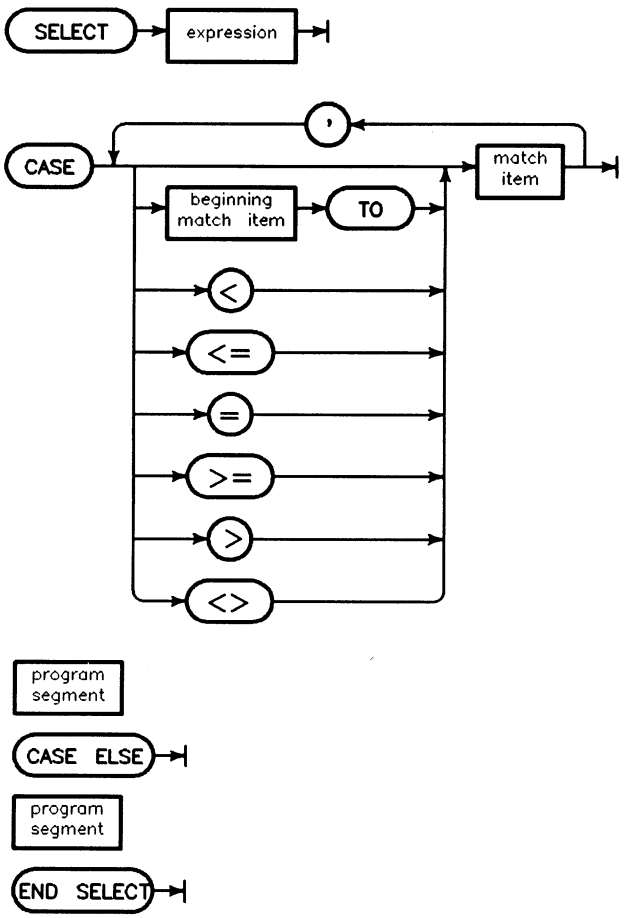
Caution

Do not SECURE the only copy of your program. Make a copy of your program, SECURE the *copy*, and save the original "source code" version of your program in a safe place. There is no way to "unsecure" a program once you have protected it with the SECURE statement. This prevents unauthorized users from listing your program.

SELECT ... CASE

SELECT ... CASE provides conditional execution of one program segment chosen from several.

Syntax



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 Statements

```
600  SELECT String$
610    CASE "0" TO "9"
620      GOSUB Digits
630    CASE ";"
640      GOSUB Delimiter
650    CASE <CHR$(32),>CHR$(126)
660      GOSUB Control_chr
670    CASE ELSE
680      GOSUB Ignore
690  END SELECT
```

Details

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.

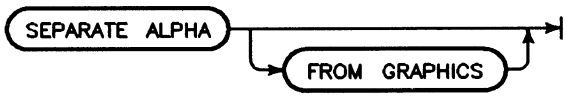
Nesting Constructs Properly

SELECT ... END SELECT constructs may be nested, provided inner construct begins and ends before the outer construct can end.

SEPARATE ALPHA

This statement may not work on all instruments. It is included for compatability with RMB-UX. It has no affect except in RMB Workstation. Like RMB-UX, this will generate a runtime error of 713. Request is not supported by dev.

Syntax



Details

Color map entries below the lowest alpha pen value have their default colors set by PLOTTER IS CRT,"INTERNAL". Using a value in this range as an alpha pen will produce transparent text (i.e., is equivalent to using pen 0). Setting up the color or gray map as given in the table causes the alpha text to be dominant over graphics images. If the COLOR MAP option is used with PLOTTER IS, the SET PEN statement can still be used to set all color or gray map entries, not just those dedicated to graphics pens.

Here is a BASIC program that performs similar configuration of the planes of a 4-plane display:

```
100 PLOTTER IS CRT, "INTERNAL";COLOR MAP!Series 300 display
110 FOR I=8 TO 15
120 SET PEN I INTENSITY 0,1,0 ! Set alpha colors (green).
130 NEXT I
140 ALPHA PEN 0 ! Set alpha pen to black (temp).
150 ALPHA MASK 15 ! Enable all planes (temp).
160 CLEAR SCREEN
170 ALPHA MASK 8 ! Enable plane 4 for alpha.
180 ALPHA PEN 8 ! Set alpha pen.
190 INTEGER Gm(0) ! Declare array for GESCAPE.
200 Gm(0)=7 ! Set bits 2,1,0, which select
210 GESCAPE CRT,7,Gm(*) ! graphics planes 3,2,1.
220 ALPHA ON ! Display alpha plane.
230 GRAPHICS ON ! Display graphics planes.
240 PLOTTER IS CRT,"INTERNAL" ! Return to non-color-map
250 END ! mode.
```

Note that when using this operation with AREA COLOR and AREA INTENSITY, there may be unexpected results. The algorithm that AREA COLOR and AREA INTENSITY use to select graphics pens does not account for the graphics write-enable or display-enable masks. If the pens selected by these statements have bits outside of the write-enable mask, then the planes corresponding to these bits will not be affected. The result is that the area fill colors will not be what is expected.

Example Statements

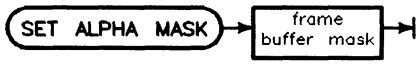
```
SEPARATE ALPHA
```

SET ALPHA MASK

SET ALPHA MASK is included for compatibility with RMB-UX. It has no effect except in RMB Workstation.

The behavior of this statement will be instrument specific. Refer to the instrument specific manual for more information.

Syntax



Item	Description/Default	Range Restrictions
frame buffer mask	numeric expression, rounded to an integer	1 through 2^n − 1, where n equals the number of display planes

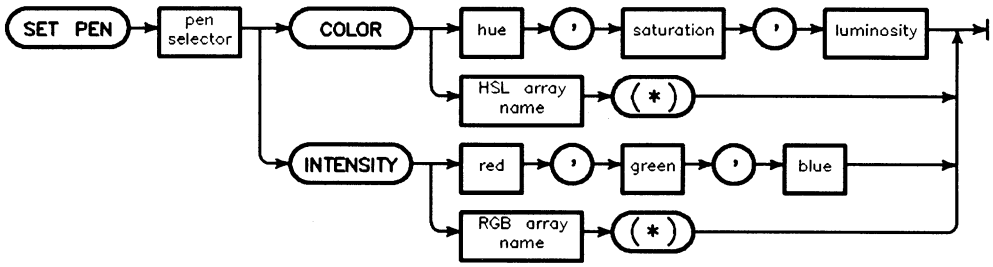
Example Statements

```
SET ALPHA MASK Frame_mask
SET ALPHA MASK 3
SET ALPHA MASK IVAL("1100",2)
IF Total_frames = 5 THEN SET ALPHA MASK 8
```

SET PEN

SET PEN assigns a color to one or more graphics pens. SET PEN has an effect only when color-mapped mode is active (PLOTTER IS ... COLOR MAP).

Syntax



Item	Description	Range
pen selector	numeric expression, rounded to an integer	0 through 32 767
hue	numeric expression	0 through 1
saturation	numeric expression	0 through 1
luminosity	numeric expression	0 through 1
HSL array name	name of a two-dimensional, three-column REAL array	any valid name
red	numeric expression	0 through 1
green	numeric expression	0 through 1
blue	numeric expression	0 through 1
RGB array name	name of a two-dimensional, three-column REAL array	any valid name

Note

The colors defined with SET PEN become active *only* after a PLOTTER IS ... COLOR MAP statement, such as:

```
PLOTTER IS CRT, "INTERNAL"; COLOR MAP
```

Example Statements

```
SET PEN P_num COLOR Hue,Saturate,Luminous

SET PEN Selector INTENSITY Red,Blue,Green

SET PEN Start_pen COLOR Hsl_array(*)

SET PEN 2 INTENSITY 4/15,1/15,9/15
```

Details

SET PEN will operate correctly *only* if the computer's video interface supports at least 256 colors. If the video interface supports fewer than 256 colors, SET PEN executes without error and without effect. Note that a Super VGA or better interface is required to obtain 256 colors.

SET PEN defines the color or gray value for one or more entries in the color map. In simpler terms, SET PEN allows you to define a color and assign it to one of the HP Instrument BASIC PEN numbers. SET PEN has an effect only in the color-mapped mode. The color-map mode is controlled by PLOTTER IS ... COLOR MAP. In non-color-mapped mode, SET PEN executes without error and without effect.

For both SET PEN COLOR and SET PEN INTENSITY, the pen selector specifies the first color or gray map entry to be defined. If individual RGB or HSL values are given, that entry in the color or gray map is the only one defined. If an array is specified, the color or gray map is redefined, starting at the specified pen, and continuing until either the highest-numbered entry in the map is redefined or the source array is exhausted.

Specifying color or gray with the SET PEN and AREA PEN statements (resulting in non-dithered color) results in a much more accurate representation of the desired color than specifying the color with an AREA COLOR or AREA INTENSITY statement.

SET PEN COLOR

The hue value specifies the color. The hue ranges from zero to one, in a circular manner, with a value of zero resulting in the same hue as a value of one. The hue, as it goes from zero to one, proceeds through red, orange, yellow, green, cyan, blue, magenta, and back to red.

The saturation value, classically defined, is the inverse of the amount of white added to a hue. What this means is that saturation specifies the amount of hue to be mixed with white. As saturation goes from zero to one, there is 0% to 100% of pure hue added to white. Thus, a saturation of zero results in a gray, dependent only upon the luminosity; hue makes no difference.

The luminosity value specifies the brightness per unit area of the color. A luminosity of zero results in black, regardless of hue or saturation; if there is no color, it makes no difference which color it is that is not there.

If you are using a gray scale display, hue and saturation are not used, and the brightness per unit area of gray is specified by the luminosity value. A luminosity of zero results in black.

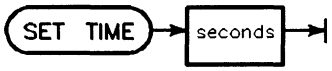
SET PEN INTENSITY

For a color display, the red, green, and blue values specify the intensities of the red, green, and blue displayed on the screen.

SET TIME

This statement resets the time-of-day given by the real-time clock.

Syntax



Item	Description	Range
seconds	numeric expression, rounded to the nearest hundredth	0 through 86 399.99

Example Statements

```
SET TIME 0
SET TIME Hours*3600+Minutes*60
SET TIME TIME("8:37:30")
```

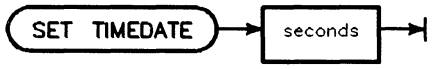
Details

SET TIME changes only the time within the current day, not the date. The new clock setting is equivalent to (TIMEDATE DIV 86 400)×86 400 plus the specified setting.

SET TIMEDATE

This statement resets the absolute seconds (time and day) given by the real-time clock.

Syntax



Item	Description	Range
seconds	numeric expression, rounded to the nearest hundredth	2.086 629 12 E+11 through 2.143 252 223 999 9 E+11

Example Statements

```
SET TIMEDATE TIMEDATE+3600
SET TIMEDATE Strange_number
SET TIMEDATE DATE("1 Jan 1989") + TIME("13:57:20")
```

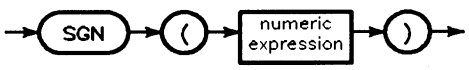
Details

The volatile clock is set to 2.086 629 12 E+11 (midnight March 1, 1900) at power-on (BASIC Workstation semantics). 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.

SGN

SGN returns 1 if the argument is positive, 0 if it equals zero, and -1 if it is negative.

Syntax



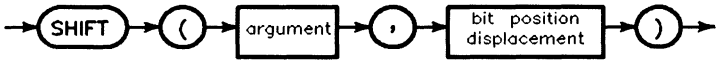
Example Statements

```
Root=SGN(X)*SQR(ABS(X))
Z=2*PI*SGN(Y)
```

SHIFT

SHIFT returns an integer which equals the value obtained by shifting the 16-bit binary representation of the argument the number of bit positions specified, without wraparound.

Syntax



Item	Description	Range
argument	numeric expression, rounded to an integer	−32 768 through +32 767
bit position displacement	numeric expression, rounded to an integer	−15 through +15

Example Statements

```
New_word=SHIFT(Old_word,-2)
```

```
Mask=SHIFT(1,Position)
```

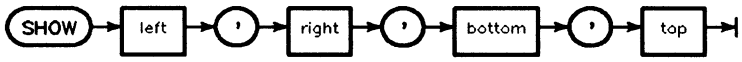
Details

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.

SHOW

SHOW defines an isotropic current unit-of-measure for graphics operations.

Syntax



Item	Description	Range
left	numeric expression	—
right	numeric expression	≠ left
bottom	numeric expression	—
top	numeric expression	≠ bottom

Example Statements

```
SHOW -5,5,0,100
SHOW Left,Right,Bottom,Top
```

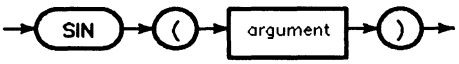
Details

SHOW defines the values which must be displayed within the hard clip boundaries, or the boundaries defined by the VIEWPORT statement. SHOW creates isotropic units (units the same in X and Y). The direction of an axis may be reversed by specifying the left greater than the right or the bottom greater than the top.

SIN

SIN returns the sine of the angle represented by the argument.

Syntax



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

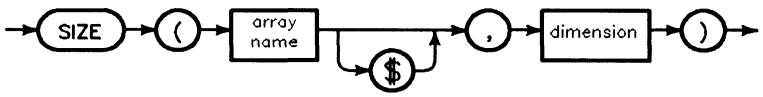
Example Statements

```
Sine=SIN(Angle)
PRINT "Sine of";Theta;"=";SIN(Theta)
```

SIZE

SIZE returns the number of elements in a dimension of an array.

Syntax



Item	Description	Range
array name	name of an array	any valid name
dimension	numeric expression, rounded to an integer	1 through 6; ≤ the RANK of the array

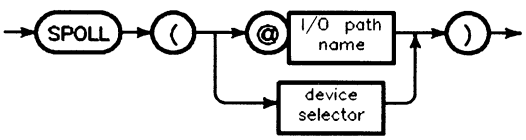
Example Statements

```
Total_words=SIZE(Words$,1)
Upperbound(2)=BASE(A1,2)+SIZE(A1,2)-1
```

SPOLL

SPOLL returns an integer containing the serial poll response from the addressed device.

Syntax



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
```

Details

A SPOLL may be executed under the following conditions:

- the computer must be the active controller
- multiple listeners are not allowed
- one secondary address may be specified to get status from an extended talker

Refer to the documentation provided with the polled device for information concerning the device's status byte.

Summary of SPOLL Bus Actions

Interface Select Code Only	Primary Address Specified
Error	ATN UNL MLA TAD SPE $\overline{\text{ATN}}$ Read data ATN SPD UNT

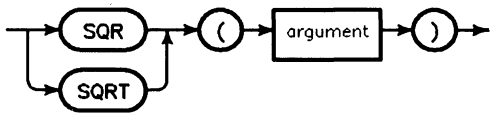
SQR

See SQRT.

SQRT

SQRT returns the square root of the argument. The keywords SQR and SQRT behave identically.

Syntax



Summary of Bus Actions

Item	Description/Default	Range Restrictions
argument	numeric expression	any valid INTEGER or REAL value for INTEGER and REAL expressions

Example Statements

```
Amps=SQRT(Watts/Ohms)
PRINT "Square root of";X;"=";SQR(X)
```

STATUS

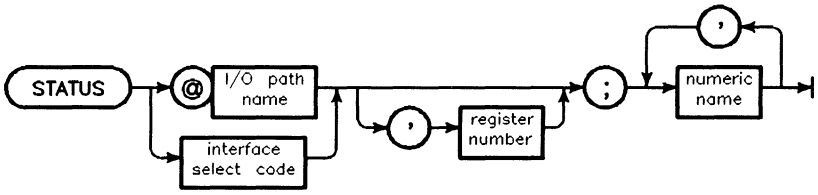
The behavior of this statement will be instrument specific. Refer to the instrument specific manual for more information.

Appendix C contains more information on registers for I/O path names, interfaces, and pseudo select code 32.

STATUS can perform a variety of functions:

- For I/O paths assigned to files, STATUS returns the values of registers that describe the size, format, and status of the file.
- For I/O paths assigned to hardware interfaces, such as GPIB and serial interfaces, STATUS returns the values of registers that describe the configuration and status of the interface.

Syntax for Files



Item	Description	Range
I/O path name	name assigned to a device, devices, mass storage file, buffer, or pipe	any valid name (see ASSIGN)
interface select code	numeric expression, rounded to an integer	1 through 40
register number	numeric expression, rounded to an integer; default = 0	interface dependent
numeric name	name of a numeric variable	any valid name

Example Statements

```
STATUS Interface,Reg;Val1 ! Read status of a hardware interface.

STATUS @File,5;Record ! Read status of a file.
```

Details

Using STATUS with Files and Hardware Interfaces

The value of the beginning register number is copied into the first variable, the next register value into the second variable, and so on. The information is read until the variables in the list are exhausted; there is no wrap-around to the first register. An attempt to read a nonexistent register generates an error.

The register meanings depend on the specified interface or on the resource to which the I/O path name is currently assigned. Register 0 of I/O path names can be interrogated with STATUS even if the I/O path name is currently invalid (i.e., unassigned to a resource). Note that the status registers of an I/O path are different from the status registers of an interface.

STOP

STOP terminates execution of the program.

Syntax

Example Statements

```
STOP
IF Done THEN STOP
```

Details

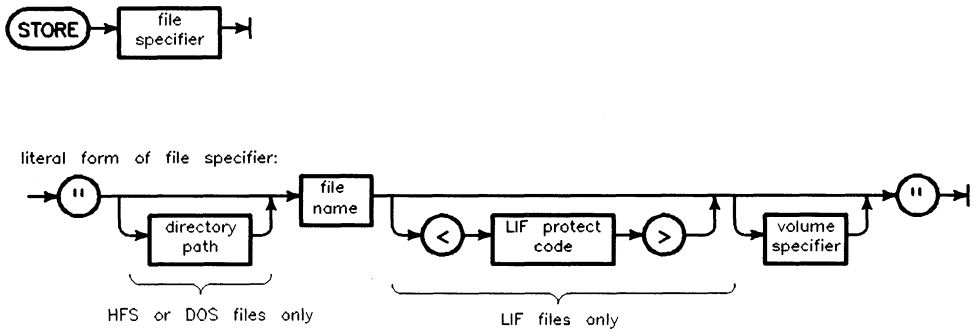
Once a program is stopped, it cannot be resumed by CONTINUE. RUN must be executed to restart the program. PAUSE should be used if you intend to continue execution of the program.

A program can have multiple STOP statements. Encountering an END statement or clicking on Stop in the control panel has the same effect as executing STOP. After a STOP, variables that existed in the main context are available from the keyboard.

STORE

STORE writes the program currently in memory to a PROG file in a special binary form used only by HP Instrument BASIC.

Syntax



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)

Example Statements

```
STORE Filename$
```

Details

STORE writes the program in memory to the specified IBPRG file. This format is different from the ASCII format used by SAVE. The IBPRG format loads more quickly into HP Instrument BASIC. The ASCII format is more portable (it can be read by other versions of HP BASIC) and it is human-readable.

In all STORE statements, an error occurs if the storage media cannot be found, the media or directory is full, or the file specified already exists. To update a file which already exists, see RE-STORE.

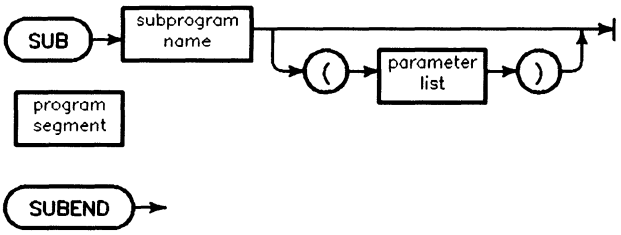
STORE

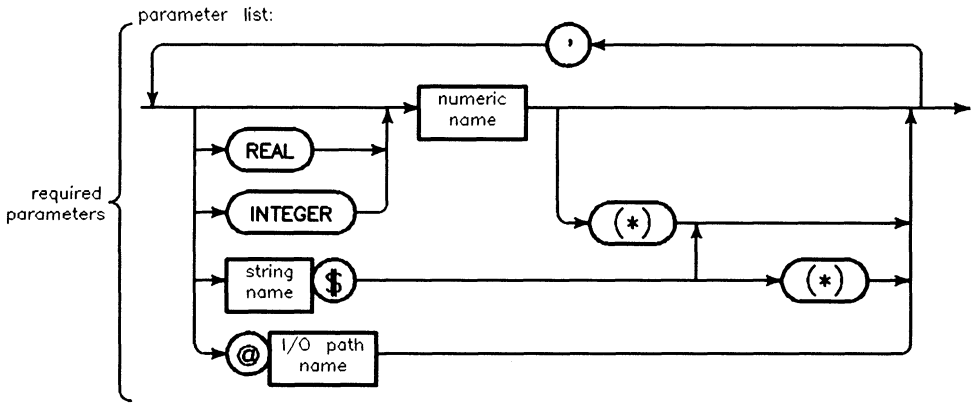
The STORE statement creates a IBPRG file and stores an internal form of the program into that file.

SUB

SUB is the first statement in a SUB subprogram and specifies the subprogram's formal parameters. SUB subprograms must follow the main program's END statement and must be terminated by a SUBEND statement.

Syntax





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 Process

SUB Transform(@Printer,INTEGER Array(*),Text$)
```

Details

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 must 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 can contain local variables. A uniquely labeled COM must be used if the local variables are to preserve their values between invocations of the subprogram.

Use SUBEXIT to exit the subprogram at some point other than the SUBEND. Multiple SUBEXITs are allowed; SUBEND may only occur once in a subprogram. ERROR SUBEXIT can be used in the same way as SUBEXIT.

SUBEND

SUBEND is the last statement of a SUB subprogram. SUBEND marks the end of the subprogram and returns control to the calling context.

Example Statement

```
SUBEND
```

SUBEXIT

SUBEXIT returns program control from a subprogram at some point other than the SUBEND statement.

Syntax

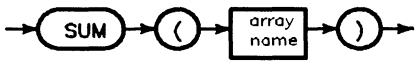
Example Statements

```
SUBEXIT

IF Done THEN SUBEXIT
```

SUM

SUM returns the sum of all the elements in a numeric array.

Syntax



Item	Description	Range
array name	name of a numeric array	any valid name

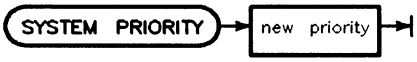
Example Statements

```
Total=SUM(Array)
PRINT SUM(Squares)
```

SYSTEM PRIORITY

SYSTEM PRIORITY sets the system priority to a specified level.

Syntax



Item	Description	Range
new priority	numeric expression, rounded to an integer	0 through 15

Example Statements

```
SYSTEM PRIORITY Level

SYSTEM PRIORITY 15
```

Details

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.

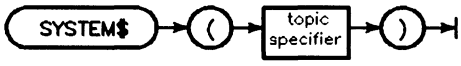
If SYSTEM PRIORITY is used to change the minimum system priority in a subprogram context, the former value is restored when the context is exited.

Error 427 results if SYSTEM PRIORITY is executed in a service routine for an ON ERROR GOSUB or ON ERROR CALL statement.

SYSTEM$

SYSTEM$ returns a string containing system status and configuration information.

Syntax



Item	Description	Range
topic specifier	string expression	see the following table

Example Statements

```
System_prior=VAL(SYSTEM$("SYSTEM PRIORITY"))
PRINT SYSTEM$("VERSION:INSTR") ! The version no. of HP IBASIC .
```

Details

The topic specifier is used to specify what system configuration information SYSTEM$ will return.

The following table lists the valid topic specifiers and the information returned for each one.

Topic Specifier	Information Returned
MASS STORAGE IS, MSI	The mass storage unit specifier of the current MASS STORAGE IS device, as it appears in a CAT heading.
PRINTER IS	A string containing numerals which specify the device selector of the current PRINTER IS device or the path name of the current PRINTER IS file.
SYSTEM ID	Returns instrument name.
SYSTEM PRIORITY	A string containing numerals which specify the current system priority.
VERSION:INSTR	Returns the current version of IBASIC, for example, A.00.00

TAB

See PRINT and DISP.

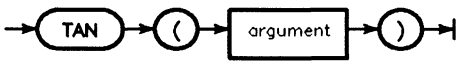
TABXY

See PRINT.

TAN

TAN returns the tangent of the specified angle.

Syntax



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

Example Statements

```
Tangent=TAN(Angle)
PRINT "Tangent of";Z;"=";TAN(Z)
```

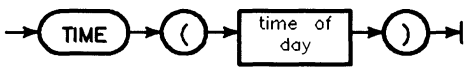
Details

Error 31 is reported for INTEGER and REAL arguments when trying to compute the TAN of an odd multiple of 90 degrees.

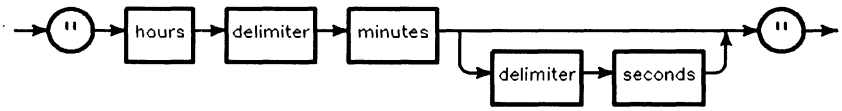
TIME

TIME converts a formatted time-of-day string into a numeric value of seconds past midnight.

Syntax



literal form of time of day



Item	Description	Range
time of day	string expression representing the time in 24-hour format	(see drawing)
hours	literal	0 through 23
minutes	literal	0 through 59
seconds	literal; default = 0	0 through 59.99
delimiter	literal; single character	(see text)

Example Statements

```
Seconds=TIME(T$)
```

Details

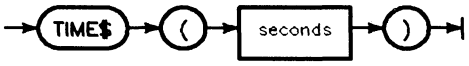
TIME returns a REAL whole number, in the range 0 through 86 399, equivalent to the number of seconds past midnight.

While any number of non-numeric characters may be used as a delimiter, a single colon is recommended. Leading blanks and non-numeric characters are ignored.

TIME$

TIME$ converts the number of seconds past midnight into a string representing the formatted time of day (HH:MM:SS).

Syntax



Item	Description	Range
seconds	numeric expression, truncated to the nearest second; seconds past midnight	0 through 86 399

Example Statements

```
PRINT "It is ";TIME$(TIMEDATE)

IF VAL(TIME$(T1))>17 THEN Overtime
```

Details

TIME$ takes time (in seconds) and returns the time of day in the form HH:MM:SS, where HH represents hours, MM represents minutes, and SS represents seconds. A modulo 86 400 is performed on the parameter before it is formatted as a time of day.

TIMEDATE

TIMEDATE returns the current value of the real-time clock.

Syntax

Example Statements

```
DISP TIME$(TIMEDATE),DATE$(TIMEDATE)

Elapsed=TIMEDATE-T1

DISP "Seconds since midnight = ";TIMEDATE MOD 86400
```

Details

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 time value accumulates from that setting unless it is reset.

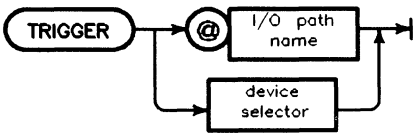
The resolution of the TIMEDATE function is .01 seconds. If the clock is properly set, this expression equals the number of seconds since midnight:

```
TIMEDATE MOD 86400
```

TRIGGER

TRIGGER sends a trigger message to a selected device, or to all devices addressed to listen, on the GPIB .

Syntax



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 Device

TRIGGER @Hpib
```

Details

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.

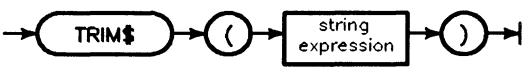
Summary of Bus Actions

	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$

TRIM$ returns a string formed by stripping all leading and trailing blanks from its argument.

Syntax



Example Statements

```
Left$=TRIM$("     center     ")
Clean$=TRIM$(User_input$)
```

Details

Only CHR$(32) (the space character) is trimmed. Only leading and trailing ASCII spaces are removed; embedded spaces are not affected.

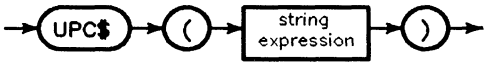
UNTIL

See REPEAT ... UNTIL.

UPC$

UPC$ returns a string formed by replacing any lowercase characters with the corresponding uppercase characters.

Syntax



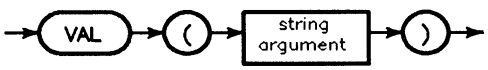
Example Statements

```
Capital$=UPC$(Mixed$)
IF UPC$(Yes$)="Y" THEN True_test
```

VAL

VAL converts a string expression into a numeric value.

Syntax



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
```

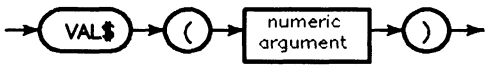
Details

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$

VAL$ converts a numeric expression to a string.

Syntax



Item	Description	Range
numeric argument	numeric expression	—

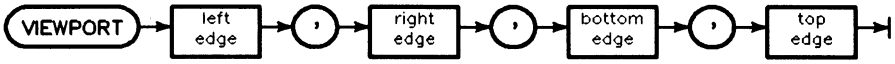
Example Statements

```
PRINT Esc$;VAL$(Cursor-1)
Special$=Text$&VAL$(Number)
```

VIEWPORT

VIEWPORT defines an area (in GDUs) onto which WINDOW and SHOW statements are mapped. It also sets the soft clip limits to the boundaries it defines.

Syntax



Item	Description	Range
left edge	numeric expression	—
right edge	numeric expression	>left edge
bottom edge	numeric expression	—
top edge	numeric expression	>bottom edge

Example Statements

```
VIEWPORT 0,35,50,80
VIEWPORT Left,Right,Bottom,Top
```

Details

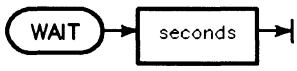
The parameters for VIEWPORT are in Graphic Display Units (GDUs). Graphic Display Units are 1/100 of the shorter axis of a plotting device. GDUs are isotropic (the same length in X and Y). The soft clip limits are set to the area specified, and the units defined by the last WINDOW or SHOW are mapped into the area. Since the dimensions of the graph window can be changed, the length of the longer side must be determined using RATIO. If RATIO is greater than one, the Y axis is 100 GDUs long, and the length of the X axis is (100*RATIO). If the ratio is less than one, then the length of the X axis is 100 GDUs and the length of the Y axis is (100*RATIO).

A value of less than zero for the left edge or bottom is treated as zero. A value greater than the hard clip limit is treated as the hard clip limit for the right edge and the top. The left edge must be less than the right edge, and the bottom must be less than the top, or error 704 results.

WAIT

WAIT causes the computer to wait the number of seconds specified before executing the next statement.

Syntax



Item	Description	Range
seconds	numeric expression, rounded to the nearest thousandth	less than 2 147 483.648

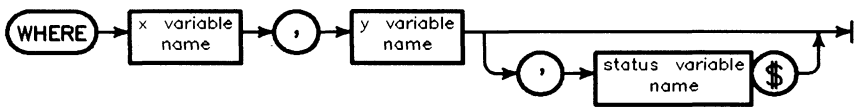
Example Statements

```
WAIT 3
WAIT Seconds/2
```

WHERE

WHERE returns the current logical position of the graphics pen.

Syntax



Item	Description	Range
x variable name	name of a numeric variable	any valid name
y variable name	name of a numeric variable	any valid name
status variable name	name of a string variable whose dimensioned length is at least 3	any valid name

Example Statements

```
WHERE X_pos,Y_pos
WHERE X,Y,Status$
```

Details

The characters in the status string have the following meaning:

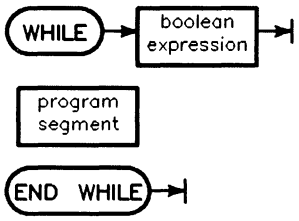
Character Position	Value	Meaning
1	"0"	Pen is up
	"1"	Pen is down
2	comma	(delimiter)
3	"0"	Current position is outside hard clip limits.
	"1"	Current position is inside hard clip limits but outside viewport boundary.
	"2"	Current position is inside viewport boundary and hard clip limits.

WHILE

WHILE defines a loop which is repeated until the expression in the WHILE statement evaluates to false (zero).

Example Statements

Syntax



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).	—

```
330 WHILE Size>=Minimum
340   GOSUB Process
350   Size=Size/Scaling
360 END WHILE
```

Details

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

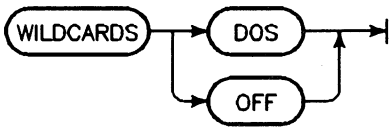
WHILE ... END WHILE constructs may be nested within other constructs, provided the inner construct begins and ends before the outer construct can end.

WILDCARDS

WILDCARDS enables and disables wildcard recognition in various file-related commands.

Wildcard recognition is disabled at power-up and after SCRATCH A. To use wildcards, you must explicitly enable them using WILDCARDS DOS.

Syntax



Example Statements

```
WILDCARDS DOS
WILDCARDS OFF
```

Details

Definitions for WILDCARDS DOS

Wildcard	Meaning
?	Matches 0 or 1 characters. For example, `X??` matches file names of *up to 3* characters that begin with the letter `X` (for example "X", "Xa", and "Xab").
*	Matches any sequence of 0 or more characters either before or after a "." in a file name. For example, `X*` matches all file names (with null extensions) of *one or more* characters that begin with `X`. Similarly, X*.b* would match "Xabc.bat" or "Xyz.bak". You can use only one asterisk before the period and one asterisk after the period to match file names.

Here is an example program segment using WILDCARDS DOS:

```
100  WILDCARDS DOS
110  PURGE "FILE_*"      deletes all files prefixed FILE with no extension
120  PURGE "*.DAT"       deletes all files with .DAT extension
```

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.

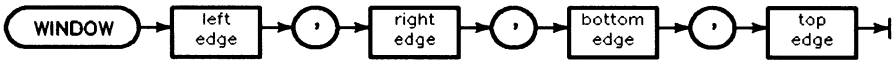
Commands Allowing Wildcards

File Name Expansion	File Name Completion
CAT	ASSIGN
PURGE	GET
COPY	LOAD
	LOAD SUB
	MSI
	PRINTER IS
	RENAME
	RE-SAVE
	RESTORE

WINDOW

WINDOW is used to define an anisotropic current unit-of-measure for graphics operations.

Syntax



Item	Description	Range
left edge	numeric expression	—
right edge	numeric expression	≠ left edge
bottom edge	numeric expression	—
top edge	numeric expression	≠ bottom edge

Example Statements

```
WINDOW -5,5,0,100
WINDOW Left,Right,Bottom,Top
```

Details

WINDOW defines the values represented at the hard clip boundaries, or the boundaries defined by the VIEWPORT statement. WINDOW may be used to create non-isotropic (not equal in X and Y) units. The direction of an axis may be reversed by specifying the left edge greater than the right edge, or the bottom edge greater than the top edge. (Also see SHOW.)

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 GPIB , 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 GPIB statement on a non- GPIB 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 GPIB 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

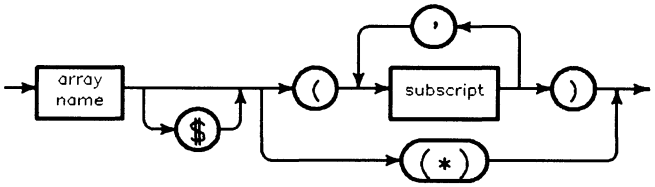
angle mode — The current units used for expressing angles. Either degrees or radians may be specified, using the DEG or RAD statements, respectively. The default at power-on and SCRATCH A is radians.

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.

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:

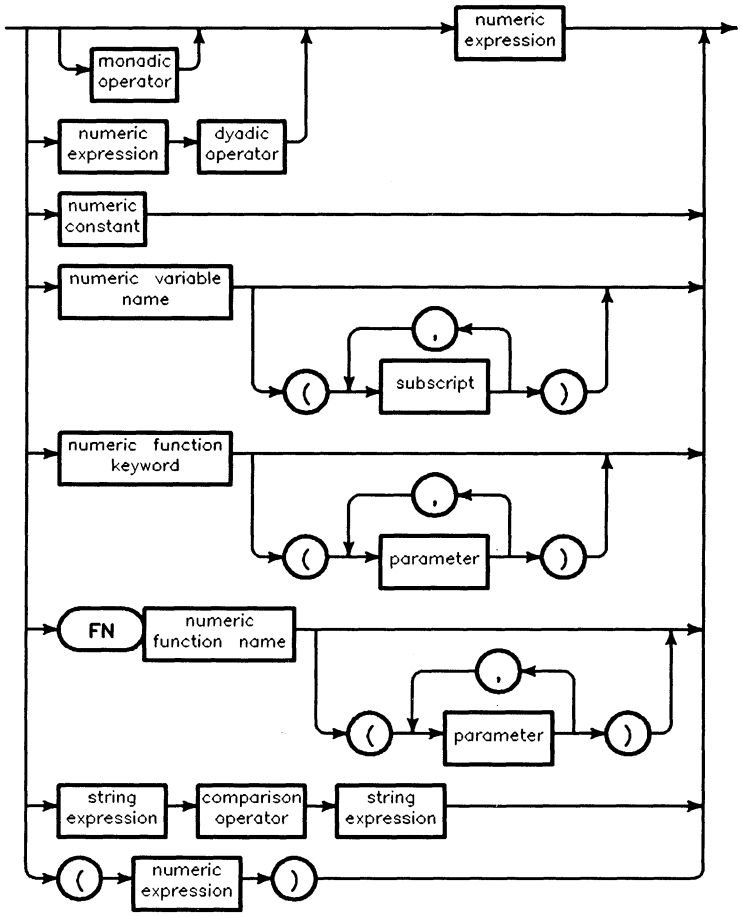
Monadic Operator	Operation
-	Reverses the sign of an expression
+	Identity operator
NOT	Logical complement

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 GPIB 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 GPIB . 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.

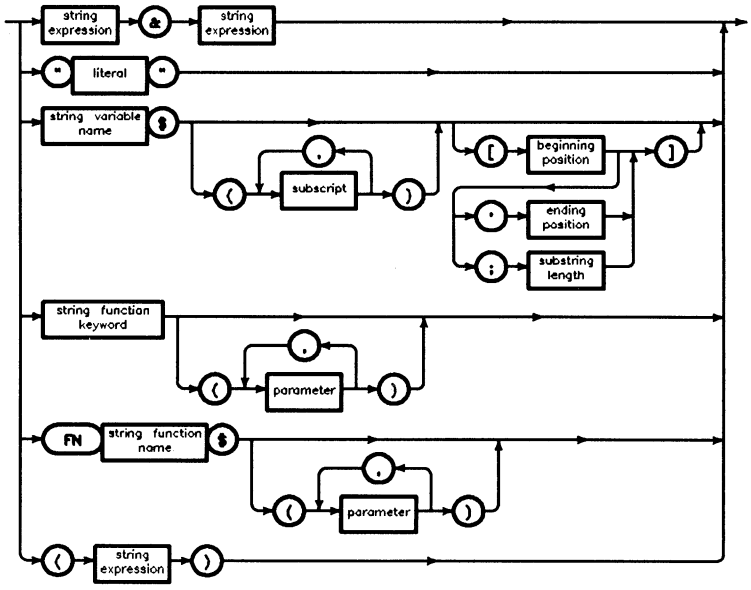
string — A data type comprised of a contiguous series of characters. Strings require one byte of memory for each character of declared length, plus a two-byte length header. Characters are stored using an extended ASCII character set. The first character in a string is in position 1. The maximum length 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.

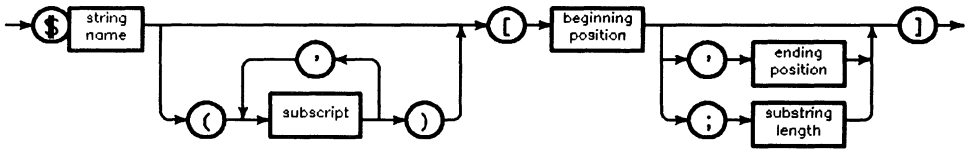
SUB subprograms are invoked by CALL. Function subprograms are invoked by an FN function occurring in an expression. A function subprogram returns a value that replaces the occurrence of the FN function when the expression is evaluated. Subprograms may alter the values of parameters passed by reference 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.

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.

C

Interface Registers

This section lists the STATUS and CONTROL registers for I/O path names, interfaces, and pseudo-select code 32.

I/O Path Registers

Registers for All I/O Paths

STATUS Register 0	0=Invalid I/O path name 1=I/O path name assigned to a device 2=I/O path name assigned to a data file 3=I/O path name assigned to a buffer 4=I/O path name assigned to an HP-UX special file

I/O Path Names Assigned to an ASCII File

STATUS Register 1	File type = 3
STATUS Register 2	Device selector of mass storage device (not supported for HF on BASIC/UX)
STATUS Register 3	Number of records
STATUS Register 4	Bytes per record = 256
STATUS Register 5	Current record
STATUS Register 6	Current byte within record
STATUS Register 9	File I/O buffering in use
CONTROL Register 9	Set file I/O buffer. BASIC/WS allows you to write to this register but no action is taken. Writing zero (0) enables buffering. Writing one (1) disables buffering.
CONTROL Register 10	In BASIC/DOS, writing a 1 to this register writes the pending buffer to the disk file and updates the directory entry for the file. However, this command has no effect on the buffering mode as defined by Control Register 9. Note that BASIC/WS and BASIC/UX allow this command but perform no action.

I/O Path Names Assigned to a BDAT File

STATUS Register 1	File type = 2
STATUS Register 2	Device selector of mass storage device (not supported for HF on BASIC/UX)
STATUS Register 3	Number of defined records
STATUS Register 4	Defined record length
STATUS Register 5	Current record
CONTROL Register 5	Set record
STATUS Register 6	Current byte within record
CONTROL Register 6	Set byte within record
STATUS Register 7	EOF record
CONTROL Register 7	Set EOF record
STATUS Register 8	Byte within EOF record
CONTROL Register 8	Set byte within EOF record
STATUS Register 9	File/I/O buffering in use
CONTROL Register 9	Set file I/O buffer. BASIC/WS and BASIC/DOS allow you to write to this register but no action is taken. Writing zero (0) enable buffering. Writing one (1) disables buffering.
CONTROL Register 10	In BASIC/DOS, wriging a 1 to this register writes the pending buffer to the disk file and updates the directory entry for the file. However, this command has no effect on the buffering mode as defined by CONTROL Register 9. Note that BASIC/WS and BASIC/UX allow this command but perform no action.

I/O Path Names Assigned to a DOS File

STATUS Register 0	0 = Invalid path name 1 = I/O path assigned to device 2 = I/O path assigned to data file 5 = I/O path assigned to widget
STATUS Register 1	File type = 4
STATUS Register 2	Device selector of mass storage device
STATUS Register 3	Number of defined records
STATUS Register 4	Defined record length (fixed record length = 1)
STATUS Register 5	Current record
CONTROL Register 5	Set record
STATUS Register 6	Current byte within record
CONTROL Register 6	Set byte within record
STATUS Register 7	EOF record
CONTROL Register 7	Set EOF record
STATUS Register 8	Byte within EOF record
CONTROL Register 8	Set byte within EOF record
STATUS Register 9	File/I/O buffering in use
CONTROL Register 9	Set file I/O buffer.
CONTROL Register 10	Flush file I/O buffer contents immediately; all buffered data is written to the disk

CRT and CONTROL Registers

See the instrument specific HP Instrument BASIC manual.

Keyboard STATUS and CONTROL Registers

See the instrument specific HP Instrument BASIC manual.

HP-IB STATUS and CONTROL Registers

See the instrument specific HP Instrument BASIC manual.

RS232C Serial STATUS and CONTROL Registers

See the instrument specific HP Instrument BASIC manual.