# Programming with HP BASIC

HEWLETT
PACKARD

Edition 1 June 1989

Reorder Number
82301-90015

## Notice

**Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.** Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

**Corvallis Information Systems**
**1000 N.E. Circle Blvd.**
**Corvallis, OR 97330, U.S.A.**

## Printing History

| | | |
|---|---|---|
| **Edition 1** | June 1989 | Mfg. No. 82301-90016 |

# Contents

## Introduction

## Part I: General Programming Techniques

### Chapter 1: Program Structure and Flow

## Chapter 2: Numeric Computation

## Chapter 3: Numeric Arrays

## Chapter 4: String Manipulation

## Chapter 5: Subprograms and User-Defined Functions

## Chapter 6: Data Storage and Retrieval

## Chapter 7: Using a Printer

## Chapter 8: The BASIC Clock

## Chapter 9: Errors and Debugging

# Part II: Graphics Techniques

## Chapter 10: Creating Graphics

## Chapter 11: External Graphics Displays and Plotters

## Part III: Interfacing Techniques

## Chapter 12: Introduction to I/O

## Chapter 13: Outputting and Entering Data

## Chapter 14: Advanced Interfacing Topics

## Chapter 15: Transfers and Buffered I/O

## Chapter 16: Techniques for Specific Interfaces

# Introduction

This manual describes selected HP BASIC programming techniques in three parts:

- Part I, "General Programming Techniques," (chapters 1 through 9) covers general programming techniques that you can use for a wide variety of applications.
- Part II, "Graphics Techniques," (chapters 10 and 11) covers graphics programming techniques. You can use these techniques to present information graphically on the CRT, or on an external printer or plotter.
- Part III, "Interfacing Techniques," (chapters 12 through 16) covers I/O interfaces and I/O programming techniques. You can use these techniques to output and enter data, and to control peripheral devices from your computer.

This manual should be used in conjunction with other HP BASIC manuals:

- For information about installing BASIC in the HP BASIC Language Processor system, and about the language processor programming environment, refer to *Installing and Using HP BASIC in the MS-DOS Environment.*
- For additional information about the syntax of HP BASIC keywords, refer to the keyword dictionary in the BASIC *Language Reference* manual, or to the BASIC *Condensed Reference* manual.

*This manual covers programming techniques for HP BASIC 5.0/5.1 and later versions of BASIC.* Most of the techniques presented are also applicable to earlier versions of BASIC. However, you should be aware that there may be some differences.

If you want to learn more about HP BASIC programming techniques, particularly I/O programming techniques, an excellent self-paced course is available. The product number is HP 82302A and the title is *Using HP BASIC for Instrument Control, a Self-Study Course.*

**Example Programs.** Several of the example programs used in this manual are provided on disk for your convenience. For the HP BASIC Language Processor these examples are found in the DFS directory named "EXAMPLES" on the "Manual Examples and Selected CSUBs" disk.

**Conventions.** Throughout this manual, when you are asked to "execute" an HP BASIC statement, type the statement on the BASIC command line and then press the Enter key.

Softkeys are indicated by a dot screen, for example EDIT.

Examples of BASIC statements and programs are given in a "computer"-style type. If a statement contains a variable for which you must supply a value, the variable will be in italics, for example:

```
PEN pen_number
```

**Program EDIT Mode.** Press the EDIT softkey (or type "EDIT") and then press Enter to go into the EDIT mode. You can now enter a program from the keyboard using the editor. Type in the program lines, pressing Enter after each one. The editor will number the program lines automatically. To exit the EDIT mode, press the Pause key. To clear a program from memory, press SCRATCH, followed by Enter. For further information, refer to *Installing and Using HP BASIC in the MS-DOS Environment.*

**Additional References.** For additional information about HP BASIC, you may want to refer to these additional manuals which are available from Hewlett-Packard:

| Manual Name | Part Number | Description |
|---|---|---|
| *BASIC Programming Techniques* | 98613-90012 | Detailed description of programming in HP BASIC. (Volume 2 provides information on porting BASIC programs from earlier versions of HP Series 200/300 BASIC.) |
| *BASIC Graphics Techniques* | 98613-90032 | Detailed description of graphics programming techniques. |
| *BASIC Interfacing Techniques* | 98613-90022 | Detailed description of interfaces and I/O programming techniques. (Volume 2 provides additional information on Series 200/300 interfaces.) |
| *Installing and Maintaining the BASIC System* | 98613-90042 | Information on installing and maintaining HP BASIC on an HP 9000 Series 200/300 system. |
| *Using the BASIC System* | 98613-90000 | Information on using HP BASIC on an HP 9000 Series 200/300 system. |

# Part I: General Programming Techniques

Chapters 1 through 9 cover general programming techniques that you will find useful for a wide variety of applications.

1

# Program Structure and Flow

There are four general categories of program flow. These are *sequence*, *selection* (conditional execution), *repetition*, and *event-initiated branching*. This chapter tells you how to use all of these types of program flow.

## Sequence

Sequence is fundamental to program flow — it provides order for what the computer is to do.

### Linear Flow

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.
- Linear flow is the default mode of execution. Unless you include a statement that stops or alters program flow, the computer will always "fall through" to the next higher numbered line after finishing the line it is on.

### Halting Program Execution

There are three statements that can be used to block execution of the next line and halt program flow.

1. The END statement. The primary purpose of the END statement is to mark the end of the main program, however when an END statement is executed, program flow stops and the program moves into the stopped (non-continuable) state.
2. The STOP statement. This acts just like an END statement in that it stops program flow. You use a STOP statement when you desire program flow to stop at some point other than the end of the main program.

3. The PAUSE statement. You use the PAUSE statement to temporarily halt program execution, leaving the program variables intact. Execution is halted until you press CONTINUE on the keyboard.

To demonstrate, type in the following program:

```
10 Radius = 5
20 Circum = PI*2*Radius
30 PRINT INT(Circum)
40 PAUSE
50 Area = PI*Radius^2
60 PRINT INT(Area)
70 END
```

Now run the program by pressing RUN or type "RUN" and press ENTER. The computer prints 31 on the CRT and the Run Indicator in the lower right corner of the CRT is replaced with a -, indicating the program is in a paused state. Now press CONTINUE. The computer prints 78 on the CRT.

## Simple Branching

The simplest form of branching uses the statements GOTO and GOSUB. Both statements cause an unconditional branch to a specified location in the program.

**The GOTO Statement.** GOTO causes the program to branch to some line that is not the next line in the program. GOTO can reference the line to branch to either by line number or line label. (Line labels are discussed later in this section.) In the following example, both GOTO statements cause a branch to the PAUSE statement at line 300 (or "Label").

```
100 GOTO 300

150 GOTO Label
  .
  .
  .
300 Label:    PAUSE
310           END
```

**The GOSUB Statement.** GOSUB is used to transfer program execution to a subroutine. A subroutine is a segment of a program that is entered with a GOSUB statement and exited with a RETURN statement. There are no parameters passed and no local variables are allowed in the subroutine. The GOSUB statement can specify either the line number or a line label as a designated entry point for the subroutine being called. Here is an example:

```
100 GOSUB 500

150 GOSUB Subrtn
  .
  .
  .
500 Subrtn:    ! Start of subroutine called "Subrtn".
  .
  .
  .
550 RETURN
```

Remember that each time a subroutine is called by a GOSUB, control is returned to the line immediately following the GOSUB when the RETURN is encountered in the subroutine. Therefore you must have a RETURN for each subroutine. Note that if you omit the RETURN, the program will continue executing beyond the point at which you expected it to return, until it encounters another RETURN, STOP, or END. Obviously, this could produce surprising results.

**Line Labels and Comments.** Before we go further, there are two topics that we should cover: *line labels* and *comments*.

- *Line labels* are used within the program so that the computer can identify a line for branching purposes. You may want to use a line label instead of a line number in a GOTO or GOSUB statement because the label won't change, even if you renumber the program. Also, the label can make it easier for a programmer to "read" the program. Line labels must immediately follow the line number and must be followed by a colon (:). Line labels consist of an initial capital letter followed by lower-case letters. *A line label only will affect program execution if referenced in a GOTO or GOSUB statement.*
- *Comments* never affect program execution. They are included only to clarify the program. Comments are always preceded by an exclamation point (!) — anything in a program line that follows an exclamation point is a comment. Many of the examples in this manual include comments for clarification of the program steps.

Note that a line label *precedes* the executable statement in a program line, while a comment *follows* the executable statement (if there is one). The following program illustrates the use of both line labels and comments:

```
10 Start:      ! Program begins here.
20             PRINT "Hello"
30             GOTO Finish
40             DISP "Stop"         ! This line is never executed.
50 Finish:         PRINT "Goodbye"   ! Print = executable statement.
60                 GOTO Start
70                 END
```

Line 10 includes the line label "Start" and a comment, but no executable statement. Line 50 includes the line label "Finish", the executable PRINT statement, and ends with a comment. If you were to type in and run this program it would simply print "Hello" and "Goodbye" repeatedly until you pause the program by pressing Pause. Normally you will want to avoid such "endless loops." To do this you can use conditional branching, which is covered in the next section.

*Note that you cannot have a program line with only a line label.* There must be an executable statement or at least a comment. For example:

```
100   Label:
```

is not allowed. However, the following line is legal even though the "comment" consists of only the comment symbol (!).

```
100   Label:   !
```

## Selection

The heart of a computer's decision-making power is the category of program flow called selection, or conditional execution. A certain set of the program either is or is not executed, depending on 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:

1. Conditional execution of one segment.
2. Conditionally choosing one of two segments.
3. 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 an expression that is evaluated as being either true or false. If true, the conditional segment is executed. If false, the conditional segment is bypassed. The conditional segment can be either a single BASIC statement or a program segment containing any number of statements. The following 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...) which appear on either side of the keyword THEN. If the value of Ph is greater than 7.7 the PRINT statement is executed. If the value of Ph is equal to or less than 7.7 the PRINT statement is not executed. In either case, the line number immediately following line 100 would be executed next. (Although "pH" is the correct chemical expression, it is not valid as an HP BASIC variable name. Thus "Ph" has been substituted in the example.)

## Conditional Branching

Powerful control structures can be developed by using branching statements in an IF...THEN statement. Here are some examples:

```
110 IF Free_space < 100 THEN GOSUB Expand_file
120 !THE LINE AFTER IS ALWAYS EXECUTED
```

The statement checks the value of a variable called Free_space, and if it is less than 100, a subroutine called Expand_file is executed. If the value is not less than 100, the subroutine is not executed. 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 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
```

## 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 "Ph value is";Ph
130   GOSUB Setup
140 END IF
150 ! Program continues here
```

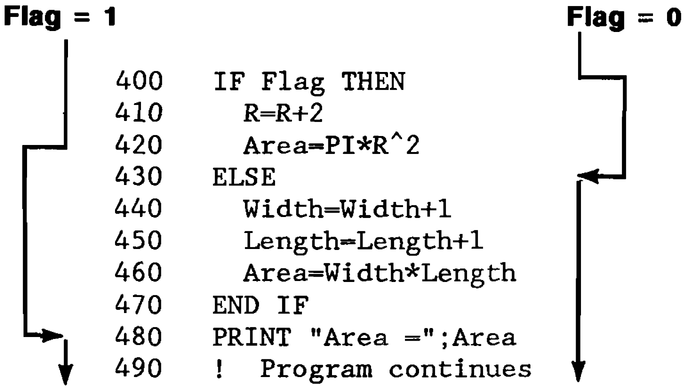
If Ph is less than or equal to 7.7, the computer skips all the statements between the IF. . .END IF statements and continues with the line following the END IF. If the value of Ph is greater than 7.7, then the statements between the IF. . .END IF are executed before continuing on to the line after the END IF. Any number of program lines can be placed between an IF. . .END IF statement, including other IF. . .END IF statements. For example:

```
100 IF Flag THEN
110   IF End_of_page THEN
120     FOR I = 1 TO Skip_length
130       PRINT
140       Lines = Lines + 1
150     NEXT I
160   END IF
170 END IF
```

Remember, you can use the INDENT command to improve the readability of your programs.

## 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:



```
400   IF Flag THEN
410     R=R+2
420     Area=PI*R^2
430   ELSE
440     Width=Width+1
450     Length=Length+1
460     Area=Width*Length
470   END IF
480   PRINT "Area =";Area
490   !  Program continues
```

This example has an IF...THEN...ELSE structure which makes the one-of-two choice easy and readable.

## Choosing One of Many Segments

The SELECT...END SELECT is similar to the IF...THEN...ELSE...END IF construct, but allows several conditional program segments to be defined. Only one segment is 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 SELECT statement.

Consider the processing of readings from a voltmeter. Readings which contain a function code have been taken. The 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 | Resistance |

The following example shows the use of the SELECT construct. The function code is contained in the variable Funct$.

```
2000 SELECT Funct$
2010 CASE "DV"
2020 !
2030 ! Processing Statements For DC Volts
2040 !
2050 CASE "AV"
2060 !
2070 ! Processing Statements For AC Volts
2080 !
2090 CASE "DI"
2100 !
2110 ! Processing Statements For DC Current
2120 !
2130 CASE "AI"
2140 !
2150 ! Processing Statements For AC Current
2160 !
2170 CASE "OM"
2180 !
2190 ! Processing Statements For Resistance
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 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 continues with the line following the END SELECT.

You should be aware that 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 the SELECT statement means that there is an error in that line or in one of the CASE statements following it.

### Using the ON Statement

The same type of program flow can be generated with an ON statement and some additional processing. The ON statement transfers program control to one of several destinations depending on the value of a pointer. The pointer can be a numeric expression rounded to an integer, but its final value must be an integer.

```
100 ON X1 GOTO 150,200,300
```

In the above example, X1 is the pointer whose value will be evaluated. If the value is 1, program control will be transferred to line 150; if it is 2, control is transferred to line 200; and if it is 3, control is transferred to line 300. If X1 has a value other than 1, 2, or 3, an error results:

```
ERROR 19 IN 100  Improper value or out of range
```

You can also use the ON statement with GOSUB instead of GOTO. In this case, the RETURN from the GOSUB is to the line following the ON...GOSUB statement.

```
100 ON X1 GOSUB FIRST,SECOND,THIRD,LAST
110 PRINT "NEXT STATEMENT"
```

The variable X1 is evaluated and the subroutine beginning at the line identifier FIRST, SECOND, THIRD, or LAST, is executed depending on whether X1 is 1, 2, 3, or 4. Control is returned to line 110 regardless of which subroutine is executed. As before, an error results if X1 is not 1, 2, 3, or 4.

## Repetition

There are four structures available for creating repetition. The FOR...NEXT structure is used for repeating a program segment a predetermined number of times. Two other structures, REPEAT...UNTIL and WHILE, are used for repeating 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. 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 the variables involved are integers, the number of iterations of any loop can be predicted using the formula

(STEP SIZE + FINAL VALUE − STARTING VALUE) ÷ STEP SIZE

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

The NEXT statement performs an "increment and compare" on the loop counter. This means that the loop counter is incremented by the step size and then compared to the final value. If the loop counter has passed the specified value, the loop is exited, otherwise the loop is repeated. Note that if the number of iterations evaluates to zero or less, the loop is not executed and program execution goes immediately to the line following the NEXT statement.

The loop counter retains the exit value after the loop is finished.

## Conditional Number of Iterations

Some applications need a loop that is executed until a certain condition is true, without specifically stating the number of iterations involved. For example, suppose you want to be able to print the value of successive powers of two, but only until the value is greater than 1000. The REPEAT...UNTIL is more flexible than the FOR...NEXT in this case. Consider the following example program (found in file REPEAT1 on your Manual Examples disk).

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

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

```
2 4 8 16 32 64 128 256 512 1024
```

The WHILE loop is used for the same purpose as the REPEAT loop. The only difference between the two is the location of the test for exiting the loop. The REPEAT loop has its test at the bottom. This means that the loop is always executed at least once, regardless of the value of the test condition. The WHILE loop has its test at the top, therefore it is possible for the loop to be skipped entirely. The following example (found in file WHILE1 on your Manual Examples disk) shows this.

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

The results obtained from this example should be identical to the example using the REPEAT. . .UNTIL loop. Try these examples on your computer, and don't be afraid to experiment with them. Change them to suit your own needs. This will help you to understand the concepts of iterative processing.

## Arbitrary Exit Points

The loop structures discussed so far do not allow for conditional exit points within the program segment between the top and bottom of the loop. The LOOP...EXIT IF construct allows you to do this. It also allows you to have more than one exit point. 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 the 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. The following two examples demonstrate this.

In this example, the EXIT IF statement is nested deeper than the LOOP statement because it is placed in an IF...THEN structure.

```
100 LOOP
110   Test = RND -.5
120   IF Test < 0 THEN
130     PRINT "NEGATIVE"
140   ELSE
150     EXIT IF Test > 0.4
160     PRINT "POSITIVE"
170   END IF
180 END LOOP
190 END
```

Here is the proper structure to use.

```
100 LOOP
110   Test = RND -.5
120 EXIT IF Test > 0.4
130   IF Test < 0 THEN
140     PRINT "NEGATIVE"
150   ELSE
160     PRINT "POSITIVE"
170   END IF
180 END LOOP
190 END
```

If you enter the "wrong" example and try to run it, you will get the following error message:

```
ERROR 347 IN 150  Structures improperly matched
```

Now try the “right” example. The program should print the words “positive” and “negative” a random number of times, and will stop when the value of the variable TEST is greater than 0.4. In effect, since the RND function returns a fractional value between 0 and 1, the program stops the first time RND returns a value greater than 0.9

## Event-Initiated Branching

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

ON CYCLE
ON DELAY
ON END
ON EOR
ON EOT
ON ERROR
ON HIL EXT
ON INTR
ON KBD
ON KEY
ON KNOB
ON SIGNAL
ON TIME
ON TIMEOUT

The ON END event is used to detect when the end of a mass storage file is reached. The ON CYCLE, ON DELAY, and ON TIME events are used to direct program flow using the clock. The ON ERROR event is used to trap run-time errors and provide for error recovery routines. The ON KBD, ON KEY, and ON KNOB events pertain to various parts of the keyboard, and are used to enhance the “human interface” of programs. The ON EOR, ON EOT, ON SIGNAL, ON INTR, ON HIL EXT, and ON TIMEOUT events pertain to data transfer, interfaces, and I/O operations.

The best way to understand how event-initiated branches operate in a program is to try a few examples on your computer. Try the following example (found in file ONKEY1 on your Manual Examples disk).

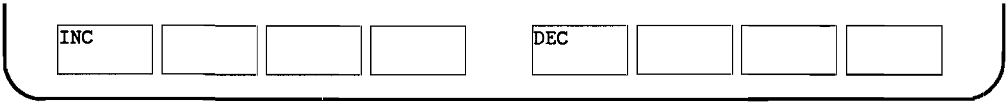
```
10  ON KEY 1 LABEL "INC" GOSUB Plus
20  ON KEY 5 LABEL "DEC" GOSUB Minus
30  !
40  Spin:DISP X
50   GOTO Spin
60  !
70  Plus:X=X+1
80   RETURN
90  !
100 Minus:X=X-1
110  RETURN
120 END
```

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 lines 40 and 50 because of the ON KEY action. Either the "Plus" or "Minus" subroutines are selected as a result of softkey presses. If no softkey is pressed, the computer continues to display the value of X. The following section of pseudocode shows the program flow of the "Spin" segment looks like.

```
Spin: DISPLAY X
  IF KEY 1 THEN GOSUB PLUS
  IF KEY 5 THEN GOSUB MINUS
  GOTO Spin
```

Note that the only way to terminate this program is to type "STOP" and press ENTER.

Now run the sample program you have just entered. Notice that the bottom two lines of the display screen display an inverse-video label area.



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

2

# Numeric Computation

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

The most fundamental numeric operation is the assignment operation, achieved with the LET statement. The LET statement originally required the keyword LET, but the HP BASIC system makes it optional. Thus, the following statements are equivalent:

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

## Numeric Data Types

There are three numeric data types in BASIC:

- REAL.
- INTEGER.
- COMPLEX.

Any numeric variable that is not declared COMPLEX or INTEGER is a REAL variable.

### REAL Variables

The valid range for REAL variables is approximately $-1.797\,693\,134\,862\,315 \times 10^{308}$ through $1.797\,693\,134\,862\,315 \times 10^{308}$ (or −MAXREAL through +MAXREAL). However, the smallest non-zero REAL value allowed is approximately $\pm 2.225\,073\,858\,507\,202 \times 10^{-308}$ (or ±MINREAL). A REAL variable can also have a value of zero.

## INTEGER Variables

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

## COMPLEX Variables

A COMPLEX number is written as the sum of a real and an imaginary number. An imaginary number is any real number multiplied by $\sqrt{-1}$, and is expressed by mathematicians in the following manner:

$a + ib$

where $i = \sqrt{-1}$. In the above representation, a is the real part of the complex number, and ib is the imaginary part. The *i* in front of the b forms the imaginary number, and is the same as multiplying b by $\sqrt{-1}$. For example, you would write $\sqrt{-9}$ as $\sqrt{-1} * \sqrt{9}$ or simply 3*i*. Electrical engineers use the letter *j* instead of i, to avoid confusion with the symbol for electric current. COMPLEX numbers are stored as two REAL variables, thus a COMPLEX number will require 16 bytes of memory.

## Variable Names

Variable names can be up to 15 characters long. The first character must be a capital letter. The rest of the name can consist of *lower-case* letters, numerals, and the underscore character (_), in any combination. No other characters are allowed in a numeric variable name.*

Here are some examples of valid numeric variable names:

```
Location_1_a
A1
I
I_no_2
Income_1988
Integer_variabl
```

* String variable names are like numeric variable names, but have an appended dollar sign ($). For example: "String_variable$".

## Declaring Variables

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

```
COMPLEX B, C, Phasor1(10), Phasor2(10)
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 is a variable which can represent a single value. An array is a subscripted variable, and can contain 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 existing OPTION BASE statement as the lower bound.You may declare an array using the DIM statement:

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

You may use an ALLOCATE statement to declare REAL, INTEGER, and COMPLEX arrays:

```
ALLOCATE REAL Coords(2,1:Points), INTEGER Status(1:Points)
ALLOCATE COMPLEX Poles(2,1:Points), REAL Location(2,1:Points)
```

The ALLOCATE statement allows you to dynamically allocate memory in programs which need tight control over memory use. Arrays will be discussed in detail in chapter 3, “Numeric Arrays.”

## Type Conversions

The computer will automatically convert between REAL and INTEGER values in assignment statements and when parameters are passed by value in program and function calls. When parameters are passed by reference the conversion is not made, and a TYPE MISMATCH error will be reported if the calling parameter and the subprogram parameters are of 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 REAL will not restore the fractional part. Also, because of the difference in ranges between the two types, not all REAL values can be converted into an equivalent INTEGER value. This problem can generate INTEGER OVERFLOW errors. The rounding problem does not generate an execution error, but the range problem can generate an execution error, and you should protect yourself from this possibility. One way to do this is as follows.

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

## Resident Numerical Functions

The resident functions are the functions that are part of the BASIC language. Your BASIC language includes numerous functions to make mathematical operations easier. This section covers these functions by placing them in the following categories:

- Arithmetic Functions.
- Array Functions.
- Exponential Functions.
- Trigonometric Functions.
- Hyperbolic Functions.
- Binary Functions.
- Limit Functions.
- Rounding Functions.
- Random Number Function
- Complex Functions.
- Time and Date Functions.
- Base Conversion Functions.
- General Functions.

## Arithmetic Functions

Your BASIC language includes the following arithmetic functions:

ABS Returns the absolute value of an expression. Takes a REAL, INTEGER, or COMPLEX 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 BASIC. Its value is approximately 1.797 693 134 862 32E+308.

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

SQRT (or SQR) Returns the square root of an expression. Takes a REAL, INTEGER, or COMPLEX number as its argument.

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

## Array Functions

These functions are available when the MAT binary is loaded. They return specific information about numeric arrays (for example, the number of dimensions in the array, the determinant of an array, and so forth). For more information on the numeric array functions listed below, refer to chapter 3, “Numeric Arrays.”

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

DET Returns the determinant of a matrix.

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

RANK Returns the number of dimensions in an array.

SIZE Returns the number of elements in a dimension of an array.

SUM Returns the sum of all elements in a numeric array.

## Exponential Functions

This section provides a list of functions used for determining the natural and common logarithms of an expression. All exponential functions use REAL, INTEGER, or COMPLEX numbers as their argument.

EXP Raise the Naperian *e* to a power ($e \simeq 2.71828182845905$).

LGT Returns the base 10 logarithm of the expression.

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

## Trigonometric Functions

Six trigonometric functions and the constant $\pi$ are provided for dealing with angles and angular measure. Note that all trigonometric functions take REAL, INTEGER, or COMPLEX 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 the expression.

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

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

*The default mode for all angular measure is radians.* Degrees can be selected with the DEG statement. Radians may be re-selected with 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 since the subprogram inherits the angular mode from the context that calls it. (The angular mode is part of the calling context. If it is changed in a subprogram, it is restored when the calling context is restored.)

## Hyperbolic Functions

Six hyperbolic functions are available when the COMPLEX binary is loaded.

| | |
|---|---|
| SINH | Returns the hyperbolic sine of a number. |
| COSH | Returns the hyperbolic cosine of a number. |
| TANH | Returns the hyperbolic tangent of a number. |
| ASNH | Returns the hyperbolic arcsine of a number. |
| ACSH | Returns the hyperbolic arccosine of a number. |
| ATNH | Returns the hyperbolic arctangent of a number. |

## Binary Functions

All computer operations use the binary number representation. You usually don't see this because the computer changes decimal numbers that you input into binary representation. The operations you specify are performed on the binary numbers, and results are changed back into decimal numbers before displaying or printing them. The following 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 two arguments. |
| 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 operations are used, the arguments are first converted to integer (if they are not already in integer) and then the specified operation is performed. You should restrict bit-oriented binary operations to declared INTEGER variables. If it is necessary to operate on REAL variables, be sure to use the precautions described under type conversions in the previous section, to avoid INTEGER OVERFLOW errors.

## Limit Functions

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

| | |
|---|---|
| MAX | Returns a value equal to the greatest value in a list of arguments. |
| MIN | Returns a value equal to the least value in a list of arguments. |

These functions work with both INTEGER and REAL values and require the MAT binary.

## Rounding Functions

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

| | |
|---|---|
| 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 one, 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.

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

The above statement will return an integer between OFFSET and OFFSET + RANGE. Try the following example, which will simulate ten throws of a die.

```
10 FOR I=1 TO 10
20   Die=INT(RND*6)+1
30   PRINT "DIE IS";Die
40 NEXT I
50 END
```

If you run the above program several times, you will see that the values for the die do not change from one run to the next. This is because the RND function is using the same seed for each run. The random number generator is seeded with the value 37480660 at power-on, during pre-run, and when SCRATCH or SCRATCH A are executed. You can change the seed by using the RANDOMIZE statement, which will give a new pattern of numbers. Edit the program above to add a RANDOMIZE statement as line 05 and see what happens.

## Complex Functions

These functions are obtained by loading the COMPLEX binary. Topics which are covered in this section are:

- Assigning COMPLEX Variables.
- Evaluating COMPLEX Numbers.
- Complex Arguments and the Trigonometric Mode.
- Determining the Parts of Complex Numbers.
- Converting from Rectangular to Polar Coordinates.

**Assigning COMPLEX Variables.** To assign complex variables, the variables must first be declared as complex, and one or more of the variables must have already been created using the CMPLX function. For example, the following program creates a complex variable C and assigns it to the complex variable B. It then displays the results.

```
10 COMPLEX B,C
20 REAL Real_part,Imaginary_part
30 Real_part=3.5
40 Imaginary_part=.5
50 C=CMPLX(Real_part,Imaginary_part)
60 B=C
70 PRINT C,B
80 END
```

Executing the above program produces these results:

```
3.5   .5   3.5   .5
```

**Evaluating COMPLEX Numbers.** The BASIC expression evaluation uses two separate routines for dealing with REAL, INTEGER and COMPLEX data types. There is a routine for dealing with REAL and INTEGER numbers and one for COMPLEX numbers. For example, taking the square root of a negative INTEGER or REAL number will produce an error. For instance, SQR(-1) results in

```
ERROR 30  SQR of negative number
```

If you have a need to compute the square root of a negative REAL or INTEGER number, assign the value to the real part of a complex number using the CMPLX function. For instance, SQR(CMPLX(-1,0)) results in

```
0  1
```

where "0" is the real part and "1" is the imaginary part of the COMPLEX number.

**Complex Arguments and the Trigonometric Mode.** When a trigonometric function call is made using a complex value as its parameter, BASIC will evaluate that call using the radian mode regardless of the current trigonometric mode setting (DEG, RAD, or GRAD). After the function call has been evaluated, the system returns to the current trigonometric mode. For example, enter and run this program:

```
10 DEG
20 PRINT SIN(30)
30 PRINT
40 PRINT SIN(CMPLX(30,0)) ! Always evaluated in the RAD mode.
50 PRINT
60 PRINT SIN(30)
70 END
```

The results from executing this program are as follows:

```
0.5                     (degree mode)
-.988031624093  0       (radian mode)
0.5                     (degree mode)
```

**Note** Any complex function whose definition includes a sine or cosine function will be evaluated in the radian mode regardless of the current trigonometric mode (i.e. RAD or DEG).

**Determining the Parts of Complex Numbers.** In some applications, such as network design, it is useful to be able to determine the real and imaginary parts of complex numbers, and the conjugate of a complex number. This section provides the functions necessary for performing these operations.

REAL(C) Returns the real part of a complex number. For example,

```
DISP REAL(CMPLX(10,-3))
```

Executing this statement produces:

```
10
```

IMAG(C) Returns the imaginary part of a complex number. For example,

```
DISP IMAG(CMPLX(10,-3))
```

Executing this statement produces:

```
-3
```

CONJG(C) Returns the complex conjugate of a complex number. This function returns both the real and imaginary parts of a complex number;however, the imaginary part is changed to a negative value. For example:

```
DISP CONJG(CMPLX(10,-3))
```

Executing this statement produces the following results:

```
10 3
```

**Converting from Rectangular to Polar Coordinates.** BASIC stores and uses complex numbers in a representation called rectangular coordinates. Rectangular coordinates locate a point in the complex plane. The complex plane is similar to the plane formed by the Cartesian coordinate system except the X axis represents the real part of the complex number and the Y axis represents the imaginary part of the complex number. An alternate representation is polar coordinates. Polar coordinates consist of a magnitude and an argument (angle). The function used to obtain the magnitude is ABS(C) and the function used to obtain the argument is ARG(C).

The following program converts the rectangular coordinates 5 and 6 of the complex number 5 + j6 to polar coordinates.

```
140 RAD
150 PRINT "The magnitude of 5 + j6 is: ";ABS(CMPLX(5,6))
160 PRINT "The argument of 5 + j6 is: ";ARG(CMPLX(5,6))
170 END
```

Executing this program produces the following results in radian mode (RAD):

```
The magnitude of 5 + j6 is: 7.81024967591
The argument of 5 + j6 is: .876058050598
```

If you change line 140 above to be:

```
140 DEG
```

and run the program again, the results in the degree mode (DEG) are:

```
The magnitude of 5 + j6 is: 7.81024967591
The argument of 5 + j6 is: 50.1944289077
```

## Time and Date Functions

The following functions return time/date information in seconds:

DATE — Converts a formatted date string ("DD MMM YYYY") into a numeric value in seconds.

TIME — Converts a formatted time-of-day ("HH:MM:SS") string into a numeric value of seconds since midnight.

TIMEDATE — Returns the current BASIC software clock value in Julian seconds; for example, 2.11404868285E+11. (If there is no battery-backed real-time clock, the software clock is set at power-on to 2.08662912E+11, which represents midnight March 1, 1900. If the computer is connected to an SRM system, the BASIC software clock is set to the value of the SRM system clock when the SRM binary is loaded.)

For further information on this subject, refer to chapter 8, "The BASIC Clock." Also included in chapter 8 are the DATE$ and TIME$ string functions.

## Base Conversion Functions

There are two functions you can use to convert binary, octal, decimal, or hexadecimal string values into a decimal number.

DVAL

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

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

returns the value:

```
-4
```

IVAL

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

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

returns the value:

```
5600
```

## 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 statement

```
ENTER 2;Numeric_value
```

allows you to enter a numeric value from the keyboard. The above statement is not as easy to understand as

```
ENTER KBD;Numeric_value
```

where you know the function KBD stands for keyboard. Functions which return a select code or device selector are as follows.

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

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

PRT Returns the INTEGER 701. This is the default (factory set) device selector for an external HP-IB printer.

SC Returns the interface select code associated with an I/O path name.

## Evaluating Scalar Expressions

The arithmetic operations that you can perform on the system are:

- Addition (+)
- Subtraction (−)
- Multiplication (*)
- Division (/)
- Exponentiation (^)
- Integer Division (/ or DIV)
- Modulo (MOD or MODULO)

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

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

When an expression is being evaluated it is read from left to right, and operations are performed as they are encountered, depending upon the hierarchy. If the computer cannot immediately perform the operation, it is stacked, and the evaluation continues. Consider the following expression:

```
4*(3^2/2)+5*SIN(Y)
```

The computer will evaluate this expression in the following the manner:

1. Perform the calculations inside the parentheses and multiply by 4.
2. Compute the sine of Y.
3. Multiply the sine of Y by 5.
4. Add the value found in step 1 to the value found in step 3.

## Strings in Numeric Expressions

You can include string expressions in numeric expressions if they are separated by comparison operators. The comparison operators always yield boolean results, which are numeric values in BASIC.

## Step Functions

The comparison operators are useful for conditional branching, but you can also use them 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 $= \sqrt{A^2 + B^2}$.
- If variable $\geq 1$ then output $= 15$.

You could achieve the desired result by using a series of IF..THEN statements, but you could also use 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 before the expression is evaluated is used to determine the result.

## Comparing REAL Numbers

When you compare INTEGER numbers, no special precautions are necessary. When you compare REAL numbers, especially the results of calculations and functions, it is possible to encounter problems due to rounding. For example, consider the use of comparison operators in IF..THEN statements to check for equality in 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. A repeating decimal or irrational number cannot be represented exactly in any finite machine.

Another good example of equality error occurs when multiplying or dividing data values. A product of two non-integer values nearly always results in more digits beyond the decimal point than exists in either of the two numbers being multiplied. Any tests for equality must consider the exact variable value to its greatest resolution. If you cannot guarantee that all digits beyond the required resolution are zero, you can use the DROUND function to eliminate unwanted resolution before comparing results. The following example (found in file DROUND1 on your Manual Examples disk) shows how you can use DROUND:

```
10  A=32.5087
20  B=31.625
30  C=A*B   ! PRODUCT IS 1028.08763750
40  D=32.5122
50  E=31.621595509
60  F=D*E   ! PRODUCT IS 1028.08763751
70  IF C=F THEN 90
80   PRINT "C is not equal to F."
90  C=DROUND(C,7)
100 F=DROUND(F,7)
110 IF C=F THEN
120   PRINT "C equals F after DROUND."
130 ELSE
140   PRINT "C is not equal to F after DROUND."
150 END IF
160 END
```

You can experiment with the concept by substituting other values for the variables A, B, D, and E, and by changing the number of digits specified in the DROUND function.

# Numeric Arrays

An array is a multi-dimensioned structure of variables that are given a common name. The array can have one through six dimensions. Each location in an array can contain one variable value, and each value has the characteristics of a single variable, depending on whether the array consists of REAL, INTEGER or COMPLEX values. 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. You can specify up to six dimensions for any array in a program. REAL arrays require eight bytes of memory for each element, plus overhead, and COMPLEX arrays require 16 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.

**Note**

Many of the statements that deal with arrays (such as MAT) require the MAT binary. If you do not have this binary loaded in your system, or you are not sure how to determine if it is loaded, refer to *Installing and Using HP BASIC in the MS-DOS Environment* for more information.

## 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, REAL or COMPLEX statements. For example,

```
COMPLEX 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 or COMPLEX type in the dimensioning statement, arrays default to REAL type. The same array can only be dimensioned once in a context.[*]

However, as we explain later in this section, you can redimension arrays by using the REDIM statement.

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 example,

```
DIM Array(3,4)
```

dimensions a 3 × 4 two-dimensional array with the first subscript (3) representing three rows and the second subscript (4) representing four columns. For a four-dimensional array, for instance, each element is identified by four subscript values. Each unique set of subscript values points to one, and only one, array element. The actual size of an array is governed by the number of dimensions and the subscript range of each dimension. If A is a three-dimensional array with a subscript range of 1 thru 4 for each dimension,

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

then its size is 4 × 4 × 4, or 64 elements. Note that 1 on the left side of the colon in the dimension statement above is the lower bound and 4 on the right is the upper bound. Therefore, when you dimension an array you must give not only the number of dimensions, but also the subscript range of each dimension. Subscript ranges can be specified by giving the lower and upper bounds, as shown above, or by giving just the upper bound. If you give only the upper bound, the lower bound defaults to the current option base setting. Each context initializes to an option base of 0 (arrays appearing in COM statements with an (*) will keep the base with which they were originally dimensioned). 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.

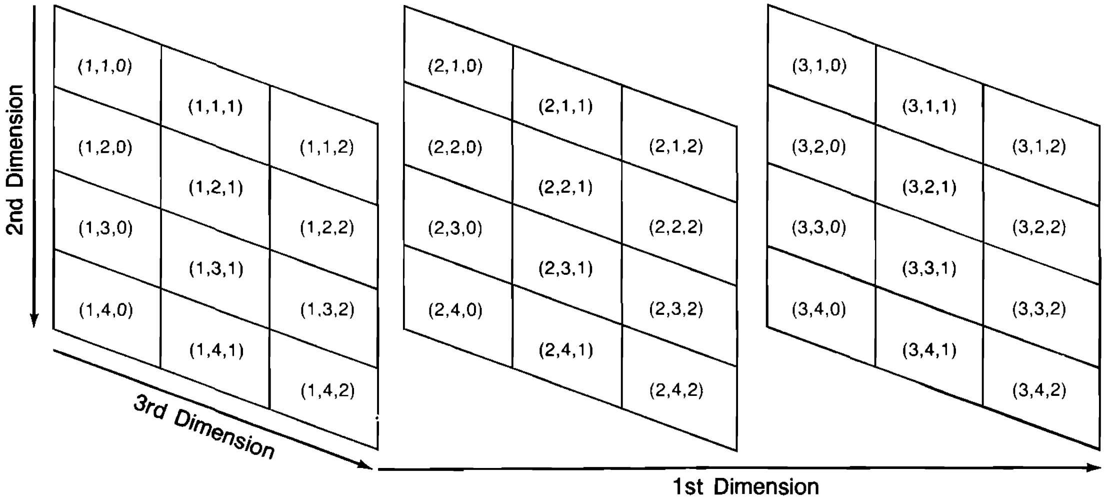
[*] There is one exception to this rule: If you ALLOCATE an array, and then DEALLOCATE it, you can dimension the array again.

## Some Examples of Arrays

The following examples illustrate some of the flexibility you have in dimensioning arrays. *Unless specified otherwise, all examples in this section use option base 1.*

In the first example the DIM statement is used to dimension a three-dimensional array:

```
10 DIM A(3,4,0:2)
```



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

In this example we portray the first dimension as planes, the second dimension as rows, and the third dimension as columns. In general, the last two dimensions of any array always refer to rows and columns, respectively. When we discuss two-dimensional arrays, the first dimension will always represent rows, and the second dimension will always represent columns. Note also in the above example that the first two dimensions use the default setting of 1 for the lower bound, while the third dimension explicitly defines 0 as the lower bound. The numbers in parentheses are the subscript values for the particular elements. These are the numbers you use to identify each array element.

In the following example, COM is used to dimension a two-dimensional array:

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

| | | | | |
|---|---|---|---|---|
| (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) |

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

The following is another two-dimensional array, but this time the ALLOCATE statement is used:

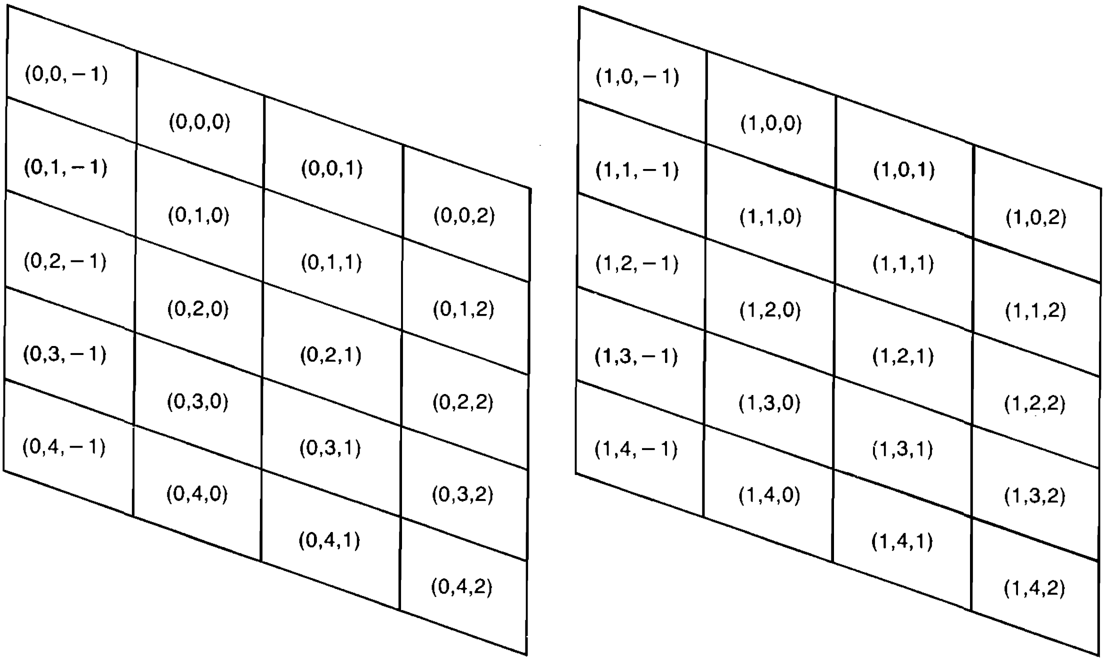
```
10 ALLOCATE INTEGER C(2:4,-2:2)
```

| | | | | |
|---|---|---|---|---|
| (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) |

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

Now let's look at a three-dimensional array dimensioned with the INTEGER statement, in this case using OPTION BASE 0:

```
10 OPTION BASE 0
20 INTEGER D(1,4,-1:2)
```



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

Arrays are limited to six dimensions, and the subscript range for each dimension must lie between −32767 and 32767. (REDIM and ALLOCATE allow the subscript range to go down to −32768, but the total size of each dimension must be less than 32768 elements.) For the most part, we use only two-dimensional examples since they are easier to illustrate. However, the same principles apply to arrays of more than two dimensions as well. 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. You can specify the location of a particular book by using the number of the floor, the stack, the shelf, and the particular book on that shelf. You can dimension an array for the library with the statement:

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

This means that there are 100,000 book locations. You identify a particular book by specifying its subscripts. For example, the statement:

```
Library(2,12,3,35)
```

identifies the 35th book on the 3rd shelf of the 12th stack on the 2nd floor. You can imagine accessing a particular page of a book by using a 5-dimensional array. For example, if we dimension an array:

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

then

```
Page(1,7,2,19,130)
```

designates page 130 of the 19th book on the 2nd shelf of the 7th stack on the 1st floor. You can specify words on pages by using a 6-dimensional array. Remember that six dimensions is the maximum, so you cannot specify letters of words. Also, you can dimension more than one array in a single statement by separating the declarations with a comma. For example

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

dimensions all three arrays A, B, and C.

## Problems With Implicit Dimensioning

In any environment, an array must have a dimensioned size. You can pass this size into an environment through a passed parameter list or a COM statement. You can explicitly dimension the array by using the COM, INTEGER, REAL, COMPLEX or ALLOCATE statements. You can also implicitly dimension an array by using a subscripted reference to it in a program statement other than a MAT or a REDIM statement. If you attempt to use an array that does not have a dimensioned size in the current environment in a MAT or REDIM statement, you will get an error. In other words, MAT and REDIM statements cannot be used to implicitly dimension an array.

## Using Array Elements

This section will show you how to assign and extract values from individual elements within an array.

### Assigning an Individual Array Element

Once an array has been dimensioned, the next step is to fill it with useful values. Every element in an array is initially set to zero, but there are a number of different ways you can change the values. The most obvious is to assign a particular value to each element. You do this by specifying the element's subscripts. For example, the statement:

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

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

### Extracting Single Values From Arrays

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

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

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

## Filling Arrays

This section will provide you with three methods for filling an entire array. The topics covered are as follows:

- Assigning Every Element in an Array the Same Value.
- Using the READ Statement 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. For example:

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

assigns the value 10 to every element in array A, regardless of A's size. 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, REAL or COMPLEX. Let's look at an example of assigning a COMPLEX value to every element of a COMPLEX array:

```
MAT C=(CMPLX(1,2))
```

This statements assigns the complex number 1 + 2i to every element of the complex array C.

## Using the READ Statement to Fill an Entire Array

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

```
10 OPTION BASE 1
20 DIM A(3,3)
30 DATA -4,36,2.3,5,89,17,-6,-12,42
40 READ A(*)
50 END
```

The asterisk in line 40 is used to designate the entire array rather than a single element. The system will fill an entire row before going to the next one. The READ/DATA statements are discussed further in chapter 6, "Data Storage and Retrieval."

## Copying Arrays into Other Arrays

Another way to fill an array is to copy the elements from one array into another. Suppose, for example, that you have the two arrays A and B shown below.

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

Note that A is a 3 × 3 array that 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 resulting 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 the system cannot redimension the result array to the proper size, it will return an error.

Automatic redimensioning of an array will not affect the lower bounds, only the upper bounds. Therefore, the BASE values of each dimension of the result array will remain the same. 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 the statement

```
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
```

The reason for this is that the nine elements originally reserved for array A remain available until the program is scratched. Array A now becomes a 2 × 4 matrix. After

```
MAT A=C
```

you could not execute:

```
MAT B=A or MAT B=C
```

since in each of these cases, you are trying to copy a larger array into a smaller one. You 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

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

### Printing an Entire Array

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

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

The semicolon at the end of the statement is equivalent to putting a semicolon between each element. When the elements are displayed they will be separated by a space. The default is to place elements in successive columns.

### Examples of Formatting Arrays for Display

You can use the following subprogram, named "Printmat," to display a two-dimensional INTEGER array:

```
240 SUB Printmat(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 the array you intend to display is a five-by-five two-dimensional array, your results should look similar to this:

$$\begin{bmatrix} 11 & 12 & 13 & 14 & 15 \\ 21 & 22 & 23 & 24 & 25 \\ 31 & 32 & 33 & 34 & 35 \\ 41 & 42 & 43 & 44 & 45 \\ 51 & 52 & 53 & 54 & 55 \end{bmatrix}$$

In order to use the above subprogram with COMPLEX arrays, you only need to change program line 240 to the following:

```
240 SUB Printmat(COMPLEX Array(*))
```

Each element position in the COMPLEX array will have two values in it one being the real part of the complex number and the other being the imaginary part. For example, the following array is a COMPLEX array called Complex_ array:

$$\begin{bmatrix} 3 & 9 & -6 & 1 \\ -1 & 5 & 2 & 4 \end{bmatrix}$$

where the element Complex_array (1,2) contains the real part of the complex number −6 and the imaginary part 1.

## Passing Entire Arrays

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

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

## Copying Subarrays

Topics discussed in this section are as follows:

- Subarray specifier.
- Copying a subarray into an array.
- 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.

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

## Subarray Specifier

A subarray is a subset of an array (an array within an array). To specify a subarray, subscripts are used in parentheses after the array name as follows:

```
Array_name(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 assigned to the subarray_ specifier. Note that "size" refers to the number of elements in the subarray 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)). Before looking at ways you can express a subarray, let's learn a few terms related to the subarray specifier.

1. 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.
2. Subscript expression is an expression which reduces the RANK of the subarray. For example if you wanted to select an element 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 represents the whole range of elements in row 2 of the array.
3. 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: (*,1:1), where * represents all the rows in the array and 1:1 represents only the first column.

Some examples of subarray specifiers are as follows:

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

2. (1:2) a given subscript range which represents the first two elements of a one-dimensional array.
3. (*,−1:2) a default range and subscript range which represents all of the elements in the first four columns of a two-dimensional array.
4. (3,1:2) a subscript expression and subscript range which represent the first two elements in the third row of a two-dimensional array.
5. (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.
6. (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.
7. (1,2,*) two subscript expressions and a default range which represent the entire second row in the first plane of the first-dimension.
8. (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.

**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:

$$\textbf{Des\_array} \begin{bmatrix} 11 & 12 & 13 & 14 & 15 \\ 21 & 22 & 23 & 24 & 25 \\ 31 & 32 & 33 & 34 & 35 \\ 41 & 42 & 43 & 44 & 45 \\ 51 & 52 & 53 & 54 & 55 \end{bmatrix} \quad \textbf{Sor\_array} \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix}$$

You can 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
```

A two-dimensional plane with the following values in it would be the result of executing the above statement.

$$\textbf{Des\_array}\begin{bmatrix} 11 & 12 & 13 & 14 & 15 \\ 21 & 22 & 23 & 24 & 25 \\ 31 & 11 & 12 & 13 & 35 \\ 41 & 21 & 22 & 23 & 45 \\ 51 & 52 & 53 & 54 & 55 \end{bmatrix}$$

**Rules for Copying Subarrays.** This section should help limit the number of syntax and run-time errors you could make when copying subarrays. A previous section entitled "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 and 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 and 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 subscripts 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,

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

- is legal; however,

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

- is not legal, 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

The system automatically redimensions an array during array assignment, 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 REDIMed 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.

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

You can redimension it to a 2 × 4 array by executing the following

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

The new array will look like the figure below:

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

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)
```

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

We could then initialize all elements to 0:

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

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

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

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

results in:

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

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 COMPLEX 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.

## Arrays and Arithmetic Operators

BASIC allows you to multiply, divide, add, and subtract scalars to an array, as well as to add, subtract, multiply, and divide one array to another. It is also possible for you to add all the elements in an array to produce a single result. This section covers a function and operations which allow you to perform these tasks with INTEGER, REAL, and COMPLEX data types.

### Using the MAT Statement

All arithmetic functions involving arrays must be preceded by the MAT keyword. The specified operation is performed on each individual element in the operand array(s) and the results are placed in the result array. The result array must be dimensioned to be at least as large as the current size of the operand array(s). If it is of a different shape than the operand array(s), the system will redimension it. Given the array A that follows, note how these arithmetic functions are performed.

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

To add 3 to each element of array A, you would use the following statement:

```
MAT B= A+(3)
```

The result of the above addition is array B below:

$$B = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

To divide each element of array B above by 2, you would use the following statement:

```
MAT C = B/(2)
```

The result of the above division is array C given below:

$$C = \begin{bmatrix} 2 & 2.5 & 3 \\ 3.5 & 4 & 4.5 \\ 5 & 5.5 & 6 \end{bmatrix}$$

To multiply each element in array C by a scalar expression, you would use a statement similar to the following:

```
MAT C= C*(1+1+1)
```

The above statement multiplied each element in array C by 3 and placed that result in array C as shown below:

$$C = \begin{bmatrix} 6 & 7.5 & 9 \\ 10.5 & 12 & 13.5 \\ 15 & 16.5 & 18 \end{bmatrix}$$

Note that the result array can be the same as the operand array. Also, the scalar must be enclosed in parentheses. In addition to performing arithmetic operations with scalars, you can also add, subtract, divide and multiply two arrays together. Except for multiplication with an asterisk, which is described later, these functions proceed as follows: Corresponding elements of each operand array are processed according to the specified operation, and the result is placed in the result array. The two operand arrays must be exactly the same size though their particular subscript ranges can be different. For multiplication, use a period rather than an asterisk. Using arrays A and B, the statement,

```
MAT D= A+B
```

would give the array:

$$\mathbf{D} = \begin{bmatrix} 5 & 7 & 9 \\ 11 & 13 & 15 \\ 17 & 15 & 21 \end{bmatrix}$$

The statement,

```
MAT B=A.B
```

would give:

$$\mathbf{B} = \begin{bmatrix} 4 & 10 & 18 \\ 28 & 40 & 54 \\ 70 & 88 & 108 \end{bmatrix}$$

Again, the dimensioned size of the result array must be as large as the current size of each operand array. The two operand arrays must be identical in shape and size, but not necessarily in subscript ranges. For instance, A and B could have been dimensioned:

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

## Performing Arithmetic Operations with Complex Arrays

Remember that each of the operations mentioned in the previous section can be performed with complex numbers. The resulting array if it is of type COMPLEX will have both a real and imaginary part in each element location. For example, you may have a two-dimensional complex array that looks like this:

$$\mathbf{Op\_array} = \begin{bmatrix} 2 & 4 & -1 & 5 \\ -6 & 1 & 9 & 3 \end{bmatrix}$$

where the dimension statement is given as follows:

```
COMPLEX Op_array(-1:0,1:2)
```

The element Op_array(−1,1) contains the values:

```
2 4
```

where 2 is the real part of the complex number and 4 is the imaginary part.

If you were to multiply each of the complex values in the above matrix by a scalar value of 2, you would use the following statement:

```
MAT Complex_result= Op_array*(2)
```

The previous statement would produce the following complex array:

$$\textbf{Complex\_result} = \begin{bmatrix} 4 & 8 & -2 & 10 \\ -12 & 2 & 18 & 6 \end{bmatrix}$$

Note that if the resulting array (Complex_result) had been of type REAL or INTEGER, the results in array Complex_result would look like this:

$$\begin{bmatrix} 4 & -2 \\ -12 & 18 \end{bmatrix}$$

This is due to the automatic type conversion made from COMPLEX to REAL or INTEGER. Notice that the imaginary parts of the complex numbers in the array were dropped.

### Summing the Elements in an Array

The statement that returns the sum of all elements in an array, however, works for arrays of any dimension. Given the array A below,

$$\mathbf{A} = \begin{bmatrix} 4 & 2 & -1 \\ 3 & 8 & 16 \\ -5 & 2 & 0 \end{bmatrix}$$

the function, SUM(A) would return 29.

## Boolean Arrays

In addition to the arithmetic operators, you can also use relational operators with arrays. The result is a boolean[*] array, an array composed entirely of 1's and 0's.

Given array B above, suppose you wanted to know how many elements were greater than 50. First you execute the statement,

[*] Strictly speaking, these are not really boolean arrays since the values of the elements are not TRUE and FALSE.

```
MAT F = B>(50)
```

which results in the array:

$$\mathbf{F} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Then you execute the statement,

```
PRINT SUM(F)
```

which causes the computer to display "4" on the current PRINTER IS device.

**Note**

The only comparison operators allowed with COMPLEX data types are: = and < >. The only dyadic operators allowed with COMPLEX data types are: ^, +, −, *, /, < >, and =. The only monadic operators allowed with COMPLEX data types are: +, −, and NOT.

You can also compare two arrays to each other. For example, if you wanted to compare the two arrays below,

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 8 & 7 \\ 1 & 4 & 6 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 1 & 3 & 4 \\ 2 & 0 & 7 \\ 1 & 4 & 4 \end{bmatrix}$$

you could execute the statement:

```
MAT C = A=B
```

By looking at C, you can tell which elements are the same for both A and B.

$$\mathbf{C} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

# 4

# String Manipulation

It is often desirable to store non-numerical information in the computer. You can use any sequence of characters in a string. Quotation marks are used to 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". BASIC allows the dimensioned length of a string to range from 1 to 32,767 characters and the current length (number of characters in the string) to range from zero to the dimensioned length. A string of zero characters is often called a null string or an empty string. The default dimensioned length of a string is 18 characters. The DIM, COM, and ALLOCATE statements are used to 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".`

* A string variable name can consist of up to 15 alpha-numeric characters *plus* the dollar sign at the end. As with numeric variables, the first character must be a capital letter. The rest of the name can consist of *lower-case* letters, numerals, and the underscore character (_), in any combination.

## String Storage

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

- DIM Long$[400] Reserve space for a 400 character string.
- COM Line$[80] Reserve an 80 character common variable.
- ALLOCATE Search$[Length] Dynamic variable allocation.

The maximum length of any string must not exceed 32,767 characters. A string may also be dimensioned to a length less than the default length of 18 characters. 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]
```

The ALLOCATE statement allows dynamic allocation of string storage. When the maximum length of of a string cannot be determined ahead of time, the ALLOCATE statement can be used to reserve enough memory space for the string without wasting space.

```
ALLOCATE Line$[Length]
```

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$(1000)[80]
```

This statement 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 disk as an ASCII type file can be entered into a string array, manipulated, and written back out to the disk.

## 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.

### String Concatenation

You can combine two strings together by using the concatenation operator "&". The following program demonstrates this feature:

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

When you run the program it will print the following:

```
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” | True |
| “6” > “7” | False |
| “2” < “12” | False |
| “long” < = “longer” | True |
| “RE-SAVE” > = “RESAVE” | False |

Any of these relational operators may be used: <, >, < =, > =, =, < >. Testing begins with the first character in the string and proceeds, character by character, until the relationship has been determined. The outcome of a relational test is based on the characters in the strings not on the length of the strings. For example:

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

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

**Note**

When the LEX binary is loaded, the outcome of a string comparison is based on the character’s lexical value rather than the character’s ASCII value. See the LEXICAL ORDER IS statement later in this chapter for more details.

# 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 within the range from 1 to the current length of the string plus 1.* Note that the brackets now indicate the substring 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. For example:

```
"JABBERWOCKY"[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. For example:

```
"JABBERWOCKY"[4;6]
```

Specifies the substring "BERWOC". In the following examples the variable B$ has been assigned to the literal "ENLIGHTENMENT".

| Statement | Output |
|---|---|
| `PRINT B$[1,13]` | `ENLIGHTENMENT` |
| `PRINT B$[1,9]` | `ENLIGHTEN` |
| `PRINT B$[3,7]` | `LIGHT` |
| `PRINT B$[3;7]` | `LIGHTEN` |
| `B$[13,26]` | (error) |
| `PRINT B$[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
```

When you run this program, it will print:

```
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
```

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

# String-Related Functions

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

## String Length

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

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

Prints: 7

## Substring Position

You can determine the position of a substring within a string by the using the POS function. This 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

## String-to-Numeric Conversion

You can use the VAL function to convert a string expression into a numeric value. The number returned by the VAL function will be converted to and from scientific notation when necessary. For example:

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

Prints: 123400

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

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

You can use the NUM function to convert 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

You can use the VAL$ function to convert 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

Several additional string functions are available when the BASIC system has been loaded into the computer.

## String Reverse

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

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

Prints: `snac kcanS`

## String Repeat

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

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

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

### Trimming a String

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

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

Prints: `*1.23*`

TRIM$ is often used to extract fields from data statements or keyboard input.

### 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. Roman Extension characters will be converted according to the current lexical order. See the LEXICAL ORDER IS statement later in this chapter for the case conversion listings.

The following program demonstrates the case conversion functions:

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

## MAT Functions and String Arrays

MAT functions (available with the MAT binary) are commonly used to manipulate data in numeric arrays. However, several of these functions can be used with string arrays. For example, a string array is copied into another string array by the following.

```
MAT Copy$ = Original$
```

Note that only the variable name is necessary. The array specifier “(*)” need not be included when using the MAT statement.

Every element in a string array will be initialized to a constant value by the following statement.

```
MAT Array$ = (Null$)
```

The constant value can be a literal or a string expression and is enclosed in parentheses to distinguish it from being an array name.

A list of items can be sorted very quickly by the MAT SORT statement. Load and run the following program from file MATSORT on your Manual Examples disk.

```
10  ! Program: SORT_LIST
20  DIM List$(1:5)[6]
30  DATA Bread,Milk,Eggs,Bacon,Coffee
40  READ List$(*)
50  !
60  PRINT "original order"
70  PRINT List$(*)
80  !
90  PRINT "ascending order"
100 MAT SORT List$(*)
110 PRINT List$(*)
120 !
130 PRINT "descending order"
140 MAT SORT List$(*) DES
150 PRINT List$(*)
160 END
```

Running this program produces:

```
original order
Bread Milk Eggs Bacon Coffee
ascending order
Bacon Bread Coffee Eggs Milk
descending order
Milk Eggs Coffee Bread Bacon
```

## 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. Each function has two parameters: the string to be converted and the radix. The radix is limited to the values 2, 8, 10, or 16, and represents the numeric base of the string to be converted. The IVAL and IVAL$ functions are restricted to the range of INTEGER variables (−32,768 thru 32,767). The DVAL and DVAL$ functions allow "double length" integers and thus allow larger numbers to be converted (−2,147,483,648 thru 2,147,483,647). IVAL and IVAL$ operate on 16-bit values, while DVAL and DVAL$ operate on 32-bit values. The following statements show valid usage of these functions.

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

## Changing the Lexical Order

The LEXICAL ORDER IS statement lets you change the collating sequence (sorting order) of the character set. Changing the lexical order affects the results of all string relational operators and operations, including the CASE, MAT SEARCH, and MAT SORT statements. In addition to redefining the collating sequence, the case conversion functions, UPC$ and LWC$, are adjusted to reflect the current lexical order. The LEXICAL ORDER IS statement is available if the LEX binary is installed.

You can create lexical orders for special applications. To do this you would create a one-dimensional INTEGER array with at least 257 elements, which specifies the desired lexical order. (Just list the ASCII characters in the desired order.) For example, the statement:

```
LEXICAL ORDER IS Lex_table(*)
```

specifies the lexical order contained in the array named Lex_table. (Refer to the BASIC *Language Reference* for further information.)

The LEXICAL ORDER IS statement provides the following predefined lexical orders: ASCII (American Standard Code for Information Interchange), FRENCH, GERMAN, SPANISH, SWEDISH, and STANDARD. For example:

```
LEXICAL ORDER IS GERMAN
```

selects the correct lexical order for the German language.

```
LEXICAL ORDER IS STANDARD
```

selects the correct lexical order for the language corresponding to the keyboard connected to the system. (The STANDARD lexical order is determined by an internal keyboard jumper, which is set at the factory.)

The computer executes a LEXICAL ORDER IS STANDARD statement when the Advanced Programming Binary is first loaded or after a SCRATCH A is executed. The result will be the correct lexical order for the language on the keyboard. This can be checked by examining the keyboard status register in a program (STATUS 2,8;Language), or by either of the following statements.

```
SYSTEM$("LEXICAL ORDER IS")
SYSTEM$("KEYBOARD LANGUAGE")
```

The following table shows the language indicated by the value returned by the STATUS statement. For example, if the value returned indicates a French keyboard, the STANDARD lexical order is FRENCH. On the other hand, the STANDARD lexical order for the Dutch keyboard is GERMAN.

| Value | Keyboard Language | Lexical Order |
|---|---|---|
| 0 | ASCII | ASCII |
| 1 | FRENCH | FRENCH |
| 2 | GERMAN | GERMAN |
| 3 | SWEDISH | SWEDISH |
| 4 | SPANISH* | SPANISH |
| 5 | KATAKANA | ASCII |
| 6 | CANADIAN ENGLISH | ASCII |
| 7 | UNITED KINGDOM | ASCII |
| 8 | CANADIAN FRENCH | FRENCH |
| 9 | SWISS FRENCH | FRENCH |
| 10 | ITALIAN | FRENCH |
| 11 | BELGIAN | GERMAN |
| 12 | DUTCH | GERMAN |
| 13 | SWISS GERMAN | GERMAN |
| 14 | LATIN† | SPANISH |
| 15 | DANISH | SWEDISH |
| 16 | FINNISH | SWEDISH |
| 17 | NORWEGIAN | SWEDISH |
| 18 | SWISS FRENCH | FRENCH |
| 19 | SWISS GERMAN | GERMAN |

** European Spanish keyboard.*

*† Latin Spanish keyboard.*

The CHR$ function may be used to produce characters not readily available on the keyboard.

5

# Subprograms and User-Defined Functions

One of the most powerful constructs available in any language is the subprogram (a user-defined function is a special form of 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 pressing RUN or executing the RUN command.

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 you to take advantage of the "top-down" method of designing programs.
- The program is much easier to read using 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.
- Finally, libraries of commonly used subprograms can be assembled for widespread use.

## 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 FN) and ending statements (SUBEND or FNEND).

## Naming

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

```
Initialize
Read_dvm
Sort_2_d_arry
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.

## The Difference Between a 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 (a REAL number, a COMPLEX number, or a string).

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

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

Using the capability of defining your own function subprograms, you can essentially extend the language if you need a feature not provided in BASIC.

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

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

## Numeric Functions and String Functions

A function is allowed to return either a REAL or COMPLEX numeric value or a string value. Previously, we saw some examples of functions returning REAL numbers. 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 which is to return a string. This is used both in the definition of the function (the DEF statement) and when the function is invoked. The second difference is that the RETURN statement in the function returns a string instead of a number.

```
200    PRINT FNAscii_to_hex$(A$)
  .
  .
  .
1550   DEF FNAscii_to_hex$(A$)
1560   !  Each ASCII byte consists of two hex
1570   !    digits; pretty formatting indicates 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$)
1700   ALLOCATE Temp$[3*Length]
1710   FOR I=1 TO Length
1720      Hexupper=SHIFT(NUM(A$[I]),4)
1730      Hexlower=BINAND(NUM(A$[I]),15)
1740      Temp$[3*I-2;1]=FNHex$(Hexupper)
1750      Temp$[3*I-1;1]=FNHex$(Hexlower)
1760      Temp$[3*I;1]=" "
1770   NEXT I
1780   RETURN Temp$
1790   FNEND
```

```
1800  DEF FNHex$(INTEGER X)
1810  ! Assume 0<=X<=15
1820  ! Return ASCII representation of the
1830  !   hex digit represented by the four
1840  !   bits of X.
1850  ! If X is between 0 and 9, return
1860  !   "0"..."9"
1870  ! If X > 9, return "A"..."F"
1880  IF X<=9 THEN
1890     RETURN CHR$(48+X)   ! ASCII 48 through 57
1900                         !   represent "0" - "9"
1910  ELSE
1920     RETURN CHR$(55+X)   ! ASCII 65 through 70
1930                         !   represent "A" - "F"
1940  END IF
1950  FNEND
```

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

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

Thus, it is perfectly legal to use expressions in the pass parameter list of a subprogram. (By the way, such expressions may also invoke function subprograms.)

## Calling and Executing a Subprogram

Subprograms are invoked explicitly using the CALL statement, while functions are invoked implicitly just by using the name in an expression, an output list, etc. A nuance of SUB subprograms is that the CALL keyword is optional when invoking a SUB subprogram. The omission of the CALL keyword when invoking a SUB subprogram is left solely to the discretion of the programmer; some will find it more aesthetic to omit CALL, others will prefer its inclusion. There are, however, three instances which require the use of CALL when invoking a subprogram:

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

# Communication

As mentioned earlier, there are two ways for a subprogram to communicate with the main program or with other subprograms: parameter lists, and COM (blank and labeled).

## 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, REAL, or COMPLEX, 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. It is perfectly legal for both the formal and pass parameter lists to be null, or nonexistent.

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

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

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

`A(*)` is a REAL array. Its size is declared by the calling context. Without MAT, there is no way to find the size of the array except through information supplied explicitly by the calling context; hence the parameters Lower and Upper.

`Lower` and `Upper` 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$` is a simple string which presumably could be used to return the status of the voltmeter to the main program. The length of the string is defined by the calling context.

`Errflag` 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.

There are two ways for the calling context to send values to a subprogram: pass by value, and pass by reference. Using pass by value, the calling context supplies a value and nothing more. Using pass by reference, the calling context actually gives the subprogram access to the calling context's value area. The distinction 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 sent using pass by value or pass by reference. That is determined by the calling context's pass parameter list. 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. 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). Let's look at our previous example from the calling program:

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

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

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

`1,400` 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$` is passed by reference here. If it were enclosed in parentheses, it would be passed by value. Notice that if it were passed by value, it would be totally useless as a method for returning the status of the voltmeter to the calling context.

`Errflag` is passed by reference.

## OPTIONAL Parameters

Another important feature of formal parameter lists is the OPTIONAL keyword. Any formal parameter list (the one defining the subprogram) may contain the keyword OPTIONAL somewhere, although it isn't required to. The OPTIONAL keyword indicates that any parameters that follow it are not required in the pass parameter list of a calling context — they are optional. On the other hand, all parameters preceding the OPTIONAL keyword are required. If no OPTIONAL appears in the subprogram's parameter list, then all the parameters must be specified, or an error will be generated. The rules requiring matching of parameter types apply to OPTIONAL parameters as well as to ordinary parameters. There is a standard function called NPAR which can be used inside the subprogram to find out how many pass parameters the calling context actually did use. (NPAR will return 0 if used inside the main program, or if no parameters were passed to a subprogram.)

## COM Blocks

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

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

A blank COM block might look like this:

```
10 OPTION BASE 1
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,Subvalves910,15),@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 to which it needs to have access. 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.

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

- COM survives pre-run. In general, any numeric variable is set to 0, strings are set to the null string, and I/O path names are set to undefined when the program is run, or upon entering a subprogram. This is true of COM the first time the program is run, but after COM block variables are defined, they retain their values until:
  1. SCRATCH A or SCRATCH C is executed.
  2. A statement declaring a COM block is modified by the user.
  3. A new program is brought into memory using the GET or LOAD commands which doesn't match the declaration of a given COM block, or which doesn't declare a given COM block at all.
- COM blocks can be arbitrarily large. One limitation on parameter lists (both pass and formal parameter lists) is that they must fit into a single program line along with the line's number, possibly a label, the invocation or subprogram header, and possibly (in the case of a function) a string or numeric expression. Depending upon the situation, this can impose a restriction on the size of your parameter lists.

  COM blocks can take as many statements as necessary. COM statements can be interwoven with other statements (though this is considered a poor practice). All COM statements within a context which have the same name will be part of the dėfinition 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 pre-run time. Pre-run is caused by pressing RUN, executing a RUN command, executing LOAD or GET from the program, or executing a LOAD or GET from the keyboard and specifying a run line. Thus if you want to create libraries of subprograms which share their own labeled COM blocks, it is wise to collect all the COM declarations together in one subprogram to make it easy to append them to the rest of the program for inclusion at prerun time. (The subprogram need not contain anything but the COM declarations.)

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

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

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

Consider the following COM block definition:

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

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

```
2000 COM /Dvm_stat/ 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$
```

# Context Switching

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

- The calling context has a DATA pointer which points to the next item in the current DATA block which will be used the next time a READ is executed. This pointer is saved away whenever a subprogram is called, and then the DATA pointer is reset to the first DATA statement in the new subprogram context.
- The RETURN stack for any GOSUBs in the current context is saved and set to the empty stack in the new context.
- The system priority of the current context is saved, and the called subprogram inherits this value. Any change to the system priority which takes place within the subprogram (or any of the subprograms which it calls in turn) is purely local, since the system priority is restored to its original value upon subprogram exit.
- Any event-initiated GOTO/GOSUB statements are disabled for the duration of the subprogram. If any of the specified events occur, this will be logged, but no action will be taken. 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 value of OPTION BASE is saved, and the value for the subprogram (0 or 1, explicitly declared or defaulted) is used.
- 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, INTEGER, or COMPLEX) or implicitly (just by using the variable name). The entire value area is initialized as part of the subprogram's prerun. All numeric values are set to zero, all strings are set to the null string, and all I/O path names are set to undefined.

### Subprograms and Softkeys

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

### Subprograms and the RECOVER Statement

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

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

## Calling Subprograms from the Keyboard

Functions and subprograms can be called by using FN and CALL at the keyboard. There are some restrictions:

- Since variables cannot be created by the user from the keyboard (variables can only be defined by the program), it is legal to use only parameters that already exist in the current context.
- Constants may be used in the pass parameter list.
- When calling a SUB subprogram from the keyboard, the CALL keyword must be used.

## Using Subprogram Libraries

If you have a program which is quite large, along with sizable data arrays, you could run out of memory in your computer. But the program you're working on just has to remain one program, and external factors prevent your reducing data array size. What to do? There are several options which address this problem.

If you want to load a specific subprogram from a PROG file, you would use the LOADSUB <subprogram name> FROM statement. If you want to load all the subprograms from a specific PROG file, you would use the LOADSUB ALL FROM statement. And, if you wanted to see which subprograms are still missing or load all those still needed, you would use the LOADSUB FROM command. Note that this is a command, and not a statement. Therefore, LOADSUB FROM cannot be invoked programmatically.

### 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 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, for which you need the PDEV binary, 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.

## Deleting Subprograms Programmatically

The utility of the LOADSUB commands would be greatly reduced if one could not delete subprograms from memory at will. So, there is a way to delete subprograms during execution of a program: DELSUB. If you want to delete only selected ones, you could use a program line like this:

```
100 DELSUB Sort_data,Print_report,FNPoly_solve
```

If you are sure of the positioning of the subprograms in memory, here is a method of deleting whole groups of subprograms:

```
100 DELSUB Print_report TO END
```

You can combine these methods:

```
100 DELSUB Sort_data,Print_report,FNGet_name$ TO END
```

The subprograms to be deleted do not have to be contiguous in memory, nor does the order in which you specify the subprograms in a DELSUB statement have to be the order in which they occur in memory. The computer deletes each subprogram before moving on to the next name.

If there are any comments after an FNEND or SUBEND, but before the next SUB or DEF FN, these will be deleted as well as the rest of the subprogram body.

If the computer attempts to delete a nonexistent subprogram, an error occurs, and the DELSUB statement is terminated. This means that subprograms whose names are listed after the error-causing one will not be deleted.

A subprogram can be deleted only if it is not currently active and if it is not referenced by a currently active ON RECOVER/CALL statement. This means:

1. A subprogram can not delete itself.
2. A subprogram can not delete the subprogram that called it, either directly or indirectly. (Otherwise it wouldn't have anywhere to return to when finished!)

Between the time that a subprogram is entered and the time it is exited, the computer keeps track of an activation record for that subprogram. Thus, if a subprogram calls a subprogram that calls a subprogram, etc., none of the subsequently-called subprograms can delete the original one or any of the ones in between because the system knows from the activation record that control will eventually need to return to the original calling context. A similar situation exists with active event-initiated CALL/RECOVER statements. As long as the possibility of the specified event occurring exists, the system will not let the subprogram be deleted. In essence, the system will not let you execute two mutually-exclusive, contradictory commands simultaneously.

## Editing Subprograms

You can edit subprograms by *inserting*, *deleting*, or *merging* them.

**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. STORE the program.
2. Delete all lines above the point where you want to insert your subprogram (refer to the DEL statement).
3. STORE the remaining segment of the program in a new file.
4. LOAD 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 LOADSUB ALL from the new file created in step 3.

If you have the PDEV binary installed, the job is much easier:

1. Write your new subprogram at the end of the program.
2. Perform a MOVELINES command where:
   a. The Starting Line in the MOVELINES command is the line which you want to immediately follow your new subprogram.
   b. The Ending Line in the MOVELINES command is the line immediately prior to the SUB or DEF FN of the new subprogram.
   c. The Destination Line is any line number greater than the highest line number currently in memory.

In either case there is an optional final step. It is not required that you do a REN to renumber the program at this point, but often it is desirable to close up the void left in the program line numbering which resulted from the block of subprograms being moved to the end of memory.

**Deleting Subprograms.** It is not possible to delete either DEF FN or SUB statements with DEL LINE 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). This can be done either with the DEL command, or with the DELSUB command.

**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.

Once again, with PDEV, your job is greatly simplified:

Execute a MOVELINES command in which you move everything from one subprogram — excluding the SUB/DEF FN and SUBEND/FNEND statements — into the desired position in the other subprogram. If there are any declarative statements in the moved code, you will probably want to move those up next to the declarative statements in the receiving code. Don't forget to go back to the place where the code came from and delete the SUB/DEF FN statement and the SUBEND/FNEND statements.

## SUBEND and FNEND

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

# Recursion

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

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

```
DEF FNFactorial (N)
IF N=0 THEN RETURN 1
RETURN N*FNFactorial(N-1)
FNEND
```

**References.** For further information you may want to refer to one of the following:

1. Wirth, Niklaus, "Program Development by Stepwise Refinement", Communications of the ACM, April 1971, Vol. 14, No. 4, pp. 221-227.
2. Yourdan, Edward, Techniques of Program Structure and Design, (Prentice-Hall, Englewood Cliffs, NJ, 1975).
3. Dahl, Dijkstra, & Hoare, Structured Programming (Academic Press, New York, 1972).

6

# 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 BASIC program*. With this method, *DATA statements* specify data to be stored in the memory area used by 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 the program, *mass storage* files are more appropriate. Files provide means of storing data on mass storage devices. This chapter describes the available *file types*, how to choose a file type, and how to access such files.

## Storing Data in Programs

This section describes a number of ways you can 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 STORE and SAVE). These techniques allow extremely fast access to the data. They provide good use of the computer's memory for storing relatively small amounts of data.

## Storing Data in Variables

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

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

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

## Data Input by the User

You also can assign values to variables at run-time with the INPUT and LINPUT statements as shown in the following examples:

```
100 INPUT "Type in the value of X, please.",Id
```

or

```
200 DISP "Enter the value of X,Y, and Z."
210 LINPUT "",Response$
```

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 data stream shown.

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

*Data stream:*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | A | 50 | BB | 20 | 45 | X | Y | 77 |

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

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

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

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

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

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

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

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

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

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

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

```
  The date is November 26, 1981.
```

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

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

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

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 first 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*.

The sections that follow describe the types of data files, how to choose a file type, and how to access data files. However, you will also need to be familiar with such mass storage concepts as *volumes* and *directories*. Refer to the following sources:

- For detailed information about LIF, DFS,* and SRM directories for the HP BASIC Language Processor, refer to the "File Systems and Mass Storage" chapter in *Installing and Using HP BASIC in the MS-DOS Environment.*
- For detailed information about LIF, HFS, and SRM directories for the HP 9000 Series 300 computers, refer to the "Mass Storage Concepts" and "Using Directories and Files" chapters in *Using the BASIC System.*

## Brief Comparison of File Types

You can store data in ASCII files, BDAT files, and HP-UX or DOS files. This section describes each of these file types, and the advantages and disadvantages of each.

**Note**

*File type* is essentially independent of *volume and directory type*. ASCII and BDAT files can exist in LIF, HFS, DFS, or SRM directories. HP-UX files can exist in LIF, HFS, or SRM directories (for either an HP 9000 Series 200/300 computer or the HP BASIC Language Processor). DOS files can exist only in a language processor DFS directory.

* The HP BASIC Language Processor implements a DOS File System (DFS) that is similar to the Series 300 Hierarchical File System (HFS).

- **ASCII** — used for general text and numeric data storage. ASCII files provide fairly compact storage for string data and are compatible with a wide range of computers. However, ASCII files can be accessed *only serially, not randomly*. They can be written only in *default ASCII format* (no formatting is possible and the data cannot be stored in internal representation). ASCII files containing BASIC program lines can be read with GET and written with SAVE.
- **BDAT** — provide the most compact and flexible data storage mechanism for HP BASIC. BDAT files can be accessed *either serially or randomly*. BDAT files provide greater flexibility in data formats and access methods, as well as faster transfer rates. They are generally more space-efficient than ASCII files (except for string data items). BDAT files allow data to be stored in ASCII format, internal format, or in a custom format (which you can define with IMAGE specifiers). However, for a BDAT file you must know how the data items were written (as INTEGER, REAL, or COMPLEX values, strings, etc.) in order to properly read the data back. Also, BDAT 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. HP-UX files are implemented by HP 9000 Series 300 BASIC systems to provide interchangeability with other systems using the HFS file system (HP-UX systems and the HP 9000 Series 300 Pascal workstations). Like BDAT files, HP-UX files can be accessed either serially or randomly, and they can use ASCII, internal, or custom data representations. HP-UX files containing BASIC program lines can be read with GET and written with RE-SAVE. The HP BASIC Language Processor implements HP-UX files for the LIF, SRM, and HFS file systems. *However, for the DFS file system the language processor implements the DOS file type, rather than HP-UX (see below).*
- **DOS** — similar to HP-UX files, but implemented by the HP BASIC Language Processor DFS binary to provide file compatibility with MS-DOS. The DOS file type can only exist in a DFS directory.

## 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.* You can execute any of these statements as a command from the keyboard, or within a program.

For example, the statements:

```
CREATE ASCII "Text",100
CREATE BDAT "File",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.

For a BDAT file you can also specify a record size. The default is 256 bytes per record. The statement:

```
CREATE BDAT "File",100,128
```

would create a BDAT file with 100 records of 128 bytes each.

Note that you can use CREATE, CREATE ASCII, and CREATE BDAT to create files within LIF volumes, HFS volumes, and DFS volumes, as well as within an SRM system. Each of these statements contains a file specifier which can include a volume and directory specification. If no volume or directory is specified, the file is created in the current volume and directory as determined by the last MASS STORAGE IS statement. For further information about the syntax of these statements, refer to your HP BASIC *Language Reference* manual.

## 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.

* The CREATE statement creates a DOS file for the DFS file system. Otherwise, an HP-UX file is created.

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

Line 400 specifies the *default mass storage device*, which is to be used whenever a mass storage device is *not explicitly specified* during subsequent mass storage operations. The term *mass storage volume specifier* (*msvs*) describes the string expression used to uniquely identify which mass storage volume (or device) is to be used. In this case, ":,700,1"is the msvs.* This msvs specifies a LIF volume on an *external* disk drive at select code 7, address 00, unit number 1.

For the HP BASIC Language Processor you can change line 400 to access the *internal* Vectra PC (or AT-compatible PC) disk drives. For example:

- The statement `MASS STORAGE IS ":DOS,A"` (or `MSI ":DOS,A"`) specifies a *DFS volume* in drive A.
- The statement `MASS STORAGE IS ":,1500,0"` (or `MSI ":,1500,0"`) specifies a *LIF volume* in drive A.

In order to store data in mass storage, a data file must be created (or already exist) on the mass storage medium. In this case, line 440 creates a BDAT file with 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) in a mass storage volume other than the default, the appropriate msvs must be appended to the file name. If that volume has a hierarchical directory format (such as an HFS, DFS, or SRM volume), you may also have to specify a directory path.

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 (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 so that the data can be read from the beginning. (Re-opening the I/O path name @Path_1 would have also reset the file

* The full msvs is ":CS80,700,1". However, this can normally be shortened to ":,700,1". (Most Hewlett-Packard LIF-format disk drives use the CS80 mass storage protocol.)

pointer.)

Notice that the msvs is *not* included with the file name. (The current default mass storage volume (":,700,1") is assumed.)

The subsequent ENTER statements read the data items into variables. With BDAT files (also HP-UX and DOS files), *the data type of each variable must match the data type of each data item.* On the other hand, with ASCII files you can, for example, read INTEGER items into REAL variables without problems.

## I/O Paths and 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, ENTER, and TRANSFER) which move the data to and from the file. I/O path names are also used to transfer data to and from devices.

**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, assign the I/O path name to a file specifier by using an ASSIGN statement. For example, executing the following statement:

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

assigns an I/O path name called @Path1 to the file Example. The file that you open must already exist and must be a data file. If the file does not satisfy one of these requirements, the system will return an error. If you do not use an msvs 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 msvs syntax described earlier. For instance, the statements:

```
ASSIGN @Path2 TO "Example:CS80,700,0"
ASSIGN @Path3 TO "Example:CS80,1500,0"
ASSIGN @Path4 TO "Example:DOS,A"
```

all open I/O paths to the file Example, but on different mass storage volumes. (You must include the protect code if the file has one.) Note that PROTECT is not implemented with DFS.

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

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

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

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

```
ASSIGN @Path2 TO *
```

closes `@Path2`. `@Path2` cannot be used again until it is Re-ASSIGNed. You can Re-ASSIGN a path name to the same file or to a different file.

## A Closer Look at ASCII Files

You have already been introduced to general file I/O techniques in the preceding section. Now let's take a closer look at using ASCII files.

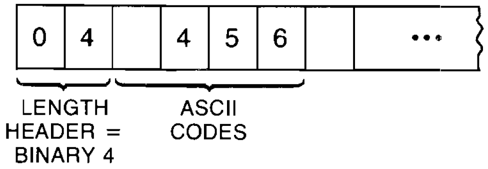
### Example of ASCII File I/O

Storing data in ASCII files requires a few simple steps. The following program segment shows a simple example of placing several items in an ASCII data file. Note that this example is *nearly identical* to the example given in "Overview of File I/O," 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$[100]
  .
  .
  .
390  ! Specify default mass storage.
400  MASS STORAGE IS ":,700,1"
410  !
420  ! Create ASCII data file with ten sectors
430  ! on the default mass storage device.
440  CREATE ASCII "File_2",10
450  !
460  ! Assign (open) an I/O path name to the file.
470  ASSIGN @Path_1 TO "File_2"
480  !
490  ! Write various data items into the file.
500  OUTPUT @Path_1;"Literal"            ! String literal.
510  OUTPUT @Path_1;Real_array1(*)       ! REAL array.
520  OUTPUT @Path_1;255                  ! Single INTEGER.
530  !
540  ! Close the I/O path.
550  ASSIGN @Path_1 TO *
  .
  .
  .
790  ! Open another I/O path to the file (assume same default drive).
800  ASSIGN @F_1 TO "File_2"
810  !
820  ! Read data into another array (same size and type).
830  ENTER @F_1;String_var$              ! Must be same data types.
840  ENTER @F_1;Real_array2(*)
850  ENTER @F_1;Integer_var
860  !
870  ! Close I/O path.
880  ASSIGN @F_1 TO *
```

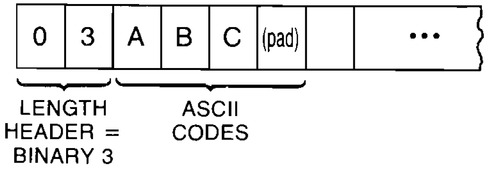
## Data Representations in ASCII Files

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

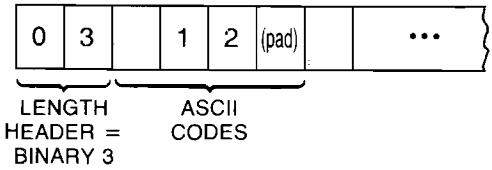


Note that there is a space at the beginning of the data item. This signifies that the number is positive. If a number is negative, a minus sign precedes the number instead.

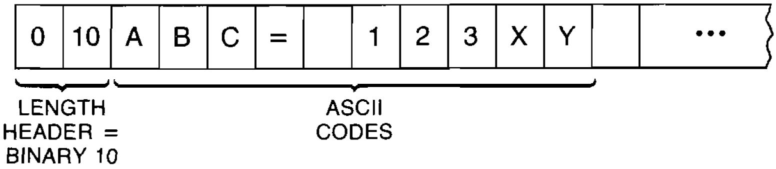
If the length of the data item is an odd number of characters, the system "pads" the item with a space to make it come out even. The string "ABC", for example, would be stored as follows:



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.

*In general, you should only use ASCII files when you want to transport data between machines.* There may be other instances where you will want to use ASCII files, but you should be aware that they cause a noticeable performance degradation compared to BDAT files.

## 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. Thus, *random access of the information in string variables is not allowed* from OUTPUT and ENTER statements; all data must be accessed serially. For example, 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 did, overwriting the data initially output to the variable.

The string variable's length header (two 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).

## Formatted ENTER With ASCII Files

Data is entered from string variables in much the same manner as it is 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, HP-UX, and DOS Files

As mentioned earlier, BDAT, HP-UX, and DOS files are designed for flexibility (random and serial access, choice of data representations), storage-space efficiency, and speed. Let's take a closer look at these file types.

## Data Representations Available

The data representations available are:

- BASIC internal data 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 rest of this section gives more details for each type of representation.

## Random Versus Serial Access

Random access means that you can directly read from and write to any record within the file. 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, HP-UX, and DOS files can be accessed either serially or randomly, 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, HP-UX, and DOS 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 with ASCII characters).

This section describes the details of internal (FORMAT OFF) representations for numeric and string data. For information about ASCII and user-defined formatting, refer to chapter 13, "Outputting and Entering Data," and chapter 14, "Advanced Interfacing Topics."

**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.

Because BDAT files use almost the same format as internal memory, very little interpretation is needed to transfer data from the computer to a BDAT file, or vice versa. BDAT files, therefore, not only save space but also time.

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

| | |
|---|---|
| **INTEGER** | 2 bytes. |
| **REAL** | 8 bytes. |
| **COMPLEX** | 16 bytes (same as two REALs). |
| **String** | 4-byte length header, followed by 1 byte per character (plus 1 pad byte if the string length is an odd number). |

* Actually, the format for BDAT files is slightly different than internal format. Instead of using a 2-byte length header for strings, BDAT files use a 4-byte length header. Otherwise, the two formats are identical, so we refer to both as "internal".

**INTEGER** values are represented in BDAT files which have the FORMAT OFF attribute by using a 16-bit, two's-complement notation, which provides a range of −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.

Here are some examples:

| Binary Representation | Decimal Equivalent |
|---|---|
| 00000000 00010111 | 23 |
| 11111111 11101000 | −24 |
| 10000000 00000000 | −32,768 |
| 01111111 11111111 | 32,767 |
| 11111111 11111111 | −1 |
| 00000000 00000001 | 1 |

**REAL** values are stored in BDAT files by using their internal format: 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 1023 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):

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

The real number "1/3" would be stored as follows:

| Byte | 1 | 2 | 3 | 4 | ... | 8 |
|---|---|---|---|---|---|---|
| Decimal value of character | 63 | 213 | 85 | 85 | ... | 85 |
| Binary value of characters | 00111111 | 11010101 | 01010101 | 01010101 | ... | 01010101 |

mantissa sign    exponent    mantissa

**COMPLEX** values are always stored as two REAL values.

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

- A 4-byte length header contains a value that specifies the length of the string.
- Every character in a string is represented by one byte which contains the character's ASCII code. 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.

If stored as a string value, the number "45" would be:

$$\underbrace{00000000\ 00000000\ 00000000\ 00000010}_{\text{Length = 0002 (binary)}}\ \underbrace{00110100}_{\text{ACSII 52}}\ \underbrace{00110101}_{\text{ASCII 53}}$$

The string "A" would be stored:

$$\underbrace{00000000\ 00000000\ 00000000\ 00000001}_{\text{Length = 0001 (binary)}}\ \underbrace{01000001}_{\text{ASCII 65}}\ \underbrace{00100000}_{\text{ASCII 32}}$$

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. Refer to chapter 13, "Outputting and Entering Data," for further information.

## Data Representations With HP-UX and DOS Files

HP-UX and DOS files are very similar to BDAT files. The *only differences* 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 have. (The system sector is covered in the next section.)

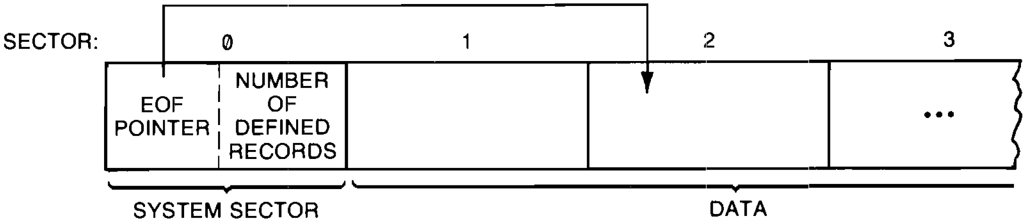
The FORMAT ON representations for HP-UX and DOS files are the same as for devices. FORMAT attributes are covered in detail in chapter 14, "Advanced Interfacing Topics."

**Note**

Throughout this section, you may assume that — unless otherwise noted — the techniques shown will apply to HP-UX and DOS files, as well as BDAT files.

## BDAT File System Sector

On the disk, every BDAT file is preceded by a system sector that contains an End-Of-File pointer and the number of defined records in the file. (See the figure that follows.) All data is placed in succeeding sectors. You cannot directly access the system sector. However, 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 the file. For example, the statement:

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

would create a file called `Example` with 7 defined records, each record being 128 bytes long. 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. Further, the record length is rounded up to the nearest even 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 BDAT "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-access OUTPUT and ENTER statements. It is not important for serial access. The most important consideration in selecting a proper record length is the type of data being stored and the way you want to retrieve it. For optimum performance, the record size should be an *even multiple of the size of the data elements* stored in the record — *2 bytes for integers* and *8 bytes for real numbers*.

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 enter each string in random mode. In other words, select a record length that will hold the longest string and then write each string into its own record. Suppose, for example, that you wanted to OUTPUT the following 5 names into a BDAT file and be able to access each one individually by specifying a record number.

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

The longest name, "Steve Anderson", is 14 characters. To store it in a BDAT file would require 18 bytes, including four bytes for the length header. 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 an I/O path.
120 OUTPUT @File,1;"John Smith"       ! Write names to successive
130 OUTPUT @File,2;"Steve Anderson"   ! records in file.
140 OUTPUT @File,3;"Mary Martin"
150 OUTPUT @File,4;"Bob Jones"
160 OUTPUT @File,5;"Beth Robinson"
```

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

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 10 | J | o | h | n | | S | m | i | t | h | x | x | x | x | 0 | 0 | 0 | 14 | S | t | e | v | e | | A | n | d | e |
| r | s | o | n | 0 | 0 | 0 | 11 | M | a | r | y | | M | a | r | t | i | n | @ | x | x | 0 | 0 | 0 | 9 | B | o | b | | J | o |
| n | e | s | @ | x | x | x | x | 0 | 0 | 0 | 13 | B | e | t | h | | R | o | b | i | n | s | o | n | @ | x | x | x | x | x | x |

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

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

## 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 ahead of the end condition.

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 disk. Any item enclosed in quotes is a string. Numeric variables and expressions are OUTPUT according to their type (8 bytes for REALs and 2 bytes for INTEGERs). 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 semicolon. (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.

**Serial OUTPUT.** Data is written sequentially (serially) to BDAT files (or HP-UX or DOS files) whenever you do not specify a record number in an OUTPUT statement. When data is written serially, each data item is stored immediately after the previous item without any type of separator. 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 next byte. After all of the data items have been OUTPUT, the file pointer points to the first 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 at once. For example, if you have a list of data items that you want to store in a file and you know that you will never want to read any of the items individually, you should write the data serially. The fastest way to write data serially is to place the data in an array and then OUTPUT the entire array at once.

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

**Random OUTPUT.** Random OUTPUT allows you to write to one record at a time. As with serial OUTPUT, there are End-Of-File (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, use either a CONTROL statement to position the EOF in the last record, or 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 return 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 if they are not trapped by ON END. 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 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."*

When reading data into a string variable, it is important to remember that the system will interpret 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.

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, 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 causes problems.

Entering data does not affect the EOF pointers. However, you cannot read data at or beyond the byte marked by the EOF pointers. If you attempt to read past an EOF pointer, the system will return an EOF condition.

In addition to making sure that data types agree, it is also advisable to make sure that access modes agree. If you wrote data serially, you should read it serially. If you wrote it randomly, you should read it randomly. Mixing access modes will often lead to erroneous results unless you are aware of the precise mechanics of the file system.

**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.

In the program below, we OUTPUT five data items serially, and then retrieve the data items with a serial ENTER statement.

```
10 CREATE BDAT "STORAGE",1
20 ASSIGN @Path TO "STORAGE"
30 INTEGER Num,First,Fourth
40 Num=5
50 OUTPUT @Path;Num,"squared"," equals",Num*Num,".",END
60 ASSIGN @Path TO "STORAGE"
70 ENTER @Path;First,Second$,Third$,Fourth,Fifth$
80 PRINT First;Second$;Third$;Fourth;Fifth$
90 END
```

```
5 squared equals 25 .
```

Note that we re-ASSIGNed the I/O path in line 60. This was done to re-position the file pointer to the beginning of the file. If we had omitted this statement, the ENTER would have produced an EOF condition. Note also that the OUTPUT statement includes END, which specifies that the EOF pointer is to be moved to match the file pointer at statement completion. In this case, the END is redundant.

**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 (found in file OUTPUT1 on your Manual Examples disk), data is randomly OUTPUT to five successive records, and then ENTERed into an array in reverse order.

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

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

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

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.

You can define records to be just one 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.

## Trapping EOF and EOR Conditions

An EOF (End-Of-File) condition exists whenever the system attempts to read data at, or beyond, the byte marked by the EOF pointers. The EOR (End-Of-Record) condition will arise if you attempt to randomly read or write beyond the particular record specified. If, for example, you try to randomly OUTPUT a 20-character string into a 10-byte record, an EOR condition will occur. EOF conditions will also result whenever you try to read or write beyond the physical end-of-file.

EOF and EOR conditions can be trapped with an ON END statement. ON END is similar to ON ERROR except that it only traps EOF/EOR conditions and is only applicable to the specified I/O path. If you do not have an ON END statement in a program, the EOF/EOR condition will produce an error that is trappable by the ON ERROR statement. Encountering a logical or physical end of file will produce Error 59. Encountering an end of record in random mode produces Error 60.

You can have any number of ON END statements in a program context. ON END statements that refer to different I/O paths will not interfere with each other, even if the paths go to the same file. If you have more than one ON END to the same I/O path, the system will use whichever one it most recently executes during program flow.

An ON END is canceled by the OFF END statement. OFF END only cancels the ON END branch for the specified I/O path. Re-ASSIGNing an I/O path will also cancel any existing ON END branch for the particular path.

## Extended Access of Directories

The 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, 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 a few examples:

```
CAT ":,700,0"
CAT ":DOS,A"
CAT "\BLP\PROJECTS:DOS,A"
CAT ":,1500,0"
```

All of the preceding statements send catalog listings to the current system printer (specified in the last PRINTER IS statement). The default system printing device is the 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 #701
CAT TO #26
CAT TO #External_prtr
CAT TO #Device_selector
```

The parameter following the # is known as a device selector, and is described in chapter 7, "Using a Printer."

## Cataloging Selected Files

The directory entries of files that begin with certain characters can be obtained by using the secondary keyword SELECT. Suppose that you want to catalog only files beginning with the letters "Proj". The following example shows how this may be accomplished.

```
10 Beginning_chars$="Proj"
20 CAT;SELECT Beginning_chars$
30 END
```

The directory entries of the files beginning with the letters "Proj" are sent to the PRINTER IS device.

SELECT may also be used to get the catalog of an individual file entry by selecting the entire file name, as shown in the following statement:

```
CAT;SELECT "Chap3"
```

## Getting a Count of Selected Files

It is often desirable to determine the total number of files on a disk, or the number that begin with a certain character or group of characters. The COUNT option directs the computer to return the number of selected files in the variable that follows the COUNT keyword.

```
10 CAT;COUNT Files_and_headr
20 END
```

```
10 CAT;SELECT "Data",COUNT Selected_files
20 END
```

The first CAT operation returns a count of all files in the directory (plus the header lines), since not including SELECT defaults to "select all files." The second operation returns a count of the specifically selected files (plus the header lines).

## Skipping Selected Files

If there are many files that begin with the same characters, it is often useful to be able to skip some of the directory entries so that the catalog is not as long. This may be especially useful when using a drive such as an HP 7912, which has the capability of storing more than 10,000 files.

The following statement shows an example of skipping file entries before sending selected entries to the destination.

```
CAT;SELECT "BCD",SKIP 5
```

The first five "selected" files (that begin with the specified characters) are "skipped" (i.e., not sent with the rest of the catalog information).

It is also important to note the order of options in the CAT statement. This order is required when several options are used. If the NO HEADER option is used, it must be the last option in the list, as shown in the following example.

```
CAT;SELECT "BCD",SKIP 5,COUNT Selected_files,NO HEADER
```

7

# Using a Printer

BASIC supports a wide range of printers that can be connected to your computer. This chapter covers the statements commonly used to communicate with external printers.

## Fundamentals

The PRINT statement normally directs text to the screen of the CRT. Text may be re-directed to an external printer by using the PRINTER IS statement. The default system printer is the screen of the CRT. The PRINTER IS statement is used to change the system printer.

Before a printer will print the first character, several steps are required to set up the printer. These steps are fully documented in the appropriate printer installation manual.

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 the following statement.

```
PRINTER IS 1
```

This statement defines 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.

## Primary Addresses

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

Each printer has a set of switches, usually located on the back panel, which set the primary address of the printer. The primary address, determined by the switch settings, is combined with the interface select code to make up the device selector. In the following example, the primary address 01 is appended to the interface select code 7 to produce the device selector 701.

```
PRINTER IS 701
```

This statement tells the computer to use the internal HP-IB interface (select code 7) to communicate with a printer whose switches are set to the primary address 01. If the printer's primary address is set to 11, the device selector would be 711.

## Using Device Selectors

A device selector is used by several different statements. In each of the following, the numeric constant is a device selector.

`PRINTER IS 1` specifies the internal CRT (default).

`PRINTER IS 701` specifies a printer connected to the internal HP-IB interface (interface select code 7) with primary address 01.

`PRINTER IS 9` specifies a printer at interface select code 9.

`PRINTER IS 26` specifies a printer at interface select code 26.*

`CAT TO #701` prints a disk directory on the printer at 701 (interface select code 7, primary address 01).

`PRINTALL IS 707` logs information on a printer at interface select code 7 and primary address 07.

* Select code 26 is used by the HP BASIC Language Processor to access a standard parallel printer at MS-DOS printer port LPT1.

`LIST #701` lists the program currently in memory on the printer at 701 (interface select code 7, primary address 01).

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 #Line_printer
```

The following three-letter mnemonics have pre-assigned values:

| | |
|---|---|
| PRT: | 701 (the system HP-IB printer). |
| KBD: | 2 (the system keyboard). |
| CRT: | 1 (the system display). |

For example, the following statements perform the same action:

```
PRINTER IS PRT
PRINTER IS 701
```

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 to by the I/O path name.

## Using the External Printer

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

## Control Characters

In addition to a "printable" character set, printers usually respond to control characters. These non-printing characters produce a response from the printer. One way to send control characters to the printer is the CHR$ function. Execute the following: *

```
PRINTER IS 701
PRINT CHR$(12)
```

The printer responds with a formfeed. To resume printing on the internal CRT, execute the following:

```
PRINTER IS 1
PRINT "Back to the CRT."
```

Refer to your printer manual for a complete listing of control characters and their effect on your printer. Some control characters will only affect the current line of text.

# 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 values: A = 1.1, B = −22.2, C = 3E + 5, D = 5.1E + 8

```
PRINT A,B,C,D
```

Produces: †

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

* If you want to access a parallel printer (LPT1) from the HP BASIC Language Processor, use select code 26.

† For your convenience, a rule with "tick" marks is shown to indicate the spacing in several examples in this chapter. Of course, this rule isn't actually printed.

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.

```
PRINT A;B;C;D
```

Produces:

```
 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 an array, the comma or semicolon can be used to format the output.

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

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

Produces:

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

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

```
PRINT TAB(25);-1.414
```

Produces:

```
                        -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 the internal CRT.

A more powerful formatting technique employs the ability of the PRINT or OUTPUT 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 an item according to the image.

One way to specify an image is to include it in the PRINT or OUTPUT 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. For example:

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

Produces:

```
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 its 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;A,B,C
```

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.

| 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 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). (Requires IO binary.) |
| R | Print the comma (European radix). (Requires IO binary.) |
| * | Like Z, except that asterisks are printed instead of leading zeros. (Requires IO binary.) |

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

| 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.

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

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

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

Produces:

```
    3.98    5.95   27.50  139.95
----^----^----^----^----^----^----^----^
```

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

## String Image Specifiers

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

| 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"
     Tom         Smith
----^----^----^----^----^----^----^----^

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

PRINT USING """PART NUMBER"",2x,10D";90001234
PART NUMBER    90001234
----^----^----^----^----^----^----^----^
```

## Additional Image Specifiers

Each of the following image specifiers serves a special purpose.

| 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 BASIC Language Reference manual for details on redefining the EOL sequence. |
| / | Send a carriage-return and a linefeed. |
| @ | Send a formfeed. |
| + | Send a carriage-return as the EOL sequence. (Requires IO binary.) |
| – | Send a linefeed as the EOL sequence. (Requires IO binary.) |

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

# Special Considerations

If nothing prints, be sure the printer is ON LINE. When the printer is 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. To clear the error press Clr I/O, check the interface cable and switch settings, then try again.

8

# The BASIC Clock

HP BASIC provides a software clock that is started whenever you boot the BASIC system. You can set and read this BASIC clock to monitor the time of day and date. The BASIC clock is "volatile." That is, it stops keeping time when you turn off the computer. However, if your computer has a battery-backed, non-volatile clock (a "real-time clock"), the BASIC clock will be reset according to that clock each time you boot the BASIC system.*

This chapter describes using the BASIC clock and the related statements and functions. Many of the statements described in this chapter require the CLOCK binary. Refer to the BASIC *Language Reference* manual for the specific requirements of each statement.

## Initial Value, Range, and Accuracy

When you boot the BASIC system, the BASIC software clock is set to an initial value as follows:

- If your computer has a non-volatile real-time clock, the clock value is read from that clock into the BASIC clock.
- If your computer does not have a non-volatile clock, but it is on a Shared Resource Management (SRM) system, the clock value is taken from the SRM system. (This occurs provided the SRM and DCOMM binaries are loaded.)
- If your computer does not have a non-volatile clock and is not on an SRM system, the time is set to 12:00:00 a.m. (midnight), March 1, 1900.

The range of the BASIC clock is March 1, 1900 through August 4, 2079. The clock maintains time to within ±5 seconds per day.

* All HP Vectra PCs, all HP 9000 Series 300 computers, and some HP 9000 Series 200 computers have a battery-backed real-time clock. To set this clock, refer to your computer owner's documentation.

## Reading and Setting the BASIC Clock

You can use the functions and statements described below to read and set the BASIC clock.

**Note** The functions that follow read and set the BASIC software clock. When you set the BASIC clock on the HP BASIC Language Processor, the Real-Time Clock on the PC will also be set if you are using MS-DOS version 3.3 or later.

### Reading the Clock Value

Internally, the BASIC clock maintains the year, month, day, hour, minute, and second as a single real number. This number is scaled to an arbitrary "dawn of time," thus allowing it to also represent the Julian date. The current value of the clock is returned by the TIMEDATE function.

```
PRINT TIMEDATE
```

While the value returned contains all the information necessary to uniquely specify the date and time to the nearest one-hundredth of a second, it needs to be "unpacked" to provide understandable information.

### Determining Date and Time of Day

The following functions are available to extract the date and time of day from TIMEDATE.

The DATE$ function extracts the date from the value of TIMEDATE.

```
PRINT DATE$(TIMEDATE)
```

The TIME$ function returns the time of day.

```
PRINT TIME$(TIMEDATE)
```

## Setting the Clock Value

The SET TIMEDATE statement is used to set the value of the BASIC clock.

```
SET TIMEDATE DATE("2 OCT 1986") + TIME("8:37:30")
```

The time of day can be changed without affecting the date by the SET TIME statement.

```
SET TIME TIME("9:55")
```

Note that an error is reported if you try to set the clock to a value outside the legal range.

## Setting the Time

The time of day is changed by SET TIME X, where X is the number of seconds past midnight. The value of X must be in the range: 0 through 86399.99 seconds. The TIME function will convert twenty-four hour formatted time (HH:MM:SS) into the value needed to set the BASIC clock.

The TIME function converts an ASCII string representing a time of day, in twenty-four hour format, into the number of seconds past midnight. For example:

```
SET TIME TIME("15:30:10")
```

Is equivalent to:

```
SET TIME 55810
```

Either of these statements will set the time of day without changing the date. Use the SET TIMEDATE statement to change the date.

To display the new time, the TIME$ function formats the clock's value (TIMEDATE) into hours, minutes, and seconds.

```
PRINT TIME$(TIMEDATE)
```

Prints: 15:30:16

Even though TIMEDATE returns a value containing both time of day and the Julian date, TIME$ performs an internal modulo 86400 on the value passed to the function and will always return a string in the range: 00:00:00 thru 23:59:59.

## Setting the Date

The date is changed by SET TIMEDATE X, where X is the Julian date multiplied by the number of seconds in a day (86400). The DATE function converts a formatted date (DD MMM YYYY) into the value needed to set the clock. Due to the wide range of values allowed by the DATE function, negative years can be specified, but not when using the function to set the clock.

The following statement will set the clock to the proper date.

```
SET TIMEDATE DATE("1 Jun 1984")
```

When programming without CLOCK, the user-defined function FNDate can be used.

```
SET TIMEDATE FNDate("1 Jun 1984")
```

Both of these statements are equivalent to the following statement.

```
SET TIMEDATE 2.113216992E+11
```

The DATE function converts the accompanying string (or string expression) into the numeric value needed to set the clock. To read the clock, the DATE$ function formats the clock's value as the day, month, and year. For example, the following line will print the date.

```
PRINT DATE$(TIMEDATE)
```

Prints: 1 Jun 1984

## Day of the Week

An advantage of Julian dates is the simplicity of finding the day of the week. TIMEDATE DIV 86400 MOD 7 returns a number which represents the day of the week. Monday is represented by zero (0), and the numbering continues through the week to Sunday which is represented by six (6).

# Branching on Clock Events

Several additional branching statements, available with CLOCK, allow end-of-statement branches to be triggered according to the BASIC clock value.

- ON TIME enables a branch to be taken when the clock reaches a specified time of day.
- ON DELAY enables a branch to be taken after a specified number of seconds has elapsed.
- ON CYCLE enables a recurring branch to be taken with each passage of a specified number of seconds.

The specified time can range from 0.01 thru 167772.15 seconds for the ON CYCLE and ON DELAY statements and 0 thru 86399.99 seconds for ON TIME. The value specified with ON TIME indicates the time of day (in seconds past midnight) for the branch to occur.

Each of these statements has a corresponding statement to cancel the branch (OFF TIME, OFF DELAY, and OFF CYCLE). A statement is also canceled by executing another ON TIME, ON DELAY, or ON CYCLE statement.

All of the statements use the BASIC clock. You should take care to avoid writing programs that could change the clock's setting during execution. Since only one resource is dedicated to each statement, certain restrictions apply to the use of these statements.

## Cycles and Delays

Both the ON CYCLE and ON DELAY statements enable a branch to be taken as soon as the specified number of seconds has elapsed. ON CYCLE remains in effect, re-enabling a branch with each passage of time. For example, load and run the program found in file ONCYCLE on your Manual Examples disk.

```
10  ON CYCLE 1 GOSUB Five ! Print 5 random numbers every second.
20  ON DELAY 6 GOTO Quit ! After 6 seconds quit.
30  !
40  T: DISP TIME$(TIMEDATE) ! Show the time.
50  GOTO T
60  !
70  Five:FOR I=1 TO 5
80    PRINT RND
90  NEXT I
100 PRINT
110 RETURN
120 !
130 Quit:END
```

The program will print five random numbers every second for six seconds and then stop.

Only one ON CYCLE and one ON DELAY statement can be active in a program context. Executing a second ON CYCLE or ON DELAY statement in the same program context deactivates the first ON CYCLE or ON DELAY statement. If a branch is missed due to priority restrictions or execution of a subprogram, the event is logged and the branch will be taken when the restriction is removed or the original context is restored. If an active ON CYCLE or ON DELAY statement gets canceled in an alternate context (subprogram) the branch is restored when execution returns to the defining context. (See Branching Restrictions for more information about this).

## Branching on Time of Day

The ON TIME statement allows you to define and enable a branch to be taken when the clock reaches a specified time of day, where time of day is expressed as seconds past midnight. Using the TIME function simplifies setting an ON TIME statement by allowing a formatted time of day to be used.

For example:

```
ON TIME TIME("11:30") GOTO Lunch
```

Typically, the ON TIME statement is used to cause a branch at a specified time of day. By adding an offset to the current clock value, the ON TIME statement can be used as an interval timer. In the following example (found in file ONDELAY on your Manual Examples disk), both ON DELAY and ON TIME are used as interval timers.

```
10  ON DELAY 5 GOSUB Takeoff ! delay 5 seconds
20  ON TIME (TIMEDATE+10) MOD 86400 GOSUB Touchdown ! delay 10 seconds
30  PRINT "STARTING... ",TIME$(TIMEDATE)
40  Clock:DISP TIME$(TIMEDATE)
50  GOTO Clock
60  !
70  Takeoff:PRINT "TAKEOFF at ",TIME$(TIMEDATE)
80  RETURN
90  Touchdown:PRINT "TOUCHDOWN at ",TIME$(TIMEDATE)
100 RETURN
110 END
```

The starting time is printed when the program is executed. Five seconds later the first subroutine is executed. Ten seconds after the program starts, the second subroutine is executed.

Only one ON TIME statement can be active in a program context. If a branch is missed, due to priority restrictions or execution of a subprogram, the event is logged and the branch will be taken when the restriction is removed or the original context is restored. If an active ON TIME statement gets canceled in an alternate context (subprogram) the branch is restored when execution returns to the defining context. (See Branching Restrictions for more information about this).

Due to the "match an exact time" nature of the ON TIME statement, if the specified time occurs when the statement is temporarily canceled (by an OFF TIME in an alternate context), no branch will be taken when the defining context is restored.

## Priority Restrictions

A priority can be assigned to the branch defined by ON CYCLE, ON DELAY, and ON TIME. For example:

```
ON CYCLE Seconds,Priority GOTO Label
```

If the system priority is higher than the branch priority at the time specified for the branch, the event will be logged but the branch will not be taken until the system priority falls below the branch priority. An example program, found in file PRIORITY on your Manual Examples disk, follows.

```
10  COM Start
20  P=0
30  Up:P=P+1
40  IF P>15 THEN Quit ! Priority from 1 thru 15
50  PRINT
60  PRINT "Priority:";P;
70  Start=TIMEDATE  ! Save the start-time for subprogram.
80  ON CYCLE 1,P RECOVER Up ! New priority every second if not Busy.
90  ON DELAY .5,6 CALL Busy ! DELAY overrides CYCLE until priority
100 ! (P) is greater than 6.
110 W:GOTO W
120 Quit:END
130 !----------------- SUB has priority of 6 --------------------
140 SUB Busy
150 COM Start
160 PRINT "SUB";
170 WHILE I<10
180   IF TIMEDATE>Start+1 THEN ! Has ON CYCLE time been exceeded?
190     PRINT "*";  ! YES (only prints if Priority<7)
200   ELSE
210     PRINT ".";  ! NO
220   END IF
230   I=I+1   ! Loop ten times
240   WAIT .1
250 END WHILE
260 PRINT "DONE";
270 SUBEND
```

Once the priority assigned to the ON CYCLE statement is greater than the priority assigned to the ON DELAY statement (6), the subprogram will be interrupted. After running the program, change line 80 in the above program to the following:

```
80 ON CYCLE 1,P GOTO Up
```

Running the new version of the program will show that GOTO (or GOSUB) will not interrupt a subprogram regardless of the assigned priority. The branch will be logged but not taken until execution returns to the main program. If you write a program that makes extensive use of subprograms and branching statements, use CALL and RECOVER to insure proper operation.

## Branching Restrictions

Certain restrictions apply to the use of ON TIME, ON CYCLE, and ON DELAY because only one resource is dedicated to each statement. Assuming an active branch has been defined in the main program, execution of a subprogram which sets up a new branch will cause the loss of the original time. When the main program context is restored, the original branch will be restored, but at the time defined in the subprogram.

9

# Errors and Debugging

This chapter covers error handling and debugging techniques that you can use to ensure that your program will work correctly.

## Error Handling

Most programs are subject to errors happening at run time. There are three courses of action 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 program "roll over and die" if an error happens.

The last alternative, which may seem frivolous at first glance, is certainly the easiest to implement and 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 every effort to make the program foolproof.

**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 crashes with an error 31. It is far better if you plan for such an occurrence. One method is shown in the following example.

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

## Error Trapping

Despite the programmer's best efforts at screening the user's inputs in order to avoid errors, sometimes an error 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 command sets up a branching condition which will be taken any time a recoverable error is encountered at run time. The branching action taken may be either GOTO, GOSUB, 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 set up, the CALL or RECOVER will be executed any time an error occurs, regardless of subprogram environment.

When an ON ERROR statement is executed, the language system will make sure that the specified line or subprogram exists in memory before the program will proceed. If ON ERROR GOTO/GOSUB/RECOVER are specified, then the line identifier must exist in the current context. If an ON ERROR CALL is given, then the specified subprogram must currently be in memory. In either case, if the system can't find the given line, an error 49 is issued.

If you use either ON ERROR GOSUB or ON ERROR CALL and an error occurs, the specified branch will take place, and when the RETURN or SUBEXIT is executed, then program execution will resume at the line which caused the error, and an attempt will be made to execute the line again.

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

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

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

ERRL is a function which is used to find the line in which the error was encountered. ERRL is a boolean function. The program feeds it a line identifier, and either a 1 or a 0 is returned, depending upon whether or not the specified identifier indicates the line which caused the error. ERRL is a local function, which means it can only be used in the same environment as the line which caused the error. This implies that ERRL cannot be used in conjunction with ON ERROR CALL, and that it can be used with ON ERROR GOTO and ON ERROR GOSUB. ERRL can be used with ON ERROR RECOVER only if the error did not occur in a subprogram which was called by the environment which set up the ON ERROR RECOVER.

The ERRL function will accept either a line number or a line label.

```
1140 DISP ERRL(710)
910 IF ERRL(Compute) THEN Fix_compute
```

**ON ERROR GOSUB.** The ON ERROR GOSUB statement should only be used when you can guarantee that the problem causing the error can be fixed and the line can be re-executed safely. Remember that if the action taken in the error service routine is not sufficient to correct the problem, the program will dive into an infinite loop. Every time an error occurs, a GOSUB will cause a branch to the error service routine which will RETURN execution to the line causing the error.

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.

**ON ERROR GOTO.** The ON ERROR GOTO statement is generally more useful than ON ERROR GOSUB, especially if you are trying to service more than one error condition. The only advantage that ON ERROR GOSUB has over ON ERROR GOTO is that system priority is maintained at the highest possible level until the error subroutine is finished.

By using the ON ERROR GOTO statement, the same 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 ERRL (where it went wrong) functions, proper recovery procedures can be taken.

**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 16 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.

You should only use the ON ERROR CALL statement when you can guarantee that the problem causing the error can be fixed and the line can be re-executed safely. Remember that if the action taken in the error service routine is not sufficient to correct the problem, the program will dive into an infinite loop. Every time an error occurs, a CALL will cause a branch to the error service routine which will return execution to the line causing the error when a SUBEXIT statement is executed.

Remember that an ON...CALL statement can not pass parameters to the specified subprogram, so the only way to communicate between the environment in which the error is declared and the error service routine is through a COM block.

The ERRL function will not work in a different environment than the one in which the ON ERROR statement is declared, so when using an ON ERROR CALL, you should set things up in such a manner that the line number either doesn't matter, or can be guaranteed to always be the same one when the error occurs. This can be accomplished by declaring the ON ERROR immediately before the line in question, and immediately using OFF ERROR after it.

```
5010  ON ERROR CALL Fix_disk
5020  ASSIGN @File TO "Data_file"
5030  OFF ERROR
  .
  .
  .
7020  SUB Fix_disk
7030  SELECT ERRN
7040  CASE 80
7050    DISP "Door open -- shut it and press CONT"
7060    PAUSE
7080  CASE 83
7090    DISP "Write protected -- fix and press CONT"
7100    PAUSE
7120  CASE 85
7130    DISP "Disk not initialized -- fix and press CONT"
7140    PAUSE
7160  CASE 56
7170    DISP "Creating Data_file"
7180    CREATE BDAT "Data_file",20
7190  CASE ELSE
7200    DISP "Unexpected error ";ERRN
7210    PAUSE
7220  SUBEND
```

**ON ERROR RECOVER.** The ON ERROR RECOVER statement sets up an immediate branch to the specified line whenever an error occurs. The line specified must be in the context of the ON...RECOVER statement. ON ERROR RECOVER is global in scope — it is active not only in the environment in which it is defined, but also in any subprograms called by the segment in which it is defined.

If an error is encountered while an ON ERROR RECOVER statement is active, the system will restore the context of the program segment which actually set up the branch, including its system priority, and will resume execution at the given line.

## Program Debugging

The problem of debugging a program is distinct from the issues raised in the "Error Handling" section. The "Error Handling" section is based on the premise that you are satisfied that the program works as it should, and that it then should be made as foolproof as possible. This could be construed as putting the cart before the horse — before you can make a program foolproof, you must get it to run correctly in the first place. One of the key characteristics of a "bug" is that it doesn't necessarily have to cause an error condition to occur — it only has to cause your program to give a wrong answer. This section deals with the methods available to diagnose problems in logic and semantics.

Naturally, the ideal way to debug a program is to write it correctly the first time through. Hopefully, the techniques that have been been discussed in this manual will help you get a little closer to this goal. The practice of writing self-documenting code and designing programs in a top-down fashion should help immensely.

The computer itself has several features which aid in the process of debugging.

### Using Live Keyboard

One of the pleasing characteristics of HP BASIC is that the keyboard is "live" during program execution. That is, you can issue commands to the computer while it is running a program the same way that you issue commands to it while it is idle. For example, you can add two numbers together, examine the catalogue of the disk currently installed in the drive, list the running program to a printer, scroll the CRT alpha buffer up and down, or output a command to a function generator over HP-IB. Practically the only thing you can't do from live keyboard while a program is running is write or modify program lines, or attempt to alter the control structures of the program. (A complete list of illegal keyboard operations is given a little later on.)

By way of illustration, key in the following program, press RUN, and then execute the commands shown underneath the listing.

```
10 FOR I=1 TO 1.E+5
20 NEXT I
30 END
CAT
2+2
SQR(6^2+17.2^2)
PRINT "THE QUICK BROWN FOX"
TIMEDATE
```

This program will take a fair amount of time to complete (about 18 seconds), so to find out how far the program has gone, merely type I and press ENTER. The current value of I will be displayed at the bottom of the screen. If you don't want to wait for the program to go through all one hundred thousand iterations, you can merely change the value of I by executing the command:

```
I=99999
```

Thus, we have seen that live keyboard can be used to examine and/or change the contents of the program's variables.

One aspect of live keyboard you should remember is that the computer will only recognize variables that exist in the current program environment. For example, suppose that we change our example program to call a subprogram inside the loop.

```
10 FOR I=1 TO 1.E+5
15   CALL Dummy
20 NEXT I
30 END
40 SUB Dummy
50 FOR J=1 TO 10
60 NEXT J
70 SUBEND
```

While this program is running and you try and test the variable I from the keyboard, chances are that you will only get a message saying that I doesn't exist in the current context — most of the time will be spent in the subprogram. On the other hand, if you test the value of J, it is highly likely that you will get an answer.

Similarly, operations like ASSIGN and ALLOCATE, which are declarative types of statements, must use variables that are already known to the current environment when they are executed from the keyboard. For example, it is perfectly legal to perform the operation

```
ASSIGN @Dvm TO *
```

from the keyboard, but it is not legal to perform

```
ASSIGN @File TO "DATA"
```

from the keyboard.

Live keyboard operations are allowed to use variables already known by the running program. Live keyboard operations are not allowed to create variables.

Although the GOTO and GOSUB commands are illegal from the keyboard, it is perfectly legal to call subprograms from the keyboard. The only restriction on using SUB and function subprograms from the keyboard is that the parameters that are passed must either be constants or must be variables that exist in the current context.

Here is a list of commands which may not be executed from the keyboard while a program is running, although they may be executed from the keyboard if the computer is idle:

RUN
CONT
EDIT
DEL
SCRATCH
SCRATCH A
SCRATCH C
SCRATCH BIN
GET
LOAD
LOAD BIN

## Stepping

One of the most powerful debugging tools available is the capability of single-stepping a program, one line at a time. This process allows the programmer to examine the values of his variables and the sequence in which the program is running at each statement. This is done with the STEP function.

There are three ways to use STEP:

1. If the program is stopped (i.e., a prerun has to be performed), pressing STEP[*] will cause the system to perform a pre-run on the program, but no program lines will actually be executed. The first line that will be executed will appear in the system message line at the bottom of the screen. Pressing STEP again will cause that line to be executed, and the next line after that to be executed will appear in the message line. If STEP is pressed causing the next line to appear in the display, and a live keyboard operation (such as examining the value of a variable) is performed, the contents of the message line will change. Pressing STEP again will still cause the line to be executed, even though it is no longer visible in the display line. After the statement has completed, the next line will appear.

[*] The exact key to press depends on your keyboard and computer. Refer to your keyboard overlay to find the STEP key.

2. If the program is in an INPUT or LINPUT statement, pressing STEP is sufficient to terminate the operation. Any data entered from the keyboard will be entered into the correct variables, just as though CONTINUE or ENTER had been pressed, but program execution will be PAUSEd, and the statement immediately following the INPUT or LINPUT will appear in the system message line.
3. If the program is in a PAUSEd state, pressing STEP will cause the next line to be executed. The program counter will not be reset, nor will a prerun be performed. Again, the next line to be executed will appear in the system message line after the last one has been completed. A paused state is indicated by a dash in the run light in the lower right-hand corner of the screen.

Type in the following example and execute it by pressing STEP repeatedly.

```
10  DIM A(1:5)
20  ! This is an example
30  S=0
40  FOR I=1 TO 5
50    INPUT "Enter a number",A(I)
60    S=S+A(I)
70  NEXT I
80  PRINT S
90  PRINT A(*);
100 END
```

Notice that STEP caused every statement to appear in the system message line, one at a time, even those statements that are not really executed, like DIM and comments.

## Tracing

The process of single-stepping, wonderful though it is, can be quite slow, especially if the programmer has little or no idea which part of his program is causing the bug. An alternative way of examining variable changes and program flow is available in the form of the TRACE ALL statement.

**TRACE ALL.** When the TRACE ALL command is executed, it causes the system to issue a message prior to executing every line (this shows the order in which the statements were executed), and if the statement caused any variables to change value, a message telling the variables involved and their new values is also issued. The messages are issued to the system message line, and the most useful way to use the TRACE ALL feature is to turn PRINT ALL on. Press PRINT ALL (as shown by your keyboard overlay). A message ("Printall on" or "Printall off") will appear on the screen. The printall mode will cause all information from the DISP line, the keyboard input line, and the system message line to be logged on the PRINTALL IS device.)

Press PRINT ALL to turn on PRINT ALL. Load and run the following example (found in file TRACEALL on your Manual Examples disk) to see how TRACE ALL works:

```
10  TRACE ALL
20  FOR I=1 TO 10
30    PRINT I;
40    IF I MOD 2 THEN
50      PRINT " is odd."
60    ELSE
70      PRINT " is even."
80    END IF
90  NEXT I
100 END
```

There are two optional parameters that can be used with TRACE ALL. Both parameters are line identifiers (line numbers or line labels). The first parameter tells the system when to start tracing, and the second one (if it's specified) tells the system when to stop tracing.

It is usually more useful to use the TRACE ALL command from the keyboard rather than from the program because a program modification is not necessary if you want to trace a different part of the program. All that's necessary is to type in a new TRACE ALL command from the keyboard to override the old one. For example, to trace a loop from lines 30 to 40, type in TRACE ALL 30,40 from the keyboard.

The program will begin tracing at line 30, and keep on tracing until it's ready to execute line 40, at which time it will terminate the trace messages and will continue executing the program normally.

If the TRACE ALL statement uses a line label instead of a line number, be aware of what happens if you have more than one occurrence of a given line label in your program. For instance, it is perfectly legal to have the same line label in two or more different program environments — line labels are local to subprograms and branching operations addressing a given line label are treated separately in different subprograms. However, when a TRACE ALL using a line label is executed, the first line label in memory is the one that gets used, regardless of the environment the program was in when the TRACE ALL statement was executed. If two line identifiers are used, their location with respect to each other does not matter. Tracing will start when the line specified first is encountered, and it will stop when (or if) the second line is encountered.

## PRINTALL IS

The PRINTALL IS command is useful for switching the tracing messages between the CRT and a hardcopy printer. (Again, to get any record at all of the trace messages, PRINT ALL must be on.) To cause the trace messages to be logged on the CRT, execute PRINTALL IS CRT. (The CRT is the default PRINTALL IS device that the system assumes when it wakes up.) To cause the messages to be logged on a printer, merely change the select code to the appropriate value.*

## TRACE PAUSE

The TRACE PAUSE command can be used to set a "break point" in the program. The program will execute at a reduced speed until the specified line is reached, at which time the program will pause, and the specified line will be shown in the display line, indicating that the program will execute it when execution is resumed. Execution may be resumed by pressing CONTINUE, or by executing CONTINUE from the keyboard (the specified line identifier must be located in the current environment).

By executing the command TRACE PAUSE Printout from the keyboard, the following program (found in file TRPAUSE on your Manual Examples disk) will pause every time it reaches line 60.

```
10  DIM A(1:10)
20  FOR I=1 TO 10
30    GOSUB Printout
40  NEXT I
50  STOP
60  Printout: !
70  FOR J=1 TO 10
80    PRINT A(J);",";
90  NEXT J
100 PRINT
110 RETURN
120 END
```

* For example, use PRINTALL IS 701 for an HP-IB system printer. Use PRINTALL IS 26 for a printer at LPT1 on a Vectra PC with the HP BASIC Language Processor installed.

Try the following ways of continuing execution:

- Press STEP.
- Press CONTINUE.
- Execute CONT 110 ENTER.

As with TRACE ALL, a new TRACE PAUSE statement overrides a previous one. The same rules are applied when a line label is used in a TRACE PAUSE statement as are applied to the TRACE ALL statement — the first line in memory having that label is used.

**TRACE OFF.** TRACE OFF cancels the effects of any active TRACE ALL or TRACE PAUSE statements. The status of Print All and the PRINTALL IS device will be unchanged.

TRACE OFF may be executed either from the program, or from the keyboard.

**The CLR I/O Key.** The CLEAR I/O key suspends any active I/O operation and pauses the program in such a way that the suspended statement will restart once CONTINUE or STEP is pressed. This is useful for operations which appear to "hang" the machine, such as printing to a printer which isn't turned on.

Most devices will not respond to ENTER requests unless they have first been instructed to respond. If improper values are sent to a device, it may refuse to respond. Therefore, CLEAR I/O can help in debugging these situations.

Here are the operations that can be suspended with CLEAR I/O.

PRINT
LIST
CAT
OUTPUT
HP-IB commands
SEND
PRINTALL outputs
ENTER
INPUT
DUMP ALPHA
ASSIGN
PURGE
CREATE
DUMP GRAPHICS
External plotter commands

# Part II: Graphics Techniques

Chapters 10 and 11 cover programming techniques that you can use to present information graphically on the CRT, or on an external printer or plotter.

10

# Creating Graphics

Your HP BASIC system is an excellent tool for creating graphics. You can use graphics to display data or to amplify your reports with artistic creations. This chapter covers the basics of computer graphics design, and gives techniques for using graphics effectively. You may want to try the examples in this chapter on your computer as you read. For further information about the graphics statements, refer to your BASIC *Language Reference* manual. Most of the techniques in this chapter require the GRAPH and GRAPHX binaries.

## Introduction to Your BASIC Graphics System

Let's begin with some background information about your graphics system.

### The CRT Display

In this chapter you will be using your CRT monitor as a plotter. It is easiest to develop and edit graphics programs using the CRT. Then, with minor changes, the image can be reproduced on an external plotter (refer to chapter 11).

**Combined Alpha and Graphics Mode.** The CRT display can present both alphanumeric data (the alpha display) and graphics (the graphics display). HP 9000 Series 300 computers and the HP 82300C BASIC Language Processor (software version C.00.00 or higher) normally present *combined* alpha and graphics information on the CRT display screen. That is, the default mode is combined alpha and graphics. (HP 9000 Series 200 computers and the earlier versions of the language processor software provide only *separate* alpha and graphics mode, which is described in the next section.)

The GRAPHICS ON/OFF and ALPHA ON/OFF statements have no effect in combined mode. This means that you cannot separately turn the graphics and alpha displays on and off.

For most applications, combined mode is adequate. However, if you want to be able to control the alpha and graphics displays separately, you can use the *separate* mode with most color monitors.

**Separate Alpha and Graphics Mode.** To enter *separate* alpha/graphics mode, execute the following HP BASIC statement:

```
SEPARATE ALPHA FROM GRAPHICS
```

If you are using a color monitor that supports separate mode, it will now have two separate displays, an alpha display and a graphics display, which can be displayed either individually or simultaneously. The alpha display outputs alphanumeric characters such as error messages or commands, while the graphics display, of course, outputs graphics. *If you execute this statement with a monitor that doesn't support separate mode, you will receive an error message and the display will remain in combined mode.*

In separate mode, the alpha display is controlled with the following statements:

```
ALPHA ON
```

```
ALPHA OFF
```

The graphics display is controlled with the following statements:

```
GRAPHICS ON
```

```
GRAPHICS OFF
```

You can also use the "Alpha" and "Graphics" keys to turn the two displays on and off.

When you enter separate mode, the alpha display and the graphics display are both on. During execution of a graphics program, the alpha display may be turned off, but execution of a command or sending output to the alpha display turns the alpha display on and leaves it on.

To return to combined mode, execute the following HP BASIC statement:

```
MERGE ALPHA WITH GRAPHICS
```

## Initializing and Clearing the Displays

To bring the graphics system to a known starting point, execute the command:

```
GINIT
```

This initializes the graphics system by resetting all the attributes, viewing operations, plotters, and other system variables. It is a good idea to include a GINIT statement at the beginning of any graphics program.

Once GINIT is executed, you want to make it easy to see the graphics display. One problem encountered is that data in the alpha display covers the graphics display. If you are in separate mode (refer to the last section), you can solve this problem by executing:

```
ALPHA OFF
```

The alpha data is preserved in memory, waiting for the next "ALPHA ON" statement.

If you are in separate mode, you can delete all data in the alpha screen with the statement:

```
CLEAR SCREEN
```

or by pressing the "Clear Screen" key. In separate mode, this doesn't affect the graphics display. *However, in combined mode, "CLEAR SCREEN" clears the entire screen, including graphics.* You can also clear the alpha display with a formfeed character, CHR$(12).

To clear the graphics display without affecting the alpha display, execute:

```
GCLEAR
```

*This works in both separate mode and combined mode.* In separate mode, GCLEAR clears the graphics display with no effect on the alpha display. In combined mode, the alpha display is "refreshed" from memory once the graphics display is cleared. In either mode, any graphics information is lost, so be sure that's what you want to do.

If you execute GINIT, any subsequent graphics statement causes any previous graphics to be cleared from the display before the new graphics are output.

## The Current Position

When dealing with graphics output, this manual uses the concept of a *current position*. It is the point relative to which graphics are currently output. Usually you can think of this as the pen's current location or the location at which graphics can be currently output.

The current position is not always where the physical pen is located. For example, if you instruct the pen to move to a point outside the edge of the plotter, the physical pen only moves to the edge, but the current position is updated to the point specified.

Although the current position is referred to when discussing where graphics are output, the concept of a pen is still important. Knowledge of what the pen type is, and whether it is "up" or "down," is needed. Thus the pen, on a CRT, is defined as the effect which gives the appearance of an invisible pen creating lines on the display. On a paper plotter, it is the control arm that holds the ink pens.

## The XY Plane

Graphics primitives are output on an imaginary plane known as the XY plane; X is a horizontal axis and Y is a vertical axis on this plane. Any two-dimensional images which you create are assigned a position on this plane using XY coordinates. Obviously, this plane cannot be infinite in size because your system can only process numbers up to a certain size. BASIC graphics uses data of type SHORT; therefore, the largest absolute value of x or y that can be input is approximately $3.4 \times 10^{38}$. That is, you can't plot any coordinates greater than this value.

Think of the display as a window to this plane. You can look at the whole coordinate system or a very small part of it at any time through this window. When you turn on the computer, the lower left-hand corner of the display is (0,0). A typical value for the upper right-hand corner is about (133,100). The exact coordinates of the upper right-hand corner are display dependent. In general, these coordinates are (RATIO*100,100). The RATIO statement returns the ratio of the x-axis hard clip limit to the y-axis hard clip limit for the current PLOTTER IS device.

Enter and run the following program:

```
10 GINIT
20 GRAPHICS ON
30 FRAME
40 WAIT 5
50 END
```

**Note**

In combined alpha and graphics mode (the default at power on), the GRAPHICS ON (line 20) and WAIT (line 40) statements aren't needed because the graphics display is always on. However, in separate mode, the WAIT statement allows time for you to view the display. Many of the examples in this chapter contain WAIT statements for this purpose. If you are using separate mode, you can recall the graphics screen after the program ends by pressing the "Graphics" key. You can then return to the alpha mode by pressing any alpha key.

A frame outlined in white will appear on the display. This frame surrounds the entire plotting surface, and is smaller than the CRT screen. Any coordinates to which you can actually plot are within this frame. You can execute graphics output statements that are beyond the edge of the display, but no primitives are output. Placing a frame around the usable plotting area can help when composing a picture.

**Finding the Current Position.** Besides knowing the plotting area that you have to work with, you need to know where the current position is. This is the point on the display relative to which subsequent graphics are positioned. To find it, use the WHERE command. Enter and execute the following program:

```
10 WHERE X,Y
20 PRINT "X =";X,"Y =";Y
30 END
```

X returns the x coordinate of the current location and Y returns the y coordinate. Right now, X and Y should equal 0 because whenever GINIT is executed, the current position is (0,0).

**Changing the Current Position.** The next step is to place the current position where you want to start drawing. To do this, use the MOVE statement. Execute the command:

```
MOVE 50,50
```

Although nothing looks different on the display, you moved the current position to the point (50,50). Use WHERE to see that the current position has changed.

You can use any expression in the range of SHORT values in MOVE. (In fact, almost all the graphics statements can work with expressions in the range of SHORT values.) After a GINIT, the resolution of the CRT is such that coordinates have two significant decimal places. Other plotters have different resolutions; you have to experiment to determine these. You may want to displace the current position by a specific amount. In this case, instead of working out the coordinates of the new position, you can specify an incremental movement. You do this with the IMOVE (Incremental MOVE) command. Execute:

```
IMOVE 10,5
```

You've moved the current position by 10 units along the X axis and 5 units along the Y axis to the point (60,55). Again, WHERE can confirm this.

**Digitizing the Current Position.** Often you can see where you would like to move the current position, but you can't tell what the exact coordinates of the point are. To determine the coordinates of a location on the display, use digitizing.

Here is how to do this.

- Put a crosshair on the display.
- Move the crosshair to the point on the display that you want to be the current position by using the arrow keys of the keyboard.
- Tell the system to return the coordinates of the crosshair's position.

You can then use these coordinates to move the current position to that point. Execute the commands:

```
TRACK CRT IS ON
DIGITIZE X,Y
```

The left and bottom edges of the display have a bright white line. The TRACK. . .IS ON command sets a full-screen crosshair at the point (0,0). TRACK CRT IS ON tells the system that you want to "track" or mimic the keyboard arrow keys on the internal CRT with a crosshair.

Use the arrow keys on the keyboard or the optional mouse to move the crosshair to the point you wish to digitize. Digitize tells the system that you want to digitize or record the arrow key's position. At this point, you can still move the crosshair around if you wish. DIGITIZE doesn't store the coordinates until the next time ENTER is pressed.

When you have the crosshair positioned at the desired point, press ENTER again to get the system to actually digitize the point and place the coordinate values in the variables X and Y. X has the x coordinate and Y has the y coordinate. When the system is waiting for you to press ENTER to digitize a point, the run light looks like an asterisk.

You can move to the point you've just found by executing the command:

```
MOVE X,Y
```

This form of digitizing only works for the internal CRT and keyboard.

The crosshair does not disappear after you're done digitizing. You can still move it around and digitize another point or move the crosshair out of the way.

If your display is generated by a program, you can turn the crosshair off with:

```
TRACK CRT IS OFF
```

then execute:

```
GCLEAR
```

and then regenerate the display by running the program again without the TRACK CRT IS ON statement.

## Graphics Fundamentals

Now that you are familiar with the BASIC graphics system, let's try some techniques for creating graphics.

### Drawing Lines

At this point, you should be able to move the current position to any point. This section explains how to draw lines on the CRT.

Execute these commands:

```
GINIT
```

```
MOVE 50,50
```

```
DRAW 60,60
```

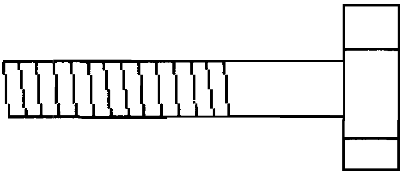
You have a line on the display from the point (50,50) to the point (60,60). You can draw any line by first moving the current position to the starting point and then drawing to the end point. In addition, the current position is updated to the point (60,60). Use WHERE to see this.

If you want to draw a line of a certain length but you don't want to figure out the coordinates of the end point, use the IDRAW (Incremental DRAW) command. Execute:

```
IDRAW 10,10
```

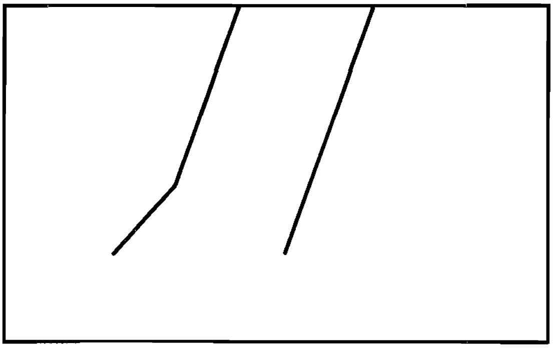
This command draws a line to a point 10 units along the X axis and 10 units along the Y axis from the current position. It also updates the current position by the specified increments.

See if you can recreate the following picture using 21 statements. You may use any of the statements presented up to this point.



Load and run the file "BOLT" from the Manual Examples disk. You can list the file on your CRT or printer to compare your version.

As mentioned before, you can plot to points beyond the edge of the display, but they do not appear. If part of the line is within the display area, that part is output. As an example, load and execute the file "BIGLINES" from the Manual Examples disk. You should see the figure that follows.



All the graphics output is handled in a similar manner. Those points within the display area are output. Those points outside the display area are not shown, but the current position is updated.

DRAW, IDRAW, IMOVE and MOVE have an additional effect beyond moving the current position and drawing a line. They also determine whether the physical pen is up or down (that is, touching the plotting surface). For a CRT, when the pen is "down," a dot appears. When the pen is "up," no dot is created.

One way to control whether the pen is up or down on a plotter is to use IMOVE AND IDRAW.

```
IMOVE 0,0
```

lifts the pen, but does not change the current position.

```
IDRAW 0,0
```

lowers the pen, but does not change the current position.

Another way to raise the pen is by executing the PENUP statement.

```
PENUP
```

The WHERE statement has an additional parameter with which you can determine the status of the pen. It is a string variable whose contents signals whether the pen is up or down and whether it is within the display area. The string also returns other information that is explained later. For example, execute the following program.

```
10 WHERE X,Y,STATUS$
20 PRINT "X =";X,"Y =";Y,"STATUS =";STATUS$
30 END
```

STATUS$ is a string which looks like:

```
1,2
```

The first digit describes the pen's vertical position (0 = up, 1 = down). The second digit describes the pen's horizontal position within the display area (0 = outside of the display area, 1 or 2 = inside the display area).

When you use a paper plotter make sure that the pen is lifted if it is going to rest in one spot for very long; otherwise, an ink blot occurs.

## Scaling

Some graphics may not show much information. There may not be enough variation in the data as presented. For example, load and run the file "SCALE" from the Examples disk.

Probably the first reaction you had when looking at the plot was "That doesn't show me anything...". That's true; it doesn't show much information. There are two reasons for this. The first is that there is not enough variation in the curve; it's too straight to show anything. The second is that it is not centered.

Both of these problems can be remedied by scaling. In this context, scaling could be defined as "defining the values the computer considers to be at the edges of the plotting surface." By definition, the left edge is the smaller X, the right edge is the larger X, the bottom is the smaller Y, and the top is the larger Y. Thus, any point you plot that falls into these ranges will be visible.

There are two statements available to define your own values for the edges of the plotting surface . The first one we'll deal with is SHOW, which forces X and Y units to be equal. Since the X and Y units are identical, the SHOW statement centers the specified area in the plotting area. This is called isotropic scaling, and it is often desirable. For example, when drawing a map, you will probably want one mile in the east-west direction to be the same size as a mile in the north-south direction. Here is an example of SHOW:

```
SHOW 0,100,16,18
```

This causes the plotting area to be defined such that there is a rectangle in that plotting area whose minimum X is 0, maximum X is 100, minimum Y is 16, and maximum Y is 18, using isotropic units. As mentioned above, isotropic means that one unit in the X direction is equal to one unit in the Y direction. Hence, if the plotting area were square, the above SHOW statement would define the minimum X to be 0, maximum X to be 100, minimum Y to be −33 (not 16) and maximum Y to be

67 (not 18). The reason for this is that  allowing whatever extra room it needs to insure that that rectangle is completely contained in the plotting area. There will be extra room in either the X or Y direction, but not both.

Since you (the user) were defining unit sizes with the SHOW statement, you were working with User-Defined Units (UDUs). Both the SHOW statement and the WINDOW statement (covered next) specify user-defined units. Load and run the file “SCALE2”.

As you can see, the SHOW statement takes care of centering the curve on the screen, but since the range of X values is so much larger than the range of Y values (0 to 100 versus 15 to 19), it still does not give us enough resolution to see what the data is doing. Isotropic scaling is desirable in many cases.  In many other cases, however, it is not. If this example shows the graph of the voltage from a sensor versus time, it makes no sense to force seconds to be just as “long” as volts. Since the data types are not equal, it would be better to use unequal, or anisotropic, scaling. You can do this with the other scaling statement: WINDOW. This will not force X units to be equal to Y units. Now load and run the file “SCALE3”.

This plot looks much better than the last one; you can easily see variations in the data. To test how the Y axis range 15–19 affects relative variations in the data, list the program  in file “SCALE3” and change line 30 to WINDOW 0,100,30,50 and change line 50 to PLOT X,RND + 40. Run the program again and note that the line is less ragged.

There is still one problem, though. You can see relative variations in the data, but what are the units being used? That is, is the height of the curve signifying differences of microvolts, millivolts, megavolts, dozens of volts, or what? And you probably wouldn’t want the text (explaining units, etc.) to be written in the same area that the curve is in, as it could obstruct part of the curve. Therefore, you need to be able to specify a subset of the screen for plotting the curve, and put explanatory notes outside this area. The next section tells you how to do this.

## Defining a Viewport

A viewport is a subset of the plotting area. This is called the soft clip area, and it is delimited by the soft clip limits. Clip, because any line segments which attempt to go outside these limits are cut off at the edge of the subarea. Soft, because you can override these limits by turning off the clipping with the CLIP OFF statement (more about this later). There are hard clip limits also, and these are defined to be the absolute limits of the plotting area. Under no circumstances can a line be drawn outside of these limits. There is no way to override the hard clip limits, as you could with soft clip limits.

**GDUs and UDUs.** There are two types of units used to define viewport limits. These are UDUs (User-Defined Units) and GDUs (Graphics Display Units). In order for viewports to be predictable, they must always be specified in the same units. Since UDUs are subject to change, you should use GDUs when specifying the limits of a VIEWPORT statement. GDUs are fixed for the CRT, so a viewport is associated with the screen, rather than the graphical model used in your program.

The lower left of the plotting area is always 0,0. The length of a GDU is defined as "One percent of the shorter edge (the Y axis of a CRT) of the plotting area." Unless you specify otherwise, the screen (but not necessarily an external plotter) is considered to have the following expanse:

- In the Y axis (the shorter side): 0 through 100 GDUs.
- In the X axis: 0 through RATIO*100 GDUs.

The RATIO statement returns the ratio of the x-axis hard clip limit to the y-axis hard clip limit for the current PLOTTER IS device. For a typical CRT monitor this ratio is in the range 1.25 to 1.35. For example, for a VGA monitor used with an HP Vectra PC and HP BASIC Language Processor, RATIO returns a value of approximately 1.3340. Thus, the X axis has a range of 0 through about 133.40 GDUs. Here are the approximate ranges for several monitors:

- VGA or EGA monitor (Vectra with BASIC Language Processor): 0 through 133.40 GDUs.
- HP Multimode Monitor (Vectra with BASIC Language Processor): 0 through 128.06 GDUs.
- HP 98542A and 98543A monitors: 0 through 128.07 GDUs.
- HP 98544, 98545, 98546, 98547, and 98700 monitors: 0 through 133.38 GDUs.
- HP 9000, Models 216 and 226: 0 through 133.44 GDUs.
- HP 9000, Models 236 and 236C: 0 through 131.36 GDUs.

To find the ratio for your display, type `RATIO` and press Enter (refer to your BASIC *Language Reference* manual for further information).

**Specifying the Viewport.** The VIEWPORT statement defines the extent of the soft clip limits in GDUs. It specifies a subarea of the plotting surface which acts just like the entire plotting surface, except that you can draw outside the subarea if you turn off clipping. Load and list file "SCALE4" from the Manual Examples disk. The VIEWPORT statement in this program specifies that the lower left-hand corner of the soft clip area is at 10,15 and the upper right-hand corner is at 120,90. This is the area which the WINDOW statement affects. Also note line 40; the FRAME statement. This draws a box around the current soft clip limits. It is used in this example so you can see the area specified by the VIEWPORT statement. Now run the program to see the result.

**Labelling a Plot.** With the inclusion of the VIEWPORT statement, you have enough room to include labels on the plot. Typically, in a plot like this, there is a title for the whole plot centered at the top, a Y-axis title on the left edge, and an X-axis title at the bottom.

You can use the LABEL statement to write text onto the graphics screen. You can position the label by using a MOVE or PLOT statement to get to the point at which you want the label to be placed. It is the lower left corner of the label which ends up at the point to which you moved. In other words, you move to the position on the screen at which you want the lower left corner of the text to be placed.

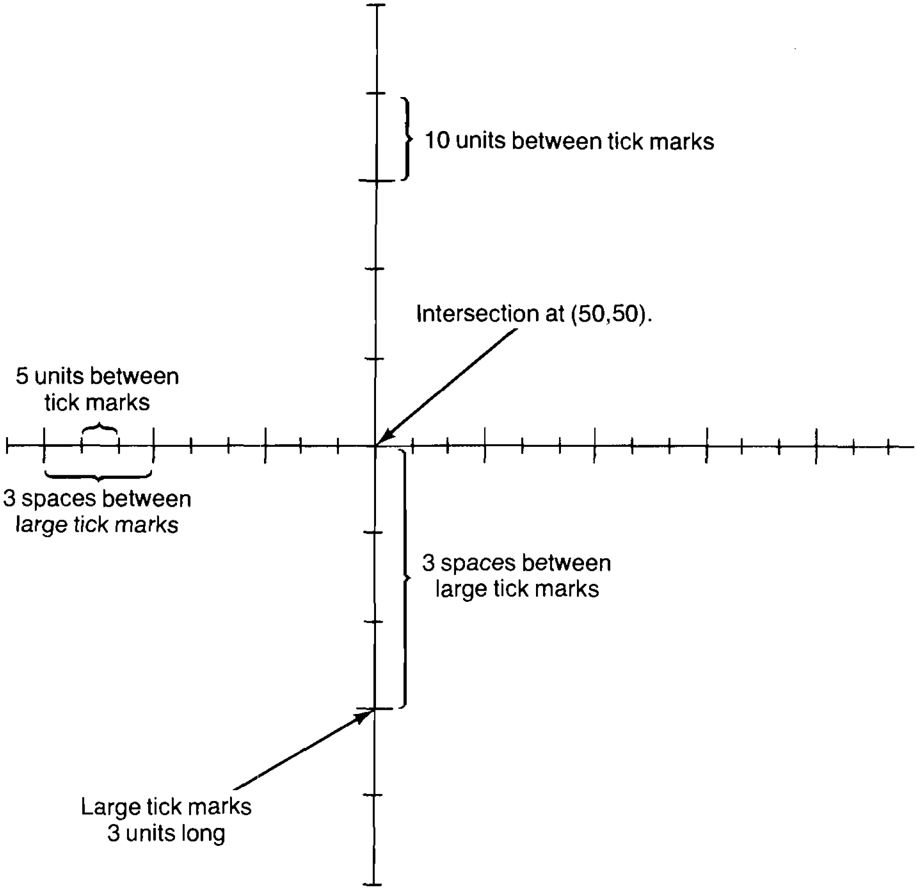
Load and run the file "LABELS" from the Manual Examples disk. Notice that the Y-axis label on the left edge of the screen is created by writing one letter at a time. You only need to move to the position of the first character in that label because each label statement automatically terminates with a carriage return/linefeed. This causes the pen to go one line down, ready for the next line of text.

Now you know what you are measuring — voltage vs. time — but you still do not know the units being used. You need an X-axis and a Y-axis labelled with numbers in appropriate places. You can use the AXES statement to accomplish this.

**Axes and Tick Marks.** You can use the AXES statement to draw X and Y axes and short lines, perpendicular to the axes, to indicate the spacing of units. These short lines are called tick marks. The axes may intersect at any point you desire. The tick marks may be any distance apart, and you can select the "major tick count" for each axis. The major tick count is the total number of tick marks drawn for every large one. This makes it convenient to count by fives or tens or whatever you chose the major tick count to be. And finally, you can specify how long you want the major tick marks to be. This is measured in GDUs. Enter the following program:

```
10  GINIT
20  FRAME
30  AXES 5,10,50,50,3,3,3
40  WAIT 5
60  END
```

When you run this program, you should see the figure below:


10 units between tick marks
Intersection at (50,50).
5 units between tick marks
3 spaces between large tick marks
3 spaces between large tick marks
Large tick marks 3 units long

In the axes statement, the first parameter specifies the distance between tick marks along the horizontal (x) axis, and the second parameter specifies the distance along the vertical (y) axis. The third and fourth parameters specify the intersection point of the axes. The fifth and sixth parameters specify the number of spaces between the large tick marks, and the last parameter specifies the size of the large tick marks. The small tick marks are drawn half the size of the large tick marks.

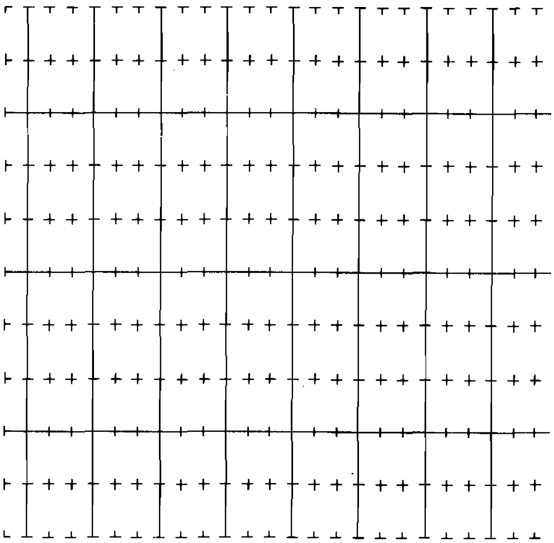
Now load the file "LABELS" again, add the AXES statement shown below to this program, and run it to see the difference.

```
145   AXES 1,.1,0,15,5,5,3
```

**Grids.** You can also create a full grid pattern. Enter and run the following program:

```
10  GINIT
20  FRAME
30  GRID 5,10,50,50,3,3,3
40  WAIT 5
50  END
```

When you run this program you should see the following:



Some of the parameters have slightly different meanings in a GRID statement than in an AXES statement. The first two still represent the distance between tick marks in the horizontal and vertical directions respectively. The next two parameters specify the intersection point of two lines in the grid. The fifth and sixth parameters still specify the number of spaces between large tick marks for each axis. The last parameter still specifies the length of the tick marks.

Like frames, axes and grids are always parallel to the edges of the display. Axes and grids are affected by the line type and pen type.

Load and run the file "AXES" from the Manual Examples disk to see various kinds of axes and grids.

You can stop this program at any time by pressing STOP.

## Other Ways to Draw or Move

There are three other ways to draw or move. The first is using the PLOT statement. With this statement, you specify the point to which the pen moves and also whether the pen is up or down before or after it is repositioned. Here is a PLOT statement.

```
PLOT 10,40,2
```

The first two items in the statement are the x and y coordinates, and the third item is an optional pen control value. The following table shows how the pen control value affects the output.

| Pen Control Value | Meaning |
|---|---|
| Negative and even (−2, −4, −6, ...) | Raise the pen before repositioning it. |
| Non-negative and even (0, 2, ...) | Raise the pen after repositioning it. |
| Negative and odd (−1, −3, −5, ...) | Lower the pen before repositioning it. |
| Positive and odd (1, 3, 5, ...) | Lower the pen after repositioning it. |

If no pen control is specified, 1 is assumed.

IPLOT (Incremental PLOT) is the second way to move or draw. It is similar to PLOT. The difference is that with IPLOT each repositioning is incremental like IMOVE OR IDRAW. The pen control values cause actions similar to those in PLOT.

RPLOT (Relative PLOT) is the third way to move or draw. It is similar to PLOT in that the RPLOT parameters are displacements from an origin, but they are displacements from a local origin. A local origin is a temporary origin for all consecutive RPLOT statements. Each coordinate given in an RPLOT statement is measured from the local origin. The local origin is defined as the current position when the first RPLOT is executed. When you stop executing consecutive RPLOTs, that is, when a graphics output statement other than an RPLOT statement is executed, the local origin ceases to exist.

The next program is an example of how RPLOT works. Load the file "STARS" and list the program. Step through it slowly and determine what happens with each statement before executing it.

Notice how the local origin is set by a MOVE and a series of IMOVEs. The pattern is actually drawn by repeating the subroutine, Rplot. RPLOT is particularly useful when drawing the same series of lines in different spots on the CRT.

PLOT, IPLOT, and RPLOT are useful for controlling the pen with a formula or variable. For instance, you might want to create a variable Pen_status. If Pen_status equals −2, no line is drawn, but the current position is updated.

## Erasing Lines

Now that you can draw lines, you're probably wondering how to erase them. GCLEAR clears the entire graphics area of the CRT screen. To eliminate one specific line or portion of a line, use the PEN statement. The PEN statement has the form:

PEN *Pen_number*

where *Pen_number* is a numeric expression which specifies the pen to use.

The PEN statement gives a choice of "pen" with which to plot. The default pen type is 1. This is the pen type that creates the white line you have seen so far.

A pen selection of −1 sets the pen type to erase white. Output statements are then executed with that color. If the primitive crosses over a point on the display which is white, that point becomes black.

Note that any output statement using this pen value not only erases lines created by DRAW or IDRAW, but also erases part of a frame or any other graphics output created with PEN 1. Obviously, a pen type of −1 is only usable with a plotter that can erase part of its display, such as a CRT. Plotters that output on paper ignore a PEN −1 statement.

## Line Attributes

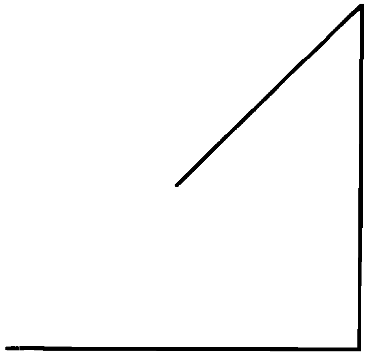
What is it about lines that distinguishes them from other lines? The color of a line can set it apart from other lines; so can the pattern used to draw it (for example, dashed or dotted). These distinguishing traits are known as attributes of the line. Your system provides a number of attributes for graphics primitives.

**Pen Types.** The PEN statement presented in the previous section does not directly create graphics output but does affect the appearance of graphics output. For this reason, it is an "attribute" statement.

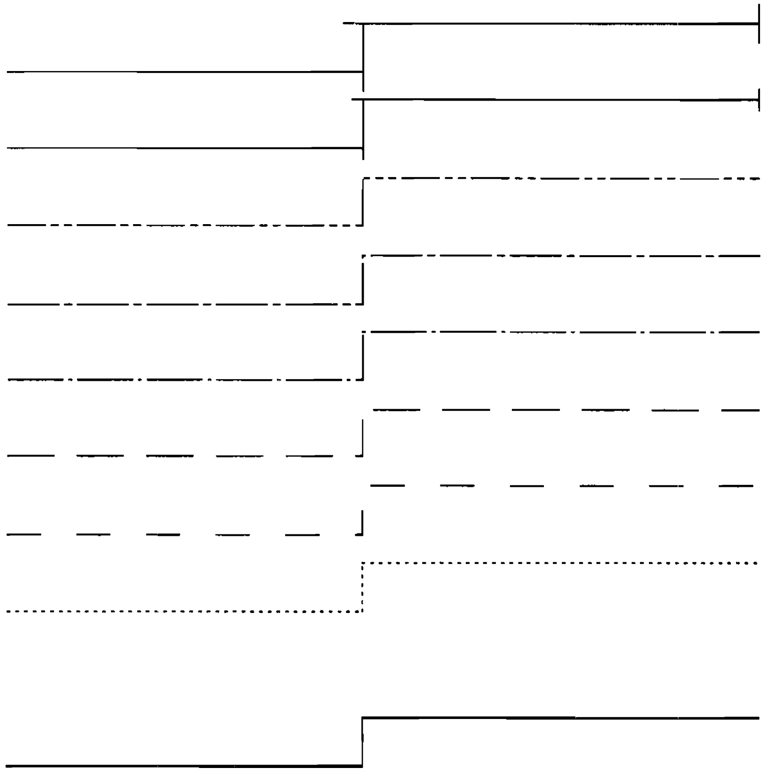
On a monochrome CRT, a pen type > 0 makes the pen color "white". A pen type < 0 makes the pen color "erase white". A pen type of 0, in effect, disables the pen; no lines or erasures occur when a DRAW or other output statement is executed. However, it does not disable updating of the current position; that is, if you execute a DRAW, no line is drawn, but the current position is now at the coordinates specified in the DRAW.

PEN can also specify other colors if you have a color CRT or other pen stalls on a paper plotter. The "Color Graphics" section provides more information about these pen types.

Load and run the file "PENDEMO" from the Manual Examples disk.



**Line Types.** To distinguish between lines, use different line types such as dashes or dots. The LINE TYPE statement gives a number of choices. The pattern of a line is considered an attribute of that line. Load and run the file "LINETYPE.S" from the Manual Examples disk. The following figure should be drawn, showing the available line types. It also shows how the lines look when drawn straight or around corners.



## Creating Simple Shapes

BASIC Graphics has specific statements to create many kinds of regular polygons quickly and easily.

**Rectangles.** The simplest polygon is probably the rectangle created used the RECTANGLE statement. Enter and execute the following program:

```
10  GINIT
20  MOVE 20,20
30  RECTANGLE 10,30
40  WAIT 5
50  END
```

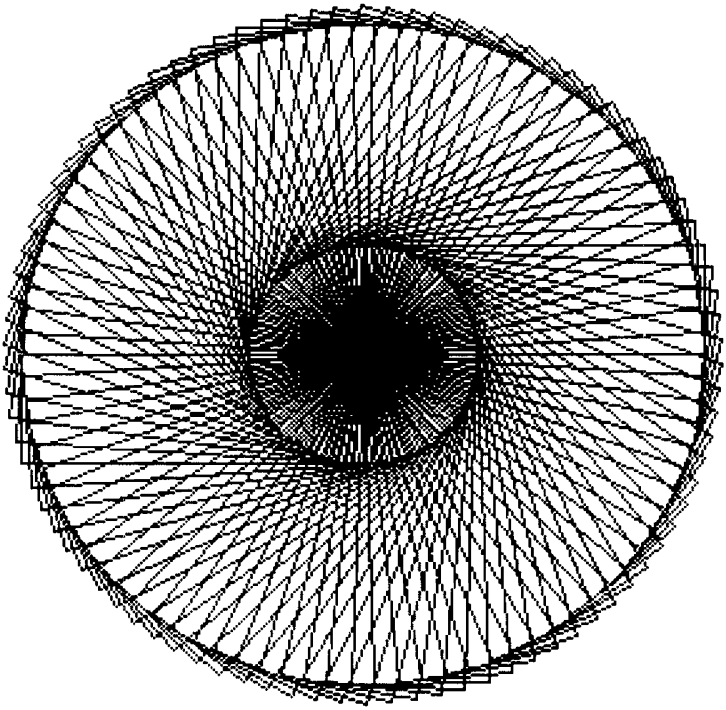
The first parameter is the width of the rectangle and the second parameter is the height of the rectangle. In this case, you've drawn a rectangle 10 units wide and 30 units high.

RECTANGLE is another statement which has, in effect, a local origin. The rectangle is drawn with the current position (local origin) in the lower-left corner. In the example above, the local origin is at 20,20. The sides of the box are parallel to the sides of the CRT. This is only the default form. You can use the PDIR (Plot DIRection) statement to rotate the rectangle to any angle about the Z axis. Its form is:

`PDIR` *Angle*

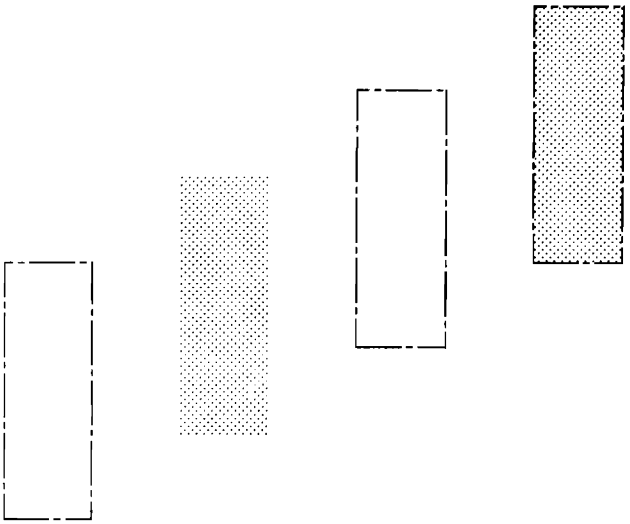
where *Angle* is the amount the polygon is rotated in degrees, radians or grads.

As an example, load and run the file "PDIRDEMO" from the Manual Examples disk. You should see the following:



Notice that the rotation is about the local origin (lower-left corner) of the rectangle. This means that when placing a rectangle on the display, you should note where the lower-left corner should go, not where the center of the rectangle should go.

The RECTANGLE statement has two other options. They are FILL and EDGE. If you specify FILL, the rectangle is filled to create a solid block. If you specify EDGE, the rectangle edge is drawn with the current pen and line type. Load and run the file "FILLEDGE" from the Manual Examples disk. You should see the following:



**FILL Attributes.** You can control the shade of the fill color to various degrees of grey using the AREA COLOR or AREA INTENSITY statements. The following paragraphs tell how to use these statements with a monochrome display. Refer to the "Color Graphics" section if you are using a color monitor.

The AREA COLOR statement defines the fill color based on a hue, a saturation and luminosity. Only the third parameter, the luminosity parameter, has any effect on a monochrome CRT. For example:

```
AREA COLOR .1,.1,.1
```

```
AREA COLOR .5,.6,.1
```

```
AREA COLOR .7,.2,.1
```

All specify the same shade of grey. The luminosity parameter specifies the approximate percentage of pixels to be "on" in the filled area. The previous three statements all turn on 10 percent of the pixels in the fill area. The range of the luminosity is from 0 through 1.

The AREA INTENSITY statement defines the fill color based on the largest of the three parameters in the statement. For example:

```
AREA INTENSITY .1,.3,.4

AREA INTENSITY .4,.2,.01

AREA INTENSITY .2,.4,.03
```

All specify the same shade of grey. The largest parameter specifies the approximate percentage of pixels to be "on" in the filled area. The previous three statements all turn on 40 percent of the pixels in the fill area. Again, the range of the parameters in the statement is 0 through 1.

**Polygons.** To create more general polygons use the POLYGON statement. Enter and execute the following:

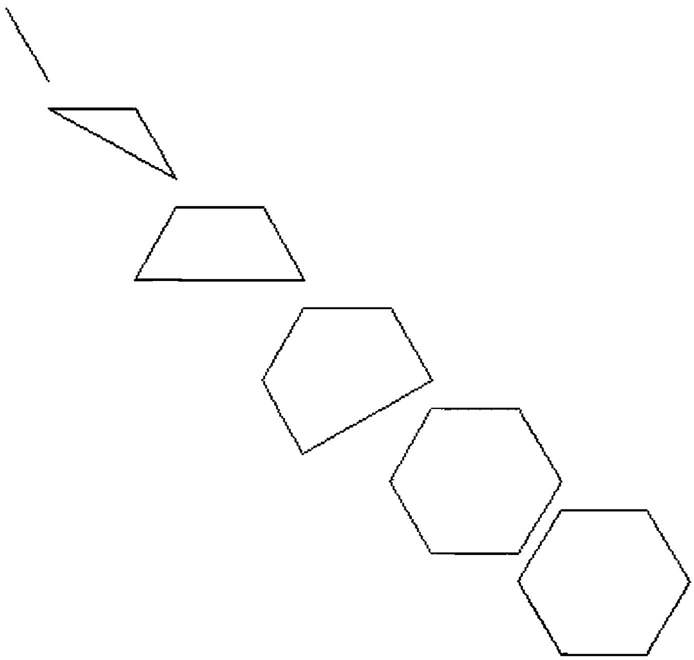
```
10  GINIT
20  GRAPHICS ON
30  MOVE 50,50
40  POLYGON 10,6
50  WAIT 5
60  END
```

The first parameter of the POLYGON statement is the radius of the polygon and the second is the number of sides. Thus, POLYGON 10,6 creates a six-sided polygon with a radius of 10 centered about the point (50,50).

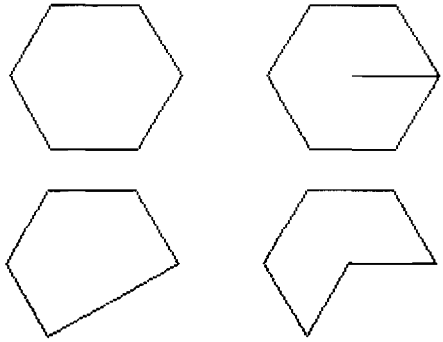
The polygon is output with the first vertex at an angle of 0 degrees from the X axis.

The POLYGON statement forces the shape to be a closed polygon as opposed to an open figure.

You may specify the number of edges to draw and the POLYGON statement then closes the shape. For example, load and run the file "POLYGON6" from the Manual Examples disk. You should see the following:



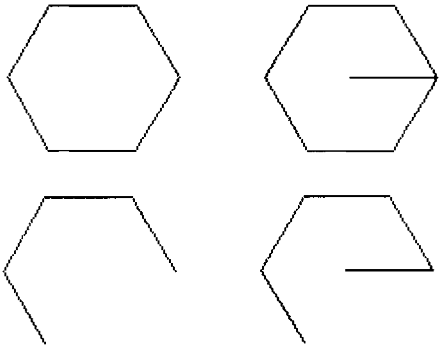
Whether the pen is up or down before the polygon is drawn makes a difference. Load and run the file "POLYGON4" from the Manual Examples disk. You should see the following:



If the pen is down when this statement is executed, the first line is from the pen's starting position out to the first vertex and the last line is from the last vertex back to the pen's original position. If the pen is up when the statement is executed, the last line is from the last vertex back to the first. In either case, the current position is unchanged.

POLYGON also has the FILL and EDGE options like RECTANGLE.

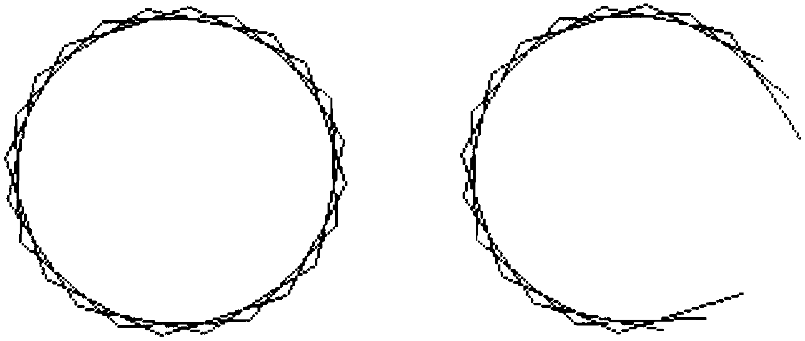
Sometimes you don't want a closed polygon. In these cases, use the POLYLINE statement. Load and run the file "POLYLINE".



You can see that polylines are similar to polygons, but they don't have to be closed.

The general rule is: if the pen is down when this statement is executed, the first line is from the pen's starting position out to the first vertex. If the pen is up when this statement is executed, the figure starts at the first vertex. The figure always stops at the last vertex. The current position remains at its original location.

PDIR also affects polygons and polylines. The local origin is at the center of the polygon/polyline so the rotation is about the center of the object. Load and run the file "CIRCLES2" from the Manual Examples disk to see PDIR's effect.



## Additional Pen Control

There are additional pen control values available when using PLOT with an array that can't be used with PLOT x,y,z. These help you to make more complex drawings than are possible with PLOT x,y,z. For instance with PLOT x,y,z, a pen control value which is a positive odd integer lowers the pen after repositioning it. However, when using PLOT with an array, only a pen control value of 1 causes this. A pen control value of 3 tells the system that the x coordinate value is to be interpreted as a pen number. Thus, you can switch pens while plotting from an array. Here is a complete table of all the pen control elements and how they are interpreted.

| X Coord. Element | Y Coord. Element | Z Coord Element | Pen Control Element | |
|---|---|---|---|---|
| This element is interpreted as: | This element is interpreted as: | This element is interpreted as: | If this element is: | Action: |
| x coord. | y coord. | z coord. | negative even number (−2, −4, −6, ...) | Raises the pen and then repositions it. |
| x coord. | y coord. | z coord. | negative odd number (−1, −3, −5, ...) | Lowers the pen and then repositions it. |
| x coord. | y coord. | z coord. | 0 or 2 | Repositions the pen and then raises it. |
| x coord. | y coord. | z coord. | 1 | Repositions the pen and then raises it. |
| pen number | n/a | n/a | 3 | Selects this pen. |
| line type | repeat length | n/a | 4 | Selects this line type and repeat length. |
| fill color (see the following text) | n/a | n/a | 5 | Selects this fill color. |
| n/a | n/a | n/a | 6 | Starts a polygon with FILL. |
| n/a | n/a | n/a | 7 | End of a polygon. |
| row number | n/a | n/a | 8 | End of plotting data. |

| **X Coord. Element** | **Y Coord. Element** | **Z Coord Element** | **Pen Control Element** | |
|---|---|---|---|---|
| This element is interpreted as: | This element is interpreted as: | This element is interpreted as: | If this element is: | Action: |
| n/a | n/a | n/a | 9 | Index of End-of-plotting-data row |
| n/a | n/a | n/a | 10 | Starts a polygon with EDGE. |
| n/a | n/a | n/a | 11 | Starts a polygon with FILL and EDGE. |
| n/a | n/a | n/a | 12 | Frames the current display area. |
| n/a | n/a | n/a | >12 | Row Ignored. |

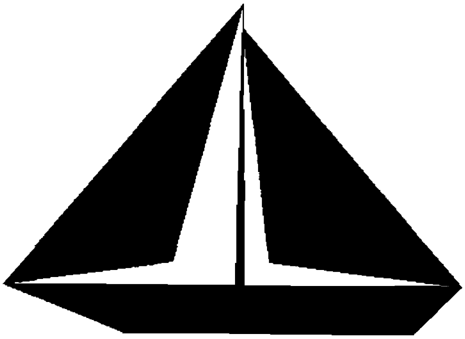
The fill color gives you the ability to change the color of a filled polygon. Refer to "Color Graphics," later in this chapter, for further information.

Now load and list the "SHIP" from the Manual Examples disk. The DATA lines in this program are treated as an example array. A step-by-step explanation of the data elements as they would be interpreted in a two-dimensional PLOT is shown in the matrix that follows.

| | | | |
|---|---|---|---|
| 0 | 0 | 12 | Frames the display. |
| 40 | 10 | −2 | Raises the pen and moves it to (40,10). |
| 11 | 0 | 5 | Selects the fill color defined by the value 11. The 0 is ignored. |
| 12 | 34 | 11 | Signals the start of a polygon that is filled and has an edge. The values 12 and 34 are ignored. Every consecutive line from this point on is considered part of the polygon until the figure is closed. The current position is still (40,10). |
| 100 | 10 | −1 | Lowers the pen and draws a line from (40,10) to (100,10). |
| 110 | 20 | −1 | Keeps the pen down and draws a line from (100,10) to (110,20). |
| 15 | 20 | −1 | Keeps the pen down and draws a line from (110,20) to (15,20). |
| 40 | 10 | −1 | Keeps the pen down and draws a line from (15,20) back to (40,10). After this line is completed there is a closed figure so it is filled and the edge is kept. |
| 0 | 0 | 7 | Signals the end of the polygon. If this line is not included and any DRAW actions are included before a MOVE, those DRAWs are considered part of the polygon. Thus, when a MOVE is executed the DRAWs are closed off to create a polygon. |
| 65 | 20 | −2 | Raises the pen and moves it to (65,20). |

|  |  |  |  |
|---|---|---|---|
| 0 | 0 | 6 | Signals the start of another polygon with the FILL attribute. The zeros in the first two columns are more appropriate than the values left in the first two elements of row 4. |
| 64 | 80 | −1 | Draws a line from (65,20) to (64,80). |
| 63 | 20 | −1 | Draws a line from (64,80) to (63,20). |
| 0 | 0 | 7 | Signals the end of a polygon so the two lines are closed off to form a triangle and then the polygon is filled. |
| 64 | 80 | 0 | Draws a line from (63,20) to (64,80) and then lifts the pen. |
| 1 | 0 | 5 | Selects the fill color defined by the value 1. |
| 0 | 0 | 6 | Signals the start of a polygon with FILL. |
| 50 | 25 | −1 | Lowers the pen and draws a line to (15,20). |
| 15 | 20 | −1 | Keeps the pen down and draws a line to (15,20). |
| 64 | 80 | −1 | Keeps the pen down and draws a line to (64,80). |
| 0 | 0 | 7 | End of the polygon. |
| 64 | 75 | −2 | Moves the pen to (64,75). |
| 0 | 0 | 6 | Start of another polygon with FILL. |
| 70 | 25 | −1 | Lowers the pen and draws a line from (64,75) to (70,25). |
| 110 | 20 | −1 | Keeps the pen down and draws a line to (110,20). |
| 64 | 75 | −1 | Keeps the pen down and draws a line to (64,75). |
| 0 | 0 | 7 | End of the polygon. |

Now run the program to see the results:



## Using Graphics Effectively

The last section discussed the more elementary graphics operations. This section will present more detailed information on those statements and introduce several other graphics operations.

Most of the demonstration programs in this section are stored on the Manual Examples disk which was shipped with HP BASIC. You are encouraged to load and run these programs while you are reading the manual, as this will make understanding the concepts much easier.

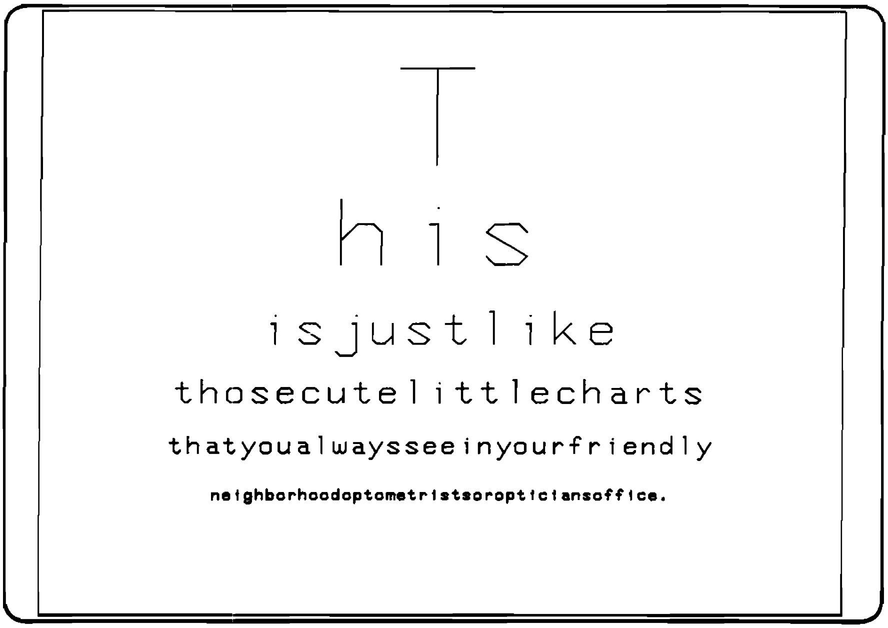
### More on Labelling a Plot

There are three statements that complement the LABEL statement; they expand its capabilities greatly.

The first is CSIZE, which means character size. CSIZE has two parameters: character cell height (in GDUs) and aspect ratio. The height measures the character cell size. A character cell contains a character and some blank space above, below, left of, and right of the character. This blank space allows packing character cells together without making the characters illegible. The amount of blank space depends, of course, on which character is contained in the cell.

The first of these small programs shows how the CSIZE statement changes the size of characters. You may load this program from file "CSIZE" on the Manual Examples disk.

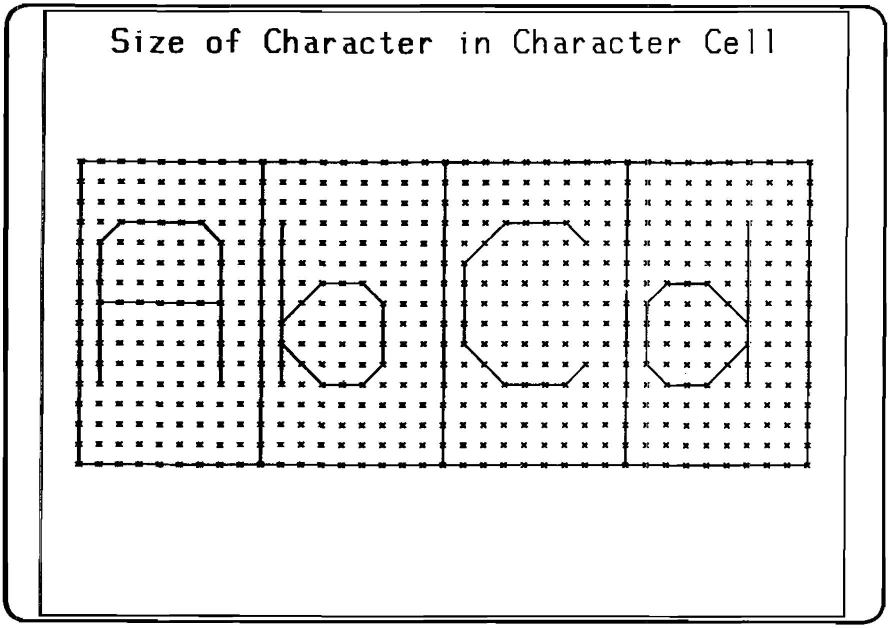
When you run the program you should see something like this:



```
T
his
isjustlike
thosecutelittlecharts
thatyoualwaysseeinyourfriendly
neighborhoodoptometristsoropticiansoffice.
```

The FOR-NEXT loop writes lines of text on the screen with different character sizes. The DATA statements contain both pieces of information. Incidentally, notice also the WINDOW statement. It specifies a Ymin larger than the Ymax. This causes the top of the screen to have a lesser Y-value than the bottom. This is perfectly legal.

The next program deals with the relationship between the size of the character, per se, and the size of the character cell — that rectangle in which the character is placed. This program is on file "CHARCELL" on the Manual Examples disk.



As the diagram shows, a character is drawn inside a rectangle, with some space on all four sides. The rectangle's height is specified by the CSIZE statement, and is measured in GDUs. The rectangle's width (also measured in GDUs) is the height multiplied by the aspect ratio. This rectangle is subdivided into a grid of 9 wide by 15 high. Characters are drawn in this framework, called the symbol coordinate system. Of course, the little Xs in the plot above are not drawn when you label a string of text; they are there solely to show the position of the characters within the character cell.

Again, character cell height is measured in GDUs, and the definition of aspect ratio for a character is identical to the definition of aspect ratio for the hard clip limits mentioned earlier: the width divided by the height. Thus, if you want short, fat letters, use an aspect ratio of 1.5 or larger. If you want tall, skinny letters, use an aspect ratio less than about 0.5.

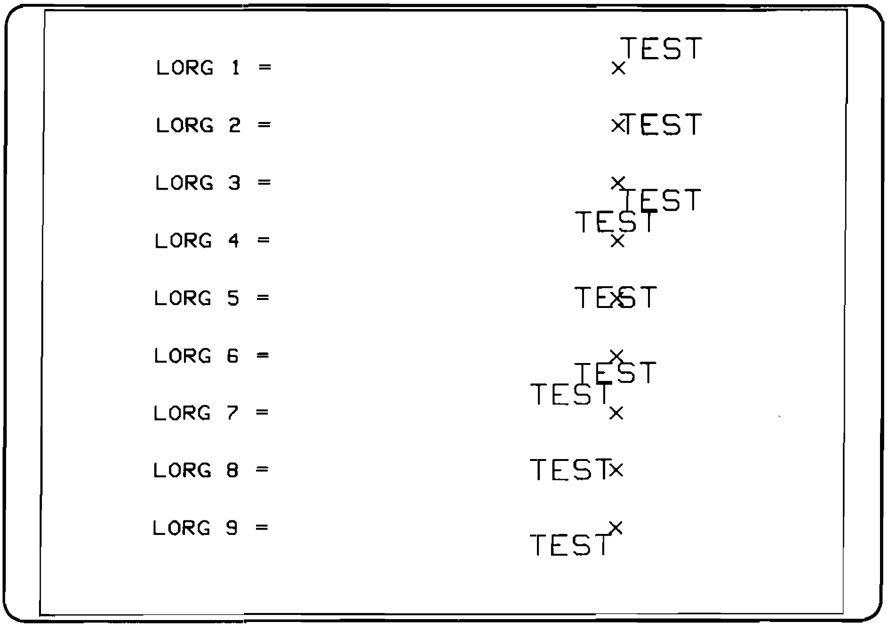
`CSIZE 3` Cell 3 GDUs high, aspect ratio .6 (default).

`CSIZE 6,.3` Cell 6 GDUs high, aspect ratio .3 (tall and skinny).

`CSIZE 1,2` Cell 1 GDU high, aspect ratio 2 (short and fat).

Note that you do not have to specify a second parameter (the aspect ratio) in the CSIZE statement. This defaults to 0.6.

The second statement you need is LORG, which means label origin. This lets you specify which point on the label ends up at the point moved to before writing the label. You may load the following program from the file "LORG" on the Manual Examples disk.



The x's indicate where the pen was moved to before labelling the word "TEST". This diagram shows that, for example, if LORG 1 is in effect and you move to 4,5 to write a label, the lower left of that label would be at 4,5. This automatically compensates for the character size, aspect ratio, and label length. It makes no difference whether there is an odd or even number of characters in the label. If LORG 6 had been in effect, and you had moved to 4,5, the center of the top edge of the label would be at 4,5. You can readily see how useful this statement is in centering labels, both horizontally and vertically.

The third statement you need to know is LDIR, meaning label direction. This specifies the angle at which the subsequent labels will be drawn. The angle is specified in the current angular units, and is either DEG (degrees) or RAD (radians). For example, assuming degrees is the current angular mode:

`LDIR 0` Writes label horizontally to the right.

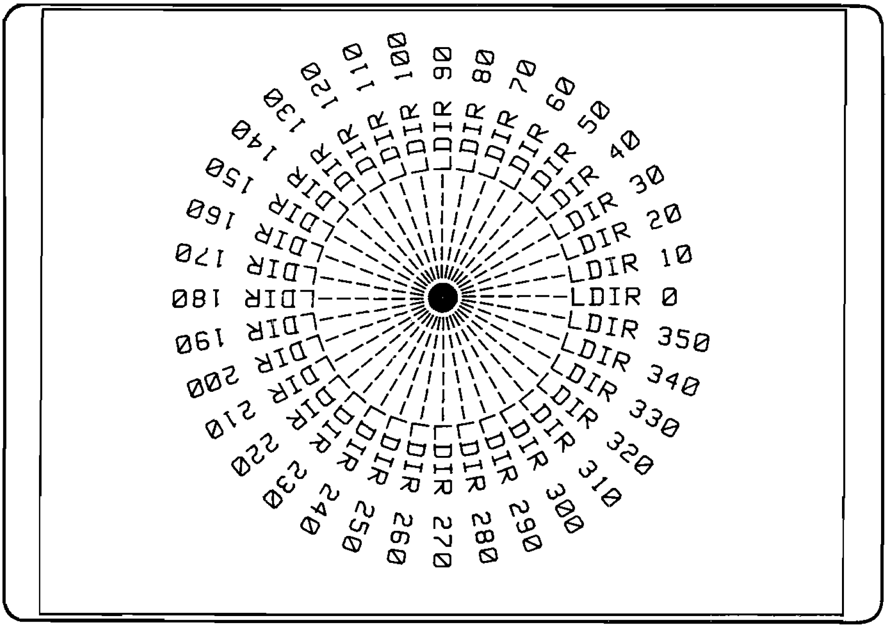
`LDIR 90` Writes label vertically, ascending.

`LDIR 14` Writes label ascending a gentle slope, up and right.

`LDIR 180` Writes label upside down.

`LDIR 270` Writes label vertically, descending.

Load and list the file "LDIR" on the Manual Examples disk. You'll note that LORG 2 was specified (line 70), and this remained in effect for many LDIRs. Each label is centered on the left edge (relative to the label, remember). Now run the program and you should see the following:



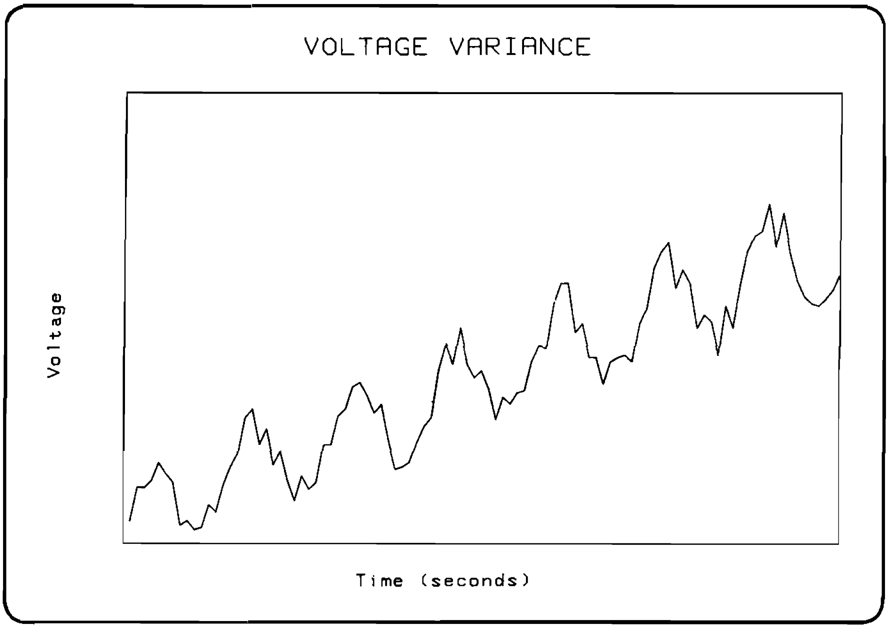
The label origin specified by LORG is relative to the label, not the plotting surface, and it is independent of the current label direction. For example, if you have specified

```
LORG 3
DEG
LDIR 90
MOVE 6,8
```

and then write the label, it is written going straight up, not horizontally. Therefore, it is the upper left corner of the label which is at point 6,8 relative to the rotated label. Relative to the plotting device, however, it is the lower left corner of the label which is at 6,8 (in this example) because the label has been rotated.

Now that the prerequisites have been taken care of, we can discuss the statement which actually causes labels to be written: LABEL. LABEL takes into account the most recently-specified CSIZE, LDIR and LORG when it writes a label. You must position the label to get to the point at which you want the label to be placed. You can use MOVE to accomplish this.

All four statements have been utilized in the following update to our progressive plotting program. You may load this program from file "SINLABEL" on the Manual Examples disk.



Notice that the title of the graph, "VOLTAGE VARIANCE", is now displayed in bold face. This effect was achieved by plotting the label several times, moving the label origin just slightly each time. Notice lines 60 through 90. The loop variable, I, goes from −.3 to .3 by tenths. This is the offset in the X direction (in GDUs) of the label origin.*

Since this is being labelled with LORG 6 in effect, the label origin (the point moved to immediately prior to labelling) represents the center of the top edge of the label. This method can also be used for offsetting in the Y direction. Or, offset both X and Y. This will give you characters which are thick in a diagonal direction, which makes them look like they are coming out of the page at you. However, a more typical boldface is produced by offsetting only in the X direction.

Now you know what you're measuring — voltage vs. time — but the units are still not shown. What is needed is an X-axis and a Y-axis, and they need to be labelled with numbers in appropriate places.

## Miscellaneous Graphics Concepts

**Clipping.** Something that occurs completely "behind the scenes" in your computer when drawing is a process called clipping. Clipping is the process whereby lines that extend over the defined edges of the drawing surface are cut off at those edges. There are two different clipping boundaries at all times: the soft clip limits and the hard clip limits. The hard clip limits are the absolute boundaries of the plotting surface, and under no circumstances can the pen go outside these limits. The soft clip limits are user-definable limits, and are defined by the CLIP statement.

```
CLIP 10,20.5,Ymin,Ymax
```

This statement defines the soft clip boundaries only; hard clip limits are completely unaffected. After this statement has been executed, all lines which attempt to go outside the X limits (in UDUs) of 10 and 20.5, or the Y limits (in UDUs) of Ymin and Ymax will be truncated at the appropriate edge. Clipping at the soft clip limits can be turned off by the statement:

```
CLIP OFF
```

It can be turned back on, using the same limits, by

```
CLIP ON
```

* Technically, a MOVE uses UDUs for its units, but UDUs and GDUs are identical until a SHOW or WINDOW is executed.

If you want the soft clip limits to be somewhere else, use the CLIP statement with four different limits. Only one set of soft clip limits can be in effect at any one time. Clipping at the hard clip limits cannot be disabled.

The VIEWPORT statement, in addition to defining how WINDOW coordinates map into the VIEWPORT area, turns on clipping at the specified VIEWPORT edges.

**Drawing Modes.** On a monochromatic CRT, there are three different drawing modes available. (For information on selecting pens with a color CRT, refer to "Color Graphics," later in this chapter.) The three pens perform the following actions:

| Number | Function |
|---|---|
| 1 | Draws lines (turns on pixels). |
| −1 | Erases lines (turns off pixels). |
| 0 | Complements lines (changes pixels' state). |

A characteristic of drawing with pen −1 or pen 1 is that if a line crosses a previously drawn line, the intersection will be the same "color" as the lines themselves. When drawing with pen 0, and a line crosses a previously drawn line, the intersection becomes the opposite state from that of the lines. For example, assume a black background. You select PEN 0 and draw a pair of AXES. When the first axis is drawn, all pixels are off. Thus, as the line is being drawn all pixels are turned on along its length. However, as the second axis is drawn all pixels along its length are turned on, except where it crosses the first axis. At the crossing, the pixels previously turned on are turned off. Thus, the crossing contrasts with the lines.

**Storing and Retrieving Images.** If a picture on the screen takes a long time to draw, or the image is used often, it may be advisable to store the image itself — not the commands used to draw the image — in memory or in a disk file.

This may be done with the GSTORE command. First, you must declare an INTEGER array of sufficient size to hold all the data in the graphics raster. This array holds the picture itself, and it doesn't care how the information got to the screen, or in what order the different parts of the picture were produced. You can use GLOAD to restore the picture to the CRT display.

To find the minimum array size for a monochrome display, multiply the number of pixels in the X direction by the number of pixels in the Y direction and divide by 16 bits per word. Thus, for a 512 by 390 monochrome monitor, the minimum array size is:

512 × 390 ÷ 16 = 12,480 words

For a color display you have to multiply the number of pixels by the number of color planes (three). Thus, a 512 by 390 color display image requires an array size of 37,440 words. *However, you cannot specify an array with more than 32,767 elements in any dimension.* To get around this restriction, make one dimension the number of memory planes (three) and the other dimension the number of pixels ÷ 16 (12,480). The following statement declares such an array:

```
INTEGER Image(1:12480,1:3)
```

If your array is larger than necessary to store an image, GSTORE will fill it only to the point where the image is exhausted. If your image is larger than the array, GSTORE will fill the array completely and ignore the rest of the image.

## Data-Driven Plotting

When plotting data points, you will often find that they do not form a continuous line. You must have the ability to control the pen's position. Earlier in this chapter, a third parameter in the PLOT statement was mentioned. This third parameter is the pen-control parameter, and its function is to raise or lower the pen so that many lines can be drawn with one set of data.

When using a single X-position and Y-position in a PLOT statement (as opposed to plotting an entire array; we'll cover this a little later), the third parameter is defined in the following manner. Though it need not be of type INTEGER, its value should be an integer. If it is not, it will be rounded. The third parameter is either positive or negative, and at the same time, either even or odd. Whether the number is odd or even determines which action will be performed on the pen, and the sign of the number determines when that action will be performed: before or after the pen is moved.

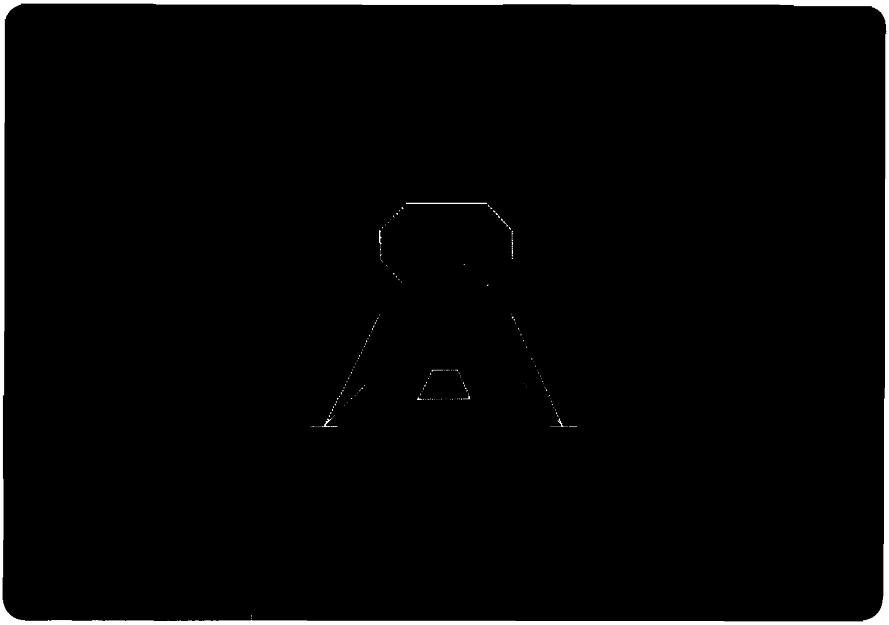
The following table shows how the pen control parameter works:

**Pen Control Parameter**

| | Even (Up) | Odd (Down) |
|---|---|---|
| Positive (after) | Pen up after move | Pen down after move |
| Negative (before) | Pen up before move | Pen down before move |

The default parameter is +1 — positive odd — therefore, the pen will drop after moving, and if the pen is already down, it will remain down, drawing a line. Zero is considered positive.

The program LEM1 (file "LEM1" on Manual Examples disk) is a good example of pen control. It draws a LEM (Lunar Excursion Module). There are two arrays used: a two-column REAL array for the X and Y data, and a one-column INTEGER array containing pen-control data. The data is read from DATA statements. Load and list the program and then run it.



Having the pen-control parameter in a third column of the data array is generally a good strategy; it reduces the number of array names you must declare, and when you have the data points for the picture, you also have the information necessary to draw it. Nevertheless, an array must be entirely of one type, and usually you'll want the data to be real. So if you're pressed for memory, you may want to have a two-column data array of type REAL, and a one-column pen-control array of type INTEGER. Integer numbers take only one-fourth the memory real numbers take to store.

## Translating and Rotating a Drawing

Often there is an application where a segment of a drawing must be replicated in many places; the same sub-picture needs to be drawn many times. Using the PLOT statement, it is possible but rather tedious to do. There is another statement called RPLOT, which draws a figure relative to a point of your choice. RPLOT means Relative PLOT, and it causes a figure to be drawn relative to a previously-chosen reference point. RPLOT's parameters may be two or three scalars, or a two-column or three-column array; the parameters are identical to those of PLOT.

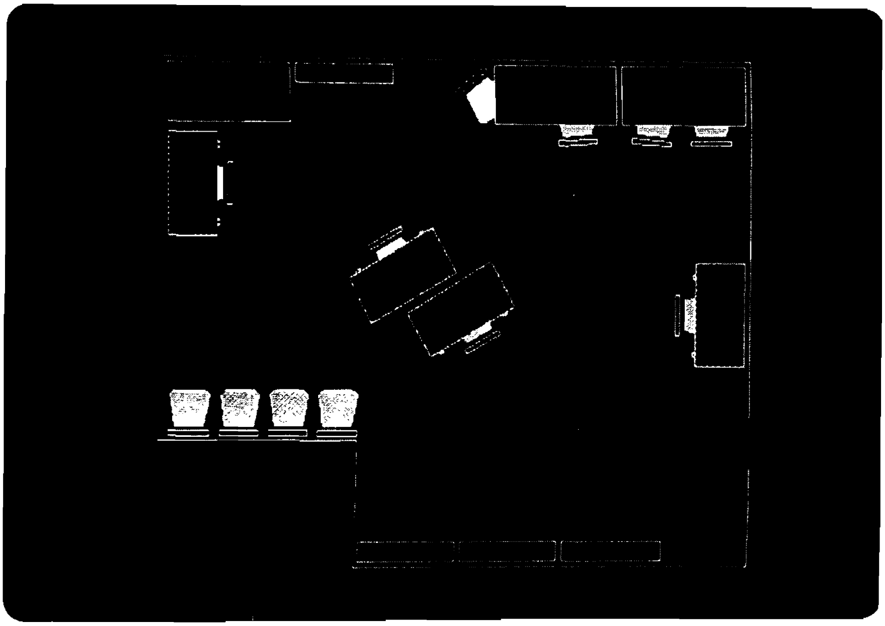
The picture defined by the data given to an RPLOT statement is drawn relative to a point called the current relative origin. This is not necessarily the same as the pen position. The current relative origin is the last point resulting from any one of the following statements:

- AXES
- DRAW
- FRAME
- GINIT
- GRID
- IDRAW
- IMOVE
- IPLOT
- LABEL
- MOVE
- PLOT
- POLYGON
- POLYLINE
- RECTANGLE

Typically, a MOVE is used to position the current relative origin at the desired location, then the RPLOT is executed to draw the figure. After the RPLOT statement has executed, the pen may be in a different place, but the current relative origin has not moved. Thus, executing two identical RPLOT statements, one immediately after the other, results in the figure being drawn precisely on top of itself.

A figure drawn with RPLOT can be rotated by using the PIVOT statement before the RPLOT. PIVOT's single parameter is a numeric expression designating the angular distance through which the figure is to be rotated when drawn. This value is interpreted according to the current angular mode: either DEG or RAD.

A program using RPLOT can be found on the Manual Examples disk under the file name "RPLOT". Load and list this file. Various figures are defined with DATA statements: a desk, a chair, a table, and a bookshelf. The program displays a floor layout. Here again, the "end polygon mode" codes (the 0,0,7s in the desk and chair definitions) are unnecessary; when a polygon mode starts, any previous one ends by necessity. Now run the program to see the following figure.



There are two points of interest in this program. First, notice that you can specify the EDGE and/or FILL parameters in the RPLOT statement itself (as in lines 230 and 260), or in the array (as in lines 180 and 210). FILLs and EDGEs are specified in the array by having a 6, a 10, or an 11 in the third column of the array. If FILL and/or EDGE are specified in both the PLOT statement and in the data, and the instructions differ, the value in the data replaces the FILL or EDGE keyword on the statement.

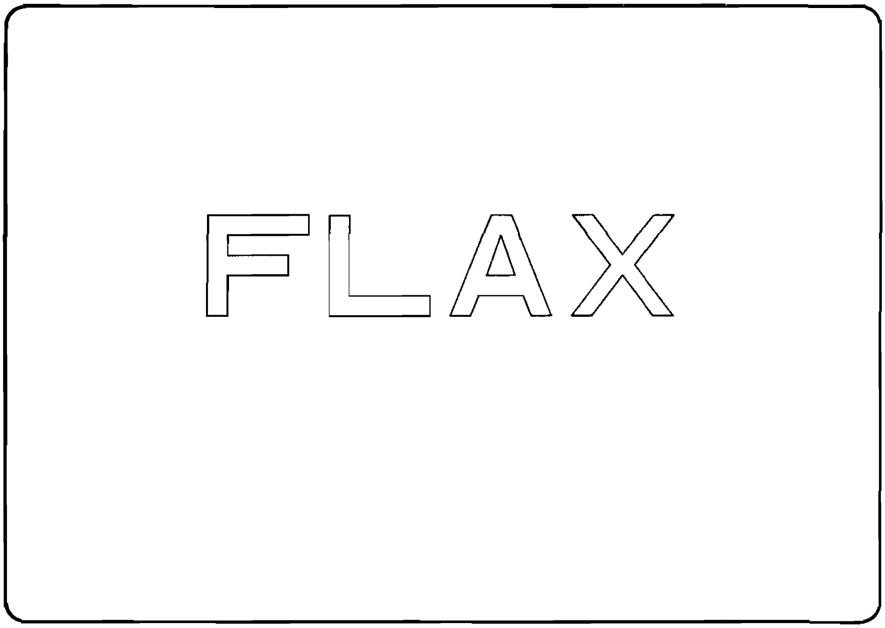
The second interesting point is that some of the chairs appear to be under the desks and tables; that is, parts of several chairs are hidden by other pieces of furniture. This is accomplished by drawing the chair, and then drawing the desk or table partially over the chair, and filling the desktop or tabletop with its own fill pattern, which may be black.

## Incremental Plotting

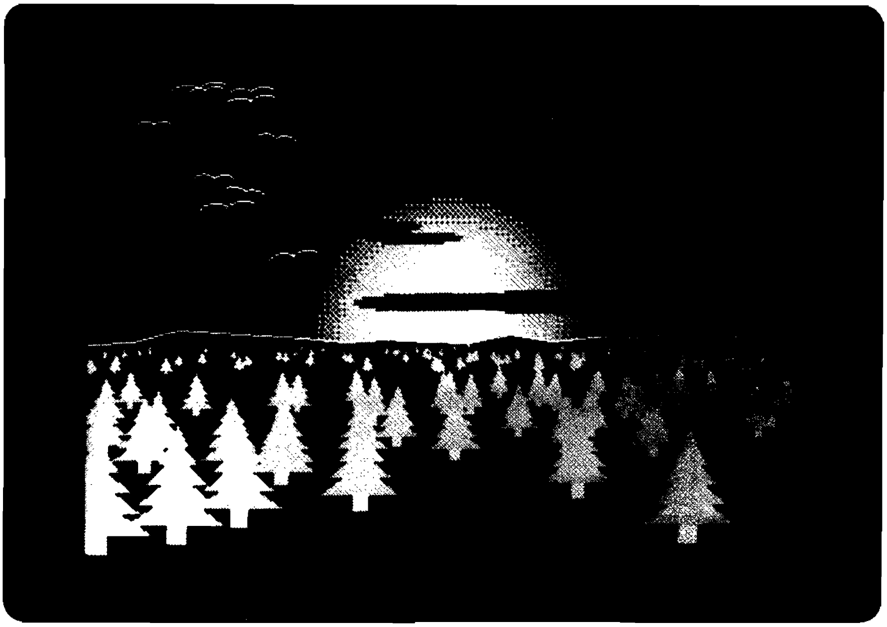
Incremental plotting is similar to relative plotting, except that the origin — the point considered to be 0,0 — is moved every point. Every time you move or draw to a point, the origin is immediately moved to the new point, so the next move or draw will be with respect to that new origin.

There are three incremental plotting statements available: IPLOT, which has the same parameters as PLOT and RPLOT; and IMOVE and IDRAW, which have the same parameters as MOVE and DRAW, respectively.

An example program using IPLOTs can be found in the file "FLAX" on the Manual Examples disk. It reads data from data statements describing the outlines of certain letters of the alphabet, and then plots them. Load and run this file.



A program which demonstrates the use of POLYGON, POLYLINE, PLOT, RPLOT, polygon filling, and gray-shading can be found on the file "SCENERY" on the Manual Examples disk. Load and run this program.



Points of note in this program:

- The sunrise was created with graduated gray shades in successively smaller "circles" (actually 30-sided polygons).
- The horizon was created by defining a rough edge on the top half of a polygon which blacked out the bottom section of the screen. This covered up the bottom of the sun. The white line of the horizon was simple plotting of the horizon array without the first and last points. We didn't want the lower corners of the screen to be included.
- The clouds were created by plotting "circles" after having invoked anisotropic units; thus long, thin ellipses resulted.
- The seagulls were created by drawing two arcs with POLYLINE. An arc is created by defining an N-sided polygon and drawing less-than-N sides. Note that PIVOT was used to cause the starting angle of the arcs to be other than straight to the right.
- The trees were created by defining an array whose left side is a mirror image of the right side. The array is centered around zero in the X direction to allow for scaling of the tree simply by multiplying the array by a constant. RPLOT was used to place the trees in their various positions.

## Color Graphics

Color can be used for emphasis, clarity, and to present visually pleasing images. Color is a very powerful tool, and it follows directly that it is very easy to misuse. Be careful in using color, and it will serve as a valuable tool for communication. Misuse it, and it will garble the communication.

The biggest benefit of the color computer is that it makes experimenting with color so easy. With a bit-mapped frame buffer and a color map, it is easy to test out ideas before you use them. It is also possible to use the color map for simple animation effects and some just plain impressive images.

The methods for displaying color fall into four categories:

- Background Value. Whenever GCLEAR is executed, all the pixel locations in the display are set to 0. Thus, PEN 0 is the background color.
- Line Value. The PEN statement is used to determine the color written to the display for all lines drawn. This includes all lines (including characters created by LABEL) and outlines specified by the secondary keyword EDGE.
- Fill Value. The AREA PEN statement is used to specify the color written to the display for filling areas specified by the secondary keyword FILL.
- Dithered Colors. AREA INTENSITY and AREA COLOR can also be used to specify a fill color.

The PEN, AREA PEN, AREA INTENSITY, and AREA COLOR statements control what are referred to as modal attributes. This means that the value established by one of the statements stays in effect until it is altered by another statement.

### Non-Color Mapped Color

When `PLOTTER IS CRT,"INTERNAL"` is executed, eight colors are available through the PEN and AREA PEN statements. The colors provided are:

- Black and white.
- Red, green, and blue (the additive color primaries).
- Cyan, magenta, and yellow (the complements of the additive color primaries).

The colors can be selected with the PEN statement, the same way they are for an external plotter. 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), n/c (no change), or complement (invert) the value in each color plane.

### Non-Color Map Mode

| Pen Value | Action | Plane 1 (red) | Plane 2 (green) | Plane 3 (blue) |
|---|---|---|---|---|
| −7 | Erase Magenta | 0 | n/c | 0 |
| −6 | Erase Blue | n/c | n/c | 0 |
| −5 | Erase Cyan | n/c | 0 | 0 |
| −4 | Erase Green | n/c | 0 | n/c |
| −3 | Erase Yellow | 0 | 0 | n/c |
| −2 | Erase Red | 0 | n/c | n/c |
| −1 | Erase White | 0 | 0 | 0 |
| 0 | Complement | invert | invert | invert |
| 1 | Draw White | 1 | 1 | 1 |
| 2 | Draw Red | 1 | 0 | 0 |
| 3 | Draw Yellow | 1 | 1 | 0 |
| 4 | Draw Green | 0 | 1 | 0 |
| 5 | Draw Cyan | 0 | 1 | 1 |
| 6 | Draw Blue | 0 | 0 | 1 |
| 7 | Draw Magenta | 1 | 0 | 1 |

If you are in this mode, you can draw lines in the eight colors listed above. The following program (found in file COLORLIN.E on your Manual Examples disk) shows the colors available.

```
10     GINIT
20     GRAPHICS ON
30     CLEAR SCREEN
40     MOVE 20,80
50     FOR X=1 TO 7
60       PEN X
70       IDRAW 50,0
80       IMOVE -50,-10
90     NEXT X
91     WAIT 5
100    END
```

## Color Mapped Color

If you are trying to define a complex human interface, you will need more colors and more control over the colors. This is possible after you turn on the color map. To do so, execute:

```
PLOTTER IS CRT,"INTERNAL";COLOR MAP
```

**Default Colors.** If you do not modify the color map, the colors selected by the PEN and AREA PEN values depend on the default color map values. These values are shown in the following table:

**Default Color Map and Pen Values**

| Pen Value | Color |
|---|---|
| 0 | Black |
| 1 | White |
| 2 | Red |
| 3 | Yellow |
| 4 | Green |
| 5 | Cyan |
| 6 | Blue |
| 7 | Magenta |
| 8 | Black |
| 9 | Olive Green |
| 10 | Aqua |
| 11 | Royal Blue |
| 12 | Maroon |
| 13 | Brick Red |
| 14 | Orange |
| 15 | Brown |

Pens 0–7 of the default color map are the same as in non-color map mode. The upper 8 colors (8 through 15) were selected by a graphic designer to produce graphs and charts for business applications. The colors are:

- Maroon, Brick Red, Orange, and Brown (warm colors).
- Black, Olive Green, Aqua, Royal Blue (cool colors).

These colors are one designer's idea of appropriate colors for business charts and graphs. They were chosen to avoid clashing with each other.

**Changing Default Colors.** The SET PEN statement is used to customize the color that each PEN value represents. SET PEN supports two color models, the RGB (Red, Green, Blue) model and the HSL (Hue, Saturation, Luminosity) model. Since the color models are dynamically interactive, it is much easier to understand them by experimenting with them.

You can think of the RGB model as mixing the output of three light sources (one each for red, green, and blue). The parameters in the model specify the intensity of each of the light sources. The RGB model is accessed through the secondary keyword INTENSITY used with the SET PEN statements. The values are normalized (range from 0 through 1). Thus,

```
SET PEN 0 INTENSITY 0.7, 0.7, 0.7
```

sets pen 0 (the background color) to approximately a 70% gray value. Whenever all the guns are set to the same intensity, a gray value is obtained. The parameters for the INTENSITY mode of SET PEN are in the same order they appear in the name of the model, red, green, and blue.

When using an EGA system, each primary color (red, green, and blue) can be displayed at four distinct levels:

- off
- 1/3 on
- 2/3 on
- full on

Therefore, each PEN may be set to one of 64 distinct colors.

The HSL model is closer to the intuitive model of color used by artists, and is very effective for interactive color selection. The three parameters represent hue (the pure color to be worked with), saturation (the ratio of the pure color mixed with white), and luminosity (the brightness-per-unit area). The HSL model is accessed through the SET PEN statement with the secondary keyword COLOR:

SET PEN *Current_Pen* COLOR *Hue, Saturation, Luminosity*

*Hue, Saturation*, and *Luminosity* are normalized to values from 0 to 1.

## Fill Colors

In either color-mapped or non-color-mapped mode, areas may be filled with a PEN color by first selecting that PEN with an AREA PEN statement. Filling is specified by using the secondary keyword FILL in any of the following statements:

IPLOT PLOT POLYGON

RECTANGLE RPLOT SYMBOL

It is possible to fill areas with other shades. These tones are achieved through dithering. Dithering produces different shades by combining dots of the eight colors described earlier. The screen is divided into 4-by-4 cells, and patterns of dots within the cells are turned on to match, as closely as possible, the color you specify. Dithered colors are defined with the AREA COLOR and AREA INTENSITY statements using the RGB or HSL models described in the previous section.

11

# External Graphics Displays and Plotters

## Specifying a Plotter

In the previous chapter you saw program listings containing a line with a PLOTTER IS statement:

```
PLOTTER IS 3,"INTERNAL"
```

This caused the computer to activate the internal CRT graphics raster as the plotting device, and thus all subsequent commands were directed to the screen. If you want a plotter to be the output device, only the PLOTTER IS statement needs to be changed. If your plotter is at interface select code 7 and address 5 (the factory settings), the modified statement would be:

```
PLOTTER IS 705,"HPGL"
```

"HPGL" stands for Hewlett-Packard Graphics Language, and it is the low-level language which the plotters actually speak behind the scenes. More about this later.

There are some limitations, though. If you are doing an operation on one plotting device and attempt to send the plot to another device which does not support that operation, it won't work.

For example: area fills, which are valid operations for the CRT, are not available on plotters. Color map operations, which are valid for a color display, are not valid on a plotter. Erasing lines can be done on the CRT, but, naturally, not on a hard-copy plotter. HPGL commands will be interpreted correctly by a hard-copy plotter, but not by the CRT.

## Using a Shared Printer or Plotter

Use of special Shared Resource Manager (SRM) directories called spooler directories allows you to access a shared plotter. Setting up a spooler directory is explained in the *Shared Resource Management System Manager's Guide*. The following examples assume that the spooler directory PL has been created at the root of the SRM directory structure.

```
MSI ":REMOTE"
```

Include in your plotting program:

```
CREATE BDAT "PL/plot-file",1
PLOTTER IS "PL/plot-file"
  .
  .
  .
PLOTTER IS 3,"INTERNAL"
```

The PLOTTER IS statement only works with BDAT files.

## Dumping Raster Images

In addition to generating a hard-copy plot with a plotter, as described above, you can dump a CRT's raster image to a printer. This method is called a graphics dump or screen dump. It is accomplished by copying data from the frame buffer to a printer to be printed dot for dot.

First, the image must be drawn on a CRT. Since this technique dumps a raster-type image, it prints only dots. Thus, it cannot draw a line, per se, but only the approximation of a line from the screen, made up of dots. The dump device "takes a snapshot" of the graphics screen at some point in time, and doesn't care how the dots came to be turned on or off. Thus, filled areas can be dumped to the printer; indeed, all CRT graphics capabilities (except color) are available.[*]

[*] You can use the CSUB utility "GDUMP_COLORED" to send raster dumps of color graphics to the HP PaintJet color printer. Refer to *Installing and Using HP BASIC in the MS-DOS Environment* for details.

If your printer is an HP 9876, HP 2631G, HP 2671G, HP 2673A, HP LaserJet, HP ThinkJet, HP PaintJet, or any other printer which conforms to the HP Raster Interface Standard, dumping graphics images is easy.* For example:

```
100 DUMP DEVICE IS 26
110 DUMP GRAPHICS
```

or simply,

```
100 DUMP GRAPHICS #26
```

Both of these program segments would take the image in the last specified CRT graphics frame buffer (the internal CRT by default) and send it to the printer at address 26. If no source device is specified, the image is taken from the last active CRT, whether internal or external. The default factory setting for printers is 701. You would probably use the two-statement version in an application where you wish to specify the destination device once, and have it apply to many different DUMP GRAPHICS statements. The one-statement version would probably be used where there are few and isolated DUMP GRAPHICS statements.

DUMP GRAPHICS will also send a graphics display to a printer. If a DUMP DEVICE IS statement has not been executed, the dump device is expected to be at address 701.

If a DUMP GRAPHICS operation is aborted with CLR I/O, the printer may or may not terminate its graphics mode. Sending 75 null characters (ASCII code zero) to a printer such as a HP 9876 terminates its graphics mode. For example:

```
OUTPUT Dump_dev USING "#,K";RPT$(CHR$(0),75)
```

If you want the image to be twice as large in each dimension as the actual screen size, you can specify:

```
100 DUMP DEVICE IS 701,EXPANDED
110 DUMP GRAPHICS
```

This will cause the dumped image to be four times larger than it would be if EXPANDED had not been specified. Each dot is represented by a 2 × 2 square of dots, and the resulting image is rotated 90° clockwise to allow more of the resulting image to fit on the page.

* Refer to your printer owner's manual to determine compatibility.

If you have a printer which does not conform to the HP Raster Interface Standard, all is not lost. It must, however, be capable of printing raster-image bit patterns.

## HPGL

Hewlett-Packard Graphics Language (HPGL) is a low-level language that is understood by all current HP hard-copy plotters. When you specify:

```
PLOTTER IS 705,"HPGL"
```

the plotter specifier “HPGL” notifies the computer that it will be talking with a device which understands HPGL. This causes all the user’s BASIC statements to be converted into HPGL commands and sent to the plotter. HP plotters always receive commands in HPGL.

When you are executing BASIC graphics statements and they are doing operations on an HP plotter, there is nothing preventing you from interspersing your own HPGL commands between the BASIC commands. HPGL commands can be sent to the device with OUTPUT statements; however, the preferred way is to use the GSEND statement. HPGL command sequences are terminated by a linefeed, a semicolon, or an EOI character, which is sent by the HP-IB (Hewlett-Packard Interface Bus) END keyword. Individual commands within a sequence are typically delimited by semicolons. Note that the GSEND statement sends a carriage return/line feed after the specified string.

There are many HPGL commands available, but the exact ones you will be able to use depend on the device itself. Plotters are not the only devices which use HPGL; digitizers and graphics tablets do also. By their nature, however, they use a different subset of commands than plotters do. Following are a few of the more common and useful HPGL commands.

### Controlling Pen Speed

If your plotter pens are getting old, you probably would want to make them draw more slowly to get a better quality line. (There are other factors which can affect line quality. For example, humidity can alter the line quality of a fiber-tipped pen.) To accomplish this, you could have a statement:

```
GSEND "VS10;"
```

“VS” stands for “Velocity Select” and the “10” specifies centimeters per second. Thus, this statement would tell the plotter to draw at a maximum speed of ten centimeters per second. It specifies a maximum speed rather than an only speed, because on short line segments, the pen does not have time to accelerate to the specified speed before the midpoint of the line segment is reached and deceleration must begin. The range and resolution of pen speeds, and default maximum speed depend on the plotter.

## Controlling Pen Force

On the HP 7580 and HP 7585 drafting plotters, you can specify the amount of force pressing the pen tip to the drawing medium. This is useful when matching a pen type (ball-point, fiber-tip, drafting pens, etc.) to a drawing medium (paper, vellum, or mylar, etc.). Again, if a pen is partially dried out, it may help line quality to adjust the pen force.

An example statement is:

```
GSEND "FS3,6;"
```

This statement (Force Select) would specify that pen number 6 should be pressed onto the drawing medium with force number 3. As you can see, the force specifier occurs first, the pen number second. The reason for this is that if you do not specify a pen number, all pens will be affected.

The force number is translated into a force in grams. If, for example, you have an HP 7580A plotter, the force number is converted to force as follows:

| | | |
|---|---|---|
| 1 = 10 grams | 4 = 34 grams | 7 = 58 grams |
| 2 = 18 grams | 5 = 42 grams | 8 = 66 grams |
| 3 = 26 grams | 6 = 50 grams | |

## Selecting Character Sets

Some plotters contain internal character sets which may be much more pleasing to the eye or more appropriate for your application than the character set provided by the BASIC operating system. Through HPGL, you can tell the plotter to use these character sets.

```
GSEND "CS1;"
```

tells the plotter to use character set 1 until further notice. This means, however, that to actually get these characters, you cannot use the LABEL statement in BASIC. This is because the BASIC graphics system generates all its characters as a series of line segments, and the plotter can't tell when it is told to draw a line segment whether it is going to be part of a character or not. Thus, you must use the HPGL label command, LB:

```
GSEND "LBThis is an example string."&CHR$(3)&";"
```

CHR$(3) is the End-of-text or ETX character. It is the default terminator for the LB command. If you wish, you can specify other characters to signal the end of a line of text to label. You use the Define Terminator command:

```
GSEND "DT&;"
```

This statement instructs the plotter to consider the ampersand to be the terminator. Thus, every LB command must have an ampersand as the final character.

**Note** When using a printable ASCII character as the terminator, *it will be labelled* in addition to terminating the LB command. Also, there must be a terminator as the final character in the string to indicate the end of the text, or all subsequent commands will be considered text and not commands; that is, they will merely be labelled, not executed.

## Error Detection

When using HPGL commands, there is always a possibility of making an error. When this occurs, the program should be able to respond in a friendly way, and not just hang then and there. With HPGL, it is possible to interrogate the plotting device and determine the problem. The following statements in an error-trapping routine would determine the type of error that occurred:

```
GSEND "OE;"
ENTER 705;Error
```

After these two statements have executed, the variable Error will contain the number of the most recent error. What the error code means depends on the particular device being used.

This is not by any means an exhaustive list of HPGL commands, but it serves to acquaint you with the concept of using the HPGL language, and the amount of control it gives you over the peripheral device. A thorough understanding of HPGL can only be gotten by combining information from the owner's manual of the particular device you have with actual hands-on experience.

# Part III: Interfacing Techniques

Chapters 12 through 16 cover I/O interfaces and I/O programming techniques. You can use these techniques to output and enter data, and to control peripheral devices from your computer.

12

# Introduction to I/O

This chapter introduces the functions and requirements of interfaces between your computer and its resources. If you are familiar with interfacing concepts, you may want to skim through this chapter and go on to the programming techniques in the chapters that follow.

## Interfacing concepts

This section describes the purpose of an I/O interface, gives an overview of some typical interfaces, and describes the I/O process. Let's begin by defining some terms.

### Terminology

In describing interfaces and the I/O process, several terms are used with special meanings. The term *computer* is defined as the processor, its support hardware, firmware, and operating system, and the BASIC language system. Together, these system elements manage all computer resources. The term *computer resource* is used to describe all of the "data handling" elements of the system. Computer resources include internal memory, CRT display, keyboard, disk drive, and any external devices that are under computer control (printer, plotter, instruments, etc.). These resources are often referred to as *peripheral* devices.

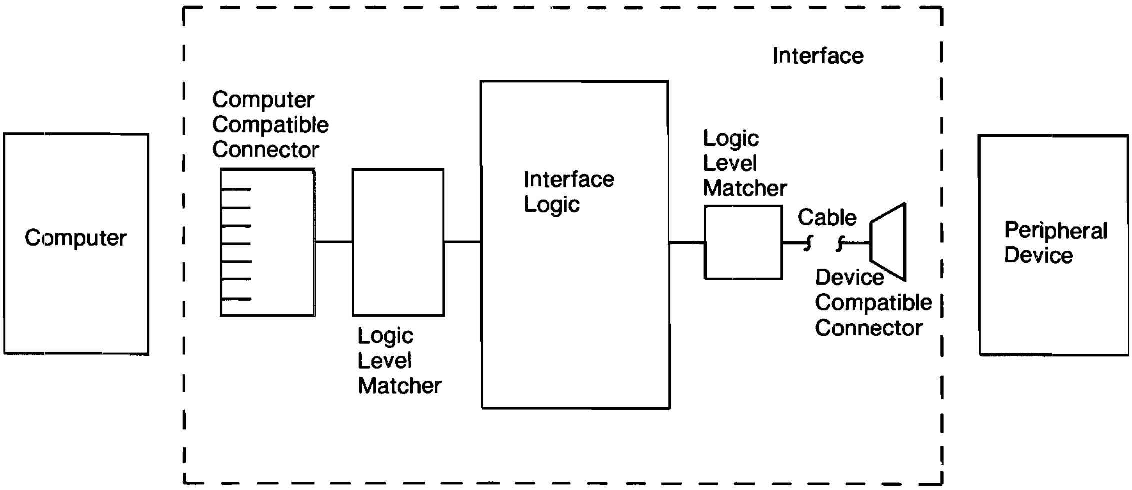
The term *hardware* describes both the electrical connections and the electronic devices that make up the circuits within the computer. Any piece of hardware is an actual physical device. The term *software* describes the user-written, BASIC language programs. *Firmware* refers to the preprogrammed assembly language programs that are invoked by BASIC language statements and commands, or assembly language routines of the operating system. You cannot modify firmware.

The term *I/O* is an acronym for "Input and Output". It refers to the process of copying data to or from the computer's memory.

The term *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 busses. The 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.

## Why Do You Need an Interface?

There are four fundamental requirements that must be met in order for a computer to communicate with a peripheral device. These are: *electrical compatibility*, *mechanical compatibility*, *data compatibility*, and *timing compatibility*. The primary function of an interface is to provide compatibility in these four areas to establish a communication path for data and commands between the computer and its resources. The following block diagram shows how an interface relates to the computer and a peripheral.



**Electrical and Mechanical Compatibility.** Electrical compatibility must be ensured before there is any thought of connecting two devices. The two devices often 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. Most interfaces have cables available that can be connected directly to the device.

**Data Compatibility.** The computer and the peripheral device must agree upon the form and meaning of data before communicating it. Some interfaces format data, but most have little responsibility for matching data formats. The computer must generally 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 the sending and receiving devices must be made.

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 send the next item. This process is known as a "handshake."

**Additional Interface Functions.** Another feature of some interface cards 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.

## Some Standard Interfaces

There are several interface *standards*, which specify compatible signals, data formats, and so forth. Let's look at some typical kinds of interfaces that you will likely encounter. These interfaces are discussed in greater detail in chapter 16.

**The HP-IB Interface.** The HP-IB interface is Hewlett-Packard's implementation of the IEEE-488 1978 Standard Digital Interface for Programmable Instrumentation. The acronym HP-IB stands for "Hewlett-Packard Interface Bus," and is often referred to as the "bus."

The HP-IB interface fulfills all four compatibility requirements (hardware, electrical, data, and timing) with no additional modification. All you need to do is connect the interface cable to the desired HP-IB.

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 several ways. The devices on the bus can be configured as senders or receivers of data and control messages, depending on their capabilities.

**The RS-232 Serial Interface.** The serial interface changes 8-bit parallel data into 8-bit-serial information and transmits the data through a two-wire cable. Data is received in this serial format and is then converted back to parallel data. This use of two-wire cable makes it more economical to transmit data over long distances than would be the case if eight individual lines were used.

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 communicating with simple devices.

**The GPIO Interface.** The GPIO (General Purpose Input/Output) interface provides the most flexibility of all the interfaces. It consists of 16 output data lines, 16 input data lines, two handshake lines, and other assorted control lines. Data is transmitted using programmable handshake conventions and logic sense.

## The I/O Process

The I/O process begins when the computer encounters an I/O statement in a program. The computer first determines the type of I/O statement to be executed (such as ENTER USING, OUTPUT, 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 are device selectors, string variable names, and path names.

For example, before executing an OUTPUT statement, the computer first evaluates the parameter which specifies the destination resource. The source parameter of an ENTER statement is evaluated in the same manner.

```
OUTPUT Dest_parameter;Source_item
ENTER Source_parameter;Dest_item
```

**Registers.** The computer must often read certain memory locations to determine which firmware routines will be called to execute the I/O procedure. The contents of these locations, known as registers, store parameters to be used and the type of interface involved in the operation.

An example of register usage by firmware occurs during output to the CRT. Characters output to this device are displayed beginning at the current screen coordinates. After the computer has evaluated the first expression in the source-item list, it must determine where to begin displaying data on the screen. Two memory locations are dedicated to storing the "X" and "Y" screen coordinates. The firmware determines these coordinates and begins copying the data to the corresponding locations in display memory.

**Data Handshake.** Each byte (or word) of data is transferred with a procedure known as data-transfer 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 received 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 is to be transferred.

## Directing Data Flow

As described in the previous section, data can be moved between computer memory and several resources, including:

- Computer memory (string variables in memory).
- Internal and external devices.
- Mass storage files.
- Buffers.

This section describes how string variables and devices are specified 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 resources. String variables are specified with their names, while devices can be specified with either their device selector or with a new data type known as an I/O path name. This section describes how to specify these resources in OUTPUT or ENTER statements.

**String-Variable Names.** Data is moved to and from string variables by specifying the variable's name in an OUTPUT or ENTER statement. Examples of each are shown in the following program (found in file OUTENTER on your Manual Examples disk).

```
100  DIM To_dest$[80],From_source$[80]
110  DIM Data_out$[80]
120  !
130  From_source$="Source data"
140  Data_out$="OUTPUT data"
150  !
160  PRINTER IS 1
170  PRINT "To_dest$ before OUTPUT= ";To_dest$
180  PRINT
190  !
200  OUTPUT To_dest$;Data_out$;
210  PRINT "To_dest$ after OUTPUT= ";To_dest$
220  PRINT
230  !
240  ENTER From_source$;To_dest$
250  PRINT "To_dest$ after ENTER= ";To_dest$
260  PRINT
270  !
280  END
```

Printed results from the program are:

```
To_dest$ before OUTPUT=
To_dest$ after OUTPUT= OUTPUT data

To_dest$ after ENTER= Source data
```

**Device Selectors.** Devices include the built-in CRT and keyboard, plus 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. The internal devices are accessed with the following permanently assigned interface select codes:

| | |
|---|---|
| Crt Display | 1 |
| Keyboard | 2 |
| Built-in HP-IB | 7 |

Other optional interfaces have select codes that you can set by means of switches on the interface card. These interfaces cannot use select codes 1 through 7; the valid range is 8 through 31. The following settings on optional interfaces have been made at the factory, but can be reset to any unique select code between 8 and 31. Refer to the interface instruction manual for further information.

GPIO 28

SRM 21

Examples of using interface select codes to access devices are shown below.

```
OUTPUT 1;"Data to CRT"
ENTER 1;Crt_line$

Int_sel_code=12
OUTPUT Int_sel_code;String$&"Expression",Num_expression
ENTER Int_select_code;Str_variable$,Num_variable

Number=2
ENTER Number+7;Serial_data$
OUTPUT 11-Number;"Data to serial card"
```

The device selector can be any numeric expression that rounds to an integer in the range 1 through 32 (32 is a pseudo select code used as a device selector for parity, cache, and float registers). If the interface select code specifies an HP-IB interface, additional information must be specified to access a particular HP-IB device, since more than one device can be connected to the computer through HP-IB interfaces.

**HP-IB Device Selectors.** Each device on the HP-IB interface has a primary address by which it is uniquely identified. Each address must be unique so that only one device is accessed when an address is specified. The device selector is therefore a combination of the interface select code and the devices address. Two examples are shown below.

To access the device on:

Interface select code 7 at primary address 01, use device selector 701.

Interface select code 10 at primary address 13, use device selector 1013.

**I/O Path Names.** All data entered into and output from the computer is moved through the "I/O path." An I/O path consists of the hardware and operating system firmware used to carry out this moving process. When a string variable or device selector is specified in an ENTER or OUTPUT statement, the operating system first evaluates the expression that specifies a resource, and then chooses the corresponding default I/O path through which data will be moved.

The I/O paths to devices and mass storage files can be assigned special names; I/O paths to string variables  can only be assigned names if the variable is declared as a buffer. Assigning names to I/O paths provides improvements in performance and additional capabilities over using device selectors.

## Assigning I/O Path Names

An I/O path name is a new data type that can be assigned to either a device or a data file on a mass storage device. Any valid name preceded by the "@" character can be used.

**Note** A name is a combination of 1 to 15 characters, beginning with an upper case alphabetic character or one of the characters CHR$(161) through CHR$(254) and followed by up to 14 lower case alphanumeric characters, the underscore character (_), or the characters CHR$(161) through CHR$(254).

The following examples show you how this is done.

```
ASSIGN @Display TO 1
ASSIGN @Printer TO 26
ASSIGN @Serial TO 9
ASSIGN @Gpio TO 12
```

Now you could use the I/O path names instead of the device selectors to specify the resource with which the communication is to take place.

```
OUTPUT @Display;"Display message"
OUTPUT @Printer;"Message to the printer"
ENTER @Serial;Variable,Variable$
ENTER @Gpio;Wordl,Word2
```

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).

Attempting to use an I/O path name that does not appear in any program line results in error 910 ("Identifier not found in this context"). This error message indicates that memory space has not been allocated for the variable.

Attempting to use an I/O path name that does appear in an ASSIGN statement in the program, but which has not yet been executed results in error 177 ("Undefined I/O path name"). This error indicates that memory space has been allocated, but no valid information has been placed into the variable since the I/O path name has not yet been assigned to a resource.

**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 used. The first action is that the I/O path name to the device is implicitly closed. A new assignment is then made as though the first never existed.

```
100  ASSIGN @Printer TO 1 !Initial assignment.
110  OUTPUT @Printer;"Data1"
120  !
130  ASSIGN @Printer TO 701 !2nd ASSIGN closes the 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 the HP-IB device 701. Since the program was paused (which maintains the program context), the I/O path name @Printer can be used in an I/O statement or reassigned to another resource from the keyboard.

**Closing I/O path names.** A second use of the ASSIGN statement is to explicitly close the name assigned to an I/O path. Examples of statements that close path names are as follows.

```
ASSIGN @Printer TO *
ASSIGN @Serial TO *
ASSIGN @Gpio 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.

# 13

# Outputting and Entering Data

This chapter discusses two very powerful BASIC statements that can be used in a wide variety of I/O applications. You can use the OUTPUT statement to output data to a peripheral, exercising considerable control over the data format. The ENTER statement allows you to enter data of different formats into your computer from a peripheral device.

## Outputting Data

There are two general types of output operations. The first type, known as "free-field outputs," use the computer'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. Let's look at free-field output first.

### Free-Field Outputs

Free-field outputs are invoked when the following types of OUTPUT statements are executed.

```
OUTPUT @Device;3.14*Radius^2

OUTPUT Printer;"String data";Num_1

OUTPUT 9;Test,Score,Student$

OUTPUT Escape_code$;CHR$(27)&"&A1S";
```

[*] The ASCII representation described briefly in the preceding chapter is the default data representation used when communicating with with devices; however, the internal representation can also be used. See the "I/O Path Attributes" section in chapter 14 for further details.

**The Free-Field Convention.** The term "free-field" refers to the number of characters used to represent a data item. During free-field outputs, 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 in this chapter). 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 the value 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. Some examples follow:

```
 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.** In the standard format no leading or trailing spaces are output with the string's characters.*

```
String characters.
No leading or trailing spaces.
```

* This statement describes the FORMAT ON attribute (ASCII data representation). When sending data with the FORMAT OFF attribute, however, the internal representation of string data is used; for strings, the data consists of a four-byte length header that contains the number of characters in the string, followed by the string characters. With FORMAT ON, there is *no* length header; only the ASCII string characters are sent.

**Item Separators and Terminators.** Data items are output one byte (or word) 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 |
|---|---|---|---|---|---|---|
| | optional | | optional | | | optional |

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.

**Note**

In the examples in this chapter, “EOL” is used to represent an end-of-line sequence, “CR” is used for a carriage-return character, and “LF” for a line-feed character.

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

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| I | t | e | m | CR | LF | – | 1 | 2 | 3 | 4 | EOL |

(The default EOL sequence is a CR/LF.)

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

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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| – | 1 | 2 | 3 | 4 | , | I | t | e | m | EOL |

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",
```

| | | | | | |
|---|---|---|---|---|---|
| I | t | e | m | CR | LF |

or:

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

| | | | | | |
|---|---|---|---|---|---|
| – | 1 | 2 | 3 | 4 | , |

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

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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| I | t | e | m | 1 | I | t | e | m | 2 | EOL |

or:

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

| | | | | | | |
|---|---|---|---|---|---|---|
| – | 1 | 2 | – | 3 | 4 | EOL |

If a semicolon separator follows the last item in the list, the EOL sequence and item terminators are suppressed.

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

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| I | t | e | m | 1 | I | t | e | m | 2 |

(Neither of the item terminators 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.)

```
100     OPTION BASE 1
110     DIM Array(2,3)
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
```

The following is the 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.

```
100     OPTION BASE 1
110     DIM Array$(2,3)[2]
120     FOR Row=1 TO 2
130       FOR Column=1 TO 3
140         Array$(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
```

The following is the 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 | | | | | | | | | | | | | | | | | | |

A pad byte may be sent following the last character of the EOL sequence when using an I/O path that possesses the WORD attribute. See "I/O Path Attributes" in chapter 14 for further information.

**Changing the EOL Sequence (Requires IO).** 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 interface-dependent END indication. When the IO binary is loaded, it is also possible to define your own special EOL sequences that include sending special characters, sending an interface-dependent END indication, and delaying a specified amount of time after sending the EOL sequence.

In order to define non-default EOL sequences to be sent by the OUTPUT statement, an I/O path must be used. The EOL sequence is specified in one of the ASSIGN statements which describe the I/O path. An example is as follows.

```
ASSIGN @Device TO 12;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. The characters are put into the output data before any conversion is performed (if CONVERT OUT is in effect).

If END is included in the EOL attribute, an interface-dependent "END" indication is sent with (or after) the last character of the EOL sequence. However, if no EOL sequence is sent, the END indication is also suppressed. The following statement shows an example of defining the EOL sequence to include an END indication.

```
ASSIGN @Device TO 20;EOL CHR$(13)&CHR$(10) END
```

With the HP-IB Interface, the END indication is an End-or-Identify message (EOI) sent with the last EOL character. Refer to "The HP-IB Interface" in chapter 16 for further details.

If DELAY is included, the system delays the specified number of seconds (after sending the last EOL character and/or END indication) before executing any subsequent BASIC statement.

```
ASSIGN @Device;EOL CHR$(13)&CHR$(10) DELAY 0.1
```

This parameter is useful when using slower devices which the computer can "overrun" if data are sent as rapidly as the computer can send them. For example, a printer connected to the computer through a serial interface set to operate at 300 baud might require a delay after receiving a CR character to allow the carriage to return before sending further characters.

The default EOL sequence is a CR and LF sent with no END indication and no delay; 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 Free-Field OUTPUT

The secondary keyword END may be optionally specified following the last source-item expression in a free-field 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).

Let's look at some examples:

```
ASSIGN @Gpio TO 12
```

```
OUTPUT @Gpio;-10,END
```

|   |   |   |   |
|---|---|---|---|
| – | 1 | 0 | , |

(Item terminator, but no EOL sequence, is sent.)

```
OUTPUT @Gpio;-10;END
OUTPUT @Gpio;-10 END
```

|   |   |   |
|---|---|---|
| – | 1 | 0 |

(Neither item terminator nor EOL sequence is sent.)

```
OUTPUT @Gpio;"AB",END
```

|   |   |   |   |
|---|---|---|---|
| A | B | CR | LF |

(Item terminator, but no EOL sequence, is sent.)

```
OUTPUT @Gpio;"AB";END
OUTPUT @Gpio;"AB" END
```

|   |   |
|---|---|
| A | B |

(Neither item terminator nor EOL sequence is sent.)

```
OUTPUT @Gpio
```

| |
|---|
| EOL |

(Only the EOL sequence is sent.)

```
OUTPUT @Gpio;END
OUTPUT @Gpio;"" END
```

(No EOL sequence is sent.)

**Note**

BASIC defines additional action when END is specified in a free-field OUTPUT statement directed to the HP-IB interface.

With the HP-IB interface, END has the additional function of sending the End-or-Identify signal (EOI) with the last data byte of the last source item; *however, if no data are sent from the last source item, EOI is not sent.* For further description of the EOI signal, see "The HP-IB Interface" in chapter 16. Here are some examples:

```
ASSIGN @Device TO 701
```

```
OUTPUT @Device;-10,END
```

| | | | |
|---|---|---|---|
| – | 1 | 0 | , |

EOI is sent with the last character (numeric item terminator).

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

| | |
|---|---|
| A | B |

EOI is sent with the last character of the item.

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

Neither EOL sequence nor EOI is sent, since no data is sent.

## Outputs that Use Images

The free-field form of the OUTPUT statement is very convenient to use. However, there may be times when the data output by the free-field convention is not compatible with the data required by the receiving device.

Several instances for which you might need to format outputs are: special control characters are to be output; the EOL sequence (carriage-return and line-feed) needs to be suppressed; or the exponent of a number must have only one digit. This section shows you how to use image specifiers to create your own, unique data representations for output operations.

**The OUTPUT USING Statement.** When this form of the OUTPUT statement is used, the data is output according to the format image referenced by the "USING" secondary keyword. This image consists of one or more individual image specifiers which describe the type and number of data bytes (or words) to be output. The image can be either a string literal, a string variable, or the line label or number of an IMAGE statement. Examples of these four possibilities are listed below.

```
100  OUTPUT 1 USING "6A,SDDD.DDD,3X";" K= ",123.45

100  Image_str$="6A,SDDD.DDD,3X"
110  OUTPUT CRT USING Image_str$;" K= ";123.45

100  OUTPUT CRT USING Image_stmt;" K= ";123.45
110  Image_stmt: IMAGE 6A,SDDD.DDD,3X

100  OUTPUT 1 USING 110;" K= ";123.45
110  IMAGE 6A,SDDD.DDD,3X
```

## Images

Images are used to specify the format of data during I/O operations. Each image consists of groups of individual image (or "field") specifiers, such as 6A, SDDD.DDD, and 3X in the preceding examples. Each of these field specifiers describe one of the following things:

- It describes the desired format of one item in the source list. (For instance, 6A specifies that a string item is to be output in a "6-character Alpha" field. SDDD.DDD specifies that a numeric item is to be output with Sign, 3 Decimal digits preceding the decimal point, followed by 3 Decimal digits following the decimal point.)
- It specifies that special character(s) are to be output. (For instance, 3X specifies that 3 spaces are to be output.) There is no corresponding item in the source list.

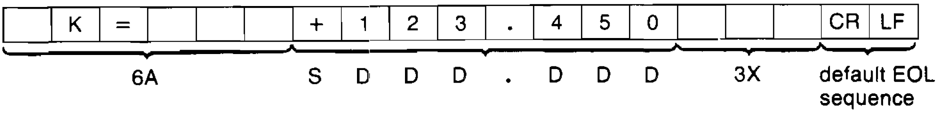
Thus, you can think of the image list as either a precise format description or as a procedure. It is convenient to talk about the image list as a procedure for the purpose of explaining how this type of OUTPUT statement is executed.

Again, each image list consists of images that describe the format of data items 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. The following example steps through exactly how BASIC executes all of the preceding equivalent statements.

**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:



The process follows the following steps:

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, six alphanumeric characters taken from the first item in the source list are to be output.
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.
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.
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.
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.
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 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 one exponent digit is output. |
| ESZZZ | Same as "E" but three exponent digits are output. |

### Sign, Digit, Radix and Exponent Specifiers (continued)

| Image Specifier | Meaning |
|---|---|
| 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). |

Now let's look at some examples using numeric image specifiers.

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

| | | | | |
|---|---|---|---|---|
| − | 1 | 2 | 4 | EOL |

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

| | | |
|---|---|---|
| − | 1 | EOL |

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

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | . | 6 | 8 | EOL |

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

| | | | |
|---|---|---|---|
| 0 | . | 4 | EOL |

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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | . | E | + | 0 | 3 | EOL |

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

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 0 | . | 0 | 0 | E | – | 0 | 5 | EOL |

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

| | | | | |
|---|---|---|---|---|
| 1 | 2 | . | 4 | EOL |

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

| | | | | | | |
|---|---|---|---|---|---|---|
| – | 1 | 2 | . | 4 | 5 | EOL |

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

| | | | | | | |
|---|---|---|---|---|---|---|
| | | 2 | . | 0 | 9 | EOL |

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

| | | | | | |
|---|---|---|---|---|---|
| + | 0 | . | 4 | 9 | EOL |

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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| – | 2 | 3 | . | 5 | 0 | E | – | 0 | 1 | EOL |

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

| | | | | |
|---|---|---|---|---|
| * | 2 | . | 6 | EOL |

**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 that 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. |

Here are some examples of OUTPUT statements with string specifiers:

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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| C | h | a | r | a | c | t | e | EOL |

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

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | B | L | i | t | e | r | a | l | EOL |

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

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | H | e | l | l | o | | | | EOL |

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

| | | | | | |
|---|---|---|---|---|---|
| | | | H | e | EOL |

**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.<br>If either an I/O path name with the BYTE attribute (refer to "I/O Path Attributes" in chapter 14) or a device selector is used to access an 8-bit interface, two bytes will be output; the first byte is most significant. 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 16-bit word is output in a single handshake operation.<br>If an I/O path name with the WORD attribute is used to access a 16-bit interface, a pad byte, CHR$(0), is output whenever necessary to achieve alignment on a word boundary.<br>If the destination is a BDAT or HPUX file, string variable, or buffer, the WORD attribute is ignored and all data are sent as bytes; however, pad byte(s) will also be output whenever necessary to achieve alignment on a word boundary. The pad byte may be changed by using the CONVERT attribute (refer to "I/O Path Attributes" in chapter 14). |
| Y | Like W, except that no pad bytes are output to achieve alignment on a word boundary. If an I/O path with the BYTE attribute is used to access a 16-bit interface, the attribute is not overridden (as with the W specifier). |

Some binary examples follow:

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

| | | | |
|---|---|---|---|
| A | B | C | EOL |

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

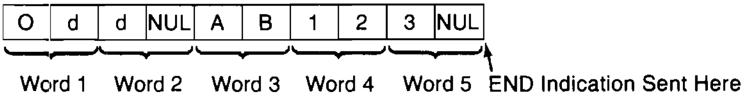
| |
|---|
| CR |

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

| | | |
|---|---|---|
| A | B | EOL |

For this example, assume that @Device possesses the WORD attribute and that the EOL sequence consists of the characters "123" with an END indication.

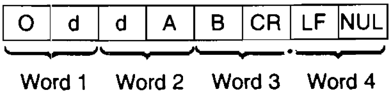
```
OUTPUT @Device USING "K,W";"Odd",256*65+66
```



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| O | d | d | NUL | A | B | 1 | 2 | 3 | NUL |

For this example, assume that @Device possesses the WORD attribute and that the EOL sequence is the default (CR/LF).

```
OUTPUT @Device USING "K,Y";"Odd",256*65+66
```



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| O | d | d | A | B | CR | LF | NUL |

**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. |

Here are some examples:

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

| | | | | | | |
|---|---|---|---|---|---|---|
| M | | | | | A | EOL |

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

| | |
|---|---|
| (50 spaces) | EOL |

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

| | | | |
|---|---|---|---|
| FF | CR | LF | EOL |

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

| | | |
|---|---|---|
| CR | LF | EOL |

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

**Termination Specifiers**

| Image Specifier | Meaning |
|---|---|
| L | Specifies that the current end-of-line sequence is to be output. The default EOL characters are CR and LF; see "Changing the EOL Sequence" for details on how to re-define these characters. If the destination is an I/O path name with the WORD attribute, a pad byte will be output after each EOL sequence when necessary to achieve word alignment. |
| # | 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). |

Let's look at some examples of termination images.

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

| | | | | | |
|---|---|---|---|---|---|
| D | a | t | a | EOL | EOL |

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

| | | | |
|---|---|---|---|
| D | a | t | a |

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

| |
|---|
| FF |

(Note that CHR$(12) is the form feed character.)

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

| | | | | |
|---|---|---|---|---|
| D | a | t | a | CR |

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

| | | | | | |
|---|---|---|---|---|---|
| EOL | D | a | t | a | LF |

## Additional Image Features

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

**Repeat Factors.** Many of the specifiers can be repeated without having to explicitly list the specifier as many times as it is to be repeated. For instance, to a character field of 15 characters, you do not need to use "AAAAAAAAAAAAAAA"; instead, you merely specify the number of times that the specifier is to be repeated in front of the image ("15A"). The following table identifies which specifiers can be repeated and which cannot.

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

Let's look at some examples using repeat factors:

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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 3 | 2 | 8 | . | 0 | 3 | 0 | EOL |

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

| | | | | | | |
|---|---|---|---|---|---|---|
| D | a | t | a | | b | EOL |

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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | D | a | EOL |

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

| | | | | | | |
|---|---|---|---|---|---|---|
| EOL | EOL | D | a | t | a | EOL |

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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| T | h | e | | E | n | d | | FF | FF | EOL |

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

| | | | | |
|---|---|---|---|---|
| CR | LF | CR | LF | EOL |

**Image Re-Use.** If the number of items in the source list exceeds the number of matching specifiers in the image list, the computer attempts to re-use the image(s) beginning with the first image.

```
110    ASSIGN @Device TO CRT
120    Num_1=1
130    Num_2=2
140    !
150    OUTPUT @Device USING "K";Num_1,"Data_1",Num_2,"Data_2"
160    OUTPUT @Device USING "K,/";Num_1,"Data_1",Num_2,"Data_2"
170    END
```

The following output will be displayed:

```
1Data_12Data_2
1
Data_1
2
Data_2
```

Since the "K" specifier can be used with both numeric and string data, the above OUTPUT statements can re-use the image list for all items in the source list. If any item cannot be output using the corresponding image item, an error results. In the following example, "Error 100 in 150" occurs due to data mismatch.

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

**Nested Images.** Another convenient capability of images is that they can be nested within parentheses. The entire image list within the parentheses will be used the number of times specified by the repeat factor preceding the first parenthesis. The following program is an example of this feature.

```
100   ASSIGN @Device TO 701
110   !
120   OUTPUT @Device USING "3(B),X,DD,X,DD";65,66,67,68,69
130   END
```

The following output will result:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | B | C | | 6 | 8 | | 6 | 9 | EOL |

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

## END with OUTPUTs that Use Images

Using the optional secondary keyword END in an OUTPUT statement that uses an image produces results which differ from those of using END in a free-field 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.

Let's look at some examples:

```
Device=12

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

| | | | |
|---|---|---|---|
| A | B | C | EOL |

(The EOL sequence is not suppressed.)

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

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EOL | CR | LF | L | i | t | e | r | a | l | | FF | EOL |

In this case, specifiers that require no source-item expressions are used to generate characters for the output; there are no source expressions. The EOL sequence is output after all specifiers have been used to output their respective characters. Compare this action to that shown in the next example.

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

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| EOL | CR | LF | L | i | t | e | r | a | l | | FF |

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

*As previously mentioned regarding free-field OUTPUT, the END secondary keyword has been defined to produce additional action when included in an OUTPUT statement directed to an HP-IB interface.*

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

Some examples:

```
ASSIGN @Device TO 701

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

| | | | | |
|---|---|---|---|---|
| D | a | t | a | EOL |

(EOI is sent with last character of the EOL sequence.)

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

| | | | |
|---|---|---|---|
| D | a | t | a |

(EOI is sent with the last character.)

EOI is sent with the last character of the last source item when the EOL sequence is suppressed, because the last source item contained data which was used in the output.

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

| | | | |
|---|---|---|---|
| D | a | t | a |

The EOI was not sent in either case, since no data were sent from the last source item *and* the EOL sequence was suppressed.

## Entering Data

The ENTER statement lets you enter data from devices into the computer. Many of the concepts discussed in the previous section regarding the OUTPUT statement are applicable to the ENTER statement as well. However, entering data can require more programming skill than outputting data because of the many ways that data can be represented in external devices. The ENTER statement allows you to covert the data to be entered to a usable format. More about this later.

As with OUTPUT, the ENTER statement allows you to enter data either with a "free-field" format or with precise control using an IMAGE. Let's look at free-field ENTER first.

### Free-Field Enters

Executing the free-field form of the ENTER invokes conventions which are the "converse" of those used with the free-field OUTPUT statement. In other words, data output using the free-field form of the OUTPUT statement can be readily entered using the free-field ENTER statement; no explicit image specifiers are required. The following statements exemplify this form of the ENTER statement.

```
100 ENTER @Voltmeter;Reading

100 ENTER 724;Readings(*)

100 ENTER From_string$;Average,Student_name$

100 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:

```
ENTER @From_a_device;N1,N2,N3     ! Enters into three variables.

ENTER @From_a_device;N1;N2;N3     ! Equivalent to first statement.

ENTER @From_a_device;N1,N2,N3,    ! Trailing comma causes an error.
```

**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, CRT, and keyboard interfaces. EOI termination is further described in the next sections.

When using an I/O path that possesses the WORD attribute, an additional byte may be entered (but ignored). Refer to "I/O Path Attributes" in chapter 14 for further information.

**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 the computer.

| Mantissa sign | Mantissa digit(s) | E | Exponent sign | Exponent digit(s) | Terminator (character or END indication) |
|---|---|---|---|---|---|
| Optional | At least one digit is required | Optional | | | Required |

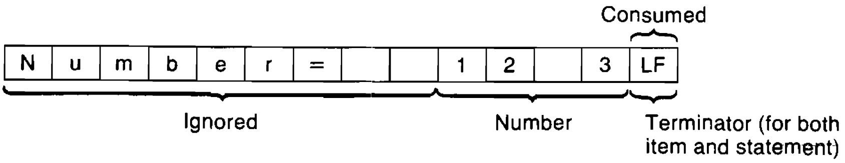
Numeric characters include decimal digits "0" through "9" and the characters ".", "+", "−", "E", and "e". These last five characters must occur in meaningful positions in the data stream to be considered numeric characters; if any of them occurs in a position in which it cannot be considered part of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

**Number Building Rules.** The following *rules* are used by the number builder to construct numbers from incoming streams of ASCII numeric characters.

**Rule 1:** *All leading non-numerics are ignored; all leading and imbedded spaces are ignored.*

For example:

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

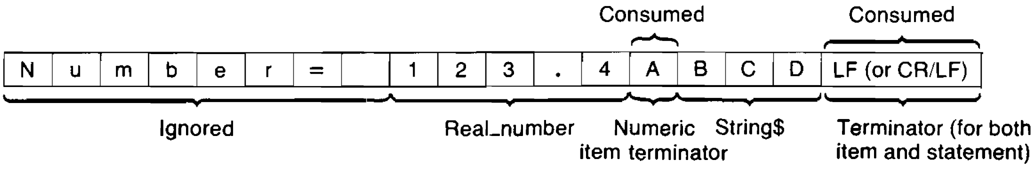


The result of entering the preceding data with the given ENTER statement is that Number receives a value of 123. The line-feed (statement terminator) is *required* since Number is the last item in the destination list.

**Rule 2:** *Trailing non-numeric characters terminate entry into a numeric variable, and the terminating characters (of both string and numeric items) are "consumed."* (A "consumed" character is used to terminate an item, but it is not itself entered into the variable. An "ignored" character is entered but is not used.)

For example:

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

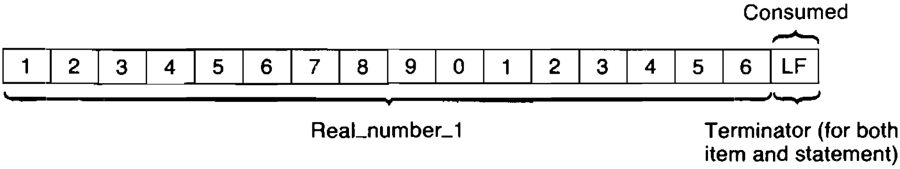

Consumed
Consumed
N u m b e r = 1 2 3 . 4 A B C D LF (or CR/LF)
Ignored
Real_number
Numeric item terminator
String$
Terminator (for both item and statement)

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.

**Rule 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.

For example:

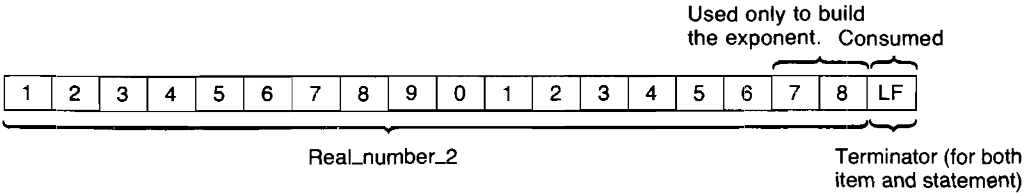
```
ENTER @Device;Real_number_1
```


Consumed
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 LF
Real_number_1
Terminator (for both item and statement)

The result of entering the preceding data with the given ENTER statement is that Real_number_1 receives the value 1.234567890123460 E+15. In order to see all digits, use a statement like this: `OUTPUT CRT USING "D.15DESZZ";Real_number_1.`

For example:

```
ENTER @Device;Real_number_2
```

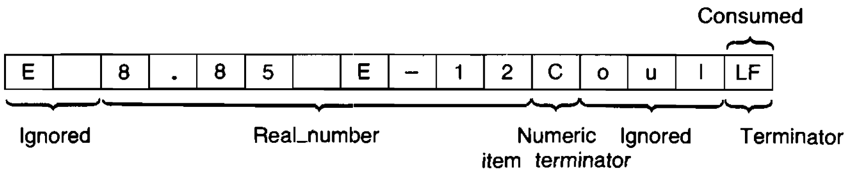

Used only to build the exponent.
Consumed
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 LF
Real_number_2
Terminator (for both item and statement)

The result of entering the preceding data with the given ENTER statement is that Real_number_2 receives the value 1.234567890123460 E+17.

**Rule 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.

For example:

```
ENTER @Device;Real_number
```

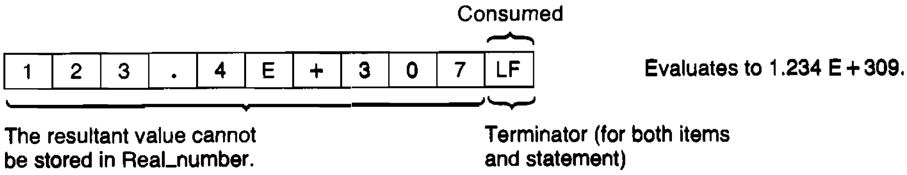


The result of entering the preceding data with the given ENTER statement is that Real_number receives a value of 8.85 E–12. The character "C" terminates entry into Real_number, and the characters "oul" are entered (but ignored) in search of the required line-feed statement terminator. If the character "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.

**Rule 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.

For example:

```
ENTER @Device;Real_number
```


Consumed
1
2
3
.
4
E
+
3
0
7
LF
Evaluates to 1.234 E+309.
The resultant value cannot be stored in Real_number.
Terminator (for both items and statement)

The data is entered but evaluates to a number outside the range of REAL numbers. Consequently, error 19 is reported, and the variable Real_number retains its former value.

**Rule 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). Termination of free-field ENTER statements is described later in this chapter.

**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).

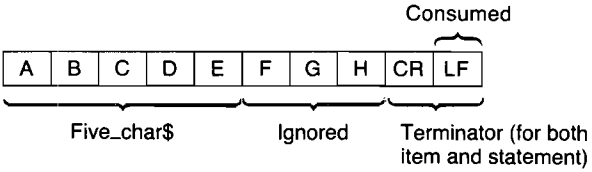
String-item terminator characters are either a line-feed (LF) or a carriage-return followed by a line-feed (CR/LF). As with numeric-item terminators characters, these characters are not entered into the string variable (during free-field enters); they are "lost" when they terminate the entry. The EOI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

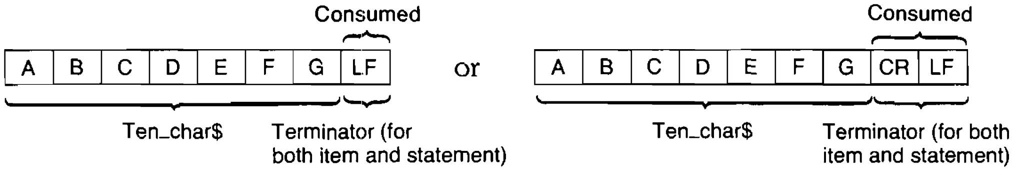
The following statements and resultant variable contents illustrate the first two conditions; the next section describes termination by EOI. Assume that the string variables Five_char$ and Ten_char$ are dimensioned to lengths of 5 and 10 characters, respectively.

```
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). This happens because Five_char$ is the last variable at the end of the ENTER statement.

```
ENTER @Device;Ten_char$
```


Consumed
A
B
C
D
E
F
G
LF
Ten_char$
Terminator (for both item and statement)
or
Consumed
A
B
C
D
E
F
G
CR
LF
Ten_char$
Terminator (for both item and statement)

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.

**Note**

Keep in mind the following points when entering free-field string data:

1. Carriage returns are not consumed unless the line feed would also fit into the string.
2. No "scan ahead to the terminator" is performed except for the last variable in an ENTER statement.

To avoid problems, always dimension strings which will be used in this manner to be at least two characters longer than the longest data item which might be read into them. This will allow room for the carriage return/line feed sequence to be read and consumed.

## 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, or 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.

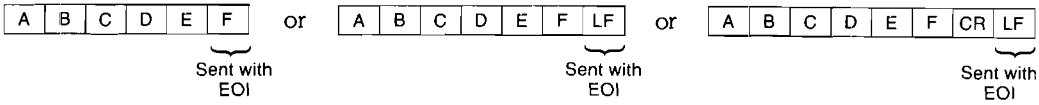
One case in which this termination condition may not be obvious can occur while entering string data. If the last variable in the destination list is a string *and* the dimensioned length of the string has been reached *before* a terminator is received, additional characters are entered (but ignored) until the terminator is found. The reason for this action is that the next characters received are still part of this data item, as far as the data *sender* is concerned. These characters are accepted from the sender so that the next enter operation will not receive these "leftover" characters.

Another case involving numeric data can also occur (see the example given with "rule 4" describing the number builder). If a trailing non-numeric character terminates the last item (which is a numeric variable), additional characters will be entered in search of either a line-feed or a character accompanied by EOI. Unless this terminating condition is found before 256 characters have been entered, an error is reported.

**EOI Termination.** A termination condition for the HP-IB Interface is the EOI (End-or-Identify) signal. When this message is sent, it immediately terminates the entire ENTER statement, regardless of whether or not all variables have been satisfied. However, if all variable items in the destination list have not been satisfied, an error is reported.

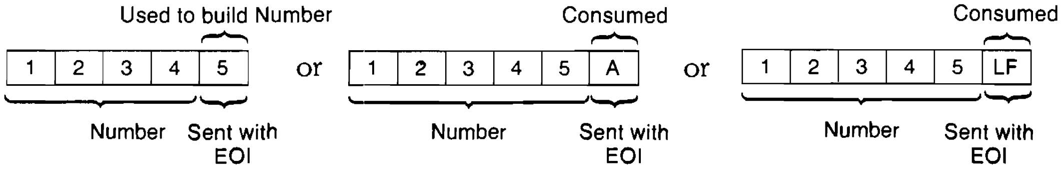
For example:

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



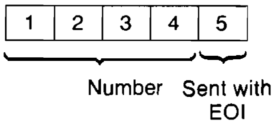
The result of entering the preceding data with the given ENTER statement is that String$ receives the characters "ABCDEF". The EOI signal being received with either the last character or with the terminator character properly terminates the ENTER statement. If the character accompanied by EOI is a string character (not a terminator), it is entered into the variable as usual.

```
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$
```



An error is reported
(Error 153 Insufficient data for ENTER).

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.

The EOI signal is implemented on the HP-IB Interface, described in the "HP-IB Interface" section of chapter 16.

## 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.

1.
```
100   ENTER @Device_x USING "6A,DDD.DD";String_var$,Num_var
```
2.
```
100   Image_str$="6A,DDD.DD"
110   ENTER @Device_x USING Image_str$;String_var$,Num_var
```
3.
```
100   ENTER @Device USING Image_stmt;String_var$,Num_var
110   Image_stmt: IMAGE 6A,DDD.DD
```
4.
```
100   ENTER @Device USING 110;String_var$,Num_var
110   IMAGE 6A,DDD.DD
```

## Images

Images are used to specify how data entered from the source is to be interpreted and placed into variables; each image consists of one or more groups of individual image specifiers that determine how the computer will interpret the incoming data bytes (or words). Thus, image lists can be thought of as a description of *either*:

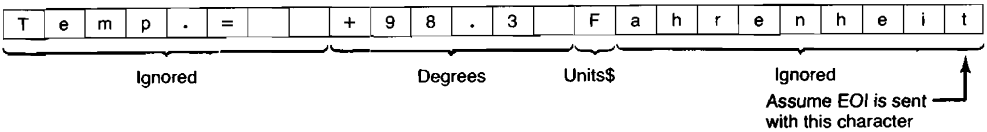
- the format of the expected data, or
- the procedure that the ENTER statement will use to enter and interpret the incoming data bytes.

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

All of the image specifiers used in image lists are valid for both enters and outputs. However, most of the specifiers have a slightly different meaning for each operation. If you plan to use the same image for output and enter, you must fully understand how both statements will use the image.

**Example of an Enter Using an Image.** This example is used to show you exactly how the computer uses the image to enter incoming data into variables. Look through the example to get a general feel for how these enter operations work. Afterwards, you should read the descriptions of the pertinent specifier(s).

Assume that the following stream of data bytes are to be entered into the computer:


T e m p . =   + 9 8 . 3   F a h r e n h e i t
Ignored
Degrees
Units$
Ignored
Assume EOI is sent with this character

Given the preceding conditions, let's look at how the computer executes the following ENTER statement that uses the specified IMAGE statement.

```
300  ENTER @Device USING Image_1;Degrees,Units$
310  Image_1:  IMAGE 8X,SDDD.D,A
```

**Step 1:** The computer evaluates the first image of the IMAGE statement. It is a special image in that it does not correspond to a variable in the destination list. It specifies that eight characters of the incoming data stream are to be ignored. Eight characters, "Temp.=  ", are entered and are ignored (i.e., are not entered into any variable).

**Step 2:** The computer evaluates the next image. It specifies that the next six characters are to be used to build a number. Even though the order of the sign, digit, and radix are explicitly stated in the image, the actual order of these characters in the incoming data stream does not have to match this specifier exactly. Only the *number* of numeric specifiers in the image, here six, is all that is used to specify the data format. When all six characters have been entered, the number builder attempts to form a number.

**Step 3:** After the number is built, it is placed into the variable "Degrees". The representation of the resultant number depends on the numeric variable type — INTEGER, REAL, or COMPLEX.*

* The number could be the real or the imaginary part of a COMPLEX value.

**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" near the end of this chapter.

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. The numeric specifiers are listed in the following table.

## 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.<br>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:

```
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
```

*(These five are equivalent.)*

```
ENTER Device USING "K";Number
```

*(Use the rules of the number builder.)*

```
ENTER @Device USING "DDRDD";Number
```

*(Enter five characters, using comma as radix.)*

```
ENTER @Device USING "H";Number
```

*(Use the rules of the number builder, but use the comma as radix and period as terminator.)*

**String Images.** The following specifiers are used to determine the number of and the interpretation of data bytes entered into string variables.

**String Specifiers**

| Image Specifier | Meaning |
|---|---|
| A | Specifies that one byte is to be entered and interpreted as a string character. Any terminators are entered into the string when this specifier is used. |
| K, H | Specifies that "free-field" ENTER conventions are to be used to enter data into a string variable; characters are entered directly into the variable until a terminating condition is sensed (such as CR/LF, LF, or an END indication). |
| -K, -H | Like K, except that line-feeds (LF's) do not terminate entry into the string; instead, they are treated as string characters and placed in the variable. Receiving an END indication terminates the image item (for instance, receiving EOI with a character on an HP-IB interface, encountering an end-of-data, or reaching the variable's dimensioned length). |
| L, @ | These specifiers are ignored for ENTER operations; however, they are allowed for compatibility with OUTPUT statements (that is, so that one image may be used for both ENTER and OUTPUT statements). Note that it may be necessary to skip characters (with specifiers such as X or /) when ENTERing data which has been sent by including these specifiers in an OUTPUT statement. Even greater care must be given to cases in which pad bytes may be sent; see "The BYTE and WORD Attributes" in chapter 14 for further explanation. |

**Examples of String Images:**

```
ENTER @Device USING "10A";Ten_chars$
```
*(Enter 10 characters.)*

```
ENTER @Device USING "K";Any_string$
```
*(Enter using free-field rules.)*

```
ENTER @Device USING "5A,K";String$,Number$
```
*(Enter two strings.)*

```
ENTER @Device USING "5A,K";String$,Number
```
*(Enter a string and a number.)*

```
ENTER @Device USING "-K";All_chars$
```
*(Enter characters until string is "full" or END is received.)*

**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:**

```
ENTER @Device USING "5X,5A";Five_chars$
```
*(Ignore first five and use second five characters.)*

```
ENTER @Device USING "5A,4X,10A";S_1$,S_2$
```
*(Ignore sixth through ninth characters.)*

```
ENTER @Device USING "/,K";String2$
```
*(Ignore first item of unknown length.)*

```
ENTER @Device USING """zz"",AA";S_2$
```
*(Ignore two characters.)*

**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. If either an I/O path name with the BYTE attribute (see "I/O Path Attributes" in chapter 14) or a device selector is used to access an 8-bit interface, two bytes will be entered; the first byte entered is most significant. If an I/O path name with the BYTE attribute is used to access a 16-bit interface, the BYTE attribute is overwritten 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 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. If an I/O path name with the BYTE attribute is used to access a 16-bit interface, the BYTE attribute is not overwritten (as with the W specifier). |

**Examples of Binary Images:**

```
ENTER @Device USING "B,B,B";N1,N2,N3
```

*(Enter three bytes, then look for LF or END indication.)*

```
ENTER @Device USING "W,K";N,N$
```

*(Enter the first two bytes as an INTEGER, then the rest as string data.)*

*Assume that @Device possesses the WORD attribute.*

```
ENTER @Device USING "B,W";Num_1,Num_2
```

*(Enter one byte, ignore one pad byte, enter one word, then search for terminator.)*

*@Device may possess either BYTE or WORD attribute.*

```
ENTER @Device USING "B,Y";Num_1,Num_2
```

*(Enter one byte, enter one word, then search for terminator.)*

## 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* or *within 256 bytes after* the byte that satisfied the last image item.
- A line-feed is received *as* the byte that satisfies the last *image item* (exceptions are the "B" and "W" specifiers) or *within 256 bytes after* the byte that satisfied the last image item.

**EOI Re-Definition.** It is important to realize that when an enter uses an image (when the secondary keyword "USING" is specified), the definition of the EOI signal is *automatically modified.* If the EOI signal terminates the *last image item*, the entire statement is properly terminated, as with free-field enters. *In addition, multiple EOI signals are now allowed and act as item terminators.* However, the EOI must be received *with* the byte that satisfies each image item. If the EOI is received *before* any image is satisfied, it is *ignored.* Thus, all images must be satisfied, and EOI will not cause early termination of the ENTER-USING-image statement.

The following table summarizes the definitions of EOI during several types of ENTER statement. The statement-terminator modifiers are more fully described in the next section.

**Effects of EOI During ENTER Statements**

| | **Free-Field ENTER Statements** | **ENTER USING** (without # or %) | **ENTER USING** (with #) | **ENTER USING** (with %) |
|---|---|---|---|---|
| **Definition of EOI** | Immediate statement terminator | Item terminator or statement terminator | Item terminator or statement terminator | Immediate statement terminator |
| **Statement Terminator Required?** | Yes | Yes | No | No |
| **Early Termination Allowed?** | No | No | No | Yes |

**Statement-Termination Modifiers.** The table on the following page lists the statement-termination modifiers. These specifiers modify the conditions that terminate enters that use images. The first one of these specifiers encountered in the image list modifies the termination conditions for the ENTER statement. If another of these specifiers is encountered in the image list, it again modifies the terminating conditions for the statement.

### Statement-Termination Modifiers

<table>
<tr><th>Image Specifier</th><th>Meaning</th></tr>
<tr><td>#</td><td>Specifies that a statement-termination condition is not required; the ENTER statement is automatically terminated as soon as the last image item is satisfied.</td></tr>
<tr><td>%</td><td>Also specifies that a statement-termination condition is not required. In addition, EOI is re-defined to be an 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.
<table>
<tr><th>Specifier</th><th>Legal Boundary</th></tr>
<tr><td>K, -K</td><td>With any character, since this specifies a variable-width field of characters.</td></tr>
<tr><td>S, M, D, E<br>Z, ., A, X<br>"literal"<br>B, W</td><td>Only with the last character that satisfies the image (e.g., with the 5th character of a "5A" image). If EOI is received with any other character, it is ignored.</td></tr>
<tr><td>/</td><td>Only with the last line-feed character that satisfies the image (e.g., with the 3rd line-feed of a "3/" image); otherwise it is ignored.</td></tr>
</table></td></tr>
<tr><td>+</td><td>Specifies that an END indication is required to terminate the ENTER statement. Line-feeds are ignored as statement terminators; however, they will still terminate items (unless a -K or -H image is used for strings).</td></tr>
<tr><td>-</td><td>Specifies that a line-feed is required to terminate the statement. EOI is ignored, and other END indications (such as EOF or end-of-data) cause an error if encountered before the line-feed.</td></tr>
</table>

**Examples of Modifying Termination Conditions:**

| | |
|---|---|
| `ENTER @Device USING "#,B";Byte` | *(Enter a single byte.)* |
| `ENTER @Device USING "#,W";Integer` | *(Enter a single word.)* |
| `ENTER @Device USING ",K";Array(*)` | *(Enter an array, allowing early termination by EOI.)* |
| `ENTER @Device USING "+,K";String$` | *(Enter characters into String$ until line-feed received, then continue entering characters until END received.)* |
| `ENTER @Device USING "-,K";String$` | *(Enter characters until line-feed received; ignore EOI, if received.)* |

## 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.** Just as with OUTPUT, the ENTER statement allows several image specifiers to be preceded by an integer that specifies how many times the specifier is to be used.

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

**Image Re-Use.** If there are fewer images than items in the destination list, the list will be re-used, beginning with the first item in the image list. If there are more images than there are items, the additional specifiers will be ignored.

For example:

```
ENTER @Device USING "#,B";B1,B2,B3
```

*(The "B" is re-used.)*

```
ENTER @Device USING "2A,2A,W";A$,B$
```

*(The "W" is not used.)*

**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.

For example:

```
ENTER @Source USING "2(B,5A,/),/";N1,N1$,N2,N2$
```

14

# Advanced Interfacing Topics

The previous chapter discussed outputting and entering data in detail. However, you may want to exercise further control over your peripheral device. You may want to monitor the status of an instrument. Or you may want your program to be interrupted by a peripheral that needs attention. This chapter covers several advanced techniques that give you greater control over your I/O application.

## Registers

A register is a memory location. Some registers are memory locations on interface cards, while others are memory locations in the computer which are maintained by BASIC to keep track of various conditions related to interfaces. Some registers store parameters that describe the operation of an interface, some store information describing the I/O path to a device, and some are in locations at which interface cards reside (remember that the computer implements "memory-mapped I/O").

Registers are accessed by the computer when executing I/O statements that specify an interface select code, a device selector, or an I/O path name. Thus, each interface and I/O path has its own set of registers. The general programming techniques used to access these registers and the specific definitions of all I/O path registers are given in this section. Refer to chapter 16 and to the BASIC *Language Reference* manual for information about the specific interface registers.

There are three levels of register access:

1. Firmware register(s) are automatically accessed by BASIC when an I/O statement is executed.

```
OUTPUT @File;Data$          ! Changes file pointer registers
ENTER @Buffer;Numeric_item  ! Changes buffer pointer registers
```

2. STATUS and CONTROL (firmware) registers are explicitly accessed by BASIC statements:

```
100  STATUS CRT,13;Crt_height
110  CONTROL CRT,13;Crt_height+3
```

3. Interface (hardware) registers are directly read or written using the READIO and WRITEIO statements. This requires an extensive knowledge of both the hardware registers and the consequences of writing to these registers. This technique is beyond the scope of this manual. In most cases you can achieve the same purpose by using STATUS and CONTROL.

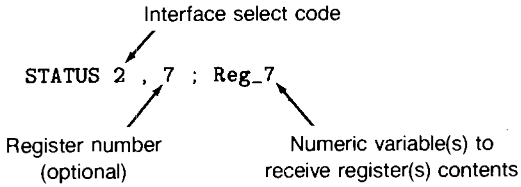
## Interface Registers

A simple example of an interface register being accessed explicitly by the program and then automatically by I/O statements is shown in the following program. Register 0 of interface select code 1 is the "X" screen coordinate at which subsequent characters output to the the CRT will begin being displayed; register 1 is the corresponding "Y" coordinate.

```
100  STATUS CRT;Reg_0,Reg_1 ! Pgrm accessing X & Y coords.
110  OUTPUT CRT;"Print coordinates before 1st OUTPUT:"
120  OUTPUT CRT;"X=";Reg_0,"  Y=";Reg_1
130  OUTPUT CRT
140  !
150  OUTPUT CRT;"1234567";  ! Note ";" is used to suppress EOL sequence.
160  STATUS CRT;Reg_0,Reg_1
170  OUTPUT CRT
180  OUTPUT CRT;"Print coordinates after OUTPUTs:"
190  OUTPUT CRT;"X=";Reg_0,"  Y=";Reg_1
200  OUTPUT CRT;" "
210  !
220  END
```

**The STATUS Statement.** The contents of a STATUS register can be read with the STATUS statement. Typical examples are shown below. A complete listing of each interface's registers is given in the BASIC *Language Reference* manual. The definitions of I/O path registers are described later in this section. Let's look at a few examples.

STATUS register 7 of the interface at select code 2 is read with the following statement. The first parameter identifies the interface and the optional second parameter identifies which register is to be read. The specified numeric variable receives the register's current contents.


Interface select code
STATUS 2 , 7 ; Reg_7
Register number (optional)
Numeric variable(s) to receive register(s) contents

STATUS register 0 of the I/O path `@Keyboard` is read with the following statement. (Note that this is *not* the same register as keyboard register 0.) Since the second parameter is optional and has been omitted in this instance, register 0 is accessed.

```
100   STATUS @Keyboard;Reg_0
```

STATUS registers 4 and 5 of the interface at select code 7 are read with the following statement:

```
100   STATUS 7,4;Reg_4,Reg_5
```

Since two numeric variables are to receive register contents, the next register (5) is accessed. If more than two variables are specified, successive registers are read.

**The CONTROL Statement.** When some I/O statements are executed, the contents of some CONTROL registers are automatically changed. For instance, in the above example registers 0 and 1 were changed whenever the OUTPUT statements to the CRT were executed. The program can also change some register's contents with the CONTROL statement, as shown in the following examples. Again, all of the CONTROL register definitions for each interface are given in the BASIC *Language Reference* manual.

Register 0 of interface select code 1 is modified with the following statement. This register determines the "X" screen coordinate at which subsequent characters output to the CRT display will appear.

```
                Interface select code
CONTROL 1 ; X_pos
                    Numeric expression(s) to be sent
                      to the appropriate register(s)
```

Register 1 of interface select code 1 is modified with the following statement. This register's contents determine the "Y" screen coordinate at which subsequent characters output to the CRT display will appear; changing the contents of this register also allows scrolling the display.

```
100   CONTROL 1,1;Line_pos
                 Register number
```

## I/O Path Registers

At this point you know how to access the registers associated with interfaces and I/O path names, but you may not know much about the differences or about the interaction between these two types of registers. Let's first review the definition of an I/O path name.

An I/O path name is a data type that contains a description of an I/O path between the computer and one of its resources sufficient to allow accessing the resource. You learned in the "Directing Data Flow" section that the computer uses this information whenever the I/O path name is used in an I/O statement. Much of this information stored in this I/O-path-name table can be accessed with the STATUS and CONTROL statements.

When an I/O path name is used to specify a resource in an I/O statement, BASIC accesses the first table entry (the validity flag) to see if the name is currently assigned. If the I/O path name is assigned, the computer reads I/O path register 0 which tells the computer the type of resource involved:

- If the resource is a device, BASIC must also access the registers of the interface specified by the device selector.
- If the resource is a file, the table contains additional entries that govern how the I/O process is to be executed.

As you can see, the set of I/O path registers is *not* the same set of registers associated with an interface. The following program is an example of using I/O path register 0 to determine the type of resource to which the I/O path name has been assigned.

```
700 Find_type: STATUS @Resource;Reg_0
710            !
720            IF Reg_0=0 THEN GOTO Not_assigned
730            !
740            IF Reg_0=1 THEN GOTO Device
750            !
760            IF Reg_0=2 THEN GOTO File
770            !
780      PRINT "Resource type unrecognized"
790      PRINT "Program STOPPED."
800      STOP
810            !
820 Not_assigned: PRINT "I/O path name not assigned"
830               GOTO Common_exit
840               !
850 Device: STATUS @Resource,1;Reg_1
860         PRINT "@Resource assigned to device"
870         PRINT "at intf. select code ";Reg_1
880         GOTO Common_exit
890         !
900         !
910 File: STATUS @Resource,1;Reg_1,Reg_2,Reg_3
920       !
930       PRINT "File type         ";Reg_1
940       PRINT "Device selector   ";Reg_2
950       PRINT "Number of sectors ";Reg_3
960       !
970       !
980 Common_exit: !  Exit point of this routine.
```

Once the type of resource has been determined, it can be further accessed with the I/O path registers or the interface registers, depending on the resource type.

- If the I/O path name has been assigned to a *device*, the *interface registers* should be accessed for further information.
- If the name has been assigned to a *mass storage file*, the *I/O path registers* should be accessed for further information.

I/O path names can be assigned to device selectors, files, and buffers. The following program shows an example of determining the interface select code of the resource to which the I/O path name has been assigned.

```
100   ! Example of determining select code
110   ! to which an I/O path name is assigned.
120   !
130 Show_sc: IMAGE "'@Io_path' assigned to ",K,"; Select code = ",D,L
140          !
150   ASSIGN @Io_path TO 701  ! Device selector.
160   Device_selector=FNSc(@Io_path)
170   OUTPUT CRT USING Show_sc;"device 701",Device_selector
180   !
190   ASSIGN @Io_path TO "Datal" ! ASCII file.
200   Device_selector=FNSc(@Io_path)
210   OUTPUT CRT USING Show_sc;"ASCII file",Device_selector
220   !
230   ASSIGN @Io_path TO "Chap1" ! BDAT file.
240   Device_selector=FNSc(@Io_path)
250   OUTPUT CRT USING Show_sc;"BDAT file",Device_selector
260   !
270   ASSIGN @Io_path TO BUFFER [1024] ! Buffer.
280   Device_selector=FNSc(@Io_path)
290   OUTPUT CRT USING Show_sc;"BUFFER",Device_selector
300   !
310   END
320   !
330   DEF FNSc(@Io_path) ! *************************************
340     ! Read I/O path register 0.
350     STATUS @Io_path;Resource_code
360     SELECT Resource_code
370     CASE 0  ! Not assigned.
380       RETURN -1  ! Return a select code out of range.
390       !
400     CASE 1  ! Assigned to a device selector.
410       STATUS @Io_path,1;Select_code
420       RETURN Select_code
430       !
```

```
440      CASE 2  ! Assigned to a file specifier.
450        STATUS @Io_path,2;Device_selector
460        RETURN Device_selector MOD 100 ! Remove addressing.
470        !
480      CASE 3 ! Assigned to a buffer.
490        RETURN 0  ! No error, but cannot determine source
500                  ! or destination of transfer to/from buffer.
510      END SELECT
520      !
530      FNEND ! ************************************************
```

The following printout shows a typical example of the program's output:

```
'@Io_path' assigned to device 701; Select code = 7

'@Io_path' assigned to ASCII file; Select code = 7

'@Io_path' assigned to BDAT file; Select code = 7

'@Io_path' assigned to BUFFER; Select code = 0
```

The user-defined function called FNSc interrogates I/O path registers to find the select code. If the I/O path name is currently not assigned, the function returns an arbitrary value of −1 (an invalid value of select code). Since STATUS Register 2 of I/O path names assigned to files contains the entire device selector, which may include addressing information, the function removes any addressing information (Device_selector MOD 100).

Notice that buffers have no select code associated with them, since they are a data type resident in computer memory; thus the function returns a value of 0.

The SC function is a feature of the "Main" BASIC system. The following statements show examples of using this function.

```
Select_code=SC(@Io_path)
IF SC(@File)=4 THEN Device_type$="INTERNAL"
```

The only difference in this language-resident function and the preceding example is that the SC function reports an error if the I/O path specified as its argument is not assigned, rather than returning a select code out of range.

## Summary of I/O Path Registers

The following list describes the information contained in I/O path STATUS and CONTROL registers. Note that only STATUS register 0 is identical for *all* types of I/O paths; the rest of the I/O path registers' contents depend on the *type* of resource to which the name is assigned.

### For All I/O Path Names.

| | |
|---|---|
| **Status Register 0** | 0 = Invalid I/O path name<br>1 = I/O path name assigned to a device<br>2 = I/O path name assigned to a data file<br>3 = I/O path name assigned to a buffer |

### I/O Path Names Assigned to a Device.

| | |
|---|---|
| **STATUS Register 1** | Interface select code |
| **STATUS Register 2** | Number of devices |
| **STATUS Register 3** | Address of 1st device |

If assigned to more than one device, the addresses of the other devices are available starting in STATUS Register 4.

### I/O Path Names Assigned to an ASCII File.

| | |
|---|---|
| **STATUS Register 1** | File type = 3 |
| **STATUS Register 2** | Device selector of mass storage device |
| **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 |

### I/O Path Names Assigned to a BDAT File.

| | |
|---|---|
| **STATUS Register 1** | File type = 2 |
| **STATUS Register 2** | Device selector of mass storage device |
| **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 current 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 |

### I/O Path Names Assigned to an HP-UX File.

| | |
|---|---|
| **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 current 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 |

**I/O Path Names Assigned to a Buffer.** When the status of register 0 indicates a buffer (3), the status and control registers have the following meanings.

| | |
|---|---|
| **STATUS Register 1** | Buffer type (1=named, 2=unnamed) |
| **STATUS Register 2** | Buffer size in bytes |
| **STATUS Register 3** | Current fill pointer |
| **CONTROL Register 3** | Set fill pointer |
| **STATUS Register 4** | Current number of bytes in buffer |
| **CONTROL Register 4** | Set number of bytes |
| **STATUS Register 5** | Current empty pointer |
| **CONTROL Register 5** | Set empty pointer |
| **STATUS Register 6** | Interface select code of inbound TRANSFER |
| **STATUS Register 7** | Interface select code of outbound TRANSFER |
| **STATUS Register 8** | If non-zero, inbound TRANSFER is continuous |
| **CONTROL Register 8** | Cancel continuous mode inbound TRANSFER if zero |
| **STATUS Register 9** | If non-zero, outbound TRANSFER is continuous |
| **CONTROL Register 9** | Cancel continuous mode outbound TRANSFER if zero |

**STATUS Register 10** Termination status for inbound TRANSFER

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 0 | TRANSFER Active | TRANSFER Aborted | TRANSFER Error | Device Termination | Byte Count | Record Count | Match Character |
| Value = 128 | Value = 64 | Value = 32 | Value = 16 | Value = 8 | Value = 4 | Value = 2 | Value = 1 |

**STATUS Register 11** Termination status for outbound TRANSFER

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 0 | TRANSFER Active | TRANSFER Aborted | TRANSFER Error | Device Termination | Byte Count | Record Count | 0 |
| Value = 128 | Value = 64 | Value = 32 | Value = 16 | Value = 8 | Value = 4 | Value = 2 | Value = 1 |

**STATUS Register 12** Total number of bytes transferred by last inbound TRANSFER

**STATUS Register 13** Total number of bytes transferred by last outbound TRANSFER

## Interrupts and Timeouts

The computer can sense and respond to the occurrence of several types of interrupt events. This section describes programming techniques for handling the interface events called “interrupts” and “timeouts” which can initiate program branches. For more information about event-initiated branches, refer to chapter 1, “Program Structure and Flow.” You may also want to refer to the keyword descriptions in the BASIC *Language Reference* manual.

### Overview of Event-Initiated Branching

Event-initiated branches are very powerful programming tools. With them, the computer can execute special routines or subprograms whenever a particular event occurs; the program doesn’t have to take time to periodically check for each event’s occurrence.

This section describes the general topic of event-initiated branching. Subsequent sections take a closer look at interrupt events.

**Types of Events.** The statements that enable events to initiate branches are summarized as follows:[*]

ON CDIAL — occurs when one of the nine "knobs" (rotary pulse generators) of an HP 46085 Control Dial Box is turned.

ON END — occurs when the computer encounters the end of a mass storage file while accessing the file.

ON ERROR — occurs when a program-execution error is sensed.

ON KEY — occurs when a currently defined softkey is pressed.

ON KNOB — occurs when the "knob" (rotary pulse generator) is turned.

ON INTR — occurs when an interrupt is requested by a device or when an interrupt condition occurs at the interface.

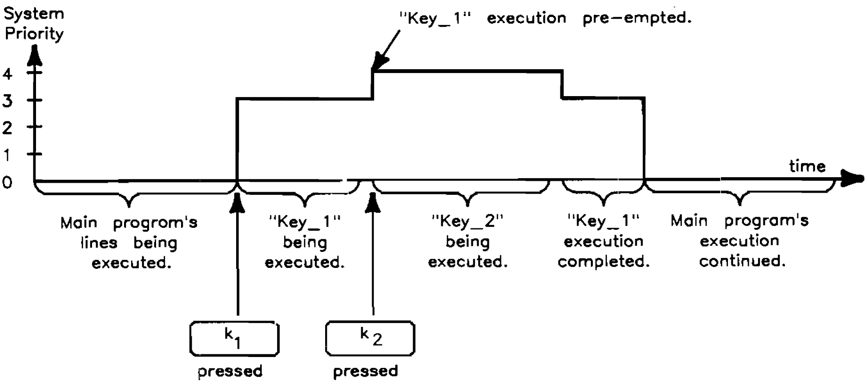
ON TIMEOUT — occurs when the computer has not detected a handshake response from a device within a specified amount of time.

**A Simple Example.** The following program shows how events are serviced by the computer. Subprograms called "Key_1" and "Key_2" are the service routines for the events of pressing softkeys [f1] and [f2] being pressed; the software priorities assigned to these events are 3 and 4, respectively. Run the program and alternately press these softkeys; the branch to each key's service routine is initiated by pressing the key. The system priority is "graphed" on the CRT display.

[*] The syntax of each of these statements is covered in the BASIC *Language Reference* manual. The ON INTR and ON TIMEOUT statements are covered in detail later in this section.

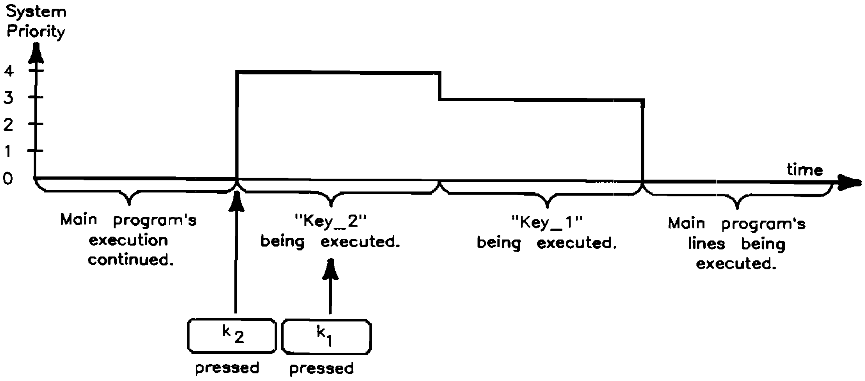
```
150 ON KEY 1,3 CALL Key_1   ! Set up events and
160 ON KEY 2,4 CALL Key_2   ! assign priorities.
170 !
180 OUTPUT CRT;" System","Priority"
190 V$=CHR$(8)&CHR$(10)     ! BS & LF.
200 OUTPUT CRT;"   4"&V$&"3"&V$&"2"&V$&"1"&V$&"0"
210 !
220 Main: CALL Bar_graph(7,"*") ! Sys. prior. is
230                             ! always >= 0.
240       BEEP 100,.1           ! Low tone.
250       FOR Jiffy=1 TO 5000
260       NEXT Jiffy
270       !
280   GOTO Main             ! Main loop.
290   !
300   END
310   !
320 SUB Key_1
330     CALL Bar_graph(4,"*") ! Plot priority.
340     BEEP 300,.1           ! Middle tone.
350     FOR Iota=1 TO 2000
360     NEXT Iota
370     CALL Bar_graph(4," ") ! Erase.
380   SUBEND
390   !
400 SUB Key_2
410     CALL Bar_graph(3,"*") ! Graph priority.
420     BEEP 400,.1           ! High tone.
430     FOR Twinkle=1 TO 2000
440     NEXT Twinkle
450     CALL Bar_graph(3," ") ! Erase.
460   SUBEND
470   !
480 SUB Bar_graph(Line,Char$)
490     CONTROL 1,1;Line   ! Locate line.
500     OUTPUT 1;Char$     ! Bar-graph character.
510     SUBEND
```

If [f2] is pressed *after* [f1] is pressed, *but while* the Key_1 routine is being executed, execution of Key_1 is *temporarily interrupted* and the Key_2 routine is executed. When Key_2 is finished, execution of Key_1 is resumed at the point where it was temporarily interrupted. This occurs because [f2] was assigned a *higher software priority* than [f1].


System Priority
4
3
2
1
0
"Key_1" execution pre-empted.
time
Main program's lines being executed.
"Key_1" being executed.
"Key_2" being executed.
"Key_1" execution completed.
Main program's execution continued.
k1
pressed
k2
pressed

**Events with Higher Software Priority Take Precedence**

On the other hand, if [f1] is pressed *while* [f2] is being serviced, the computer finishes executing Key_2 *before* executing Key_1. *The event of pressing* [f1] *was "logged" but not processed until after the routine having higher software priority was completed.* This is a very important concept when dealing with event-initiated branching. The action of the computer in logging events and determining assigned software priority is further described in the next section.



**An Event with Lower Software Priority Must Wait**

**Conditions Required for Initiating a Branch.** In order for any event to initiate a branch, the following prerequisite conditions must be met. The preceding section showed a simple example of softkey events, which are similar to interface interrupts. This section describes the additional requirements for servicing interface interrupts. Later sections show more details of meeting these requirements.

1. The branch must be *set up* by an ON-event-branch statement, and the *service routine* must exist.

   ```
   100  ON INTR GOSUB Check_device
                 .
                 .
                 .
   920 Check_device: ! Service routine for interface interrupts.
   ```

   The term *service routine* is any legal branch location for the type of branch specified (GOSUB, GOTO, CALL,* or RECOVER) and current context.

2. Before an event (which is set up) can initiate a branch, it must first be enabled to do so. With non-interrupt events (such as ON KEY, and ON KNOB), the event is *automatically* enabled when the ON-event statement is executed. However, with ON INTR, you must explicitly enable the interrupt to initiate its corresponding branch. For example, to enable the interface at select code 7 to initiate an interrupt branch:

   ```
   110  ENABLE INTR 7;Intr_mask
   ```

   Refer to "Interface Interrupts" and "Interface Timeouts," later in this section, for further details on enabling these events.

3. The event must *occur* and be *logged* by the BASIC system. (For instance, the HP-IB "Service Request" signal is sent from the device to the computer and is logged by the BASIC operating system.)

4. The *software priority* assigned to the event must be greater than the current *system priority.*†

When all of these conditions have been met, the branch is taken.

* Parameters cannot be passed to the service routine in an ON INTR CALL statement; any variables to be used jointly by the service routine and other contexts must be defined in common.

† Software priority is specified in the event's set-up statement; the range of priorities that can be specified in this statement is 0 through 15. Interfaces also have a "hardware" priority which is different from the software priority. The following sections describe details of hardware and software priority.

**Logging and Servicing Events.** The preceding events may occur at any time; however, the BASIC program is only "notified" if these events have been "set up" to initiate a branch. An example of ignoring an event is seen when an undefined softkey is pressed. Since the event has not been set up, the operating system detects the event, but does not notify the BASIC program. In this example, the computer beeps. No BASIC service routine is executed, even though the operating system was "aware" of the event. Thus, only when an event is first set up and then occurs does the BASIC program "service" its occurrence.

**Software Priority:** The computer first "logs" the occurrence of an event which is set up.* After recording that the event occurred, the computer then checks the event's software priority against that of the routine currently being executed. The priority of the routine currently being executed is known as *system priority*. If no service routine is being executed, the system priority is 0; otherwise the system priority is equal to the assigned software priority of the routine currently being executed. The following table lists the software priority structure of the BASIC system; priority increases from 0 to 17.

**Software Priorities of Events**

| System Priority | Explanation |
|---|---|
| 0 | System priority when no service routine is being executed (known as the "quiescent level"). |
| 1 thru 15 | Software-assignable priorities of service routines. |
| 16 | Effective software priority of ON END and ON TIMEOUT. The software priorities of these events *cannot* be changed. |
| 17 | Effective software priority of ON ERROR. The software priorities of these events *cannot* be changed. |

In the above example, system priority was 0 before either of the events occurred. When f1 was pressed, the system priority became 3. When f2 was subsequently pressed, the system first logged the event and then checked its priority against the current system priority. Since f2 had been assigned a priority of 4, it pre-empted f1's service routine because of its higher software priority.

It is important to note that *BASIC only services event occurrences when a program line is exited.* This change of lines occurs either:

* The process of logging event occurrences is described in the section called "Hardware Priority."

- at the end of execution of a line, or
- when the line is exited when a user-defined function is called.

When the program line is changed, the computer attempts to service all events that have occurred since the last time a line was exited. The next sections further describe logging and servicing events.

When execution of Key_2 started, the system priority was set to 4. If any event was to interrupt the execution of this service routine, it must have had a software priority of 5 (or greater). When execution of Key_2 completed, the Key_1 service routine had the highest software priority, so its execution was resumed at the point at which it was interrupted.

If [f1] was pressed *again* while its own service routine was being executed, execution of the first service routine was finished before the service routine was executed again. Thus, if an event occurs that has the *same* software priority as the system priority, its service routine will *not* interrupt the current routine. The service routine will *only* be executed if the event's software priority becomes the highest priority of any event which has been logged (i.e., after *all* other events of higher software priority have been serviced).

**Changing System Priority:** Events are assigned a software priority to allow the computer to respond to occurrences of events with high software priority before those with lower priorities. Occasionally, service routines may contain code segments that should not be interrupted once their execution begins. In such cases, the entire service routine may not require a high software priority, even though a portion of the routine needs a high priority to ensure that it will not be interrupted by most other processes.

The SYSTEM PRIORITY statement can be used in these cases to set the system priority to a level higher than the BASIC system would otherwise set it when the branch to the service routine is taken. The current system priority can also be determined by calling SYSTEM$("SYSTEM PRIORITY"), which returns a string value of the current system priority in the range 0 through 15. Examples are shown in the following program.

```
100   GINIT   ! Use default plotter is CRT.
110   GRAPHICS ON
120   VIEWPORT 0,131,30,100
130   WINDOW 0,2000,0,7
140   !
150   ON KEY 1 LABEL "Prior.1",1 GOSUB Key_1
160   ON KEY 2 LABEL "Prior.2",2 GOSUB Key_2
170   ON KEY 3 LABEL "Prior.2",3 GOSUB Key_3
180   !
190   Sys_prior$="SYSTEM PRIORITY" ! Define string for SYSTEM$.
200   !
```

```
210 Main_program: !
220   DISP "Quiescent system priority level = 0."
230   X=X+1
240   Sys_prior=VAL(SYSTEM$(Sys_prior$))
250   GOSUB Plot_priority
260   GOTO Main_program
270   !
280 Key_1:   FOR Iota=1 TO 100
290              DISP "Key 1; priority 1."
300              X=X+1
310              Sys_prior=VAL(SYSTEM$(Sys_prior$))
320              GOSUB Plot_priority
330          NEXT Iota
340          RETURN
350          !
360 Key_2:   FOR Twinkle=1 TO 100
370              DISP "Key 2; priority 2."
380              X=X+1
390              Sys_prior=VAL(SYSTEM$(Sys_prior$))
400              GOSUB Plot_priority
410          NEXT Twinkle
420          !
430          ! Critical routine raise system priority.
440          SYSTEM PRIORITY 3
450          FOR Split_second=1 TO 100
460              DISP "Subroutine set system priority to 3."
470              X=X+1
480              Sys_prior=VAL(SYSTEM$(Sys_prior$))
490              GOSUB Plot_priority
500          NEXT Split_second
510          !
520          ! System priority lowered when finished.
530          SYSTEM PRIORITY 0
540          RETURN
550          !
560 Key_3:   FOR Jiffy=1 TO 100
570              DISP "Key 3; priority 3."
580              X=X+1
590              Sys_prior=VAL(SYSTEM$(Sys_prior$))
600              GOSUB Plot_priority
610          NEXT Jiffy
620          RETURN
```

```
630          !
640 Plot_priority:  !
650                 IF X>2000 THEN ! Draw new plot.
660                     GCLEAR
670                     MOVE 0,0
680                     X=0
690                 END IF
700                 PLOT X,Sys_prior
710                 RETURN
720                 !
730                 !
740    END
```

The subroutine called Key_2 raised the system priority from its current level, 2, to level 3 during the time that the second FOR..NEXT loop was being executed. During this time, pressing f3 will not interrupt the routine, since a priority of 4 or greater is required to interrupt the Key_2 routine.

By setting the system priority level in this manner, routines can selectively allow and disallow other routines from being executed; routines with higher software priority are allowed to pre-empt the routine, while those with the same or lower priority are not. If no other events are to interrupt the process, system priority can be set to 15. However, keep in mind that END, ERROR, and TIMEOUT events have effective software priorities higher than 15 and can therefore interrupt the service routine (if a branch for one of these events is currently set up).

When the "critical" code has been executed, the program returns the system priority to the value set by the BASIC system when the branch was taken (which was 2 since the Key_2 event was being serviced). Of course, if an event with higher software priority occurs while the code segment is being executed, its service routine will pre-empt the critical code segment.

This technique can also be used within SUB and FN subprograms. Keep in mind that when program control is returned from a context, the system priority is returned to the value it had when the context was called.

**Hardware Priority:** There is a second event priority, hardware priority, that also influences the order in which the computer responds to events.

- Hardware priority determines the order in which events are *logged* by the system (explained in following paragraphs).
- Software priority determines the order in which events are *serviced.*

The hardware priority of an interface interrupt is determined by the priority-switch setting on the interface card itself.[*] *Hardware priority is independent of the software priority assigned to the event by the ON INTR statement.*

All events have a hardware priority, but not all have hardware priorities that can be changed. The following table lists the hardware-priority structure of HP 9000 Series 200/300 computers and the HP BASIC Language Processor. Only the optional interfaces' hardware priorities can be changed.

**Hardware Priorities of Interfaces**

| Hardware Priority | Interfaces and Events at this Priority |
|---|---|
| 0 | Quiescent level<br>(no interface is currently interrupting) |
| 1 | System keyboard<br>(KEY and KNOB events) |
| 3 | Built-in HP-IB interface<br>(INTR and TIMEOUT events) |
| 3-6 | Optional interface cards<br>(INTR and TIMEOUT events) |
| 7 | Non-Maskable Interrupts, such as the RESET (Break) key |

In order to fully understand the differences between hardware and software priority, it is helpful to first understand how the computer logs and services events. When any event occurs, the interface (at which the event has occurred) signals it to the computer. The computer responds by temporarily suspending execution of its current task to *poll* (interrogate) the currently enabled interfaces.

When the computer determines which interface is interrupting, it records that it has occurred on this interface (i.e., logs the event) and *disables further interrupts from this interface.* This event is now *logged* and service by the computer is *pending.* The computer can then return to its former task (unless other events have occurred which have not been logged).

[*] Setting hardware priority on an optional interface is described in the interface's installation manual.

If other events have occurred but have not yet been logged, they will be *logged in order of descending hardware priority.* This occurs because events with hardware priority lower than that of the event currently being logged are *ignored* until all events with the current hardware priority are logged.

**Servicing Pending Events.** If BASIC was interrupted while executing a program line, execution of the line is resumed (after logging all events) and continues until either the line is completely executed or a user-defined function causes the line to be exited. When the line is exited, BASIC begins servicing all pending events.

When servicing pending events, the following rules are used to determine the order in which they are serviced:

1. Highest software priority first, lowest software priority last.
2. If two or more events have the same software priority, the BASIC services the events in order of descending interface select codes.
3. If events have both the same software priority and interface select code (such as softkeys with the same software priority), the events are serviced in the order in which they occurred.

The process of logging of events is still taking place while events are being serviced. This concurrent action has two major effects.

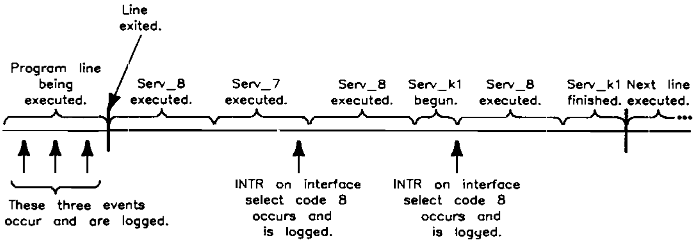
1. Events of higher hardware priority will interrupt the current activity to be logged by the computer.
2. Events which also have higher software priority will interrupt the computer's present activity to be serviced.

Thus, events of high hardware and software priority can potentially occur and be serviced many times between program lines.

For example, suppose that the following events have been set up and enabled to initiate branches. Assume that the events have the hardware priorities shown in the program's comments.

```
100   ON INTR 8,15 CALL Serv_8   ! Hardware priority 6.
110   ON INTR 7,14 CALL Serv_7   ! Hardware priority 3.
120   ON KEY 0,5 CALL Serv_k0    ! Hardware priority 1.
```

The following diagram shows the INTR event on interface select code 8 occurring and being serviced several times after one program line has been exited.

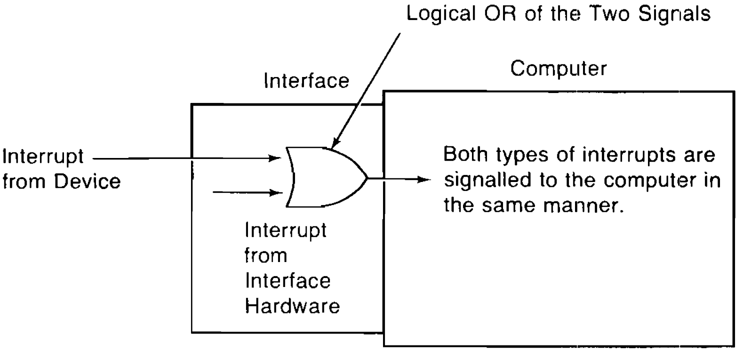

Line exited.
Program line being executed.
Serv_8 executed.
Serv_7 executed.
Serv_8 executed.
Serv_k1 begun.
Serv_8 executed.
Serv_k1 finished.
Next line executed.
These three events occur and are logged.
INTR on interface select code 8 occurs and is logged.
INTR on interface select code 8 occurs and is logged.

**INTR Event Servicing**

The main function of hardware priority is to keep events of lower priority from being logged so that more "urgent" events can be serviced quickly. Decreasing the system's response time to these urgent events may also increase overall system throughput.

## Interface Interrupts

All interfaces have a hardware line dedicated to signal to the computer that an interrupt event has occurred. The source of this signal can be either the device(s) connected to the interface or the interface hardware itself. These possibilities are shown in the following diagram:


Logical OR of the Two Signals
Interface
Computer
Interrupt from Device
Interrupt from Interface Hardware
Both types of interrupts are signalled to the computer in the same manner.

**Interface Interrupts**

There are two general types of interrupt events:

- One type of event occurs when a *device* determines that it requires the computer to execute a special procedure.
- The second type occurs when the *interface itself* determines that a condition exists or has occurred that requires the computer's attention.

The first type of interrupt event is usually called a *service request*. Service requests *originate at the device.* An example is a voltmeter signaling to the computer that it has a reading; another is a printer generating a service request when it is out of paper. The service routine takes the appropriate action, and the program (usually) resumes execution.

The second type of interrupt event is used to inform the computer of a *specific condition* at the interface. This type of event *originates at the interface.* An example of this interrupt event is the occurrence of a parity error detected by the serial interface. This error usually requires that the erroneous data just received be re-transmitted. The service routine can often correct this error by telling the sender to keep sending the data until the error no longer occurs, after which the computer can resume its former task.

**Enabling Interrupt Events.** Before the INTR event can initiate its branch, it must be enabled to do so. The following examples show how to enable interrupt events to initiate branches.

Enable interrupts occurring at interface select code 7 to initiate the branch set up by an ON-event-branch statement.

```
ENABLE INTR 7;Mask
```

The bit pattern of Mask is copied into the "interrupt-enable" register of the specified interface; in this case, register 4 of the built-in HP-IB interface receives Mask's bit pattern. *Individual bits of the mask* are used to enable different types of interrupt events for each interface. Each bit which is *set* (i.e., which has a value of 1) in the mask expression *enables* the corresponding interrupt condition defined for that bit.

For instance, bit 1 of the HP-IB's interrupt-enable register is used to enable and disable service-request interrupts. To enable this event to initiate a branch, bit 1 must be set to a "1". Specifying a mask parameter of "2" causes a value of 2 to be written into this register, thus enabling *only* service requests to initiate branches.

```
ENABLE INTR 7;2
```

Most Significant Bit

Least Significant Bit

| Bit 15 | Bit 14 | | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|
| ← Other interrupt causes described in subsequent sections → | | | | | Service Request | See Subsequent Sections |
| Value = −32 768 | Value = 16 384 | | Value = 8 | Value = 4 | Value = 2 | Value = 1 |

**HP-IB Interrupt-Enable Register**

The mask parameter is optional:

- If it is included, the specified value is written into the appropriate register of the specified interface.
- If this parameter is omitted, the mask specified in the *last* ENABLE INTR is used. If no ENABLE INTR statement has been executed for the specified interface, a value of 0 is used (all interrupt events disabled).

For example, the following statement re-enables a previously enabled interrupt event:

```
ENABLE INTR 7
```

Since no interrupt-enable mask is specified, the last mask used to enable interrupts on this interface is used.

**Service Requests.** You can program a service routine to perform any task(s) that is "requested" by the device that initiated the branch. If this event can occur for only one reason, the service routine just performs the specified action. However, with many devices, the service request can occur for several different reasons. In this case, the program must have a means of determining *which* event(s) occurred and then take action.

The following program shows an example of using a service routine that can be initiated by only one cause — a service request from a device at address 22 on the built-in HP-IB interface.

```
100   ! Example of service routine for HP-IB service requests.
110   !
120   ON INTR 7,5 CALL Intr7    ! Set up interface, priority,
130                             ! branch type, and location.
140                             !
150   ENABLE INTR 7;2           ! Only service requests
160                             ! (bit 1) are enabled.
170                             !
180 Loop: GOTO Loop             ! Idle loop.
190                             !
200   END
210                             !
220  SUB Intr7
230       Z=SPOLL(722)          ! Clear INTR cause first.
240                             !
250       ENTER 722;Reading     ! Take desired action.
260                             !
270       ENABLE INTR 7         ! Re-enable service requests.
280                             !
290     SUBEND
```

The program shows the sequence of steps required to set up and enable interrupt events. These steps are as follows:

1. The interrupt event is *set up* to be logged, as in line 120. This statement also assigns the event's software priority; in this case, the priority is 5.
2. The event must be *enabled* to initiate its branch, as in line 150. The mask value specifies that only service requests (enabled by setting bit 1) can initiate branches.
3. When the event occurs it is *logged*. Any further interrupts from this interface are automatically disabled until this interrupt event is serviced.
4. *Determine the interrupt's cause.* On HP-IB interfaces, a serial poll (line 230) must be performed by the service routine, clearing the interrupt-cause register so that the same event will not cause another branch upon return to the interrupted context. The value obtained from the serial poll operation can then be used to determine the interrupt's cause. (The serial poll is particular to the HP-IB interface, but analogous actions can be performed to determine interrupt causes on other interfaces.)
5. The actual *requested action is performed* (line 250).

6. If subsequent events are to also initiate branches, they must be *re-enabled* before resuming execution of the previous program segment, as in line 270. Since no interrupt-enable mask is explicitly specified, the previous mask is used.

**Interrupt Conditions.** The conditions that can be sensed by each type of interface are different. All interrupt conditions signal to the computer that either its assistance is required to correct an error situation or an operating mode of the interface has changed and must be made known to the computer.

The following service routine demonstrates typical action taken when a receiver-line status ("RLS") interrupt condition is sensed by the serial interface.

```
100   ! Example of interface-condition interrupt event.
110
120   ON INTR 9,4 CALL Intr_9  !  Set up for interface select
130                            !  code 9 and priority of 4.
140   ENABLE INTR 9;4          !  Bit 2 in mask enables
150                            !  "RLS"-type interrupts only.

 .

 .  Main program.

 .

600   SUB Intr_9
610         !
620         STATUS 9,10;Intr_cause  ! Clear intr.-cause reg.
630         !
640         ! Check errors and branch to "fix" routines.
650         !
660         IF BIT(Intr_cause,3)=1 THEN GOTO Framing error
670         IF BIT(Intr_cause,2)=1 THEN GOTO Parity_error
680         IF BIT(Intr_cause,1)=1 THEN GOTO Overrun_error
690         IF BIT(Intr_cause,0)=1 THEN GOTO Recv_buf_full
700         ENABLE INTR 9,4        ! Ignore others, re-enable
710         SUBEXIT                ! INTRs, and return.
720         !
```

```
730 Framing_error: ! "Fix" and re-enable.
740                SUBEXIT
750                !
760 Parity_error:  ! "Fix" and re-enable.
770                SUBEXIT
780                !
790 Overrun_error: ! "Fix" and re-enable.
800                SUBEXIT
810                !
820 Recv_buf_full: ! "Fix" and re-enable.
830                SUBEXIT
840      SUBEND
```

## Interface Timeouts

A "timeout" occurs when the handshake response from any external device takes longer than the specified amount of time. The time specified for the timeout event is usually the maximum time that a device can be expected to take to respond to a handshake during an I/O statement.

**Setting Up Timeout Events.** The ON TIMEOUT statement sets up an event-initiated branch. The software priority of this event *cannot* be assigned by the program; it is permanently assigned priority 15. The maximum time that the computer will wait for a response from the peripheral can be specified in the statement with a resolution of 0.001 seconds.

For example, the following statement sets up a timeout to occur after the Serial Interface has not detected a response from the peripheral for 0.200 seconds. A branch to a subroutine called "Serial_down" then occurs:

```
ON TIMEOUT 9,.2 GOSUB Serial_down
```

To set up a timeout after 0.060 seconds for the interface at select code 8, you could use the following statement:

```
ON TIMEOUT 8,.06 GOTO Hp_ib_status
```

**Timeout Limitations.** Timeout events cannot be set up for any of the internal interfaces except the built-in HP-IB.

Event-initiated branches are only executed at certain times during program execution, usually after a program line has been executed. Consequently, BASIC may wait up to 25 percent longer than the specified time to detect a timeout event; however, it will *always* wait *at least* the specified amount of time before generating the interrupt.

There is no default timeout time parameter. *Thus, if no ON TIMEOUT is executed for a specific interface, the computer will wait indefinitely for the device to respond.* The only way that the computer can continue executing the program is for the operator to use the CLR I/O (Break) key. This key aborts the I/O operation that was left "hanging" by the failure of the device to respond to and complete the handshake.

The times specified for timeouts are passed to subprograms. Thus, unless the time for a timeout event is changed in the subprogram, it remains the same as it was in the calling routine. If the time parameter is changed by the subprogram, it is restored to its former value upon return to the calling context.

# I/O Path Attributes

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. Since these attributes are implicit to the I/O path name, they are called *I/O path attributes*. I/O path attributes are available for such purposes as controlling data representations, generating and checking parity, and defining special end-of-line (EOL) sequences.

## The FORMAT Attributes

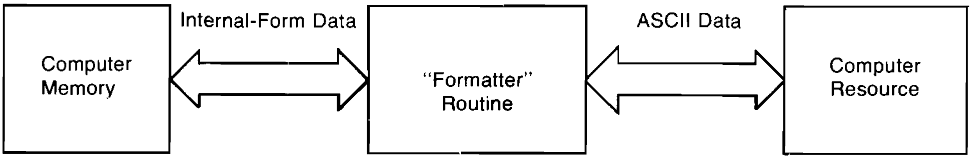
All I/O paths used as means to move data have certain attributes, which involve both hardware and software characteristics. For instance, some interfaces handle 8-bit data, while others can handle either 8-bit or 16-bit data. Some I/O operations involve sending ASCII data (for "human consumption"), while others may involve sending data in an "internal" form (that is easier for the computer to understand). This second characteristic, data representation, is what the FORMAT attributes control.

All I/O paths possess one of two available FORMAT 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.

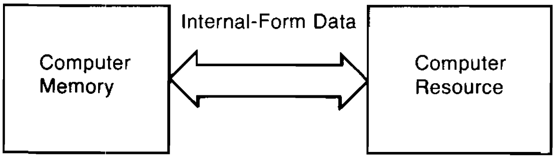
**FORMAT ON.** With FORMAT ON, internally represented numeric data must be "formatted" into its ASCII representation before being sent to the device. Conversely, numeric data being received from the device must be "unformatted" back into its internal representation. These operations are shown in the diagram below:



**Numeric Data Transfer with FORMAT ON**

For more information about the ASCII data format, refer to chapter 12, "Introduction to I/O." For more information about how items and I/O statements are terminated, refer to chapter 13, "Outputting and Entering Data."

**FORMAT OFF.** 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 BASIC representation.

Here is 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.
- COMPLEX: same as two REAL values.
- 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.
- No *non-default* CONVERT or PARITY attribute may be assigned to the I/O path (discussed later in this section).
- 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 chapter 12, 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 and BUFFERs, as discussed later in this chapter and in the next chapter.)
- *BDAT and HPUX files* — the default for BDAT and HPUX files is FORMAT OFF. (This is because for numeric quantities, 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.

**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, except WORD, are restored to their default state (such as FORMAT ON for devices.)

```
340   ASSIGN @Device  !  Restores ALL default attributes.
```

## Additional Attributes

The first section discussed the FORMAT attributes of I/O path names. Several other attributes are available to direct the BASIC system to perform the following operations whenever data are moved through the I/O path possessing the attribute:

- specify that data are to be sent and received on a byte or word basis
- perform conversions on a character-by-character basis on inbound and/or outbound data
- check for parity on inbound data, and generate parity on outbound data
- re-define the end-of-line sequence normally sent after the last data item in output operations

It is also possible to direct the system to return a numeric code to a variable which describes the outcome of an attempted ASSIGN operation. This section describes implementing these functions by using the additional I/O path attributes.

**The BYTE and WORD Attributes.** The HP Series 200/300 computers are capable of handling data as either 8-bit bytes or 16-bit words when using 16-bit interfaces. This section describes how to use the BYTE and WORD attributes to determine which way the system will handle data when using these interfaces.

*Unless otherwise specified, the system treats data as bytes during I/O operations.* For instance, when the following I/O statement is executed:

```
OUTPUT Device_selector;Integer_array(*)
```

the 16-bit INTEGER values are normally sent one byte at a time, with the most significant byte of each INTEGER sent first. Executing the following statement:

```
OUTPUT Device_selector USING "W";Integer_array(*)
```

directs the system to send the data as words *if* the interface has the ability to handle data as words. With a 16-bit interface, such as a GPIO Interface, the INTEGER data are sent one word at a time (i.e., one word per handshake cycle). If the interface is not capable of sending one word in a single operation, the word is sent as two bytes with the most significant byte first.

When the BYTE attribute is assigned to an I/O path name, the system sends and receives all data through the I/O path as bytes; one byte is sent (or received) per operation. *Thus, BYTE directs the system to treat a 16-bit interface as if it were an 8-bit interface.* The following statements show examples of assigning the BYTE attribute to an I/O path:

```
ASSIGN @Printer TO 701; BYTE
ASSIGN @Device  TO  12; BYTE
```

In the first statement, the BYTE attribute is redundant, because the WORD attribute cannot be assigned to the HP-IB Interface (since it is an 8-bit interface).

When the I/O path name assigned to an interface possesses the BYTE attribute, the system sends and receives all subsequent data through the interface one byte per handshake operation. As an example, executing either of the following statements (when the I/O path possesses the BYTE attribute):

```
OUTPUT @Device;Integer_array(*)
OUTPUT @Device USING "W";Integer_array(*)
```

directs the system to send the data as bytes, even though the interface is capable of sending the data as words (and in the second example the "W" specifier was used). Stated again, the BYTE attribute directs the system to treat 16-bit interfaces as if they were 8-bit interfaces. With BYTE, only the 8 least significant bits of the interface are used to send and receive data; the most significant bits are always zeros. Keep in mind that the logic sense of the signal lines used to send and receive these bits is determined by switch settings on the interface card.

The *WORD attribute* specifies that all data sent and received through the I/O path are to be moved as words. *In other words, this attribute directs the system to use all 16 data lines of a 16-bit interface for all subsequent I/O operations that use the I/O path name.* This attribute is designed to improve performance in two types of situations (on 16-bit interfaces): when sending and receiving FORMAT OFF data, and when sending and receiving INTEGERs with FORMAT ON. The WORD attribute can also be used under other situations; however, results may show some unexpected "side effects," which are explained later in this section. The interface to which the I/O path name is assigned must be capable of handling data words; if not, an error will be reported when the ASSIGN is executed.

When an I/O path possesses the WORD attribute, an even number of data bytes will always be sent or received by any one I/O statement that uses the I/O path. Consequently, when an operation involves an odd number of data bytes, the system will place pad byte(s) in outbound data or enter (but ignore) additional byte(s) of inbound data. These operations can be thought of as "aligning data on word boundaries." This is the main side effect that can occur with the WORD attribute.

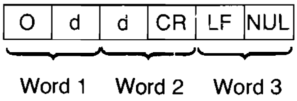
With the FORMAT OFF attribute, all data items are represented by an even number of bytes (see the discussion in "The FORMAT OFF Attributes" earlier in this section for details). Since these representations use an even number of bytes, no pad bytes are necessary.

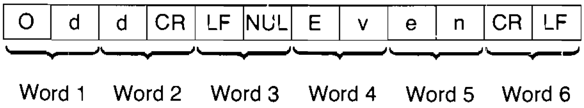
When WORD is used with FORMAT ON, the data will be buffered (automatically by the system) when necessary to allow sending all data as words. Sending INTEGERs does not usually require this type of buffering, because each INTEGER consists of two bytes of data. However, sending strings of odd length often requires that the system perform this automatic buffering. The first byte of each word is placed in a two-character buffer (created by the system); when the second byte is placed in this buffer, the two bytes are sent as one word, with the most significant eight bits representing the first byte. If an odd number of data bytes would otherwise be sent, a Null character, CHR$(0), is placed in the buffer to "flush" the last byte.

The following statements show assigning the WORD attribute and using the I/O path to send data through the GPIO Interface at select code 12. Remember that the default FORMAT attribute assigned to I/O paths to devices is FORMAT ON.

```
110  ASSIGN @Gpio TO 12;WORD
120  OUTPUT @Gpio;"Odd"
130  OUTPUT @Gpio USING "K,L,K";"Odd","Even"
```

The following diagrams show the characters that would be sent by the OUTPUT statements in lines 120 and 130, respectively.





In the first statement, a Null was sent after the EOL characters to flush the buffer and force word alignment for a subsequent OUTPUT. The second statement shows that a pad byte will be sent after any EOL sequence when required to achieve word alignment; the Null pad byte was not needed after the second EOL sequence. In addition, if a buffer or file pointer currently has an odd value, a leading pad byte will be output to force word alignment before any data are sent by the OUTPUT statement.

When executing an ENTER statement from an I/O path with the WORD attribute, the system always reads an even number of bytes from the source device, since data are sent as words. In cases where an odd number of data bytes are sent, such as when an odd number of string characters are sent with an even number of statement-terminator characters, the system enters (but ignores) the last byte sent (after the statement-terminator characters). The following statements show an example of entering the data sent by the OUTPUT statements in the preceding example.

```
ASSIGN @Device TO 12;WORD
ENTER @Device;String_var1$
ENTER @Device;String_var2$
ENTER @Device;String_var3$
```

The variables receive the following values:

```
String_var1$="Odd"
String_var2$="Odd"
String_var3$="Even"
```

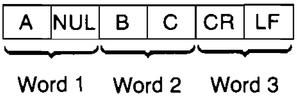
Notice that three ENTER statements were used to enter the data sent by the two preceding OUTPUT statements. This method was used to handle the pad bytes generated by the OUTPUT statement. If two ENTER statements would have been used, the pad byte sent after the second "Odd" and EOL sequence would have to have been skipped by an "X" image specifier. The following ENTER statements show how this could be done.

```
ENTER @Device USING "K,X,K";String_var1$,String_var2$
ENTER @Device USING "K";String_var3$
```

If the "X" specifier would not have been used, a pad byte would have been placed in `String_var2$`. Thus, a *general recommendation* for entering data OUTPUT through an I/O path with the WORD and FORMAT ON attributes is to enter only one item per ENTER statement.

When the WORD attribute is in effect, the "W" image specifier sends data that are always aligned on word boundaries. For instance, the following statement shows how the system defines "W" with the WORD attribute during OUTPUT.

```
OUTPUT @Device USING "B,W";65,256*66+67
```



The Null (NUL) pad byte was sent before the "W" image data to align the INTEGER specified by the "W" on a word boundary.

During ENTER, a pad byte is entered (but ignored) when necessary to align the "W" item on a word boundary. For instance, the following statement would enter the preceding data items in the same manner as they were sent.

```
ENTER @Device USING "B,W";One_byte,One_word
```

Keep in mind that these examples have been provided only to show potential problems that can arise when sending an odd number of data bytes while using the WORD attribute. It would be more appropriate to use only images that send an even number of bytes when using WORD during OUTPUT, and it will simplify matters to send only one item per OUTPUT statement. Similarly, it is generally much simpler if only one item is entered per ENTER statement.

Furthermore, if pad bytes pose a problem when working with INTEGER data (with FORMAT ON), you can also use the "Y" specifier. During OUTPUT, the "Y" does not force word alignment by sending a pad byte; during ENTER, the "Y" does not skip a byte to achieve word alignment.

Note also that the Null character pad byte may be converted to another character by using the CONVERT attribute; see the next section for further details.

The BYTE and WORD attributes affect any ENTER, OUTPUT, or TRANSFER statements that use the I/O path name. However, only the attribute specified on the non-buffer I/O path end of the TRANSFER is used; BYTE or WORD is ignored on the buffer end.

Unlike other attributes, the BYTE and WORD attribute cannot be changed once assigned to an I/O path name. For instance, executing:

```
ASSIGN @Printer TO 12
```

implicitly assigns the BYTE attribute to @Printer, since it is the default attribute. Executing the following statement results in error 600 (`Attribute cannot be modified`):

```
ASSIGN @Printer;WORD
```

The converse situation is true for the WORD attribute. Furthermore, if WORD has been assigned to the I/O path, then BYTE is not restored when `ASSIGN @Device` is executed; all other default attributes would be restored. For instance, executing:

```
ASSIGN @Device TO 12;WORD,FORMAT OFF
```

assigns the specified non-default attributes to the I/O path name @Device. Executing:

```
ASSIGN @Device
```

restores the default attribute of FORMAT ON (and also other default attributes, if currently non-default), but it *does not* restore the default BYTE attribute.

**Converting Characters.** The CONVERT attribute is used to specify a character-conversion table which is to be used for OUTPUT or ENTER operations. If data are to be converted in both directions, a separate conversion table must be defined for each direction. Two conversion methods are available — by index and by pairs. This section shows simple examples of each.

CONVERT...BY INDEX specifies that each original character's code is used to index a replacement character in the specified conversion string. For instance, CHR$(10) is replaced by the 10th character in the conversion string. The only exception is that CHR$(0) will be replaced by the 256th character in the conversion string. If the string contains less than 256 characters, characters with codes that do not index a conversion-string character will not be converted. If the string contains more than 256 characters, error 18 is reported.

The following program shows an example of setting up a conversion by index for OUTPUT operations.

```
100   DIM Conv_string$[256]
110   INTEGER Index_val
120   !
130   ! Generate conversion string.
140   FOR Index_val=1 TO 255
150     SELECT Index_val
160     CASE NUM("a") TO NUM("z")    ! Change to uppercase.
170       Conv_string$[Index_val]=UPC$(CHR$(Index_val))
180     CASE ELSE  ! No conversion.
190       Conv_string$[Index_val]=CHR$(Index_val)
200     END SELECT
210   NEXT Index_val
220   Conv_string$[256]=CHR$(0) ! 256th element has an
230                             ! effective index of 0.
240                             !
250   ! Set up conversions.
260   ASSIGN @Device TO 1;CONVERT OUT BY INDEX Conv_string$
270   !
280   OUTPUT @Device;"UPPERCASE LETTERS ARE NOT CONVERTED."
290   OUTPUT @Device;"Lowercase letters are converted."
300   OUTPUT 1;"Conversions are made only "
310   OUTPUT 1;"when the I/O path is used."
320   !
330   END
```

The program is designed to convert lowercase characters to uppercase characters. In order to make the conversion, the program first computes the characters in the conversion string; the characters are computed one at a time. If the character's original code is not in the range 97 to 122 ("a" to "z"), then no change is made. If it is in the range, an uppercase character is placed in the string at the location indexed by the original (lowercase character's) code.

The example program's output is as follows:

```
UPPERCASE LETTERS ARE NOT CONVERTED.
LOWERCASE LETTERS ARE CONVERTED.
Conversions are made only
when the I/O path is used.
```

To perform the lowercase-to-uppercase conversion, it was not necessary to include characters with codes 123 through 255 in the conversion string, since these characters are not to be converted.

They were included to emphasize that the 256th character must be included in the string if CHR$(0) is to be converted with this method. The CONVERT attribute is then assigned to the I/O path, and all subsequent data sent through the I/O path (while CONVERT is in effect) will be converted.

CONVERT...BY PAIRS specifies that the conversion string contains pairs of characters, each pair consisting of an original character followed by its replacement character. Before each character is moved through the interface, the original characters in the conversion string (the odd characters) are searched for the character's occurrence. If the character is found, it will be replaced by the succeeding character in the conversion string; if it is not found, no conversion takes place. If duplicate original characters exist in the conversion string, only the first occurrence is used. The string variable must contain an even number of characters; if not, error 18 is reported.

The following program shows an example of setting up the same conversion as in the preceding example, except that conversion by pairs is used.

```
100   DIM Conv_string$[512]
110   !
120   ! Define conversion string.
130   Conv_string$="aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpP"
140   Conv_string$=Conv_string$&"qQrRsStTuUvVwWxXyYzZ"
150   !
160   ! Set up conversions.
170   ASSIGN @Device TO 1;CONVERT OUT BY PAIRS Conv_string$
180   !
190   OUTPUT @Device;"UPPERCASE LETTERS ARE NOT CONVERTED."
200   OUTPUT @Device;"Lowercase letters are converted."
210   OUTPUT 1;"Conversions are made only "
220   OUTPUT 1;"when the I/O path is used."
230   !
240   END
```

The pairs method only requires that each character to be replaced (and its replacement) is included in the conversion string. Note that the first character of each pair is the original character and the second is the replacement. If a character does not appear in the conversion string, it will not be converted.

Conversion of inbound characters can also be performed with both of these methods. In the second example, for instance, the conversion is implemented with the following statement.

```
ASSIGN @Device;CONVERT IN BY PAIRS Conv_string$
```

Conversions in both directions will continue until disabled. The following statement could be used to disable conversions of outbound data.

```
ASSIGN @Device;CONVERT OUT OFF
```

It is important to note that the conversion string specified in the ASSIGN statement is used for each OUTPUT or ENTER statement that uses the I/O path while the conversion is enabled. Note that the conversion string's contents are not contained in the I/O path data type; only a pointer to the string variable is maintained. Thus, any changes to the string's value will immediately affect any subsequent OUTPUT or ENTER that uses that I/O path.

It is also important to note that the string must be defined for at least as long as the I/O path which references it; this "lifetime" requirement has several implications. If the I/O path and conversion string are defined in different COM blocks, an error will be reported. If the I/O path is to be used as a formal parameter in a subprogram, the conversion string variable must either appear in the same formal parameter list or be defined in a COM block accessible to that subprogram. If the I/O path name is passed to subprogram(s) by including it as a pass parameter, the string variable must currently be defined in the context which defined the I/O path.

When CONVERT OUT is in effect, the specified conversions are made after any end-of-line (EOL) sequence has been inserted into the data, but before parity generation is performed (with the PARITY attribute). When CONVERT IN is in effect, conversions are made after parity is checked (if enabled), but before the data are checked for any item- or statement-termination characters.

Keep in mind that no non-default CONVERT attribute can be assigned to an I/O path that currently possesses the FORMAT OFF attribute, and vice versa.

**Changing the EOL Sequence.** 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.

In order to define non-default EOL sequences to be sent by the OUTPUT statement, an I/O path must be used. The EOL sequence is specified in one of the ASSIGN statements which describe the I/O path. An example is as follows.

```
ASSIGN @Device TO 12;EOL CHR$(10)&CHR$(10)&CHR$(13)
```

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. The characters are put into the output data before any conversion is performed (if CONVERT OUT is in effect).

If END is included in the EOL attribute, an interface-dependent "END" indication is sent with (or after) the last character of the EOL sequence. However, if no EOL sequence is sent, the END indication is also suppressed. The following statement shows an example of defining the EOL sequence to include an END indication.

```
ASSIGN @Device TO 20;EOL CHR$(13)&CHR$(10) END
```

With the HP-IB Interface, the END indication is an End-or-Identify message (EOI) sent with the last EOL character. Refer to chapter 16 for information on the END indication for other interfaces (if implemented).

If DELAY is included, the system delays the specified number of seconds (after sending the last EOL character and/or END indication) before executing any subsequent BASIC statement.

```
ASSIGN @Device;EOL CHR$(13)&CHR$(10) DELAY 0.1
```

This parameter is useful when using slower devices which the computer can "overrun" if data are sent as rapidly as the computer can send them. For example, a printer connected to the computer through a serial interface set to operate at 300 baud might require a delay after receiving a CR character to allow the carriage to return before sending further characters. Note that the DELAY parameter is not exact; it specifies the minimum amount of delay.

The default EOL sequence is a CR and LF sent with no end indication and no delay; this default can be restored by using the EOL OFF attribute.

**Parity Generation and Checking.** Parity is an indication used to help determine whether or not a quantity of data has been communicated without error. The sending device generates the parity indication, which is then checked against the parity expected by the receiving device. If the two indications don't agree, a parity error is reported.

With this system, parity may be indicated by the most significant bit of a data byte. The parity bit is generated (during OUTPUT) or checked (during ENTER) by the system according to the current PARITY attribute in effect for the I/O path through which the data bytes are being sent or received.

Unless otherwise specified, the system will not generate or check parity (the default mode is PARITY OFF). The following optional PARITY attributes are available:

### Optional PARITY Attributes

| Option | Effect During ENTER | Effect During OUTPUT |
|---|---|---|
| OFF | No check is performed | No parity is generated |
| EVEN | Check for even parity | Generate even parity |
| ODD | Check for odd parity | Generate odd parity |
| ONE | Check for parity bit set (1) | |
| | Set parity bit (1) | |
| ZERO | Check for parity bit clear (0) | Clear parity bit (0) |

If PARITY EVEN is specified, the parity bit will be a 1 when required to make the total number of 1's in the byte an even number; for instance, a byte with a value of 1 will have the parity bit set to 1 with even parity. Conversely, PARITY ODD specifies that the parity bit will be a 1 when required to make the total number of 1's odd. PARITY ONE specifies that the parity bit will always be 1, while PARITY ZERO specifies that it will always be 0. PARITY OFF disables parity generation and checking, if currently enabled for the I/O path.

To enable parity generation during OUTPUT and ENTER operations, assign a PARITY option to an I/O path. For example:

```
ASSIGN @Serial TO 9;PARITY ODD
```

specifies that all data sent through the I/O path @Serial will use the most significant bit of each byte for parity. However, only 128 different characters will be available, since one bit of the eight is not available for data representation.

If the system detects a parity error while executing an ENTER statement, error 152 (`Parity error`) will be reported. All characters entered up to (but not including) the erroneous byte will be assigned to the appropriate variable, after which the system will report the error.

If the receiving device detects a parity error, it will be responsible for communicating the error to the computer. A typical means would be to enable the interface to signal the error by generating an interrupt. Refer to "Interrupts and Timeouts" earlier in this chapter.

Parity is generated after conversions have been made during OUTPUT and is checked before conversions during ENTER. After parity is checked on inbound data, the parity bit is cleared; however, when PARITY OFF is in effect, bit 7 is not affected.

Disabling parity generation and checking is accomplished by assigning the PARITY OFF attribute to the I/O path.

```
ASSIGN @Serial;PARITY OFF
```

Parity is also disabled when an I/O path name is explicitly closed and then re-assigned, when an I/O path name is re-assigned without being closed, and when the default attributes are restored with statements such as ASSIGN @Serial.

Keep in mind that a non-default PARITY attribute cannot be assigned to an I/O path that currently possesses the FORMAT OFF attribute, and vice versa.

**Determining the Outcome of ASSIGN Statements.** Although RETURN is not an attribute, including it in the list of attributes directs the system to place a a numeric code that indicates the outcome of the ASSIGN operation into the specified numeric variable. The following statement shows an example of enabling this error check:

```
ASSIGN @Device TO 12;RETURN Outcome
```

- If the operation is successful, a 0 is returned.
- If a non-zero value is returned, it is the error number which otherwise would have been reported. For instance, if an interface was not present at select code 12, the system would have placed a value of 163 in `Outcome`. This value is the error code for `I/O interface not present`.

The following statement shows a method of determining the Open/Closed status of the I/O path.

```
ASSIGN @Device;RETURN Closed_status
```

If @Device is currently Open, then 0 is returned; if it is Closed, then 177 is returned (`Undefined I/O path name`). When RETURN is used in this manner, the default attributes are not restored.

When RETURN is used in this manner, ON ERROR is normally disabled during the ASSIGN statement; however, there are certain errors which cannot be trapped by using RETURN in the ASSIGN statement.

If more than one error occurred during the ASSIGN, there is no assurance that the error number returned is either the first or the last error.

## Concepts of Unified I/O

One of the most powerful features of HP BASIC is that *by assigning different I/O path names to various computer resources you can use one set of I/O statements to access all of those resources.* For example, you can use the OUTPUT and ENTER statements, with a different I/O path for each resource, to access the CRT display, the keyboard, mass storage files, and buffers. This feature, called *unified I/O* avoids the need to use a separate set of BASIC statements to access each class of resources. Unified I/O, is an implicit attribute of I/O path names. *

The reason that unified I/O is so powerful is that it gives you great flexibility in program design. For example, you may want to design a program that collects data from an instrument, but you may want to debug the program before connecting the instrument. You can do this by first testing the program using a disk file to simulate the external device. This technique is described in "Unified I/O and Program Design" later in this section.

The "Directing Data Flow" section of chapter 12 tells how to assign an I/O path name to a device or mass-storage file. By using I/O path names in OUTPUT and ENTER statements you can communicate with several system resources:

- *String variables* — You can use ENTER and OUTPUT to move data to and from string variables. Refer to "I/O Operations with String Variables" later in this section.
- *Buffers* — You can use ENTER and OUTPUT, along with the TRANSFER statement, with buffers. Refer to chapter 15, "Transfers and Buffered I/O."
- *HP-IB peripherals* — Refer to "The HP-IB Interface" in chapter 16 for information on how to use OUTPUT and ENTER to communicate with HP-IB peripheral devices.
- *Mass-storage files* — You can use OUTPUT and ENTER to move data to and from mass-storage files. A discussion of file I/O follows. Refer to "Unified I/O and Program Design" for a practical application of this technique.

* Other attributes of I/O path names were covered in the previous section.

## I/O Paths and Mass-Storage Files

Before discussing I/O paths to mass storage files, let's look at some background information concerning data representation and file types.

**Data Representations and File Types.** As you know, the computer supports two general data representations — the *ASCII* and *internal* representations. The following criteria should be considered in choosing which data representation to use:

- Maximization of the rate at which computations can be made.
- Maximization of the rate at which the computer can move the data between its resources.
- Minimization of the amount of storage space required to store a given amount of data.
- Compatibility 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 be met if communication with an external resource 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.

There are three types of mass-storage *data files*: ASCII, BDAT, and HPUX.* Let's consider the data representations used with these file types:

- 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 HPUX (or DOS) files.

* The HP BASIC Language Processor implements the DOS file type rather than HPUX.

**BDAT and HPUX Files.** BDAT (BASIC Data) files and HPUX (or DOS) files have been designed to maximize performance rather than for compatibility:

- Both numeric and string computations are much faster.
- More data can generally be stored on a disk medium because there is no storage overhead (there are no "record holders") for numeric items.
- The *transfer rates* for each data type have 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 HPUX 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. (Note that this example is intended only to show how string data items are preceded by a 4-byte length header. Mixing FORMAT ON and FORMAT OFF data in this manner is *not* recommended.)

```
100   OPTION BASE 1
110   DIM Length$[4],Data$[256],Int_form$[256]
120   !
130   ! Create a BDAT file (1 record; 256 bytes/record.)
140   ON ERROR GOTO Already_created
150   CREATE BDAT "B_file",1
160 Already_created: OFF ERROR
170   !
180   ! Use FORMAT ON during output.
190   ASSIGN @Io_path TO "B_file";FORMAT ON
200   !
210   Length$=CHR$(0)&CHR$(0)  ! Create length header.
220   Length$=Length$&CHR$(0)&CHR$(252)
230   !
240   ! Generate 256-character string.
250   Data$="01234567"
260   FOR Doubling=1 TO 5
270      Data$=Data$&Data$
280   NEXT Doubling
290   ! Use only 1st 252 characters.
300   Data$=Data$[1,252]
310   !
320   ! Generate internal-form and output.
330   Int_form$=Length$&Data$
340   OUTPUT @Io_path;Int_form$;
350   ASSIGN @Io_path TO *
360   !
370   ! Use FORMAT OFF during enter (default).
380   ASSIGN @Io_path TO "B_file"
390   !
400   ! Enter and print data and # of characters.
410   ENTER Data$
420   PRINT LEN(Data$);"characters entered."
430   PRINT
440   PRINT Data$
450   ASSIGN @Io_path TO * ! Close I/O path.
460   !
470   END
```

**ASCII Files.** ASCII files are designed for interchangeability with other HP computer systems. This interchangeability imposes the restriction that the data must be represented with ASCII characters. Each data item sent to these files is a special case of FORMAT ON representation, *consisting of a two-byte length header followed by the ASCII characters*. 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 disk, because running the program will load the program it creates, destroying itself in the process.)

```
100    DIM Line$(1:3)[100]  ! Array to store program.
110    !
120    ! Create if not already on disk.
130    ON ERROR GOTO Already_exists
140    CREATE ASCII "ASC_FILE",1  ! 1 record.
150 Already_exists:  OFF ERROR
160    !
170    ASSIGN @Io_path TO "ASC_FILE"
180    STATUS @Io_path,6;Pointer
190    PRINT "Initially:   file pointer=";Pointer
200    PRINT
210    !
220    Line$(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) |
| BDAT files | FORMAT OFF | Yes |
| HPUX (or DOS) files | FORMAT OFF | Yes |
| ASCII files | FORMAT ON | No |
| String variables | FORMAT ON | No |
| Buffers | FORMAT ON | Yes |

**Note**

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.

The data representation used with ASCII files is a special case of the FORMAT ON representation.

## 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 using I/O path names.

**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; in fact, *data output to string variables is exactly like that sent to devices through I/O paths with the FORMAT ON attribute*.

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).

The following program outputs string and numeric data items to a string variable and then calls a subprogram which displays each character, its decimal code, and its position within the variable.

```
100   ASSIGN @Crt TO 1  ! CRT is disp. device.
110   !
120   OUTPUT Str_var$;12,"AB",34
130   !
140   CALL Read_string(@Crt,Str_var$)
150   !
160   END
170   !
180   !
190 SUB Read_string(@Disp,Str_var$)
200     !
210     ! Table heading.
220     OUTPUT @Disp;"---------------------"
230     OUTPUT @Disp;"Character  Code  Pos."
240     OUTPUT @Disp;"---------  ----  ----"
250     Dsp_img$="2X,4A,5X,3D,2X,3D"
260     !
270     ! Now read the string's contents.
280   FOR Str_pos=1 TO LEN(Str_var$)
290         Code=NUM(Str_var$[Str_pos;1])
300       IF Code&<32 THEN ! Don't disp. CTRL chars.
310           Char$="CTRL"
320       ELSE
330           Char$=Str_var$[Str_pos;1] ! Disp. char.
340       END IF
350       !
360       OUTPUT @Disp USING Dsp_img$;Char$,Code,Str_pos
370   NEXT Str_pos
380   !
390   ! Finish table.
400   OUTPUT @Disp;"---------------------"
410   OUTPUT @Disp ! Blank line.
420   !
430   SUBEND
```

The final display appears as follows:

```
--------------------
Character  Code  Pos.
---------  ----  ----
           32     1
1          49     2
2          50     3
,          44     4
A          65     5
B          66     6
CTRL       13     7
CTRL       10     8
           32     9
3          51    10
4          52    11
CTRL       13    12
CTRL       10    13
--------------------
```

Outputting data to a string and then examining the string's contents is usually a more convenient method of examining output data streams than using a mass storage file. The preceding subprogram may facilitate the search for control characters, because they are not actually displayed, which could otherwise interfere with examining the data stream.

The following example program shows how outputs to string variables can be used to reduce the overhead required in ASCII data files. The first method of outputting data to the file requires as much media space for overhead as for data storage, due to the two-byte length header that precedes each item sent to an ASCII file. The second method uses more computer memory, but uses only about half of the storage-media space required by the first method. The second method is also the only way to custom-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$(*); ! Consolidate.
420   OUTPUT @Ascii;String$     ! OUTPUT as 1 item.
430   !
440   STATUS @Ascii,5;Rec,Byte
450   Bytes_used=256*(Rec-1)+Byte-1
460   PRINT Bytes_used;" bytes used with 2nd method."
470   !
480   END
```

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

The program could have used a BDAT file, which would have resulted in using slightly less disk-media space; however, using BDAT files usually saves much more disk space than would be saved in this example.

The program also does not show that *ASCII files cannot be accessed randomly.* This is one of the major differences between using ASCII and BDAT files.

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.* 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
```

The results are as follows:

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

**Entering Data From String Variables.** Data items 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, every read operation also begins at the first position. If there are fewer data items in the string than in the ENTER statement, 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.

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.

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
```

## Unified I/O and Program Design

This application shows how HP BASIC's unified I/O language structure may help simplify using a "top-down" programming approach. In this example, a simple algorithm is first designed and then expanded into a program in a general-to-specific, step-wise manner. The top-down approach shown here begins with the general steps and works toward the specific details of each step in an orderly fashion.

One of the first things you *should* do when programming computers is to *plan the procedure before actually coding any software.* At this point of the design process, you need to have a good understanding of both the problem and the requirements of the program. The general tasks that the program is to accomplish must be described before the order of the steps can be chosen. The following simple example goes through the steps of taking this top-down approach to solving the problem.

**The Problem.** Our example problem is to write a program to monitor the temperature of an experimental oven for one hour.

**Step 1:** *Verbally describe what the program must do in the most general terms. You may want to make a chart or draw a picture to help visualize what is required of the program.*

Initialize the monitoring equipment. Start the timer and turn the oven on. Begin monitoring oven temperature and measure it every minute thereafter for one hour. Display the current oven temperature, and plot the temperatures vs. time on the CRT.

**Step 2:** *Verbally describe the algorithm. Again, try to keep the steps as general as possible.*

This process is often termed writing the "pseudo code." Pseudo code is merely a written description of the procedure that the computer will execute. The pseudo code can later be translated into BASIC-language code.

```
Setup the equipment.

Set the oven temperature and turn it on.

Initialize the timer.

Perform the following tasks every minute for one hour.

    Read the oven temperature.

    Display the current temperature and elapsed time.

    Plot the temperature on the CRT.

Turn the oven and equipment off.

Signal that the experiment is done.
```

**Step 3:** *Begin translating the algorithm into a BASIC-language program.*

The following program follows the general flow of the algorithm. As you become more fluent in a computer language, you may be able to write pseudo code that will translate more directly into the language. However, avoid the temptation to write the initial algorithm in the computer language, because writing the pseudo code is a *very important* step of this design approach!

```
100   ! This program: sets up measuring equipment,
110   ! turns an oven on, and initializes a timer.
120   ! The oven's temperature is measured every
130   ! minute thereafter for one hour.  The temp.
140   ! readings are displayed and plotted on the
150   ! CRT.
160   !
170   Rdgs_interval=60    ! 60 seconds between readings.
180   Test_length=60      ! Run test for 60 minutes.
190   Minutes=0
200   Seconds=0
210   !
220   CALL Equip_setup
```

```
230   CALL Set_temp
240   GOSUB Start_timer
250   !
260   ! Keep monitoring
270   LOOP
280     GOSUB Timer
290     !
300     IF Seconds>=Rdgs_interval THEN
310       Minutes=Minutes+1
320       CALL Read_temp
330       CALL Plot_temp
340     END IF
350     EXIT IF Minutes>=Test_length
360   END LOOP
370   CALL Off_equip
380   PRINT "End of experiment."
390   !
400   STOP
410   !
420   !
430   ! First the subroutines.
440   !
450 Start_timer:   Init_time=TIMEDATE
460                PRINT "Timer initialized."
470                PRINT
480                PRINT
490                RETURN
500                !
510 Timer: !
520        Seconds=TIMEDATE-Minutes*60-Init_time
530        DISP USING Time_image;Minutes,Seconds
540 Time_image: IMAGE "Time: ",DD," min  ",DD.D," sec"
550        RETURN
560        !
570   END
580   !
590   !
600   ! Now the subprograms.
610   !
620   SUB Equip_setup
630     PRINT "Equipment setup."
640     SUBEND
```

```
650       !
660   SUB Set_temp
670       PRINT "Oven temperature set."
680       SUBEND
690   SUB Read_temp
700       PRINT "Temp.= xx degrees F ";
710       SUBEND
720       !
730   SUB Plot_temp
740       PRINT "(plotted)."
750       PRINT
760       SUBEND
770       !
780   SUB Off_equip
790       PRINT
800       PRINT "Equipment shut down."
810       PRINT
820       SUBEND
```

At this point, you should run the program to verify that the general program steps are being executed in the desired sequence. If not, keep refining the program flow until all steps are executed in the proper sequence. This is also a very important step of your design process; the sooner you can verify the flow of the main program the better. This approach also relieves you of having to set up and perform the actual experiment as the first test of the program.

Notice also that some of the program steps use CALLs while others use GOSUBs. The general convention used in this example is that subprograms are used only when a program step is to be expanded later. GOSUBs are used when the routine called will probably not need further refinement. As the subprograms are expanded and refined, each can be separately stored and loaded from disk files, as shown in the next step.

**Step 4:** *After the correct order of the steps has been verified, you can begin programming and verifying the details of each step (known as step-wise refinement).*

The computer features a mechanism by which the process of expanding each step can be simplified. With it, each subprogram can be expanded and refined individually and then stored separately in a disk file. This facilitates the use of the top-down approach. Each subprogram can also be tested separately, if desired.

In order to use this mechanism, first STORE or SAVE the main program; for instance, execute:

```
STORE "MAIN1"
```

or

```
SAVE  "MAIN1"
```

Then, isolate the subprogram by deleting all other program lines in memory. In this case, executing:

```
DEL 10,620
```

and

```
DEL 660,900
```

would delete the lines which are not part of the "Equip_setup" subprogram currently in memory.

```
620   SUB Equip_setup
630       PRINT "Equipment setup."
640   SUBEND
650   !
```

At this point, two steps can be taken:

- Write the temperature-measuring device's initialization routine.
- Write a test routine that simulates the device by returning a known set of data.

The "Equip_setup" subprogram might be expanded as follows to create a disk file and fill it with a known set of temperature readings so that the program can be tested without having to write, verify, and refine the routine that will set up the temperature-measuring device. In fact, you don't even need the device at this point.

```
100  CALL Equip_setup(@Temp_meter,Temp)
110  END
120  !
130  SUB Equip_setup(@Temp_meter,Temp)
140    !
150    ! This subroutine will set up a BDAT file as
160    ! be used to simulate a temperature-measuring
170    ! device. Refine to set up the actual
180    ! equipment later.
190    !
200    ON ERROR GOTO Already
210    CREATE BDAT "Temp_rdgs",1
220    !
230    ! Output fictitious readings.
240    ASSIGN @Temp_meter TO "Temp_rdgs"
250    FOR Reading=1 TO 60
260        OUTPUT @Temp_meter;Reading+70
270    NEXT Reading
280    ASSIGN @Temp_meter TO * ! Reset pointer.
290    !
300 Already: OFF ERROR
310    !
320    ASSIGN @Temp_meter TO "Temp_rdgs"
330    !
340    PRINT "Equipment setup."
350  SUBEND
```

Notice that two pass parameters have been added to the formal parameter list. These parameters allow the main program (and subprograms to which these parameters are passed) to access this I/O path and variable. The CALL statements in the main program must be changed accordingly before the main program can be run with these subprograms. These parameters can also be passed to the subprograms by declaring them in variable common (that is, by including the appropriate COM statements).

After the subprogram has been expanded, tested, and refined, you should store it in a file with the STORE statement (not SAVE). For instance, execute:

```
STORE "SETUP1"
```

When the main program is to be tested again, the "Equip_setup" subprogram can be loaded back into memory by executing:

```
LOADSUB ALL FROM "SETUP1"
```

Since this subprogram names an I/O path which is to be used to simulate the temperature-measuring device, the "Read_remp" subprogram can also be expanded at this point. The "Read_temp" subprogram only needs to enter a reading from the measuring device (in this case, the disk file which has been set up to simulate the temperature-measuring device.) The following program shows how this subprogram might be expanded.

```
660   SUB Read_temp (@Temp_meter,Temp)
661       ENTER @Temp_meter;Temp
670       PRINT "Temp. =";Temp;" degrees F. "
680   SUBEND
```

This subprogram can also be stored in a disk file by executing:

```
STORE "READ_T1"
```

Now that both of the expanded subprograms have been stored, the main program can be retrieved and modified as necessary. Execute:

```
LOAD "MAIN1"
```

or

```
GET  "MAIN1"
```

Add the pass parameters to the appropriate CALL statements (lines 200 and 320). Since the main program still contains the initial versions of the expanded subprograms, these two subprograms should be deleted. Executing these two statements:

```
DELSUB "Equip_setup"
```

and

```
DELSUB "Read_temp
```

will delete only these two subprograms and leave the rest of the program intact.

Now that the main program has been modified to CALL the expanded/refined subprograms, you may want to store (or save) a copy of the program on the disk. This will relieve you of the effort of deleting the old subprograms from the main program every time it is retrieved. Execute:

```
STORE "MAIN2
```

or

```
SAVE "MAIN2"
```

Now load the subprograms into memory by executing:

```
LOADSUB ALL FROM "SETUP1"
```

and

```
LOADSUB ALL FROM "READ_T1"
```

Running the program "sets up" the device simulation and then accesses the file as it would access the actual temperature-measuring device.

**Conclusion.** As you can see, this approach can be used very easily with HP BASIC. *In addition, the "Read_temp" subprogram does not have to be revised to access the real device.* Only "Equip_setup" needs to be changed to assign the I/O path name "@Temp_meter" to the real device. This unified I/O scheme makes this system very powerful and reduces "throw away" code when using this "top down" approach.

15

# Transfers and Buffered I/O

This chapter discusses data transfer techniques available with the TRANS binary. While many applications will not need the specialized techniques presented here, these techniques aid in communicating with very slow and very fast devices.

## The Purpose of Transfers

When using OUTPUT and ENTER to communicate with peripheral devices, special problems can arise. Normally, program execution does not leave the statement until all data items are satisfied; therefore, a very slow device will keep the computer waiting between each byte or word. A great amount of time may be wasted while the computer waits for the device to be ready for the next item. Another problem exists when communicating with a very fast device. The device may attempt to send data faster than the computer can accept it. To overcome both problems, an alternate method of communication has been implemented — the TRANSFER statement.

The TRANSFER statement allows you to exchange information with a device or file through I/O paths. The most important difference between using TRANSFER and the regular methods of communication (OUTPUT and ENTER) is that a transfer can take place *concurrently* with continued program execution. Thus a transfer can be thought of as a "background" process or an "overlapped" operation. This has far-reaching consequences that affect the behavior of the BASIC system.

## Overview of Buffers and Transfers

Before any transfer takes place, an area of memory is reserved to hold the data being transferred (examples are shown on the following pages). This area of memory is called a buffer. Defining a buffer is somewhat analogous to creating a high-speed device inside the computer. Two advantages are gained by simulating a device in memory:

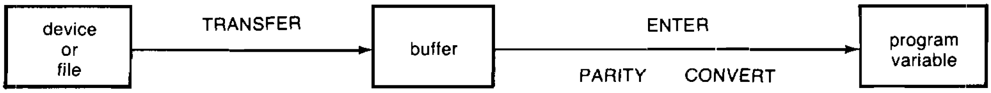
- The buffer is *fast* enough to accept incoming data from almost any device.

- The actual transfer operation can be handled *concurrently* with continued program execution (that is, it is a "background process" which can be "overlapped" with concurrent processing of other BASIC program lines).

## Inbound and Outbound Transfers

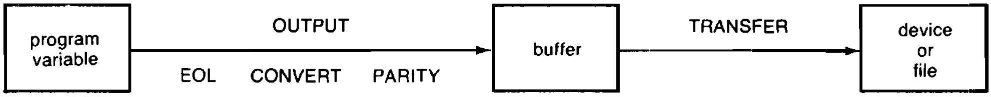
Every transfer will use a buffer as either its source or its destination. From the buffer's point of view, there are two types of transfers.

An *inbound* transfer moves data from a device or file into the buffer:



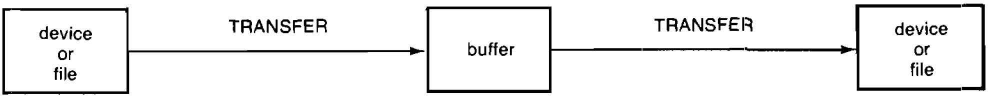
**Inbound Transfer**

An *outbound* transfer moves data from the buffer to a device or file:



**Outbound Transfer**

*Data logging* is the process of combining inbound and outbound transfers:



**Data Logging**

## Supported Transfer Sources and Destinations

TRANSFER operations are allowed only for certain types of interfaces and files.

**Note**

A transfer *cannot* involve a CRT display, a keyboard, a BCD interface, or a parallel printer interface.

One and only one buffer can be specified in a TRANSFER statement. Transfers from buffer to buffer or from device to device are *not* allowed.

Transfers to and from files on volumes with 512-byte sectors (formatting option 2) are *not* allowed since volumes with 512-byte sectors are not supported by BASIC.

Further restrictions are listed in the "Restrictions" section of this chapter.

**Interfaces.** The following table shows which interfaces are supported as TRANSFER sources or destinations for HP 9000 Series 200 and Series 300 computers and for the HP BASIC Language Processor:

**Supported Interfaces**

| | HP 9000 Series 200/300 | HP BASIC Language Processor |
|---|---|---|
| HP-IB | (built-in, HP 98624) | (built-in, HP 82990, HP 82335) |
| GPIO | (HP 98622) | (HP 82306) |
| Serial | (built-in, HP 98626, HP 98644) | - |
| Datacomm | (HP 98628) | - |

Note that the HP BASIC Language Processor does *not* support transfers through the PC serial ports, COM1 and COM2.

**File Types.** BDAT files and HPUX files (DOS files for the BASIC Language Processor) are supported sources and destinations for the TRANSFER statement. TRANSFER operations to or from ASCII files are *not* supported.

## Examples of Transfer

Here are two complete programs that show the steps in creating and using buffers. The following paragraphs describe the individual steps of the programs.

```
10    DIM Text$[1025] BUFFER
20    ASSIGN @Buff TO BUFFER Text$
30    ASSIGN @Print TO PRT          ! 'PRT' returns 701 for printer
40    !
50    FOR I=1 TO 25
60      OUTPUT @Buff;"How many times do I need to print this?"
70    NEXT I
80    !
90    TRANSFER @Buff TO @Print      ! Start the transfer
100   !                               Transfer continues as
110   FOR I=1 TO 450                  a "background" process.
120     PRINT TABXY(I MOD 15,0);"As many times as it takes."
130   NEXT I
140   END
```

Lines 10 and 20 create a named buffer. Line 30 assigns a printer that will be used as the destination for the transfer. The OUTPUT statement in line 60 fills the buffer with data (25 lines of 41 characters, including the CR/LF EOL sequence). Line 90 contains the TRANSFER statement that sends the data in the buffer to the printer. Running the program shows the overlapped operation of transfers. Buffered data is being printed on the printer while the program prints on the CRT.

A similar technique can be used for inbound transfers, as shown in the following example program.

```
10    DIM Text$[256] BUFFER,A$(100)[80]
20    ASSIGN @Buff TO BUFFER Text$
30    ASSIGN @Device TO 12            ! Some device at select code 12
40    !
50    TRANSFER @Device TO @Buff;CONT  ! Start the transfer
60    !
70    FOR I=1 TO 100
80      ENTER @Buff;A$(I)             ! Enter the items
90    NEXT I
100   ABORTIO @Device                 ! Terminate TRANSFER
110   !
120   END
```

A named buffer is created in lines 10 and 20. A device is assigned in line 30 that will be used as the source for the transfer. The buffer is filled by the TRANSFER in line 50 and the ENTER statement in line 80 empties the buffer.

## A Closer Look at Buffers

A buffer is a section of computer memory reserved to hold the data being transferred.

### Types of Buffers

Two types of buffers can be created and assigned to I/O path names.

- A *named* buffer is a string scalar, or an INTEGER, COMPLEX, or REAL array.

```
100   DIM Num_array(1:512) BUFFER       ! Named buffer.
110   ASSIGN @Buff TO BUFFER Num_array
```

- An *unnamed* buffer is a section of memory which has no associated variable name.

```
100   ASSIGN @Buff TO BUFFER [1024]     ! Unnamed buffer.
```

A named buffer can be accessed by its variable name (for instance, by using OUTPUT or assigning the variable). However, an unnamed buffer can be accessed only by its I/O path name.

### Creating Named Buffers

Named buffers are buffers which use variables declared in DIM, COM, COMPLEX, REAL, or INTEGER statements. Note that a buffer cannot be allocated by an ALLOCATE statement. Named buffers are declared by placing the keyword BUFFER after the variable name. For instance:

```
100   DIM A$[256],B$[256] BUFFER,C$

110   COM Block(1000),Temp(100) BUFFER,INTEGER X(10,10) BUFFER,Y,Z

120   REAL Fools_buff(1000), Real_buff(10) BUFFER, No_buff(10)
```

Only the variable name immediately preceding the keyword BUFFER becomes a buffer. In the first example statement, B$ is a buffer while A$ and C$ are not buffers. Declaring a variable as a buffer does not prevent it from being used in its normal manner, but care must be taken not to corrupt the information in the buffer if it is assigned to an I/O path name.

## Assigning I/O Path Names to Named Buffers

Once a named buffer has been declared, an I/O path name can be assigned to it by an ASSIGN statement. For instance:

```
ASSIGN @Path TO BUFFER B$
```

```
ASSIGN @Buff TO BUFFER X(*)
```

```
ASSIGN @Buffer TO BUFFER Real_buff(*)
```

The I/O path name can now be used to access the buffer. The keyword BUFFER must appear in both the variable declaration statement and the ASSIGN statement for named buffers.

## Assigning I/O Path Names to Unnamed Buffers

Unnamed buffers are created in ASSIGN statements and can be accessed only by their I/O path names. The following statement shows a typical unnamed buffer assignment.

```
ASSIGN @Buff to BUFFER [65536]
```

The value in brackets indicates the number of bytes of memory to be reserved for the buffer. An unnamed buffer can be larger than the maximum length (32 767 bytes) of a string variable. Named buffers using REAL, COMPLEX, and INTEGER arrays can also be larger than 32 767 bytes.

Using unnamed buffers ensures data integrity since the buffer cannot be accessed by a variable name. Closing an I/O path assigned to an unnamed buffer (`ASSIGN @Path TO *`) releases the memory reserved for the buffer. This is similar to the behavior of allocated variables.

## Buffer-Type Registers

Assigning an I/O path name to a buffer creates a control table. This control table defines STATUS and CONTROL registers which can monitor and interact with the operation of the buffer.

All I/O path names, including I/O path names assigned to buffers, use register 0 to indicate the path type.

| | |
|---|---|
| **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 |

Register 0 returns a "3" when the I/O path is associated with a buffer. Register 1 indicates whether the buffer is named or unnamed.

| | |
|---|---|
| **STATUS Register 1** | Buffer type (1=named, 2=unnamed) |

## Buffer Life Time

When I/O path names are assigned to buffers, the buffer must exist as long as the I/O path name is valid. Consider the example of a buffer created locally in a context and then assigned an I/O path name declared in COM. When execution leaves the local context, the I/O path name would still be valid but the buffer would no longer exist. If this happens, an error is reported:

```
ERROR 602 Improper BUFFER lifetime.
```

This error also occurs if the buffer and the I/O path name being assigned are in different COM areas.

## Buffer Size Register

Once a buffer has been assigned an I/O path name, Status register 2 returns the buffer's capacity (maximum size, in bytes).

| | |
|---|---|
| **STATUS Register 2** | Buffer size in bytes |

## Buffer Pointers

In order to understand I/O involving buffers, it is essential to understand how a buffer is set up and maintained.

When an ASSIGN statement associates an I/O path name with a buffer, it also creates and initializes a buffer control table. Among the entries in the *control table* are two pointers and a counter which are used to monitor and control all data transfer to and from the buffer through the I/O path.

- The buffer *fill pointer* points to the next byte of the buffer which can accept data.
- The *empty pointer* points to the next byte of data which can be read from the buffer.
- The *byte count* shows the number of bytes currently in the buffer (usually equal to fill pointer – empty pointer).

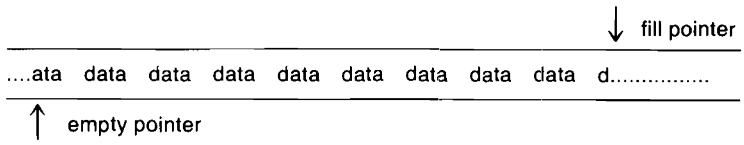
The current values of the pointers can be checked by using the STATUS statement with the following registers.

| | |
|---|---|
| **STATUS Register 3** | Current fill pointer |
| **STATUS Register 4** | Current number of bytes in buffer |
| **STATUS Register 5** | Current empty pointer |

As data is written into the buffer (OUTPUT or TRANSFER), the fill pointer is advanced as necessary to point to the next available byte of buffer storage, and the counter is incremented by the number of bytes added to the buffer.

*(inbound)*

```
TRANSFER @Device TO @Buffer
```


fill pointer
....ata data data data data data data data data d................
empty pointer

*(outbound)*

```
TRANSFER @Buffer to @File
```

Similarly, when data is read from the buffer (ENTER or TRANSFER), the empty pointer is advanced to point to the first unread byte, and the counter is decremented by the number of bytes which have been read.

It is also important to realize that the buffers used with the TRANSFER statement are *circular*. This means that when the last byte of buffer storage has been accessed, the system will wrap around and access the first byte of buffer storage. The only thing which prevents writing more data into the buffer is the byte count (Register 4) becoming equal to the buffer capacity (Register 2) which indicates that the buffer is full. Similarly, once the system has read the data from the last byte of buffer storage, it will next read from the first byte, but reading must cease when the byte count reaches zero which indicates that the buffer is empty.

A full or empty buffer has the fill pointer and the empty pointer referencing the same byte of buffer storage. The system distinguishes between full and empty by examining the byte count. If it is zero, the buffer is empty. If it is equal to the buffer's capacity, the buffer is full.

It is impossible to perform any operation which would cause the byte count to take on a value less than zero or greater than the buffer capacity. Attempting to OUTPUT more data into a full buffer or ENTER data from an empty buffer produces:

```
ERROR 59 End of file or buffer found
```

Since fill and empty pointers are updated independently of each other and a TRANSFER can execute concurrently with other statements, it is possible for one TRANSFER to be putting data into the buffer while another TRANSFER is removing data.

The amount of data which can be moved by a single transfer operation is not limited by the buffer's capacity. When two TRANSFER statements involving the same buffer are of comparable speed and execute concurrently, the buffer's fill and empty pointers may never reach the empty or full state. If the two TRANSFER statements execute at different speeds because of the transfer mode which must be used or because of the throughput capacity of the devices involved, it is still possible to keep two TRANSFER statements running concurrently by specifying the CONT parameter on both (discussed in subsequent sections). CONT directs a transfer not to terminate when the buffer becomes full or empty. Instead, the transfer "goes to sleep" until the buffer is again ready for the transfer process to continue.

## Accessing Named Buffers Using Variable Names

If you plan to transfer data through a buffer without using the I/O path name (such as by using the string varible's name or numeric array variable's name), it will be necessary to change the values of the pointers. CONTROL registers 3, 4, and 5 control the positioning of the pointers.

If either the fill or empty pointer is changed the appropriate pointer is modified and no other action is taken. Assuming no active transfer, if the byte count is changed, the empty pointer is set to zero and the fill pointer is set to correspond to the length specified. If a transfer is active in both directions, you cannot change the byte count or either pointer. If an inbound transfer is active, the empty pointer will be adjusted to set the byte count as specified. Similarly, if an outbound transfer is active, the fill pointer will be adjusted to match the byte count specified.

When the byte count is set along with either the fill or empty pointer, the pointer is moved to the position specified and the remaining pointer is adjusted to correspond to the specified length.

If all three pointers are changed, they must be a consistent set to prevent the following error:

```
ERROR 19 Improper value or out of range.
```

If both fill and empty pointers are set to the same value, the length must be either zero (buffer empty) or the maximum buffer length (buffer full).

Attempting to change a pointer used by an active TRANSFER will result in the error:

```
ERROR 612 Buffer pointer(s) in use
```

The fill pointer can be changed during an outbound transfer, but not during an inbound transfer. Similarly, the empty pointer can be changed during an inbound transfer, but not during an outbound transfer.

**Note** When string variables are used as buffers, the length of the string should *not* be changed. Although this does not affect the operation of the buffer, it can prevent access to the contents of the buffer by the variable name.

## A Closer Look at Transfers

Once a buffer has been created and an I/O path name assigned to it, data can be transferred into or out of the buffer by a TRANSFER statement. Every TRANSFER will need a buffer as either its source or destination. For example:

```
TRANSFER @Source TO @Buffer
```

or

```
TRANSFER @Buffer TO @Destination
```

From the buffer's point of view, there are two types of transfers: *inbound* and *outbound*:

- An inbound transfer will move data from a device or file into the buffer, updating a fill pointer and byte count as it proceeds.
- An outbound transfer will remove data from the buffer, updating an empty pointer and byte count as necessary.

For a complete explanation, see the "Closer Look at Buffer Pointers" section near the end of this chapter.

## Transfer Methods

The actual method of transfer is device dependent and is chosen *automatically* by the BASIC system (you cannot explicitly choose a method). The three possible transfer methods are:

- DMA (direct memory access).
- FHS (fast handshake).
- INT (interrupt).

Descriptions of each method and how the system chooses one for each TRANSFER are covered in the section called "Transfer Methods and Rates".

## OUTPUT and ENTER and Buffers

The OUTPUT and ENTER statements may be used to interact with the data sent through the buffer. If the I/O path name of the buffer is used as the source for an ENTER or the destination for an OUTPUT, the control table (pointers, size, etc.) will be updated automatically.

Accessing the data in a named buffer by using the variable name will not update the buffer pointers. This could easily lead to corruption of the data in the buffer.

## Transfer Formatting

OUTPUT and ENTER statements can format data according to a given IMAGE list and transform the data according to the attributes specified in the ASSIGN statement. *No data formatting or transformation occurs*, however, when data are transferred by a TRANSFER statement.

### Transfer Termination Branching

The ON EOT (End Of Transfer) statement allows you to define a branch to be taken upon the completion of a transfer (see the next few pages for details of TRANSFER termination conditions). When the data being transferred has been divided into records, the ON EOR (End Of Record) statement can be used to define a branch to be taken after each record is transferred.

---

**Note**

An active TRANSFER will not be terminated by stopping or pausing a program. You may use Reset (RESET) or ABORTIO to terminate a TRANSFER prematurely. The Break (CLR I/O) key will not terminate a TRANSFER.

---

### Visually Determining Transfer Status

If a TRANSFER is active while a program is paused, the "I/O" indicator (`IO` or `Transfer`) is displayed in the lower-right corner of the CRT instead of the "Pause" indicator (`-` or `Paused`). When the "I/O" indicator is displayed, any action which would make the program non-continuable (such as GET, LOAD, SCRATCH, entering a program line, etc.) will wait until the transfer completes before executing. This can give the appearance of the system being "hung." Indeed, if the TRANSFER will not complete, the system is "hung." In this last case, use Reset to recover.

## Choosing Transfer Parameters

For a standard inbound transfer, data from the device (or file) is placed in the buffer, and the TRANSFER is terminated when the buffer is full. For an outbound transfer, data is removed from the buffer, and the TRANSFER is terminated when the buffer is empty.

### Continuing Transfers Indefinitely

To allow a TRANSFER to continue indefinitely, the CONT parameter can be specified. The TRANSFER will not terminate when the buffer is full or empty.

```
TRANSFER @Source TO @Buffer;CONT
```

Several interesting things happen when a continuous TRANSFER is specified. Execution cannot leave the current program context unless the buffer and I/O path name are in COM (or passed as parameters), and you will not be able to LOAD, GET, or EDIT a program. During program development, you can terminate a transfer by RESET (Reset) or ABORTIO @Non_buff (use the I/O path name assigned to either the device or file). ABORTIO can be used in a program or executed from the keyboard.

A continuous TRANSFER can also be canceled by writing to a CONTROL register (use the I/O path name assigned to the buffer). Note that the CONTROL register only cancels the continuous mode. The TRANSFER is still active until the buffer is full or empty.

```
CONTROL @Buff,8;0        (for inbound transfers)

CONTROL @Buff,9;0        (for outbound transfers)
```

When the CONT parameter is specified for an inbound transfer, the transfer fills the buffer and is then suspended while program execution continues. The suspended transfer "sleeps" until another operation removes some data from the buffer. The transfer then "wakes up" and continues the transfer operation. When the CONT parameter is specified for an outbound transfer, the transfer empties the buffer and is then suspended. As soon as more data are available, the transfer "wakes up" and continues the transfer operation. This process proceeds until the transfer is terminated (such as with [Reset] or ABORTIO) or the CONT mode is canceled.

## Waiting for a Transfer to End (Non-Overlapped Transfers)

By default, transfers take place concurrently with continued program execution. To defer program execution until a transfer is complete, use the WAIT parameter. This allows transfers to take place serially (non-overlapped).

```
TRANSFER @Source TO @Buffer;WAIT
```

When the WAIT parameter is specified, the program statement following the TRANSFER will not be executed until the transfer has completed.

## Continuous Non-Overlapped Transfers

By combining both the CONT and WAIT parameters, a continuous non-overlapped TRANSFER can be defined. However, this is only legal if you already have an active TRANSFER for the buffer in the opposite direction.

```
TRANSFER @Source TO @Buffer;WAIT,CONT
```

## Transferring a Specified Number of Bytes

The COUNT parameter tells a transfer how many bytes are to be transferred. The following TRANSFER specifies 32 bytes to be transferred. The transfer will terminate after 32 bytes have been transferred (or when the buffer becomes full for non-continuous transfers).

```
TRANSFER @Source TO @Buffer;COUNT 32
```

## Delimiter Characters

The DELIM parameter can be used to terminate an inbound transfer when a specified character is received. The following TRANSFER will terminate when the delimiter (comma) is sent or when the buffer is full (unless the CONT parameter is specified). The DELIM parameter is not allowed on outbound transfers or WORD transfers. If the DELIM string is the null string, the DELIM clause is ignored. This allows programmatic disabling of DELIM checking. An error results if the DELIM string contains more than one character.

```
TRANSFER @Source TO @Buffer;DELIM ","
```

## Using the END Indication with Transfers

The END parameter can also be used to terminate a TRANSFER. On an outbound transfer on an HP-IB interface, for example, specifying END causes an End-or-Identify (EOI) signal to be sent with the last character of the transfer.

```
TRANSFER @Buffer TO @Device;END
```

Using an END parameter with an inbound transfer causes the transfer to be terminated by an interface-dependent signal (for devices) or by encountering the current end-of-file (for files).

```
TRANSFER @Device TO @Buffer;END
```

The END parameter is discussed in detail following the introduction of the RECORDS parameter.

## Transferring Records

It is often desirable to divide the data into records. The RECORDS parameter is then used to indicate the size of each record.

Whenever RECORDS is used, there must be a parameter which signals the end of a record. The EOR (End-Of-Record) parameter can use COUNT, DELIM, or END (discussed later) to signify the end of a record. For example, the following statement specifies 4 records of 15 bytes per record are to be transferred.

```
TRANSFER @Source TO @Buffer;RECORDS 4,EOR(COUNT 15)
```

## Multiple Termination Conditions

When multiple termination conditions are specified, the transfer will terminate when any one of the conditions occurs.

```
TRANSFER @Source TO @Buffer;COUNT 128,DELIM ";",END
TRANSFER @Source TO @Buffer;RECORDS 100,EOR(COUNT 15,END)
```

As in all transfer operations, unless the CONT parameter is specified, the TRANSFER will also terminate when the buffer is full or empty.

The END parameter specifies an inbound transfer will be terminated by receiving an interface-dependent signal (for devices) or by encountering the current end-of-file (for files). Some devices on the HP-IB send an EOI concurrently with the last byte of data. Unless the END parameter is specified, receiving an EOI will generate an error. For files, encountering the end-of-file will generate an error unless the END parameter is specified.

Using the END parameter with an outbound transfer on the HP-IB will result in the EOI signal being sent concurrently with the last byte of the transfer. If EOR(END) is specified, EOI will be sent with the last byte of each record. For files, END will cause the end-of-file pointer to be updated at the end of the transfer. Using EOR(END) will cause the pointer to be updated at the end of each record.

## TRANSFER Records and Termination

The following tables show the different system responses to the END and EOR(END) parameters.

### Inbound TRANSFER

| Parameter | File | Device |
|---|---|---|
| No END | Terminate prematurely.<br>Bit 3 of Register 10 is set.<br>Error 59 waiting. | Terminate prematurely.<br>Bit 3 of Register 10 is set.<br>Error 59 waiting. |
| END | Terminate normally.<br>Bit 3 of Register 10 is set. | Terminate normally.<br>Bit 3 of Register 10 is set. |
| EOR(END) | Finish current record.<br>ON EOR triggered.<br>Start new record. | Terminate normally.<br>Bit 3 of Register 10 is set. |
| END,EOR(END) | Terminate normally.<br>Bit 3 of Register 10 is set. | Terminate normally.<br>Bit 3 of Register 10 is set. |

An error is logged when a transfer terminates prematurely. For overlapped transfers, this error is "waiting" and will be reported the next time the non-buffer I/O path name is referenced (for example, in an ASSIGN statement). At that time, any ON ERROR or ON TIMEOUT branches will be triggered. (If the WAIT parameter is specified, the error is reported immediately.) See "Error Reporting" for further explanation.

An ON END branch will be triggered only if the END parameter is not specified.

### Outbound TRANSFER

| Parameter | File | Device |
|---|---|---|
| No END | No special action. | No special action. |
| END | Update EOF pointer after TRANSFER is finished. | Send an EOI with the last byte of each record. |
| EOR(END) | Update EOF pointer after each record. | Send an EOI with the last byte of each record. |
| END,EOR(END) | Update EOF pointer after each record and when the TRANSFER is finished. | Send an EOI with the last byte of each record and with the last byte of the TRANSFER. |

For an outbound transfer to a device, no special action is taken if the device does not support EOI. The Serial, Datacomm, and GPIO interfaces do not support EOI.

## Transfer Event-Initiated Branching

Two types of event-initiated branches can be defined for a transfer.

- The ON EOT statement defines and enables a branch to be taken upon completion of a transfer.
- The ON EOR statement defines and enables a branch to be taken every time a record is transferred.

```
ON EOT @Device CALL Process
ON EOR @File GOTO Parse
```

No ON EOR branches will be triggered unless the EOR parameter is specified in the TRANSFER statement and an item is transferred which satisfies one of the end-of-record conditions (COUNT, DELIM, or END).

To ensure that a branch receives service, the transfer must complete before attempting to leave the context in which the branches are defined. If the I/O path names are local to a program context, encountering SUBEND, SUBEXIT, or RETURN before the transfer has completed will cause the context switch to be deferred until completion of the transfer. If this happens, any ON EOR or ON EOT branch will not be serviced.

## Overlapped Nature of TRANSFER

Certain statements wait until a transfer is completed before they are executed. A complete list of these statements is provided later in this chapter. These statements can be used to prevent overlapped operation or defer a context switch until completion of the transfer. For example, if the following I/O path names were used in a TRANSFER, either of the following statements will cause program execution to wait until the transfer is finished.

`ASSIGN @Path TO *` *(can be a device, file, or buffer)*

`WAIT FOR EOT @Non_buff` *(can be a device or file)*

When a TRANSFER is used inside a loop, the entire loop may execute before the transfer has completed. If this happens, the second execution of the TRANSFER statement will wait until the completion of the first. Any event-initiated branch defined for the TRANSFER (ON EOT or ON EOR) will be serviced.

**Disabling Overlapped TRANSFER Mode.** While the WAIT parameter can be specified to ensure completion of a transfer before proceeding with the next statement (thus ensuring a branch can be serviced), this defeats any advantage of overlapped operation.

The WAIT FOR statement can be used to allow overlapped operation up to the point where the WAIT FOR statement is encountered. The WAIT FOR statement ensures the servicing of an event-initiated branch defined for the end-of-transfer or end-of-record.

## Terminating a Transfer

A transfer is usually terminated by satisfying the conditions specified by the transfer parameters. There are times, especially during program development, when you may wish to prematurely terminate (abort) a transfer.

A transfer can be aborted by pressing the [Reset] ([RESET]) key, which will stop the program, close all I/O paths, and destroy all buffer pointers.

To abort a transfer without stopping the program, the ABORTIO statement can be used from the program or the keyboard. For example:

```
ABORTIO @Non_buff
```

This statement will terminate any active transfer associated with the I/O path. ABORTIO has no effect if a transfer is not in progress. Using ABORTIO does not ensure all data in the buffer is transferred, but it does leave the buffer pointers and byte count in their correct state.

**Note**

If the destination of a TRANSFER is a mass storage file, aborting a TRANSFER with ABORTIO will not cause data already placed in the disk buffer to be written to the disk. Up to 255 bytes of data could be lost.

While most transfers are terminated by fulfilling the conditions specified by the parameters, a continuous TRANSFER (using the CONT parameter) requires a bit more effort to terminate.

To terminate a continuous TRANSFER without leaving data in the buffer, first cancel the continuous mode (with CONTROL), then wait for the transfer to complete. Use register 8 for inbound transfers and register 9 for outbound transfers. The following two methods are the safest ways of terminating a continuous TRANSFER.

```
CONTROL @Buff,8;0
WAIT FOR EOT @Path

CONTROL @Buff,8;0
ASSIGN @Path TO *
```

Remember that the buffer pointers are not reset to the beginning of the buffer when the transfer is finished. The RESET statement (RESET @Buff) can be used to reset the buffer pointers to the beginning of the buffer and the byte count to zero.

Transfers are not terminated by pausing the program. The I/O indicator in the lower-right corner of the CRT will indicate when a transfer is in progress.

While transfers may continue when the computer is in the paused state, all transfers must terminate before entering the stopped state. Pressing Return or ENTER, after editing or adding a program line, will attempt to put the computer in the stopped state. If a transfer is still in progress, the computer will "hang" until the transfer is completed. To abort the transfer without performing a hardware reset, press Break (CLR I/O) to clear the Return or ENTER and then execute an ABORTIO on the non-buffer I/O path name for each active TRANSFER. If a hardware reset can be tolerated, press Reset (RESET) to terminate the transfer.

## More Transfer Examples

Here is a short program which sets up a continuous transfer from a device through the buffer to a BDAT file. A program of this type is useful when the data being received must be saved for later analysis.

```
10    ! Data Logging Example
20    !
30    ! Buffer size should be a multiple of disk sector (256) size.
40    ASSIGN @Device TO 717          ! Assign source device on HPIB
50    ASSIGN @Buf TO BUFFER [512]    ! Assign BUFFER
60    ASSIGN @File TO "LOG_FILE"     ! Assign destination file
70    !
80    TRANSFER @Device TO @Buf;CONT  ! Continuous TRANSFER
90    TRANSFER @Buf TO @File;CONT    ! Continuous TRANSFER
100   !
110   ! Program execution continues ...
120   ! Data logging continues as a "background" task ...
130   !
140   PAUSE                  ! TRANSFER continues in paused state
150   END
```

The following program creates and fills a BDAT file and then sends its contents to a printer. Notice that the OUTPUT statement used to fill the file placed a CR/LF at the end of each record. The TRANSFER statement (line 90) looks for the carriage-return as a record delimiter.

```
10     ON ERROR CALL Makefile
20     ASSIGN @File TO "BDAT_FILE"   ! Test for file's existence
30     OFF ERROR
40     ASSIGN @Buff TO BUFFER [2046] ! Assign buffer
50     ASSIGN @Print TO PRT          ! Assign destination
60     !
70     Cr$=CHR$(13)                  ! ASCII character for carriage return
80     PRINT "Start"
90     TRANSFER @File TO @Buff;RECORDS 10,END,EOR (DELIM Cr$)
100    !
110    TRANSFER @Buff TO @Print
120    FOR I=1 TO 10000
130      PRINT "TRANSFERS RUNNING",I
140      STATUS @Buff,11;Stat
150      IF NOT BIT(Stat,6) THEN 180
160    NEXT I
170    !
180    OUTPUT @Print;CHR$(12)        ! ASCII character for formfeed
190    PRINT "File is printed"
200    END
210    !
220    SUB Makefile
230      OFF ERROR
240      CREATE BDAT "BDAT_FILE",10,12
250      ASSIGN @File TO "BDAT_FILE";FORMAT ON
260      FOR I=1 TO 10
270        DISP "Writing";I
280        READ Word$
290        OUTPUT @File;Word$
300      NEXT I
310      DISP
320      DATA ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,TEN
330    SUBEND
```

The next program continually shows the activity of the buffer. Note that a continuous TRANSFER is used (line 90). Data is placed in the buffer a few bytes at a time (line 130) and the status is displayed by the SUB called from line 140. After a few hundred bytes are transferred, the continuous mode is canceled (line 180), the program waits for the transfer to finish (line 190), and the final status is displayed.

```
20    PRINTER IS CRT
30    PRINT USING "@"                ! Clear Screen
40    COM @Buff,@Print,B$[47] BUFFER ! Declare variables
50    INTEGER Characters
60    ASSIGN @Buff TO BUFFER B$      ! Assign I/O path name to buffer
70    ASSIGN @Print TO PRT           ! Assign I/O path name to 701
80    DISP "printer is off line"     ! Transfer hangs if no printer
90    TRANSFER @Buff TO @Print;CONT  ! Continuous transfer
100   DISP                           ! Clear display line
120   REPEAT
130     OUTPUT @Buff;"AB ";          ! Fill buffer with data
140     CALL Buff_status
150     Times=Times+1
160   UNTIL Times>100
180   CONTROL @Buff,9;0              ! Cancel continuous mode
190   WAIT FOR EOT @Print            ! Wait for buffer empty
200   CALL Buff_status               ! Show final status
210   END
230   SUB Buff_status ! ---------------------------------------------
240     COM @Buff,@Print,B$ BUFFER
250     STATUS @Buff;R0
260     PRINT TABXY(1,1);"Buffer Status: ";
270     STATUS @Buff,1;R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13
280     IF R1=1 THEN PRINT "Named ";
290     IF R1=2 THEN PRINT "Unnamed ";
300     PRINT "Buffer[";VAL$(R2);"]"
310     PRINT TABXY(1,3);RPT$(" ",55)
320     PRINT TABXY(R3,3);"v"         ! Show fill pointer position
330     PRINT TABXY(1,4);"""";B$;"""" ! Show buffer contents
340     PRINT TABXY(1,5);RPT$(" ",55)
350     PRINT TABXY(R5,5);"^"         ! Show empty pointer position
360     PRINT
370     PRINT "Fill pointer:  ";R3
380     PRINT "Bytes in use:  ";R4
390     PRINT "Empty pointer: ";R5
400     PRINT
410     PRINT "           inbound/outbound"
420     PRINT "Select code:   ";R6;"/";R7
430     PRINT "Continuous? :  ";R8;"/";R9
440     PRINT "Term. status:  ";R10;"/";R11
450     PRINT "Total bytes:   ";R12;"/";R13
460   SUBEND
```

Data currently in the buffer can be reused or ignored by manipulating the pointers (with CONTROL). When it is necessary to move data through the buffer without using I/O path names, the CONTROL statement can be used to modify the pointers, thus allowing a TRANSFER to take place. The next program uses this technique. The array size used in the next program is for the Model 236; change the array size in lines 50 and 60 for other computer models.

```
10    GINIT                             ! Uses graphics
20    GCLEAR
30    GRAPHICS ON
40    PRINT CHR$(12)                    ! Clear the screen
50    INTEGER I,Graph(1:12480) BUFFER   ! (1:7500) FOR 9826/9816
60    Gbytes=2*12480                    ! 2 * 7500 FOR 9826/9816
70    ASSIGN @Buff TO BUFFER Graph(*)
80    ON ERROR GOTO Record              ! Enable ERROR trap
90    ASSIGN @Read TO "PHOTOS"          ! Test if file exists
100   ASSIGN @Read TO *                 ! Close file
110   GOTO Playback                     ! If file exists then Playback
120                                     !
130 Record:OFF ERROR
140   CREATE BDAT "PHOTOS",5,Gbytes     ! 5 "PHOTOS" of graphics screen
150   ASSIGN @Write TO "PHOTOS"         ! to be written to the BDAT file
160   FOR I=1 TO 5
170     GRID I*4,I*4
180     GSTORE Graph(*)                 ! Fill buffer with GSTORE
190     GCLEAR
200     DISP "SAVING #";I
210     CONTROL @Buff,4;Gbytes          ! Tell TRANSFER "The buffer is full"
220     TRANSFER @Buff TO @Write;WAIT
230   NEXT I
240   ASSIGN @Write TO *
250   !
260 Playback:OFF ERROR
270   ASSIGN @Read TO "PHOTOS"
280   FOR I=1 TO 5
290     DISP "LOADING #";I
300     TRANSFER @Read TO @Buff;WAIT
310     GLOAD Graph(*)
320     CONTROL @Buff,4;0               ! Tell TRANSFER "The buffer is empty"
330   NEXT I
340   DISP "DONE"
350   END
```

The program creates five "photos" of the graphics raster and writes them to a disk file. The file is then read and each picture is loaded back into the graphics raster.

# Special Considerations

## Transfer with Care

Whenever possible, a transfer will take place concurrently with continued program execution. You must carefully construct a program using transfers. *A poorly designed transfer may take longer* to execute than using OUTPUT and ENTER.

A TRANSFER which uses a local I/O path name must terminate before a SUBEXIT, SUBEND, or RETURN (from a function) can return execution to the calling context. The system will detect that such a transfer is in progress and will make the SUBEXIT wait for the transfer to terminate. If this happens, the system will not process any ON EOT (or ON EOR) branch which had been defined for the transfer. To allow servicing of the branch, any statement which cannot execute in overlap with the TRANSFER can be inserted in the subprogram before the SUBEXIT. Two of the most sensible choices are:

```
WAIT FOR EOT @Non_buff
```

or

```
ASSIGN @Path to *
```

A TRANSFER which uses only non-local I/O path names can execute in overlap with a SUBEXIT. One word of caution is necessary; if a local ON EOT (or ON EOR) statement is used in the subprogram, its branch will not be serviced if the SUBEXIT is encountered before termination of the TRANSFER. To ensure the possibility of servicing the branch, insert a statement that cannot execute in overlap with the TRANSFER. This is essentially the same technique discussed in the preceding paragraph.

More than one I/O path name can be assigned to a named buffer; however, each path name will maintain its own set of pointers. Using multiple path names on the same buffer could lead to corruption of the data in the buffer.

Special care should be taken when using REAL and COMPLEX arrays as buffers, since a device may send a bit pattern that is not a valid real number. Accessing the data as a REAL or COMPLEX value may produce an error.

## Statements That Affect Concurrency

The following statements do *not* wait for the completion of a TRANSFER statement.

| **Buffer in Use** | **Device in Use** |
|---|---|
| `STATUS @Buf` | `STATUS @Dev` |
| `CONTROL @Buf` | `ON EOR @Dev` |
| `SCRATCH A` | `ON EOT @Dev`<br>`OFF EOR @Dev`<br>`OFF EOT @Dev` |

Statements which wait for completion of inbound transfers.

```
OUTPUT @Buf
TRANSFER @Dev TO @Buf
```

Statements which wait for completion of outbound transfers.

```
ENTER @Buf
TRANSFER @Buf TO @Dev
```

Statements which wait for completion of inbound and outbound transfers.

| **Buffer in Use** | **Device in Use** |
|---|---|
| ASSIGN @Buf TO * | ASSIGN @Dev TO * |
| ASSIGN @Buf TO BUFFER[bytes] | |
| ASSIGN @Buf TO BUFFER B$ | |
| ASSIGN @Dev | ASSIGN @Dev |
| ASSIGN @Dev; (new attributes) | ASSIGN @Dev; (new attributes)<br>WAIT FOR EOT @Dev<br>OUTPUT @Dev<br>ENTER @Dev<br>TRANSFER @Buf TO @Dev<br>TRANSFER @Dev TO @Buf |
| END | END |
| SUBEXIT | SUBEXIT |
| SUBEND | SUBEND |
| SCRATCH C | SCRATCH C |
| SCRATCH | SCRATCH |
| LOAD "PROG" | LOAD "PROG" |
| GET "PROG" | GET "PROG" |
| STOP | STOP |
| | CONTROL @Dev |

## Error Reporting

If an error is encountered during an overlapped transfer, the error is logged in the non-buffer I/O path name and reported the next time the non-buffer I/O path name is referenced. Thus, the error line reported will be the most recently executed line containing the I/O path name and usually not the line containing the TRANSFER statement. For example:

```
10    !   This program shows delayed error reporting for TRANSFER
20    !
30    ON ERROR GOTO Ok
40    PURGE "bdat_file"             ! Zap file if it already exists
50 Ok:OFF ERROR
60    !
70    CREATE BDAT "bdat_file",1     ! CREATE an empty file
80    ASSIGN @Non_buf TO "bdat_file"! ASSIGN I/O path name to the file
90    INTEGER B(100) BUFFER         ! Declare a variable as a buffer
100   ASSIGN @Buf TO BUFFER B(*)    ! Assign I/O path name to buffer
110   PRINT
120   !
130   WAIT 2
140   LIST 150,150
150   TRANSFER @Non_buf TO @Buf;CONT ! Error occurs in this line
160   !
170   WAIT 2
180   LIST 190,190
190   STATUS @Buf,10;Status_byte     ! Error not reported with @Buf
200   !
210   WAIT 2
220   LIST 230,230
230   STATUS @Non_buf;Status_byte    ! Error reported with @Non_buf
240   END
```

The error displayed as a result of running the above program is:

```
ERROR 59 IN 230  End of file or buffer found
```

which indicates that the error occurred on line *230* of the program. However, the actual error occurred on line *150*. The reason for the error is that the file called bdat_file was empty and there was no END option used with the TRANSFER statement.

Since a continuous TRANSFER was specified, the error that occurs in line 150 is reported in line 230 when the non_buffer I/O path name is referenced. For continuous transfers, the error is always logged with the non-buffer I/O path name. Referencing the buffer's I/O path name (line 190) does not cause the error to be reported. After running the program, change the CONT parameter in line 150 to WAIT. The program will now report the error in line 150 since the WAIT parameter specified a serial TRANSFER.

At the time the error is reported, any ON END (for files), ON TIMEOUT (for devices), or ON ERROR statements will be triggered. However, ON END is not triggered when the END parameter is specified.

### Suspended Transfers

When a TRANSFER statement is executed, that transfer is said to be "active". The transfer proceeds until either a termination condition is reached, or until there is nothing else the transfer can do for the time being. An example of the latter is a continuous TRANSFER, which does not terminate when the buffer is full and has not yet met any other termination conditions.

This TRANSFER will be "suspended" to give some other TRANSFER operation a chance to empty the buffer. It will not be reactivated until one of the following occurs:

1. The other TRANSFER operation reaches a record boundary, fills or empties the buffer, terminates, or is suspended.
2. An OUTPUT or ENTER operation active in the other direction fills or empties the buffer, or terminates.
3. A CONTROL statement is executed to change the fill or empty pointers, or buffer's byte count.
4. A CONTROL statement is executed to cancel continuous mode.

A TRANSFER cannot be suspended unless it has CONT as one of its transfer parameters.

## Transfer Performance

### Sector Size

For the best performance when transferring BDAT and HP-UX (or DOS) files, the buffer size should be a multiple of 256 or 1024 bytes (the size of a sector on the disk).* If the buffer is not a multiple of 256 bytes, the system must do sector buffering; this is handled automatically, but reduces the transfer rate.

* Disks with 512-byte sectors are not supported by BASIC.

## Transfer Methods and Rates

The BASIC system chooses the *fastest possible* transfer method when executing a TRANSFER. *You cannot explicitly choose the method.*

There are three types of transfers available to the BASIC system.

- DMA (direct memory access).
- FHS (fast handshake).
- INT (interrupt).

**DMA Mode.** All transfers use DMA mode whenever possible. However, any one of the following reasons will prevent a DMA transfer.

- The DMA card is not present.
- Both DMA channels are busy.
- The device involved is not HP-IB or GPIO.
- The DELIM parameter is specified.

If DMA cannot be used with the HP-IB or GPIO interfaces, the FHS mode will be used if the WAIT parameter was specified and INT mode will be used if the WAIT parameter was not specified.

**INT Mode.** The INT mode will always be used for the Serial and Datacomm interfaces. Note also that the handshake lines are *not* used for Serial and Datacomm transfers. Therefore, on inbound transfers through the Serial interface, it is easy to overrun the 1-byte hardware buffer on the card. The maximum transfer rate with Serial interfaces is hard to specify, because it may be affected by other operations that attempt to alter the BASIC interrupt-logging structure (statements such as ON INTR and ON KEY). In general, using the WAIT parameter will result in a higher transfer rate, with a lower potential for overrun errors, than other methods. The WAIT parameter specifies that the TRANSFER is to complete before the next BASIC statement is executed (that is, it specifies that the transfer is to be performed in non-overlapped mode).

If a very slow device is sending a few bytes at a time, the most efficient method of transfer would be to interrupt the processor whenever data is ready. Both DMA and INT modes operate in this way. The DMA hardware "steals" a single memory cycle from the processor to transfer each byte. The INT mode must completely interrupt the processor and therefore takes more time.

Either type of interrupt (DMA or INT) can occur at any time and will be handled immediately by the system. The interrupt doesn't have to wait for a statement to end before it is serviced. This is not the same as event-initiated branches which are serviced only at the end of a statement.

**Burst Interrupt Mode.** The INT transfers implemented on the HP-IB and GPIO interfaces use a specialized "burst interrupt" mode. When an interrupt occurs, the system's interrupt service routine will transfer the byte (or word) then wait approximately 20 μs for another byte. If the device is fast enough to accept or generate another byte each 20 μs, the net transfer rate will be much faster than if the system must exit the service routine and then re-enter the routine for the next byte.

## Restrictions

All data must be buffered. This means every TRANSFER statement will have one I/O path assigned to a buffer and one I/O path assigned to a device (or file). Additionally, transfers are *not permitted* with:

- The CRT or keyboard.
- The HP 98623 BCD Interface card.
- ASCII type files.

In addition, TRANSFER to or from a mass storage device with hierarchical directories (such as HFS and SRM volumes) will not operate in overlapped mode (because of the "extensible" nature of files on these volumes).

A buffer can have only one inbound and one outbound I/O operation (using I/O path names) at any given time. The I/O operation can use TRANSFER, OUTPUT, or ENTER statements. A second I/O operation in the same direction must wait until the completion of the current operation. A second I/O operation in the opposite direction does not have to wait.

The HP-IB and GPIO interfaces support only one I/O operation at any given time. A second operation must wait until the completion of the first operation. The Serial and Datacomm interfaces allow concurrent inbound and outbound transfer operations if each TRANSFER has a unique I/O path name assigned to the device. An OUTPUT or ENTER must wait until completion of transfers in both directions. Thus, concurrent operation requires using TRANSFER statements and not a mixture of TRANSFER, OUTPUT, and ENTER statements.

The I/O path name assigned to a device can be used in only one I/O operation at a time. However, the path name can be used with OUTPUT, ENTER, and TRANSFER interchangeably. An OUTPUT or ENTER to the I/O path name will be deferred until completion of any active TRANSFER for that path name. All file operations (including CAT, CREATE, OUTPUT, and ENTER) will be deferred until completion of any TRANSFER using the same interface select code.

*Serial transfers are not supported by the HP BASIC Language Processor.*

## Interactions with Other Keywords

The TRANSFER statement restricts some of the interrupts on various devices. If an ON INTR statement and an ENABLE INTR statement have been executed for an interface, not all possible ON INTR conditions will be triggered during a transfer.

### Specific Interfaces

This section covers interface specific interactions.

**HP-IB.** For the HP-IB interface, all interrupt conditions are triggered if they occur during a transfer. However, certain interrupt conditions may occur which will cause the transfer operation to be prematurely terminated.

With the exception of the Handshake Error, the majority of interrupt conditions only occur when the HP-IB interface is configured as a non-controller. If any of the following interrupt conditions are enabled and the given interrupt occurs during a transfer to or from the interface, the user interrupt will be logged and the TRANSFER will be prematurely terminated.

- Parallel Poll Configuration Change.
- My Talk Address Received.
- My Listen Address Received.
- Talker/Listener Address Change.
- Trigger Received.
- Handshake Error.
- Unrecognized Universal Command.
- Secondary Command While Addressed.
- Clear Received.
- Unrecognized Address Command.

If one of these interrupt conditions occurs and the given interrupt condition has not been enabled, the interrupt will be ignored and the TRANSFER will not be terminated.

**Note** When an abortive interrupt condition is ignored, it is possible for data to be corrupted. It is recommended that abortive interrupt conditions be enabled during a transfer.

The Active Controller and IFC Received interrupt conditions will always prematurely terminate a TRANSFER, even if they have not been enabled.

**GPIO.** For the GPIO interface, the PFLG (data ready) interrupt is not triggered during a transfer that uses the interface. The EIR (External Interrupt Request) interrupt is triggered even if there is a transfer in progress.

**Serial.** For the Serial interface, the Transmitter Holding Register Empty and Receiver Buffer Full interrupts are not triggered during a transfer that uses the interface. The Receiver Line Status and Modem Status Change interrupts are triggered even if there is a transfer in progress.

**Datacomm.** For the Series 200/300 Datacomm interface, all interrupt conditions are triggered even if a transfer is in progress.

## Changing Buffer Attributes

You can change the I/O path name's attributes without changing the current buffer pointers. Just execute another ASSIGN statement with the new attributes. For example:

```
ASSIGN @Path;PARITY OFF
```

You will not be able to change all possible attributes in this manner. The BYTE and WORD attributes cannot be changed once assigned.

By specifying just the I/O path name, the default attributes (except BYTE) can be restored. For example:

```
ASSIGN @Path
```

See the ASSIGN statement in the BASIC *Language Reference* manual for a complete list of attributes.

**Note**

It is possible to assign more than one I/O path name to a single named buffer. Using two I/O path names on the same buffer could lead to the corruption of the data in the buffer. Although each path name maintains a separate set of buffer pointers, they are pointing to the same buffer.

## Buffer Status and Control Registers

Status register 0 indicates the resource assigned to an I/O path:

| | |
|---|---|
| **Status Register 0** | 0 = Invalid I/O path name<br>1 = I/O path name assigned to a device<br>2 = I/O path name assigned to a data file<br>3 = I/O path name assigned to a buffer |

When the status of register 0 indicates a buffer (3), the status and control registers have the following meanings.

| | |
|---|---|
| **STATUS Register 1** | Buffer type (1=named, 2=unnamed) |
| **STATUS Register 2** | Buffer size in bytes |
| **STATUS Register 3** | Current fill pointer |
| **CONTROL Register 3** | Set fill pointer |
| **STATUS Register 4** | Current number of bytes in buffer |
| **CONTROL Register 4** | Set number of bytes |
| **STATUS Register 5** | Current empty pointer |
| **CONTROL Register 5** | Set empty pointer |
| **STATUS Register 6** | Interface select code of inbound TRANSFER |

| | |
|---|---|
| **STATUS Register 7** | Interface select code of outbound TRANSFER |
| **STATUS Register 8** | If non-zero, inbound TRANSFER is continuous |
| **CONTROL Register 8** | Cancel continuous mode inbound TRANSFER if zero |
| **STATUS Register 9** | If non-zero, outbound TRANSFER is continuous |
| **CONTROL Register 9** | Cancel continuous mode outbound TRANSFER if zero |

**STATUS Register 10** Termination status for inbound TRANSFER

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 0 | TRANSFER Active | TRANSFER Aborted | TRANSFER Error | Device Termination | Byte Count | Record Count | Match Character |
| Value = 128 | Value = 64 | Value = 32 | Value = 16 | Value = 8 | Value = 4 | Value = 2 | Value = 1 |

**STATUS Register 11** Termination status for outbound TRANSFER

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 0 | TRANSFER Active | TRANSFER Aborted | TRANSFER Error | Device Termination | Byte Count | Record Count | 0 |
| Value = 128 | Value = 64 | Value = 32 | Value = 16 | Value = 8 | Value = 4 | Value = 2 | Value = 1 |

| | |
|---|---|
| **STATUS Register 12** | Total number of bytes transferred by last inbound TRANSFER |
| **STATUS Register 13** | Total number of bytes transferred by last outbound TRANSFER |

16

# Techniques for Specific Interfaces

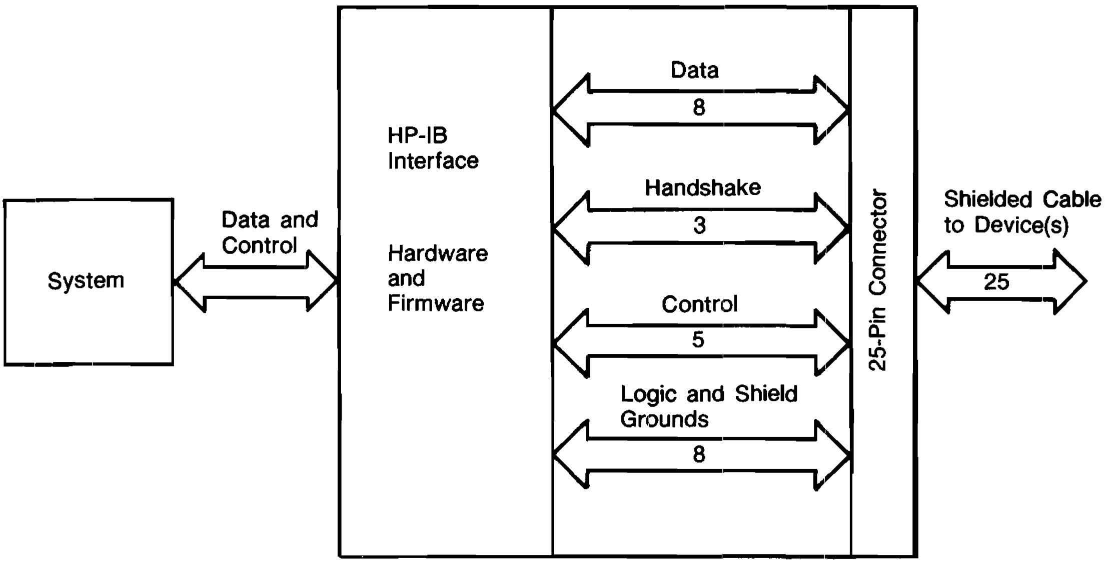
This chapter describes HP BASIC programming techniques that are specific to individual interfaces. The sections that follow cover techniques for the HP-IB, RS-232 serial, GPIO, and HP-HIL interfaces.

**Note** The "Interface Registers" appendix in the BASIC *Language Reference* manual lists the Status and Control registers for each of these interfaces. You may want to refer to that manual for additional information.

## The HP-IB Interface

This section describes the techniques necessary for programming the HP-IB interface. It also describes the specific details of how this interface works and how it is used to communicate with and control systems consisting of various HP-IB devices. Be sure you have the TRANS and IO binaries loaded in your system.

The HP-IB (Hewlett-Packard Interface Bus), commonly called the "bus", provides compatibility between the computer and external devices conforming to the IEEE 488-1978 standard. Electrical, mechanical, and timing compatibility requirements are all satisfied by this interface. The following block diagram depicts the HP-IB interface.



The HP-IB Interface is easy to use and allows great flexibility in communicating data and control information between the computer and external devices.

## Initial Installation

The built-in HP-IB interface on HP 9000 Series 200/300 computers, or on the HP BASIC Language Processor, requires no installation. It is pre-configured to hardware interrupt level 3. However, the hardware interrupt level of an external HP-IB interface can be set in the range 3 through 6. Refer to your interface owner's manual for information on installing and configuring an external interface. Refer to your computer or language processor owner's manual for information about changing the hardware configuration of the built-in HP-IB interface.

## Communicating with Devices

This section describes programming techniques used to output data to and enter data from HP-IB devices. General bus operation is also briefly described in this section.

**HP-IB Device Selectors.** Since the HP-IB allows the interconnection of several devices, each device must have a means of being uniquely accessed. Specifying just the interface select code of the HP-IB interface through which a device is connected to the computer is not sufficient to uniquely identify a specific device on the bus.

Each device on the bus has a primary address by which it is identified. This address must be unique to allow individual access of each device. Each HP-IB device has a set of switches that are used to set its address. Thus, when a particular HP-IB device is to be accessed, it must be identified with both its interface select code and its bus address.

The interface select code is the first part of an HP-IB device selector. The interface select code of the internal HP-IB is 7. External interfaces can range from 8 through 31. The second part of an HP-IB device selector is the device's primary address, which can range from 0 through 30. For example, to specify the device:

| | |
|---|---|
| On interface select code 7 with primary address 22 | Use device selector = 722 |
| On interface select code 10 with primary address 2 | Use device selector = 1002 |

Remember that each device's address must be unique. The procedure for setting the address of an HP-IB device is given in the installation manual for each device. The HP-IB interface also has an address. The default address of the internal HP-IB is 21 or 20, depending on whether or not it is a System Controller, respectively. The addresses of an external HP-IB interface's address can be determined by reading STATUS register 3 of the appropriate interface select code, and each interface's address can be changed by writing to CONTROL register 3. See "Determining Controller Status and Address" and "Changing the Controller's Address" for further details.

**Moving Data Through the HP-IB.** Data is output from and entered into the computer through the HP-IB with the OUTPUT and ENTER statements, respectively. The only difference between the OUTPUT and ENTER statements for the HP-IB and those for other interfaces is the addressing information within HP-IB device selectors. Some examples of using OUTPUT and ENTER with the HP-IB follow.

```
100  Hpib=7
110  Device_addr=22
120  Device_selector=Hpib*100+Device_addr
130  !
140  OUTPUT Device_selector;"FIR7T2T3"
150  ENTER Device_selector;Reading
```

```
320  ASSIGN @Hpib_device TO 702
330  OUTPUT @Hpib_device;"Data message"
340  ENTER @Hpib_device;Number
```

```
440  OUTPUT 822;"FIR7T2T3"
```

```
380  ENTER 724;Readings(*)
```

**General Structure of the HP-IB.** Communications through the HP-IB are made according to a precisely defined set of rules. These rules help to ensure that only orderly communication may take place on the bus. For conceptual purposes, the organization of the HP-IB can be compared to that of a committee. A committee has certain "rules of order" that govern the manner in which business is to be conducted. For the HP-IB, these rules of order are the IEEE 488-1978 standard.

On the HP-IB, the System Controller corresponds to the committee chairman. The system controller is generally designated by setting a switch on the interface and cannot be changed under program control. However, it is possible to designate an "acting chairman" on the HP-IB. On the HP-IB, this device is called the Active Controller, and may be any device capable of directing HP-IB activities, such as a desktop computer.

When the System Controller is first turned on or reset, it assumes the role of Active Controller. Thus, only one device can be designated System Controller. These responsibilities may be subsequently passed to another device while the System Controller tends to other business. This ability to pass control allows more than one computer to be connected to the HP-IB at the same time.

In a committee, only one person at a time may speak. It is the chairman's responsibility to "recognize" which one member is to speak. Usually, all committee members present always listen; however, this is not always the case on the HP-IB. One of the most powerful features of the bus is the ability to selectively send data to individual (or groups of) devices. This allows fast talkers to communicate with fast listeners without having to wait.

During a committee meeting, the current chairman is responsible for telling the committee which member is to be the talker and which is (are) to be the listener(s). Before these assignments are given, he must get the attention of all members. The talker and listener(s) are then designated, and the next data message is presented to the listener(s) by the talker. When the talker has finished the message, the designation process may be repeated.

On the HP-IB, the Active Controller takes similar action. When talker and listener(s) are to be designated, the attention signal line (ATN) is asserted while the talker and listener(s) are being addressed. ATN is then cleared, signaling that those devices not addressed to listen may ignore all subsequent data messages. Thus, the ATN line separates data from commands; commands are accompanied by the ATN line being true, while data messages are sent with the ATN line false.

On the HP-IB, devices are addressed to talk and addressed to listen in the following orderly manner. The Active Controller first sends a single command which causes all devices to unlisten. The talker's address is then sent, followed by the address(es) of the listener(s). After all listeners have been addressed, the data can be sent from the talker to the listener(s). Only device(s) addressed to listen accept any data that is sent through the bus (until the bus is reconfigured by subsequent addressing commands).

The data transfer, or data message, allows for the exchange of information between devices on the HP-IB. Our committee conducts business by exchanging ideas and information between the speaker and those listening to his presentation. On the HP-IB, data is transferred from the active talker to the active listener(s) at a rate determined by the slowest active listener on the bus. This restriction on the transfer rate is necessary to ensure that no data is lost by any device addressed to listen. The handshake used to transfer each data byte ensures that all data output by the talker is received by all active listeners.

**Examples of Bus Sequences.** Most data transfers through the HP-IB involve a talker and only one listener. For example, the following OUTPUT statement could be used (by the Active Controller) to send data to an HP-IB device:

```
OUTPUT 701;"Data"
```

The following sequence of commands and data is sent through the bus:

1. The talker's address is sent (here, the address of the computer; "My Talk Address"), which is also a command.
2. The unlisten command is sent.
3. The listener's address (01) is sent, which is also a command.
4. The data bytes "D", "a", "t", "a", CR, and LF are sent; all bytes are sent using the HP-IB's interlocking handshake to ensure that the listener has received each byte.

Similarly, most ENTER statements involve transferring data from a talker to only one listener. For instance, the ENTER statement:

```
ENTER 722;Voltage
```

invokes the following sequence of commands and data-transfer operations:

1. The talker's address (22) is sent, which is a command.
2. The unlisten command is sent.
3. The listener's address is sent (here, the computer's address; "My Listen Address"), also a command
4. The data is sent by device 22 to the computer using the HP-IB handshake.

**Addressing Multiple Listeners.** HP-IB allows more than one device to listen simultaneously to data sent through the bus (even though the data may be accepted at differing rates). The following examples show how the Active Controller can address multiple listeners on the bus.

```
100  ASSIGN @Listeners TO 701,702,703
110  OUTPUT @Listeners;String$
120  OUTPUT @Listeners USING Image_1;Array$(*)
```

This capability allows a single OUTPUT statement to send data to several devices simultaneously. It is however, necessary for all the devices to be on the same interface. When the preceding OUTPUT statement is executed, the unlisten command is sent first, followed by the Active Controller's talk address and then listen addresses 01, 02, and 03. Data is then sent by the controller and accepted by devices at addresses 1, 2, and 3.

If an ENTER statement that uses the same I/O path name is executed by the Active Controller, the first device is addressed as the talker (the source of data) and all the rest of the devices, including the Active Controller, are addressed as listeners. The data is then sent from the device at address 01 to the devices at addresses 02 and 03 and to the Active Controller.

```
130  ENTER @Listeners;String$
140  ENTER @Listeners USING Image_2;Array$(*)
```

**Secondary Addressing.** Many devices have operating modes which are accessed through the extended addressing capabilities defined in the bus standard. Extended addressing provides for a second address parameter in addition to the primary address. Examples of statements that use extended addressing are as follows.

```
100   ASSIGN @Device TO 72205  ! 22=primary, 05=secondary.
110   OUTPUT @Device;Message$

200   OUTPUT 72205;Message$

150   ASSIGN @Device TO 7220529 ! Additional secondary
160
170   OUTPUT @Device;Message$

120   OUTPUT 7220529;Message$
```

The range of secondary addresses is 00 through 31; up to six secondary addresses may be specified (a total of 15 digits including interface select code and primary address). Refer to the device's operating manual for programming information associated with the extended addressing capability. The HP-IB interface also has a mechanism for detecting secondary commands.

## General Bus Management

The HP-IB standard provides several mechanisms that allow managing the bus and the devices on the bus. Here is a summary of the statements that invoke these control mechanisms.

**ABORT** is used to abruptly terminate all bus activity and reset all devices to power-on states.

**CLEAR** is used to set all (or only selected) devices to a pre-defined, device-dependent state.

**LOCAL** is used to return all (or selected) devices to local (front-panel) control.

**LOCAL LOCKOUT** is used to disable all devices' front-panel controls.

**PPOLL** is used to perform a parallel poll on all devices (which are configured and capable of responding).

**PPOLL CONFIGURE** is used to setup the parallel poll response of a particular device.

**PPOLL UNCONFIGURE** is used to disable the parallel poll response of a device (or all devices on an interface).

**REMOTE** is used to put all (or selected) devices into their device-dependent, remote modes.

**SEND** is used to manage the bus by sending explicit command or data messages.

**SPOLL** is used to perform a serial poll of the specified device (which must be capable of responding).

**TRIGGER** is used to send the trigger message to a device (or selected group of devices).

These statements (and functions) are described in the following discussion. However, the actions that a device takes upon receiving each of the above commands are, in general, different for each device. Refer to a particular device's manuals to determine how it will respond.

**Remote Control of Devices.** Most HP-IB devices can be controlled either from the front panel or from the bus. If the device's front panel controls are currently functional, it is in the Local state. If it is being controlled through the HP-IB, it is in the Remote state. Pressing the front-panel "Local" key will return the device to Local (front-panel) control, unless the device is in the Local Lockout state (described in a subsequent discussion).

The Remote message is automatically sent to all devices whenever the System Controller is powered on, reset, or sends the Abort message. A device also enters the Remote state automatically whenever it is addressed. The REMOTE statement also outputs the Remote message, which causes all (or specified) devices on the bus to change from local control to remote control. The computer must be the System Controller to execute the REMOTE statement.

Here are some examples:

```
REMOTE 7

ASSIGN @Device TO 700
REMOTE @Device

REMOTE 700
```

**Locking Out Local Control.** The Local Lockout message effectively locks out the "local" switch present on most HP-IB device front panels, preventing a device's user from interfering with system operations by pressing buttons and thereby maintaining system integrity. As long as Local Lockout is in effect, no bus device can be returned to local control from its front panel.

The Local Lockout message is sent by executing the LOCAL LOCKOUT statement. This message is sent to all devices on the specified HP-IB interface, and it can only be sent by the computer when it is the Active Controller.

Here are some examples:

```
ASSIGN @Hpib TO 7
LOCAL LOCKOUT @Hpib

LOCAL LOCKOUT 7
```

The Local Lockout message is cleared when the Local message is sent by executing the LOCAL statement. However, executing the ABORT statement does not cancel the Local Lockout message.

**Enabling Local Control.** During system operation, it may be necessary for an operator to interact with one or more devices. For instance, an operator might need to work from the front panel to make special tests or to troubleshoot. And, in general, it is good systems practice to return all devices to local control upon conclusion of remote-control operation. Executing the LOCAL statement returns the specified devices to local (front-panel) control. The computer must be the Active Controller to send the LOCAL message.

Here are some examples:

```
ASSIGN @Hpib TO 7
LOCAL @Hpib

ASSIGN @Device TO 700
LOCAL @Device
```

If primary addressing is specified, the Go-to-Local message is sent only to the specified device(s). However, if only the interface select code is specified, the Local message is sent to all devices on the specified HP-IB interface and any previous Local Lockout message (which is still in effect) is automatically cleared. The computer must be the System Controller to send the Local message (by specifying only the interface select code).

**Triggering HP-IB Devices.** The TRIGGER statement sends a Trigger message to a selected device or group of devices. The purpose of the Trigger message is to initiate some device-dependent action; for example, it can be used to trigger a digital voltmeter to perform its measurement cycle. Because the response of a device to a Trigger Message is strictly device-dependent, neither the Trigger message nor the interface indicates what action is initiated by the device.

Here are some examples:

```
ASSIGN @Hpib TO 7
TRIGGER @Hpib

ASSIGN @Device TO 707
TRIGGER @Device
```

Specifying only the interface select code outputs a Trigger message to all devices currently addressed to listen on the bus. Including device addresses in the statement triggers only those devices addressed by the statement. The computer can also respond to a trigger from another controller on the bus.

**Clearing HP-IB Devices.** The CLEAR statement provides a means of "initializing" a device to its predefined, device-dependent state. When the CLEAR statement is executed, the Clear message is sent either to all devices or to the specified device(s), depending on the information contained within the device selector. If only the interface select code is specified, all devices on the specified HP-IB interface are cleared. If primary-address information is specified, the Clear message is sent only to the specified device. Only the Active Controller can send the Clear message.

Here are some examples:

```
ASSIGN @Hpib TO 7
CLEAR @Hpib

ASSIGN @Device TO 700
CLEAR @Device
```

**Aborting Bus Activity.** This statement may be used to terminate all activity on the bus and return all the HP-IB interfaces of all devices to a reset (or power-on) condition. Whether this affects other modes of the device depends on the device itself. The computer must be either the active or the system controller to perform this function. If the System Controller (which is not the current Active Controller) executes this statement, it regains active control of the bus. Only the interface select code may be specified; device selectors which contain primary-addressing information (such as 724) may not be used.

Here are some examples:

```
ASSIGN @Hpib TO 7
ABORT @Hpib

ABORT 7
```

**HP-IB Service Requests.** Most HP-IB devices, such as voltmeter, frequency counters, and spectrum analyzers, are capable of generating a "service request" when they require the Active Controller to take action. Service requests are generally made after the device has completed a task (such as making a measurement) or when an error condition exists (such as a printer being out of paper). The operation and programming manuals for each device describes the device's capability to request service and conditions under which the device will request service.

To request service, the device sends a Service Request message (SRQ) to the Active Controller. The mechanism by which the Active Controller detects these requests is the SRQ interrupt. Interrupts allow an efficient use of system resources because the system may be executing a program until interrupted by an event's occurrence. If enabled, the external event initiates a program branch to a routine which "services" the event (executes remedial action).

**Setting Up and Enabling SRQ Interrupts.** In order for an HP-IB device to be able to initiate a service routine as the Active Controller, two prerequisites must be met: the SRQ interrupt event must have a service routine defined, and the SRQ interrupt must be enabled to initiate the branch to the service routine. The following program segment shows an example of setting up and enabling an SRQ interrupt.

```
100  Hpib=7
110  ON INTR Hpib GOSUB Service_routine
120  !
130  Mask=2  ! Bit 1 enables SRQ interrupts.
140  ENABLE INTR Hpib;Mask
```

The value of the mask in the ENABLE INTR statement determines which type(s) of interrupts are to be enabled. The value of the mask is automatically written into the HP-IB interface's interrupt-enable register (CONTROL register 4) when this statement is executed. Bit 1 is set in the preceding example, enabling SRQ interrupts to initiate a program branch. Reading STATUS register 4 at this point would return a value of 2.

**Servicing SRQ Interrupts.** The SRQ is a level-sensitive interrupt; in other words, if an SRQ is present momentarily but does not remain long enough to be sensed by the computer, the interrupt will not be generated.

It is important to note that once an interrupt is sensed and logged, the interface cannot generate another interrupt until the initial interrupt is serviced. The computer disables all subsequent interrupts from an interface until a pending interrupt is serviced. For this reason, it was necessary to re-enable the interrupt to allow for subsequent branching.

**Polling HP-IB Devices.** The Parallel Poll is the fastest means of gathering device status when several devices are connected to the bus. Each device (with this capability) can be programmed to respond with one bit of status when Parallel Polled, making it possible to obtain the status of several devices in one operation. If a device responds affirmatively ("I need service") to a Parallel Poll, more information as to its specific status can be obtained by conducting a Serial Poll of the device.

**Configuring Parallel Poll Responses.** Certain devices can be remotely programmed by the Active Controller to respond to a Parallel Poll. A device which is currently configured for a Parallel Poll responds to the poll by placing its current status on one of the bus data lines. The logic sense of the response and the data-bit number can be programmed by the PPOLL CONFIGURE statement. No multiple listeners can be specified in the statement; if more than one device is to respond on a single bit, each device must be configured with a separate PPOLL CONFIGURE statement.

**Conducting a Parallel Poll.** The PPOLL function returns a single byte containing up to 8 status bit messages of all devices on the bus capable of responding to the poll. Each bit returned by the function corresponds to the status bit of the device(s) configured to respond to the parallel poll. (Recall that one or more devices can respond on a single line.) The PPOLL function can only be executed when the computer is the Active Controller. The following statement conducts a parallel poll of the interface at select code 7 (normally, the built-in HP-IB).

```
Response=PPOLL(7)
```

**Disabling Parallel Poll Responses.** The PPOLL UNCONFIGURE statement gives the computer (as Active Controller) the capability of disabling the Parallel Poll responses of one or more devices on the bus.

For example, the following statement disables device 5 only:

```
PPOLL UNCONFIGURE 705
```

On the other hand, the following statement disables all devices on interface select code 8 from responding to a Parallel Poll:

```
PPOLL UNCONFIGURE 8
```

If no primary addressing is specified, all bus devices are disabled from responding to a Parallel Poll. If primary addressing is specified, only the specified devices (which have the Parallel Poll Configure capability) are disabled.

**Conducting a Serial Poll.** A sequential poll of individual devices on the bus is known as a Serial Poll. One entire byte of status is returned by the specified device in response to a Serial Poll. This byte is called the Status Byte message and, depending on the device, may indicate an overload, a request for service, or a printer being out of paper. The particular response of each device depends on the device.

The SPOLL function performs a Serial Poll of the specified device; the computer must be the Active Controller.

Here are some examples:

```
ASSIGN @Device TO 700
Status_byte=SPOLL(@Device)
```

```
Spoll_24=SPOLL(724)
```

Just as the Parallel Poll is not defined for individual devices, the Serial Poll is meaningless for an interface; therefore, primary addressing must be used with the SPOLL function.

## The Computer As a Non-Active Controller

The section called "General Structure of the HP-IB" described how communications take place through HP-IB Interfaces. The functions of the System Controller and Active Controller were likened to a "committee chairman" and "acting chairman," respectively, and the functions of each were described. This section describes how the Active Controller may "pass control" to another controller and assume the role of a non-Active Controller. This action is analogous to designating another committee member to take the responsibility of acting chairman and then becoming a committee member who listens to the acting chairman and speaks when given the floor.

**Determining Controller Status and Address.** It is often necessary to determine if an interface is the System Controller and to determine whether or not it is the current Active Controller. It is also often necessary to determine or change the interface's primary address.

Let's look at an example. Executing the following statement reads STATUS register 3 (of the internal HP-IB) and places the current value into the variable Stat_and_addr. Remember that if the statement is executed from the keyboard, the variable Stat_and_ addr must be defined in the current context.

```
STATUS 7,3;Stat_and_addr
```

## Status Register 3: Controller Status and Address

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| System Controller | Active Controller | 0 | Primary Address of Interface | | | | |
| Value = 128 | Value = 64 | Value = 32 | Value = 16 | Value = 8 | Value = 4 | Value = 2 | Value = 1 |

If bit 7 is set ("1"), it signifies that the interface is the System Controller; if clear ("0"), it is not the System Controller. Only one controller on each HP-IB interface should be configured as the System Controller.

If bit 6 is set ("1"), it signifies that the interface is currently the Active Controller; if it is clear ("0"), another controller is currently the Active Controller.

Bits 4 through 0 represent the current value of the interface's primary address, which is in the range 0 through 30. The power-on default value for the internal HP-IB is 21 (if it is the System Controller) and 20 (if not the System Controller).

Let's look at an example. Calculate the primary address of the interface from the value previously read from STATUS register 3.

```
Intf_addr=Stat_and_addr MOD 32
```

This numerical value corresponds to the talk (or listen) address sent by the computer when an OUTPUT (or ENTER) statement containing primary-address information is executed.

**Changing the Controller's Address.** It is possible to use the CONTROL statement to change an HP-IB interface's address. For example:

```
CONTROL 7,3;Intf_addr
```

The value of Intf_addr is used to set the address of the HP-IB interface (in this case, the internal HP-IB). The valid range of addresses is 0 through 30; address 31 is not used. Thus, if a value greater than 30 is specified, the value MOD 32 is used (for example: 32 MOD 32 equals 0, 33 MOD 32 equals 1, 62 MOD 32 equals 30, and so forth).

**Passing Control.** The current Active Controller can pass this capability to another computer by sending the Take Control message (TCT). The Active Controller must first address the prospective new Active Controller to talk, after which the TCT message is sent. If the other controller accepts the message, it then assumes the role of Active Controller; this computer then assumes the role of a non-Active Controller.

Passing control can be accomplished in one of two ways: it can be handled by the system, or it can be handled by the program. The PASS CONTROL statement can be used. For example, the following statements first define the HP-IB Interface's select code and new Active Controller's primary address and then pass control to that controller.

```
100 Hp_ib=7
110 New_ac_addr=20
120 PASS CONTROL 100*Hp_ib+New_ac_addr
```

The following statements perform the same functions.

```
100 Hp_ib=7
110 New_ac_addr=20
120 SEND Hp_ib;TALK New_ac_addr CMD 9
```

Once the new Active Controller has accepted the TCT command, the controller passing control assumes the role of a non-Active Controller (or "HP-IB device") on the specified HP-IB Interface. The next section describes the responsibilities of the computer while it is a non-Active Controller.

**Interrupts While Non-Active Controller.** When the computer is not an Active Controller, it must be able to detect and respond to many types of bus messages and events.

The computer (as a non-Active Controller) needs to keep track of the following information.

- It must keep track of itself being addressed as a listener so that it can enter data from the current active talker.
- It must keep track of itself being addressed as a talker so that it can transmit the information desired by the active controller.
- It must keep track of being sent a Clear, Trigger, Local, or Local Lockout message so that it can take appropriate action.
- It must keep track of control being passed from another controller.

One way to do this is to continually monitor the HP-IB interface by executing the STATUS statement and then taking action when the values returned match the values desired. This is obviously a great waste of computer time if the computer could be performing other tasks. Instead, the interface hardware can be enabled to monitor bus activity and then generate interrupts when certain events take place.

The computer has the ability to keep track of the occurrences of all of the preceding events. In fact, it can monitor up to 16 different interrupt conditions. STATUS registers 4, 5 and 6 provide access to the interface state and interrupt information necessary to design very powerful systems with a great degree of flexibility.

Each individual bit of STATUS register 4 corresponds to the same bit of STATUS register 5. Register 4 provides information as to which condition caused an interrupt, while register 5 keeps track of which interrupt conditions are currently enabled. To enable a combination of conditions, add the decimal values for each bit that you want set in the interrupt-enable register. This total is then used as the mask parameter in an ENABLE INTR statement.

## Status Register 5: Interrupt Enable Mask

This is a 16-bit register:

| Bit 15 | Bit 14 | Bit 13 | Bit 12 | Bit 11 | Bit 10 | Bit 9 | Bit 8 |
|---|---|---|---|---|---|---|---|
| Active Controller | Parallel Poll Configur-ation Change | My Talk Address Received | My Listen Address Received | EOI Received | SPAS | Remote/ Local Change | Talker/ Listener Address Change |
| Value = 32,768 | Value = 16,384 | Value = 8,192 | Value = 4,096 | Value = 2,048 | Value = 1,024 | Value = 512 | Value = 256 |

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| Trigger Received | Handshake Error | Unrecog-nized Universal Command | Secondary Command While Addressed | Clear Received | Unrecog-nized Addressed Command | SRQ Received | IFC Received |
| Value = 128 | Value = 64 | Value = 32 | Value = 16 | Value = 8 | Value = 4 | Value = 2 | Value = 1 |

Bit 15 enables an interrupt upon becoming the Active Controller. The computer then has the ability to manage bus activities.

Bit 14 enables an interrupt upon detecting a change in Parallel Poll Configuration. (This condition requires accepting data from the bus and then explicitly releasing the bus.)

Bit 13 enables an interrupt upon being addressed as an active talker by the Active Controller.

Bit 12 enables an interrupt upon being addressed as an active listener by the Active Controller.

Bit 11 enables an interrupt when an EOI is received during an ENTER operation (the EOI signal line is also described in the section "HP-IB Control Lines").

Bit 10 enables an interrupt when the Active Controller performs a Serial Poll on the computer (in response to its service request).

Bit 9 enables an interrupt upon receiving either the Remote or the Local message from the active controller, if addressed to listen. The action taken by the computer is, of course, dependent on the user-programmed service routine.

Bit 8 enables an interrupt upon a change in talk or listen address. An interrupt will be generated if the computer is addressed to listen or talk or "idled" by an Unlisten or Untalk command.

Bit 7 enables an interrupt upon receiving a Trigger message, if the computer is currently addressed to listen. This interrupt can be used in situations where the computer may be "armed and waiting" to initiate action; the active controller sends the Trigger message to the computer to cause it to begin its task.

Bit 6 enables an interrupt if a bus error occurs during an OUTPUT statement. Particularly, the error occurs if none of the devices on the bus respond to the HP-IB's interlocking handshake (see "HP-IB Control Lines"). The error typically indicates that either a device is not connected or that its power is off.

Bit 5 enables an interrupt upon receiving an unrecognized Universal Command. This interrupt condition provides the computer with the capability of responding to new definitions that may be adopted by the IEEE standards committee. (This condition requires accepting data from the bus and then explicitly releasing the bus.)

Bit 4 enables an interrupt upon receiving a Secondary Command (extended addressing) after the interface receives either its primary talk address or primary listen address. Again, this interrupt provides the computer with a way to detect and respond to special messages from another controller. (This condition requires accepting data from the bus and then explicitly releasing the bus.)

Bit 3 enables an interrupt on receiving a Clear message. Reception of either a Device Clear message (addressed to the computer) will cause this type of interrupt. The computer is free to take any "device-dependent" action, such as setting up all default values again or even restarting the program, if that is defined by the programmer to be the "cleared" state of the machine.

Bit 2 enables an interrupt upon receiving an unrecognized Addressed Command, if the computer is currently addressed to listen. This interrupt is used to intercept and respond to bus commands which are not defined by the standard. (This condition requires accepting data from the bus and then explicitly releasing the bus.)

Bit 1 enables an interrupt upon detecting a Service Request.

Bit 0 enables an interrupt upon detecting an Interface Clear (IFC). The interrupt is generated only when the computer is not the System Controller, as only a System Controller is allowed to set the Interface Clear signal line. The service routine typically is used to recover from the abrupt termination of an I/O operation caused by another controller sending the IFC message.

Note that most of the conditions are state-sensitive or event-sensitive; the exception is the SRQ event, which is level-sensitive. State or event-sensitive events can never go unnoticed by the computer as can service requests; the event's occurrence is "remembered" by the computer until serviced.

For instance, if the computer is enabled to generate an interrupt on becoming addressed as a talker, it would interrupt the first time it received its own talk address. After having responded to the service request (most likely with some sort of OUTPUT operation), it would not generate another interrupt, even if it was still left assigned as a talker by the Active Controller. Thus, it would not generate another interrupt until the event occurred a second time.

An oversimplified example of a service routine that is to respond to multiple conditions might be as follows. This example can be found in file SERVER1 on your Manual Examples disk.

Register 4, the interrupt status register, is a "read-destructive" register; reading the register with a STATUS statement returns its contents and then clears the register (to a value of 0). If the service routine's action depends on the contents of STATUS register 4, the variable in which it is stored must not be used for any other purposes before all of the information that it contains has been used by the service routine.

The computer is automatically addressed to talk (by the Active Controller) whenever it is Serially Polled. If interrupts are concurrently enabled for My Address Change and/or Talker Active, the ON INTR branch will be initiated due to the reception of the computer's talk address. However, since the Serial Poll is automatically finished with the Untalk Command, the computer may no longer be addressed to talk by the time the interrupt service routine begins execution. See "Responding to Serial Polls" for further details.

**Requesting Service.** When the computer is a non-Active Controller, it has the capability of sending an SRQ to the current Active Controller. The following statement is an example of requesting service from the Active Controller of the HP-IB Interface on select code 7.

```
CONTROL 7,1;64
```

The REQUEST statement can be used to perform the same function.

```
REQUEST 7;64
```

Both of the preceding examples place a logic True on the SRQ line. (Note that the line may already be set True by another device.) Other bits may be set in the Status Byte message, indicating that other device-dependent conditions exist.

The SRQ line is held True until the Active Controller executes a Serial Poll or this computer executes a REQUEST with bit 6 equal to 0. (Note also that the line may still be held True by another device.)

**Responding to Parallel Polls.** Before performing a Parallel Poll of bus devices, the Active Controller configures selected device(s) to respond on one of the eight data lines. Each device is directed to respond on a particular data line with a logic True or False; the logic sense of the response informs the Active Controller either "I do need service" or "I don't need service." The logic sense of the response is also specified by the Active Controller. This response to the Parallel Poll is known as the Status Bit message.

**Responding to Serial Polls.** As a non-Active Controller, the response to Serial Polls is automatically handled by the system. The desired Serial Poll Response Byte is sent to HP-IB CONTROL Register 1. If bit 6 is set (bit 6 has a value of 64), an SRQ is indicated from this controller. All other bits can be considered to be "device-dependent," and can thus be set according to the program's needs.

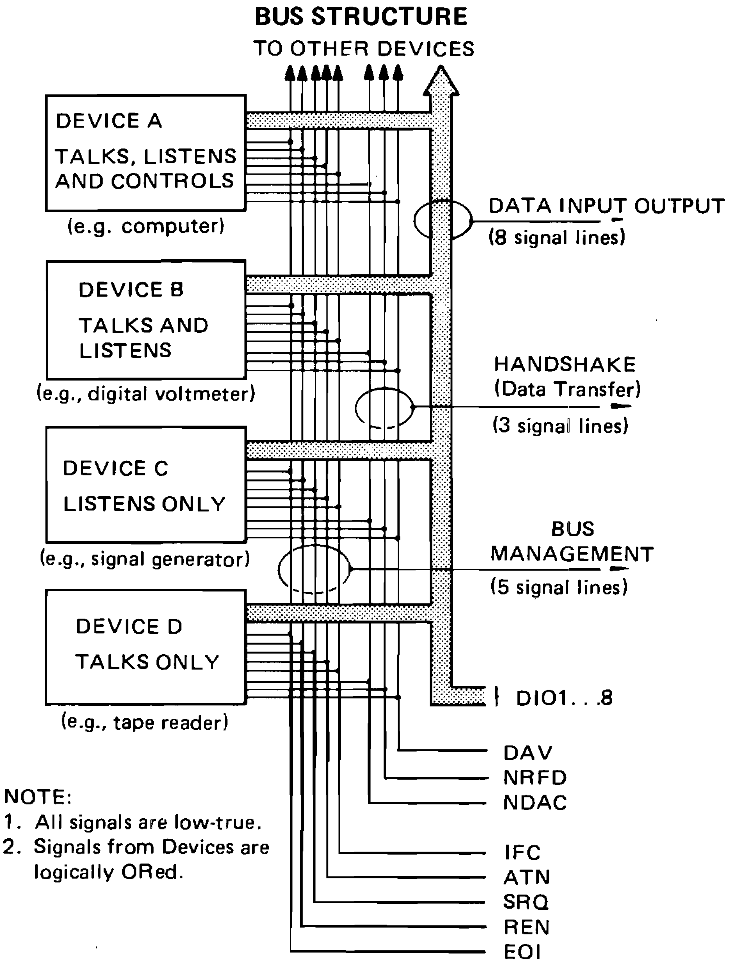
The following statement sets up a response with SRQ and bits 1 and 0 set to "1".

```
CONTROL 7,1;64+2+1
```

When the Active Controller performs a Serial Poll on this non-Active Controller, the specified byte is automatically sent to the Active Controller by the system.

## HP-IB Control Lines

The HP-IB interface provides eight data lines and eight control lines as shown in the following figure.



**HP-IB Control Lines**

The preceding figure shows the names given to the eight control lines that make up the HP-IB. These lines are described in the following paragraphs.

**Handshake Lines.** Three of the eight HP-IB control lines are designated as "handshake" lines and are used to control the timing of data byte exchanges so that the talker does not get ahead of the listener(s). The three handshake lines are as follows:

| | |
|---|---|
| DAV | Data Valid |
| NRFD | Not Ready for Data |
| NDAC | Not Data Accepted |

The *HP-IB interlocking handshake* uses the lines as follows. All devices currently designated as active listeners would indicate when they are ready for data by using the NRFD line. A device not ready would pull this line low (true) to signal that it is not ready for data, while any device that is ready would let the line float high. Since an active low overrides a passive high, this line will stay low until all active listeners are ready for data.

When the talker senses that all devices are ready, it places the next data byte on the data lines and then pulls DAV low (true). This tells the listeners that the information on the data lines is valid and that they may read it. Each listener then accepts the data and lets the NDAC line float high (false). As with NRFD, only when all listeners have let NDAC go high will the talker sense that all listeners have read the data. It can then float DAV (let it go high) and start the entire sequence over again for the next byte of data.

**The Attention Line (ATN).** Command messages are encoded on the data lines as 7-bit ASCII characters, and are distinguished from normal characters by the logic state of the attention line (ATN). That is, when ATN is *false*, the states of the data lines are interpreted as *data*. When ATN is *true*, the data lines are interpreted as *commands*. The set of 128 ASCII characters that can be placed on the data lines during this ATN-true mode are divided into four classes by the states of data lines DIO6 and DIO7. Only the Active Controller can set ATN true.

**The Interface Clear Line (IFC).** Only the System Controller can set the IFC line true. By asserting IFC, all bus activity is unconditionally terminated, the System Controller regains the capability of Active Controller (if it has been passed by another device), and any current talker and listeners become unaddressed. Normally, this line is only used to terminate all current operations, or to allow the System Controller to regain control of the bus. It overrides any other activity that is currently taking place on the bus.

**The Remote Enable Line (REN).** This line is used to allow instruments on the bus to be programmed remotely by the Active Controller. Any device that is addressed to listen while REN is true is placed in the Remote mode of operation.

**The End or Identify Line (EOI).** Normally, data messages sent over the HP-IB are sent using the standard ASCII code and are terminated by the ASCII line-feed character, CHR$(10). However, certain devices may wish to send blocks of information that contain data bytes which have the bit pattern of the line-feed character, but which are actually part of the data message. Thus, no bit pattern can be designated as a terminating character since it could occur anywhere in the data stream. For this reason the EOI line is used to mark the end of the data message.

The EOI line is used as an END indication (ATN false) during ENTER statements and as the Identify message (ATN true) during an identify sequence (the response to a parallel poll). During data messages, the EOI line is set true by the talker to signal that the current data byte is the last one of the data transmission. Generally, when a listener detects that the EOI line is true, it assumes that the data message is concluded. However, EOI may either be used or ignored by the computer when entering data with an ENTER statement that uses an image.

**The Service Request Line (SRQ).** The Active Controller is always in charge of the order of events that occur on the HP-IB. If a device on the bus needs the Active Controller's help, it can set the Service Request line true. This line sends a request, not a demand, and it is up to the Active Controller to choose when and how it will service that device. However, the device will continue to assert SRQ until it has been "satisfied." Exactly what will satisfy a service request depends on the requesting device (refer to the operating manual for the device).

---

**Note**

You can determine the current status of all of the bus hardware lines by reading Status Register 7 with the STATUS statement. Refer to the "Interface Registers" appendix in volume 2 of the BASIC *Language Reference* manual.

---

## References

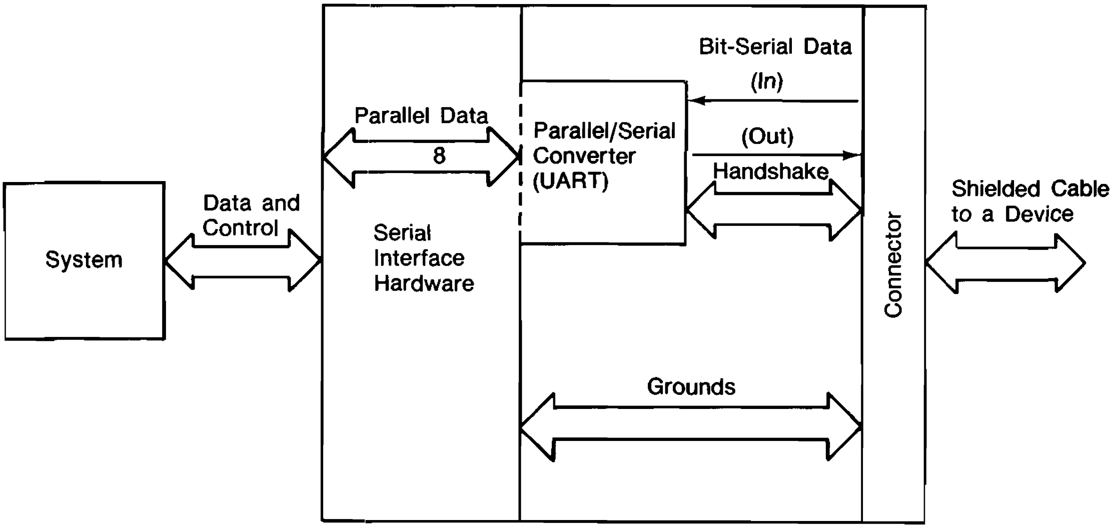
For further information, about the HP-IB (IEEE-488) interface you may want to refer to the following sources:

- *Tutorial Description of the Hewlett-Packard Interface Bus*, Hewlett-Packard Company, 1987 (HP part number 5952-0156).
- IEEE Standard 488.1-1987, "Digital Interface for Programmable Instrumentation," The IEEE, Inc., 345 East 47th St., New York, NY, June 1987.

## The RS-232 Serial Interface

The Serial Interface is an RS-232-C compatible interface used for simple asynchronous I/O applications such as driving line printers, terminals, or other peripherals. It uses a UART (Universal Asynchronous Receiver and Transmitter) integrated circuit to generate the required async signals. The computer must provide most control functions because the card does not have its own processor capability. Consequently, there is more interaction between the card and computer than when you use a more intelligent interface, except for relatively simple applications.

The following block diagram shows the RS-232 serial interface:



The RS-232-C interface standard establishes electrical and mechanical interface requirements, but does not define the exact function of all the signals that are used by various manufacturers of data communications equipment and serial I/O devices. Consequently, when you plug your serial interface into an RS-232 connector, there is no guarantee the devices can communicate unless you have configured optional parameters to match the requirements of the device you are connecting to.

**Note** RS-232-C is a data communication standard established and published by the Electronic Industries Association (EIA). Copies of the standard are available from the association at 2001 Eye Street N.W., Washington D.C. 20006. Its equivalent for European applications is CCITT V.24.

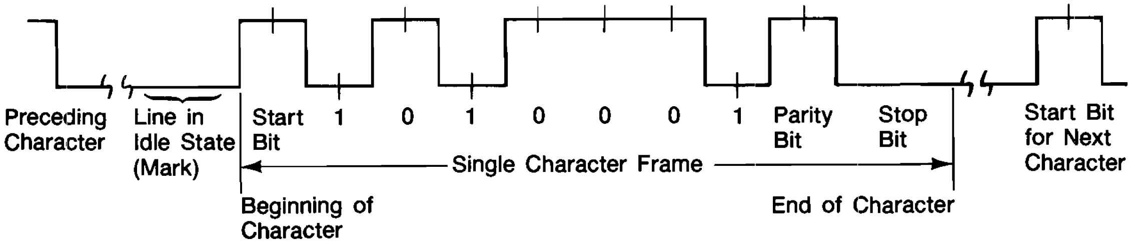
## Asynchronous Data Communication

The terms Asynchronous (Async for short) data communication and Serial I/O refer to a technique of transferring information between two communicating devices by means of bit-serial data transmission. This means that data is sent, one bit at a time, and that characters are not synchronized with preceding or subsequent data characters; that is, each character is sent as a complete entity without relationship to other events, before or after. Characters may be sent in close succession, or they may be sent sporadically as data becomes available. Start and stop bits are used to identify the beginning and end of each character, with the character data placed between them.

**Character Format.** Each character frame consists of the following elements:

- Start Bit: The start bit signals the receiver that a new character is being sent. Since the receiver knows how many bits per second are being transmitted (specified by the baud rate), it can determine the expected arrival time for all subsequent bits in that character frame. All other bits in a given frame are synchronized to the start bit.
- 5 – 8 Character Data Bits: The next bits are the binary code of the character being transmitted, consisting of 5, 6, 7, or 8 bits; depending on the application. The parity bit is not included in the character data bits.
- Parity Bit: The parity bit is optional, included only when parity is enabled.
- Stop Bit(s): One or more stop bits identify the end of each character. The serial interface has no provision for inserting the time gaps between characters.

Here is a simple diagram showing the structure of an asynchronous character and its relationship to other characters in the data stream:



**Parity.** The parity bit is used to detect errors as incoming characters are received. If the parity bit does not match the expected sense, the character is assumed to be incorrectly received. The action taken when an error is detected depends upon how the interface and your computer program are configured.

Parity sense is determined by system requirements.[*] The parity bit may be included or omitted from each character by enabling or disabling the parity function. If the parity bit is enabled, four options are available. Parity is checked by the receiver for all parity options including ONE and ZERO. Parity options include:

- NONE — Parity function is DISABLED, and the parity bit is omitted from each character frame.
- ODD — Parity bit is SET if there is an EVEN number of ones in the data character. The receiver performs parity checks on incoming character.
- EVEN — Parity bit is SET if there is an ODD number of ones in the data character. The receiver performs parity checks on incoming characters.
- ONE — Parity bit is set for all characters. Parity is checked by the receiver on all incoming characters.
- ZERO — Parity bit is cleared, but present for all characters. Parity is checked by the receiver on all characters.

**Error Detection.** Two types of incoming data errors can be detected by serial receivers:

- Parity errors are signaled when the parity bit does not match the number of ones, including the parity bit, even or odd as defined by interface configuration. When parity is disabled, no parity check is made.
- Framing errors are signaled when start and stop bits are not properly received during the expected time frame. They can be caused by a missing start bit, noise errors near the end of the character, or by improperly specified character length at the transmitter or receiver.

Two additional error types are detected by the receiver section of the serial interface:

- Overrun errors result when the desktop computer does not consume characters as fast as they arrive. The card provides only one character of buffer space, so the current character must be consumed by an ENTER before the next character arrives. Otherwise, the character is lost when the next character replaces it, and an error is sent to BASIC.

[*] Parity sense is determined by counting the number of ones in the character *including* the parity bit. Consequently, the parity sense is reversed from the number of ones in a character without the parity bit.

- Received BREAKs are detected as a special type of framing error. They generate the same type of BASIC error as framing errors.

## Data Transfers Between Computer and Peripheral

Four statements are used to transfer information between your computer and the interface card:

- The CONTROL statement is used to control interface operation and defines such parameters as baud rate, character format, or parity.
- The OUTPUT statement sends data to the interface which, in turn, sends the information to the peripheral device.
- The ENTER statement inputs data from the interface card after the interface has received it from the peripheral device.
- The STATUS statement is used to monitor the interface and obtain information about interface operation such as buffer status, detected errors, and interrupt enable status.

Since the interface has no on-board processor, ENTER and OUTPUT statements cause the computer to wait until the ENTER or OUTPUT operation is complete before continuing to the next line. For OUTPUT statements, this means that the computer waits until the last bit of the last character has been sent over the serial line before continuing with the next program statement.

## Overview of Serial Interface Programming

Serial interface programming techniques are similar to most general I/O applications. The interface card is initialized by use of CONTROL statements; STATUS statements evaluate its readiness for use. Data is transferred between your computer and a peripheral device by OUTPUT and ENTER statements. In most cases, you can use default configuration switches on the interface card to eliminate or significantly reduce the need for using CONTROL statements to initialize the card.

Due to the asynchronous nature of serial I/O operations, you should take special care to ensure that data is not lost by sending to a device before the device is ready to receive. Modem line handshaking can be used to help solve this problem. The interface registers are described in an appendix of the BASIC *Language Reference* manual.

## Initializing the Interconnection

Before you establish a connection, you must determine what certain interface parameters are.

**Determining Operating Parameters.** Before you can successfully transfer information to a device, you must match the operating characteristics of the interface to the corresponding characteristics of the peripheral device. This includes matching signal lines and their functions as well as matching the character format for both devices.

**Hardware Parameters.** To determine hardware operating parameters, you need to know the answer for each of the following questions about the peripheral device:

- Which of the following signal and control lines are actively used during communication with the peripheral?

  __Data Set Ready (DSR)
  __Data Carrier Detect (DCD or CD)
  __Clear to Send (CTS)
  __Ring Indicator (RI)
- What baud rate (line speed) is expected by the peripheral?

**Character Format Parameters.** To define the character format, you must know the requirements of the peripheral device for the following parameters:

- Character Length: How many data bits are used for each character, excluding start, stop, and parity bits?
- Parity Enable: Is Parity enabled (included) or disabled (absent) for each character?
- Parity Sense: Is the parity bit, if enabled, ODD, EVEN, always ONE, or always ZERO?
- Stop Bits: How many stop bits are included with each character: 1,1.5, or 2?

**Using Interface Defaults to Simplify Programming.** The serial interface may be preconfigured with default parameters.

## Using Program Control to Override Defaults

You can override some of the interface default configuration options by use of CONTROL statements. This not only enables you to guarantee certain parameters, but also provides a means for changing selected parameters in the course of a running program.

**Interface Reset.** Whenever an interface is connected to a modem that may still be connected to a telecommunications link from a previous session, it is good programming practice to reset the interface to force the modem to disconnect, unless the status of the link and remote connection are known. When the interface is connected to a line printer or similar peripheral, resetting the interface is usually unnecessary unless an error condition requires it.

When the interface is reset by use of a CONTROL statement to Control Register 0 with a non-zero value, the interface is restored to its default configuration, except that the current character format is not altered, whether or not it is the same as the current default configuration. If you are not sure of the present settings, or if your application requires changing the configuration during program operation, you can use CONTROL statements to configure the interface. An example of when this may be necessary is when several peripherals share a single interface through a manually operated RS-232 switch such as those used to connect multiple terminals to a single computer port, or a single terminal to multiple computers.

**Selecting the Baud Rate.** In order to successfully transfer information between the interface card and a peripheral, the interface and peripheral must be set to the same baud rate. A CONTROL statement to register 3 can be used to set the interface baud rate. To verify the current baud rate setting, use a STATUS statement addressed to register 3. All rates are in baud (bits/second).

**Setting Character Format and Parity.** Control Register 4 overrides the default configuration that controls parity and character format. To determine the value sent to the register, add the appropriate values selected from the following table:

| Parity Sense | Parity Enable | Stop Bits | Character Length |
|---|---|---|---|
| 0 ODD parity | 0 Disabled | 0 1 stop bit | 0 5 bits/char |
| 16 EVEN parity | 8 Enabled | 4 1.5 stop bits if 5 bits/char, or 2 stop bits if 6, 7, or 8 bits/char | 1 6 bits/char |
| 32 Always ONE | | | 2 7 bits/char |
| 48 Always ZERO | | | 3 8 bits/char |

For example, to configure a character format of 8 bits per character, two stop bits, and EVEN parity, use the following CONTROL statement:

```
1200 CONTROL Sc,4;3+4+8+16
```

or

```
1200 CONTROL Sc,4;31
```

To configure a 5-bit character length with 1 stop bit and no parity bit, use the following:

```
1200 CONTROL Sc,4;0
```

## Data Transfers

The serial interface card is designed for relatively simple serial I/O operations. It is not intended for sophisticated applications that use ON INTR statements extensively to service the interface. Limited ON INTR capabilities are provided by the serial interface for error trapping and other simple tasks.

**Program Flow.** When the interface is properly configured, either by use of default switches or CONTROL statements, you are ready to begin data transfers. OUTPUT statements are used to send information to the peripheral; ENTER statements to input information from the external device. Any valid OUTPUT or ENTER statement and variable(s) list may be used, but you must be sure that the data format is compatible with the peripheral device. For example, non-ASCII data sent to an ASCII line printer results in unpredictable behavior.

Various other I/O statements can be used in addition to OUTPUT and ENTER, depending on the situation. For example, the LIST statement can be used to list programs to an RS-232 line printer PROVIDED the interface is properly configured before the operation begins.

**Data Output.** To send data to a peripheral, use OUTPUT, OUTPUT USING, or any other similar or equivalent construct. Suppression of end-of-line delimiters and other formatting capabilities are identical to normal operation in general I/O applications. The OUTPUT statement hangs the computer until the last bit of the last character in the statement variable list is transmitted by the interface. When the output operation is complete the computer then continues to the next line in the program.

**Data Entry.** To input data from a peripheral, use ENTER, ENTER USING, or an equivalent statement. Inclusion or elimination of end-of-line delimiters and other information is determined by the formatting specified in the ENTER statement. The ENTER statement hangs the computer until the input variables list is satisfied. To minimize the risk of waiting for another variable that isn't coming, you may prefer to specify only one variable for each ENTER statement, and analyze the result before starting the next input operation.

Be sure that the peripheral is not transmitting data to the interface while no ENTER is in progress. Otherwise, data may be lost because the card provides buffering for only one character. Also, interrupts from other I/O devices, or operator inputs to the computer keyboard can cause delays in computer service to the interface that result in buffer overrun at higher baud rates.

**Modem Line Handshaking.** Modem line handshaking, when used, is performed automatically by the computer as part of the OUTPUT or ENTER operation. After a given OUTPUT or ENTER operation is complete, the program continues execution on the next line.

Control Register 5 can be used to force selected modem control lines to their active state(s). The Data Rate Select and Secondary Request-to-Send lines are set or cleared by bits 3 and 2 respectively. Request-to-send and Data Terminal Ready are held in their active states when bits 1 and 0 are true, respectively. If bits 1 and/or 0 are false, the corresponding modem line is toggled during OUTPUT or ENTER as explained previously.

**Incoming Data Error Detection and Handling.** The serial interface card can generate several errors that are caused when certain conditions are encountered while receiving data from the peripheral device. The UART detects a given error condition and sets the corresponding bit in Status Register 10. The card then generates a pending error to BASIC.

**Trapping Serial Interface Errors.** Pending BASIC errors can be trapped by using an ON ERROR statement in conjunction with an error trapping service routine to evaluate the error condition.

## The GPIO Interface

The GPIO Interface is a very flexible parallel interface that allows you to communicate with a variety of devices. The interface sends and receives up to 16 bits of data with a choice of several handshake methods. External interrupt and user-definable signal lines are provided for additional flexibility. The interface is known as the General-Purpose Input/Output (GPIO) Interface for these reasons. This section describes how to use the interface's features from BASIC Programs.

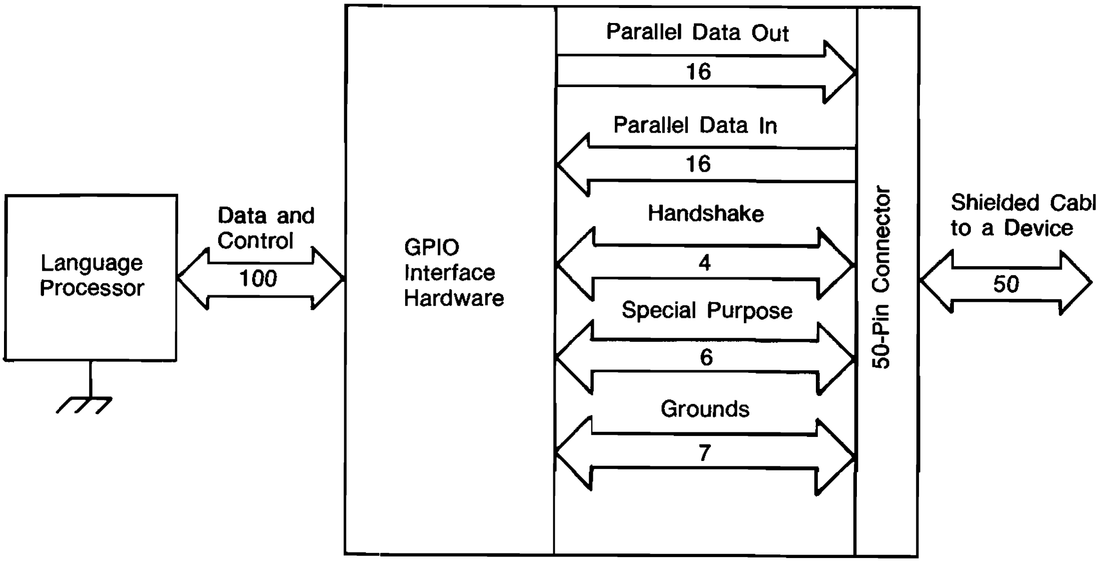
Because of the flexibility of the GPIO interface, the programmer usually needs to know quite a bit about the interface hardware. Refer to the manual that came with your GPIO interface for information about configuring the interface and connecting it to peripheral devices.

*Some of the statements and programming techniques covered in this section require that the TRANS binary be installed.*

### Interface Description

The main function of any interface is obviously to transfer data between the computer and a peripheral device. This section briefly describes the interface lines and how they function.

The GPIO Interface provides 32 lines for data input and output: 16 for input (DI0-DI15), and 16 for output (DO0-DO15). The interface is shown in the following block diagram.



Three lines are dedicated to handshaking the data from source to destination device. The Peripheral Control line (PCTL) is controlled by the interface and is used to initiate data transfers. The Peripheral Flag line (PFLG) is controlled by the peripheral device and is used to signal the peripheral's readiness to continue the transfer process. The Input/Output line (I/O) is used to indicate direction of data flow.

One line is used to signal External Interrupt Requests to the computer (EIR). The interface must be enabled to initiate interrupt branches for the interface to detect this request. The state of the line can also be read by the program.

Four general-purpose lines are available for any purpose you desire; two are controlled by the computer and sensed by the peripheral (CTL0 and CTL1), and two are controlled by the peripheral device and sensed by the computer (STI0 and STI1).

Both Logic Ground and Safety Ground are provided by the interface. Logic Ground provides the reference point for signals, and Safety Ground provides earth ground for cable shields.

## Interface Configuration

This section presents a brief summary of selecting the interface's configuration-switch settings. It is intended to be used as a checklist and to begin to acquaint you with programming the interface. Refer to the installation manual for the exact location and setting of each switch.

A sample program (found in file "GPIOCHEC.K" on your Manual Examples disk) checks a few of these switch settings on a GPIO Interface installed in the computer and displays the settings. However, many of the settings cannot be determined from BASIC programs. If any of the displayed settings are different than desired, or if any settings are not already known, refer to the installation manual for switch locations and settings.

**Interface Select Code.** In BASIC, allowable interface select codes range from 8 through 31; codes 1 through 7 are already used for built-in interfaces. The GPIO interface has a factory default setting of 12. You can change this select code by changing switch settings on the interface. (Refer to your interface owner's manual.)

**Hardware Interrupt Priority.** Two switches are provided on the interface to allow selection of hardware interrupt priority. The switches allow hardware priority level 3 through 6 to be selected. Hardware priority determines the order in which simultaneously occurring interrupt events are logged, while software priority determines the order in which interrupt events are serviced by the BASIC program.

**Data Logic Sense.** The data lines of the interface are normally low-true; in other words, when the voltage of a data line is low, the corresponding data bit is interpreted to be a "1". This logic sense may be changed to high-true with the Option Select Switch. Setting the switch labeled "DIN" to the "0" position selects high-true logic sense of Data In lines. Conversely, setting the switch labeled "DOUT" to the "1" position inverts the logic sense of the Data Out lines. The default setting is "1" for both.

**Data Handshake Methods.** As a brief review, a data handshake is a method of synchronizing the transfer of data from the sending to the receiving device. In order to use any hand shake method, the computer and peripheral device must be in agreement as to how and when several events will occur. With the GPIO Interface, the following events must take place to synchronize data transfers; the first two are optional.

- The computer may optionally be directed to perform a one-time "OK check" of the peripheral before beginning to transfer any data.
- The computer may also optionally check the peripheral to determine whether or not the peripheral is "ready" to transfer data.
- The computer must indicate the direction of transfer and then initiate the transfer.
- During OUTPUT operations, the peripheral must read the data sent from the computer while valid; similarly, the computer must clock the peripheral's data into the interface's Data In registers while valid during ENTER operations.
- The peripheral must acknowledge that it has received the data.

**The GPIO handshakes data with three signal lines.** The Input/Output line, I/O, is driven by the computer and is used to signal the direction of data transfer. The Peripheral Control line, PCTL, is also driven by the computer and is used to initiate all data transfers. The Peripheral Flag line, PFLG, is driven by the peripheral and is used to acknowledge the computer's requests to transfer data.

**Handshake Logic Sense.** Logic senses of the PCTL and PFLG lines are selected with switches of the same name. The logic sense of the I/O line is High for ENTER operations and Low for OUTPUT operations; this logic sense cannot be changed. The available choices of handshake logic sense and handshake modes allow nearly all types of peripheral handshakes to be accommodated by the GPIO Interface.

**Handshake Modes.** There are two general handshake modes in which the PCTL and PFLG lines may be used to synchronize data transfers: Full-Mode and Pulse-Mode Handshakes. If the peripheral uses pulses to handshake data transfers and meets certain hardware timing requirements, the Pulse-Mode Handshake may be used. The Full-Mode Handshake should be used if the peripheral does not meet the Pulse-Mode timing requirements.

The handshake mode is selected by the position of the "HSHK" switch on the interface, as described in the installation manual. Both modes are more fully described in subsequent sections.

**Data-In Clock Source.** Ensuring that the data are valid when read by the receiving device is slightly different for OUTPUT and ENTER operations. During OUTPUTs, the interface generally holds data valid while PCTL is in the Set state, so the peripheral must read the data during this period. During ENTERs, the data must be held valid by the peripheral until the peripheral signals that the data are valid (which clocks the data into interface Data In registers) or until the data is read by the computer. The point at which the data are valid is signaled by a transition of PFLG. The PFLG transition that is used to signal valid data is selected by the "CLK" switches on the interface. Subsequent diagrams and text further explain the choices.

**Optional Peripheral Status Check.** Many peripheral devices are equipped with a line which is used to indicate the device's current "OK-or-Not-OK" status. If this line is connected to the Peripheral Status line (PSTS) of the GPIO Interface, the computer may determine the status of the peripheral device by checking the state of the PSTS. The logic sense of this line may be selected by setting the "PSTS" switch. If the switch is enabled, the computer performs a one-time check of the Peripheral Status line (PSTS) before initiating any transfers as part of the data-transfer handshake. If PSTS indicates "Not OK," Error 172 is reported; otherwise, the transfer proceeds normally. If this feature is not enabled, this one-time check is never made. This feature is available with both Full-Mode and Pulse-Mode Handshakes.

## Interface Reset

The interface should always be reset before use to ensure that it is in a known state. All interfaces are automatically reset by the computer at certain times: when the computer is powered on, when RESET is pressed. The interface may be optionally reset at other times under control of BASIC programs. Two examples are as follows:

```
Gpio=12
CONTROL Gpio,0;1

Reset=1
CONTROL Gpio;Reset
```

The following action is invoked whenever the GPIO Interface is reset:

- The Peripheral Reset line (PRESET) is pulsed Low for at least 15 microseconds.
- The PCTL line is placed in the Clear state.
- If the DOUT CLEAR jumper is installed, the Data Out lines are all cleared (set to logic 0).
- The interrupt enable bit is cleared, disabling subsequent interrupts until re-enabled by the program.

The following lines are unchanged by a reset of the GPIO Interface:

- The CTL0 and CTL1 output lines.
- The I/O line.
- The Data Out lines, if the DOUT CLEAR jumper is not installed.

## Using OUTPUT and ENTER Through the GPIO

This section shows you how to use OUTPUT and ENTER through the GPIO Interface. The actual signals that appear on the data lines depend on three things: the data currently being transferred, how this data is being represented, and the logic sense of the data lines.

This section gives simple examples of how several representations are implemented during OUTPUTs and ENTERs through the GPIO Interface.

**ASCII and Internal Representations.** Data normally passes through the GPIO Interface one byte at a time, with the most significant byte first. This byte-mode transfer is independent of whether FORMAT ON or FORMAT OFF is the I/O path attribute.

**Example Statements Using OUTPUT.** The following examples show how you can use the OUTPUT statement to output data bytes through the GPIO interface.

```
ASSIGN @Gpio TO 12
OUTPUT @Gpio;"ASCII"

Gpio=12
Number=-4
OUTPUT Gpio USING "MD.DD";Number

ASSIGN @Gpio TO 12;FORMAT OFF
String$="1234"
OUTPUT @Gpio;String$
```

**Example Statements Using ENTER.** The following examples show how you can use the ENTER statement to enter data bytes through the GPIO interface.

```
ENTER @Gpio USING "#,B";Byte
DISP "Value Entered = ";Byte

Value Entered = 65

ENTER 12;String$
DISP "String Entered =

String Entered = ruok?

REAL Number
ASSIGN @Gpio TO 12
ENTER @Gpio;Number
DISP "Number = ;&in;Number

Number = 2
```

**Example Statements that Output Data Words.** Data are automatically sent as words when using an I/O path with the WORD attribute:

```
Word=3*256+3
OUTPUT @Gpio USING "#,W";Output_word

Output_16_bits=-1
CONTROL Gp_isc,3;Output_16_bits
```

It is important to note that no output handshake is executed when the CONTROL statement is executed; only the states of the Data Out lines and the I/O lines are affected. Handshake sequence, if desired, must be performed by BASIC statements in the program.

**Example Statements that Enter Data Words.** Data are automatically received as words when using an I/O path with the WORD attribute:

```
ENTER 12 USING "#,W";Enter_16_bits
DISP "INTEGER entered = ";Enter_16_bits

INTEGER entered = 511

STATUS Gp_isc,3;Enter_16_bits
DISP "INTEGER entered = ";Enter_16_bits

INTEGER entered = -512
```

It is important to note that no enter handshake is performed when the STATUS statement is executed. The only actions taken are the I/O lines being placed in the High state and the Data In registers being read. If an enter handshake is required, it must be performed by the BASIC program.

Remember also that the Data In Clock source is solely determined by the switch setting on the interface card. Thus, when the STATUS statement is used to read the Data In lines, the data on the lines may or may not be clocked into the registers when the statement is executed. If the data are to be clocked in by the STATUS statement, the "READ" clock source must be selected. See the installation manual for further details.

## GPIO Timeouts

This section explains how the time parameter is measured and describes typical service routines.

**Timeout Time Parameter.** There are two general time intervals measured and compared to the specified TIMEOUT time. The first interval is measured between the computer initiating the first handshake (PCTL=Set) and the peripheral signaling Ready (with the PFLG line). If the peripheral does not indicate readiness by the specified TIMEOUT time parameter, a TIMEOUT event occurs.

The time elapsed during each handshake is also measured and compared to the TIMEOUT time. The timing begins when the transfer is initiated (PCTL Set by the computer) and, in general, ends when the peripheral responds on the PFLG line.

Keep in mind that the TIMEOUT time parameter specifies the minimum time that the computer will wait before initiating the ON TIMEOUT branch. However, the computer may occasionally wait an additional 25 percent of the specified time parameter before initiating the branch. For instance, if a time of 0.4 seconds is specified, the computer will wait at least 0.4 seconds for the handshake to be completed, but it may occasionally wait up to 0.5 seconds before initiating the ON TIMEOUT branch.

**Timeout Service Routines.** The service routine usually responds by determining if the peripheral is functioning properly ("OK") or is down ("not OK"). The simplest action that might be taken by the computer is to read the state of the PSTS signal line, as shown in the following service routine (found in file GPIOSERV on you Manuals Examples disk).

A TIMEOUT has been set up to occur if the peripheral takes approximately more than .08 seconds to complete its response during a data transfer; how the peripheral completes its response depends on the handshake mode currently selected. With Pulse-Mode Handshakes, the peripheral completes its response by using PFLG to Clear PCTL; with Full-Mode Handshakes, the response is complete only after PCTL has been Cleared and PFLG is in the Ready state.

When a TIMEOUT occurs, the computer automatically executes an Interface Reset; the PCTL line is Set and then Cleared, and the PRESET line is pulsed Low. See the section called "Interface Reset" for further effects. The Service routine checks the PSTS line to see if the peripheral is OK or not OK. If not OK, a message is displayed and the program is paused; if OK, program execution is returned to the line following that in which the TIMEOUT occurred. A service routine may be programmed to attempt the transfer again, if desired; however, the automatic Reset performed when the TIMEOUT occurred may make this type of response difficult to implement.

## GPIO Interrupts

This section describes the types of and techniques for using the interrupts available on the GPIO Interface.

**Types of Interrupt Events.** The GPIO Interface can sense two interrupt events:

- The interface becoming "Ready" for subsequent handshakes.
- The External Interrupt Request Line (EIR) being driven to logic low by the peripheral.

Since both of these events initiate identical computer responses, the service routine must be able to determine which of these interrupts has occurred.

**Setting Up and Enabling Events.** When either event occurs, the interrupt is logged by the operating system. After logging the occurrence, any further interrupts from the GPIO Interface are automatically disabled until specifically enabled by a program. All further computer responses to either event depend entirely on the BASIC program currently in memory.

The following program segment shows the steps involved in setting up and enabling Ready Interrupts.

```
100  Gpio=12
110  ON INTR Gpio GOSUB Gpio_serv
120  !
130  Mask=2
140  ENABLE INTR Gpio;Mask
```

The value of the interrupt mask determines which, if any, of the GPIO interrupt events are to be enabled to initiate the corresponding branch. Bits of the Interrupt Mask register have the following definitions.

## Interrupt Enable Register: (ENABLE INTR)

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| Not Used | | | | | | Enable Interface Ready Interrupts | Enable EIR Interrupts |
| Value = 128 | Value = 64 | Value = 32 | Value = 16 | Value = 8 | Value = 4 | Value = 2 | Value = 1 |

**Interface Ready.** Setting this bit ("1") enables an interrupt to initiate the ON INTR branch when the interface detects that it is Ready to handshake data. If Full-Mode Handshake is selected (with the Option Select switch), the Ready event is PCTL=Clear and PFLG=Ready. With Pulse-Mode Handshake, the event is PCTL=Clear (independent of the state of PFLG).

**External Interrupt Request.** Setting this bit ("1") enables an interrupt to initiate the ON INTR branch when the interface senses an External Interrupt Request (EIR line=Low).

## Interrupt Service Routines

If both events are enabled, the service routine must be able to differentiate between the two. And, if both have occurred, the service routine must be able to service both causes. The following registers contain the current state of the Interface Ready flag and EIR signal lines, from which the interrupt cause(s) may be determined.

## Status Register 4: Interface Ready

The interface is ready for a subsequent data transfer; "1" = Ready, "0" = Busy.

## Status Register 5: Peripheral Status

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | PSTS<br>OK | EIR<br>Line Low | STI1<br>Line Low | STI0<br>Line Low |
| Value = 128 | Value = 64 | Value = 32 | Value = 16 | Value = 8 | Value = 4 | Value = 2 | Value = 1 |

As mentioned in preceding paragraphs, these two interrupt causes are both level-sensitive events, not edge-triggered events. This fact has two important implications. The first is that, for an event to be recognized, the corresponding signal line must be held in the interrupting state until the computer can interrogate the line's logic state. If the signal line's state is changed before the service routine checks the line, the interrupt may be "missed". This will happen only if both events are enabled; if only one event is enabled, determining the cause may not be necessary.

The second implication is that the service routine must be able to acknowledge the request in order for the peripheral device to remove the request. If the request is not removed after service, the same request may be serviced more than once.

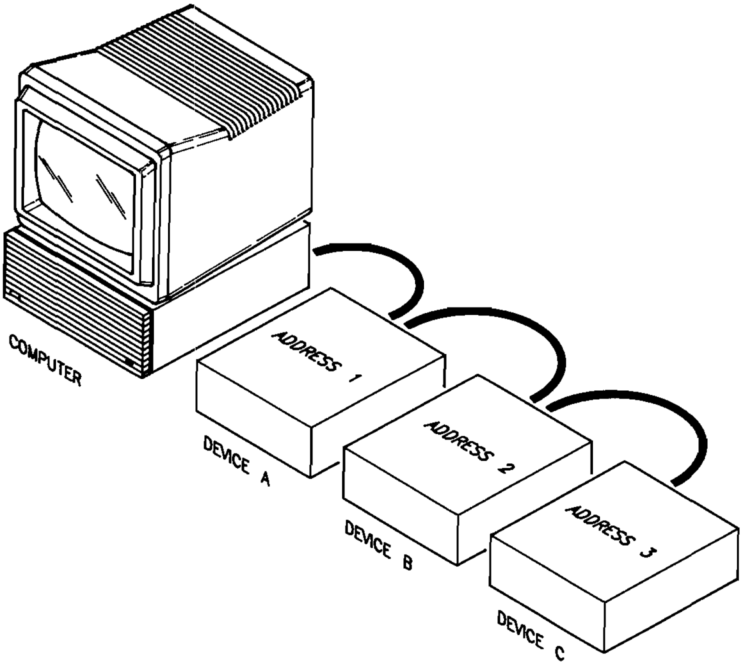
The program found in file EIRSERV on your Manuals Examples disk shows a simple example of servicing an External Interrupt Request. Note that only EIR-type interrupts have been enabled and that the peripheral device provides its own interrupt cause with signals on the STI0 and STI1 lines.

A slightly different method that peripherals use to communicate the cause of their interrupt request is to place the interrupt cause on the data lines concurrent with the interrupt request. The service routine can determine the cause by reading STATUS register 3 and take the appropriate action.

Notice that the service routine indicates a likely place for a Ready-interrupt service routine. The Service routine must check for the Ready condition, acknowledge the interrupt, and then take the desired action. In this case, no service action has been defined because Ready interrupts have not been enabled.

# The HP-HIL Interface

HP-HIL (Hewlett-Packard Human Interface Link) is an interface capable of supporting up to seven devices (such as a mouse, keyboard, or digitizer) generally related to human input. The following diagram illustrates the basic components of the HP-HIL interface.



HP-HIL initialization occurs when you boot HP BASIC. HP BASIC logs the HP-HIL devices present on the link. The link can deal with a maximum of seven devices at a time. Any devices added after the seventh are ignored. If you add a de vice to the link after HP Basic is booted, the device will not be recognized by the system unless you boot HP BASIC again. Also, if you replace an HP-HIL device with a different one, the system may misinterpret data coming from the new device. Again, you must reboot HP BASIC in order for the system to recognize the new device.

The address of a device is simply its topological order of placement along the link. In the above diagram, device A has address 1, B has address 2, and C has address 3. This is only a result of their physical order of connection. If device C had been connected between devices A and B, device A would still be address 1, but device C would be address 2 and device B address 3. The type of device has no bearing on the address assigned to it.

After the link is operational and subsequent link operations, each device looks at the data being sent down the link. If a device sees that the destination address associated with the data is the same as its address, that device receives and acts on the data. Otherwise, the data is sent on to the next device.

## Preview of HP-HIL Devices

HP-HIL devices can be divided into a number of different categories. This section provides you with a table that includes these categories as well as a list of high level and low level statements that apply to each category.

| HP-HIL Device Categories | High Level BASIC Access | Low Level BASIC Access |
|---|---|---|
| HP-HIL Keyboard | Operating system normally handles keystrokes. Programs can enter text and numbers with the INPUT, LINPUT, and ENTER statements. | ON/OFF KEY<br>ON/OFF KBD<br>KBD$ |
| Relative Positioner (mouse, etc.) | Operating system handles as cursor movement input. Can also be used with GRAPHICS INPUT IS. | ON/OFF KBD (traps movement as arrow keys and also traps mouse buttons.)<br>KBD$<br>ON/OFF KNOB<br>ON/OFF CDIAL<br>CDIAL |
| Absolute Positioner (digitizing tablet, etc.) | Can be used with GRAPHICS INPUT IS. | HIL SEND<br>ON HIL EXT<br>HILBUF$ |
| ID Module | One can be used with SYSTEM$("SERIAL NUMBER"). | HIL SEND<br>ON HIL EXT<br>HILBUF$ |
| Other Devices | None | HIL SEND<br>ON HIL EXT<br>HILBUF$ |

## Communicating Through the HP-HIL Interface

This section provides a brief description of the HP-HIL Interface Driver. This driver supports a set of statements which allow communications between the HP-HIL interface and the HP-HIL devices connected to it. Refer to the appropriate command listing in the BASIC *Language Reference* manual for detailed information on these statements.

`HIL SEND`
*Address;HIL_Command*

Allows you to send HP-HIL commands to an HP-HIL device (for example, `HIL SEND 1;IDD`). The basic HP-HIL commands are presented in the next section. Address is the location of the device in the HP-HIL link. Address 1 is as signed to the first device on the link that is addressable. Subsequent addresses are assigned in ascending order.

`ON HIL EXT`
*Address_mask Branch*

Enables end-of-line interrupts from HP-HIL devices, allowing you to receive interrupts from up to seven devices on the HP-HIL link. Address_mask is a bit-map of the locations of the device or devices in the HP-HIL link. The default value is 254 which allows up to seven devices to send interrupts. Branch refers to a branch to a program line number, label, subroutine, or subprogram using the keywords GOTO, GOSUB, RECOVER or CALL.

`OFF HIL EXT`

This statement disables all previously enabled end-of-line interrupts for HP-HIL devices. Note that this statement does not require an address mask.

`HILBUF$`

This is a function used to capture data returned from HP-HIL devices. This function provides a 256 byte buffer for data to be stored in after execution of the first two statements listed above. Once the limit of 256 bytes has been reached, the buffer will not receive any new data until it has been emptied by a read. The first byte stored in the buffer tells you how many bytes of data have been lost. This byte is initially null.

## Supported HP-HIL Devices

This section provides a brief description of those devices supported by the HP-HIL Interface Driver, and the use of a program for identifying all devices on the HP-HIL link.

**Identifying All Devices on the HP-HIL Link.** Each device in the HP-HIL link has a device ID that identifies the device and a Describe Record that provides you with device characteristics. This information can be obtained by executing the HP-HIL command IDD and parsing the string value returned by using the HILBUF$ function. A program called HIL_ID on the Manual Examples disk makes use of the IDD command and the HILBUF$ function for the following:

- Determining if a device is recognized as being on the HP-HIL link.
- Identifying the device at a specific address.
- Determining the device's characteristics.

Assume that your HP-HIL link has the following devices:

- Touchscreen located at address 1.
- ITF keyboard located at address 2.
- Function box located at address 3.

Executing the HIL_ID program will produce the following output:

```
HP 25273A (Touchscreen) located at address 1
 Describe Record Information
   Descriptor Information
   Does not support Prompts/Acknowledges 1 thru 7
   Supports Proximity Detection
   Does not report buttons
  X and Y axis information reported
   Absolute positioning device
   Returns 8 bits/axis

HP 46020/21A (ITF Keyboard) located at address 2
 Describe Record Information
  No special features

HP 46086A (Function Box) located at address 3
 Describe Record Information
  I/O Descriptor Information
   Recognizes General Prompt and Acknowledge
   Does not support Prompts and Acknowledges 1 thru 7
   Does not report buttons
  No axis information reported

NO MORE DEVICES
```

The first device is a touchscreen located at address 1 in the HP-HIL link. The Describe Record provides you with the characteristics of the device. This information is as follows:

- I/O Descriptor Byte information is reported. The information supplied in this byte tells you that when you touch your finger on the screen or remove it from the screen, it will be detected. This is called proximity in/out detection.
- It is an absolute positioning device. This means that every coordinate position on the screen is referenced to the lower left-hand corner of the screen (X coordinate = 0 and Y coordinate = 0).
- X and Y axis information is reported. This tells you that Poll Records received when communicating with this device will contain X and Y coordinate information. These are absolute coordinate positions.
- Coordinate information is returned as 8 bits per axis. This means that there will be only one byte of information for each coordinate (X and Y) returned in the poll record.

**HP-HIL Keyboards.** The following keyboards are supported HP-HIL devices:

- ITF keyboard (HP 46021).
- Integral PC keyboard (modified ITF layout).
- HP 98203C.
- Vectra HP-HIL keyboard (HP BASIC Language Processor only).

To perform interrupt branching with the keyboard keys, you need to use the following statements and function:

`ON/OFF KEY`

ON KEY defines and enables an event-initiated branch to be taken when a soft key is pressed. OFF KEY cancels event-initiated branches previously defined and enabled by and ON KEY statement.

`ON/OFF KBD`

ON KBD defines and enables an event-initiated branch to be taken when a key is pressed. OFF KBD cancels event-initiated branches previously defined and enabled by the ON KBD statement.

`KBD$`

This function returns the contents of the keyboard buffer when ON KBD is active.

**Relative Positioners.** The following devices are considered to be relative positioners:

- HP 46060A Mouse.
- HP 46083A Rotary Control Knob.
- HP 98203C Keyboard.
- HP 46094A HP-HIL Quadrature Port using HP 46095A Quadrature 3-button Mouse.

These devices support the ON/OFF KBD and KBD$ statements and functions in the same manner as described in the previous section. In addition, the following statements are also supported.

`ON/OFF KNOB`

ON KNOB defines and enables an event-initiated branch to be taken when the relative positioner is moved. OFF KNOB cancels event-initiated branches previously defined and enabled by the ON KNOB statement. Subsequent use of the relative positioner results in normal scrolling or cursor movement.

`DIGITIZE`

This statement is used when graphics input has been specified as a relative positioner by the statement

```
GRAPHICS INPUT IS KBD,"KBD"
```

It inputs the X and Y coordinates of a digitized point from the locator specified by the GRAPHICS INPUT IS statement (KBD in this case).

`READ LOCATOR`

This statement is used when graphics input has been specified as a relative positioner by the statement

```
GRAPHICS INPUT IS KBD,"KBD"
```

It samples the locator device without waiting for a digitizing operation.

In addition, the HP 46085A Control Dials is a device with nine knobs. Refer to ON CDIAL, OFF CDIAL, and CDIAL (n) in the BASIC *Language Reference* manual for information on accessing this device.

**Absolute Positioners.** The following devices are considered absolute positioners:

- HP 35723A HP-HIL Touchscreen.
- HP 46087A A-Size Digitizer.
- HP 46088A B-Size Digitizer.

These devices can generate ON HIL EXT interrupts any time *except* when the absolute positioner has been specified as the input graphics device in a GRAPHICS INPUT IS statement:

```
GRAPHICS INPUT IS KBD,"TABLET"
```

Using HIL SEND to transmit a command other than IDD to these devices in this situation will result in an error. Due to the speed with which data is returned from the digitizers, an HP BASIC program cannot keep up with them using ON HIL EXT because HILBUF$ overflows. The only device in this group capable of using the ON HIL EXT statement is the touchscreen.

When these devices are specified as the graphics input device (as above in the GRAPHICS INPUT IS statement), the statements you may use are:

`DIGITIZE X_coord, Y_coord` Inputs the X and Y coordinates of a digitized point.

`READ LOCATOR X_coord,Y_coord` Samples the locator device without waiting for a digitize operation.

**Security Device.** The HP 46084A HP-HIL Module is an HP-HIL device that returns an identification number that identifies you as the computer user. The identification number is unique to your particular ID module. This allows application programs to use the the ID module to control access to program functions, data bases and networks.

**Other Devices.** The following devices can generate ON HIL EXT interrupts and respond to various HIL SEND commands.

*HP 46086A Function Box* — The HP 46086A Function Box provides 32 keys to select software-defined functions. It has an LED that acts as a visual prompt for any purpose you as sign to it. The HP 46086A Function Box responds to the following HP-HIL commands when sent by the HIL SEND statement:

- PRM
- ACK
- DKA
- EKA 1
- EKA 2

*HP 92916A Bar Code Reader* — The HP 92916A Bar Code Reader reads all standard bar codes using a wand as the input device. It provides you with an effective and reliable alternative to the time-consuming keyboard for data entry. Note that HP BASIC supports this device in both the ASCII transmit mode (where the input from the device is ASCII characters), and in the keyboard mode* where it transmits the same keycodes as an HP 46020/21A Keyboard.

When the HP 92916A Bar Code Reader is in the ASCII transmit mode, use the following statement:

```
ON HIL EXT
```

When it is in the keyboard mode, use the following statements:

```
ON KBD
ENTER KBD
INPUT
LINPUT
```

* When in the keyboard mode, this device returns an HP-HIL ID in the same range as an HP 46021 (ITF) Keyboard.

# Index